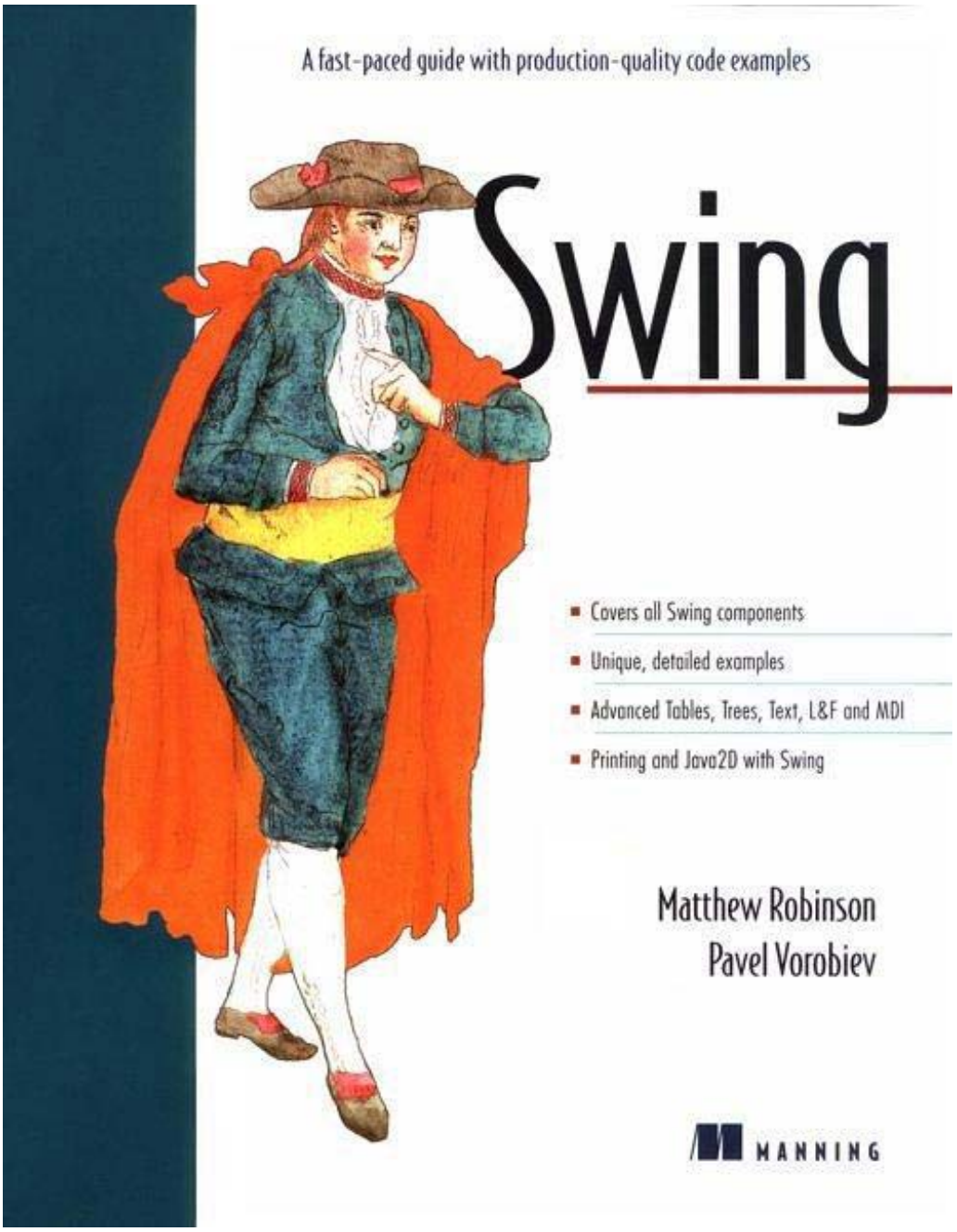


This book was compiled in a single PDF file by The Admin®. Visit website at http://theadmin.data.bg

Part I - Foundations

Part I consists of two chapters that lay the foundation for a successful and productive journey through the JFC Swing class library. The first begins with a brief overview of what Swing is and an introduction to its architecture. The second builds up into a detailed discussion of the key mechanisms underlying Swing, and how to interact with them. There are several sections on topics that are fairly advanced, such as multithreading and painting. This material is central to many areas of Swing and by introducing it in chapter 2, your understanding of what is to come will be significantly enhanced. We expect that you will want to refer back to this chapter quite often, and in several places we explicitly refer you to it in the text. At the very least, it is recommended that you know what chapter 2 contains before moving on.

Chapter 1. Swing Overview

In this chapter:

- AWT
- Swing
- MVC
- UI delegates and PLAF

1.1AWT

AWT (the Abstract Window Toolkit) is the part of Java designed for creating user interfaces and painting graphics and images. It is a set of classes intended to provide everything a developer requires in order to create a graphical interface for any Java applet or application. Most AWT components are derived from the `java.awt.Component` class as figure 1.1 illustrates. (Note that AWT menu bars and menu bar items do not fit within the `Component` hierarchy.)

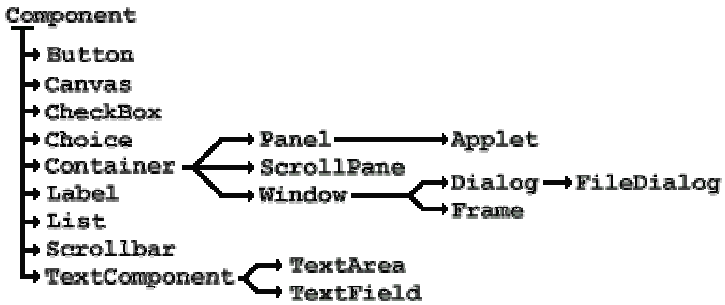


Figure 1.1 Partial Component hierarchy

<<file figure1-1.gif>>

The Java Foundation Classes consist of five major parts: AWT, Swing, Accessibility, Java 2D, and Drag and Drop. Java 2D has become an integral part of AWT, Swing is built on top of AWT, and Accessibility support is built into Swing.

The five parts of JFC are certainly not mutually exclusive, and Swing is expected to merge more deeply with AWT in future versions of Java. The Drag and Drop API was far from mature at the time of this writing but we expect this technology to integrate further with Swing and AWT in the near future. Thus, AWT is at the core of JFC, which in turn makes it one of the most important libraries in Java 2.

1.2 Swing

Swing is a large set of components ranging from the very simple, such as labels, to the very complex, such as tables, trees, and styled text documents. Almost all Swing components are derived from a single parent called `JComponent` which extends the AWT `Container` class. Thus, Swing is best described as a layer on top of AWT rather than a replacement for it. Figure 1.2 shows a partial `JComponent` hierarchy. If you compare this with the AWT `Component` heirarchy of figure 1.1 you will notice that for each AWT component there is a Swing equivalent with prefix "J". The only exception to this is the AWT `Canvas` class, for which `JComponent`, `JLabel`, or `JPanel` can be used as a replacement (in section 2.8 we discuss this in detail). You will also notice many Swing classes with no AWT counterparts.

Figure 1.2 represents only a small fraction of the Swing library, but this fraction consists of the classes you will be dealing with most. The rest of Swing exists to provide extensive support and customization capabilities for the components these classes define.

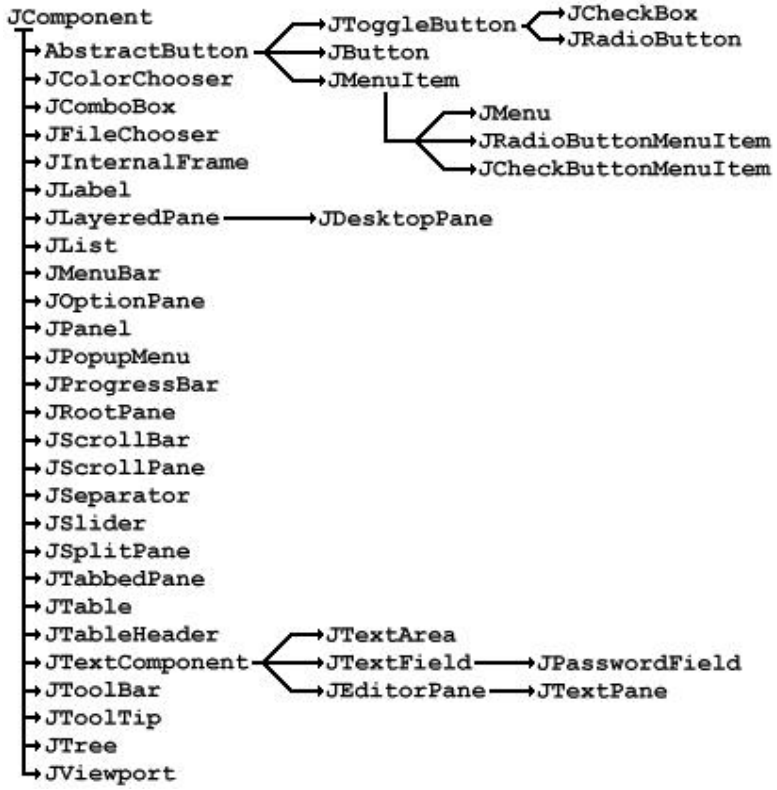


Figure 1.2 Partial JComponent hierarchy

<<file figure1-2.gif>>

1.2.1 Z-order

Swing components are referred to as *lightweight*s while AWT components are referred to as *heavyweight*s. The difference between lightweight and heavyweight components is *z-order*: the notion of depth or layering. Each heavyweight component occupies its own z-order layer. All lightweight components are contained inside heavyweight components and maintain their own layering scheme defined by Swing. When we place a heavyweight inside another heavyweight container it will, by definition, overlap all lightweights in that container.

What this ultimately means is that we should avoid using both heavyweight and lightweight components in the same container whenever possible. This does not mean that we can *never* mix AWT and Swing components successfully. It just means we have to be careful and know which situations are safe and which are not. Since we probably won't be able to completely eliminate the use of heavyweight components anytime soon, we have to find ways to make the two technologies work together in an acceptable way.

The most important rule to follow is that we should never place heavyweight components inside lightweight containers that commonly support overlapping children. Some examples of these containers are `JInternalFrame`, `JScrollPane`, `JLayeredPane`, and `JDesktopPane`. Secondly, if we use a popup menu in a container holding a heavyweight component, we need to force that popup to be heavyweight. To control this for a specific `JPopupMenu` instance we can use its `setLightWeightPopupEnabled()` method.

Note: For `JMenu`s (which use `JPopupMenu`s to display their contents) we first have to use the `getPopupMenu()` method to retrieve the associated popup menu. Once retrieved we can then call `setLightWeightPopupEnabled(false)` on that popup to enforce heavyweight functionality. This needs to be done with each `JMenu` in our application, including menus contained within menus, etc.

Alternatively we can call `JPopupMenu`'s static `setDefaultLightWeightPopupEnabled()` method, and pass it a value of `false` to force all popups in a Java session to be heavyweight. Note that this will only affect popup menus created *after* this call is made. It is therefore a good idea to call this method early within initialization.

1.2.2 Platform independence

The most remarkable thing about Swing components is that they are written in 100% Java and do not depend on peer components, as most AWT components do. This means that a Swing button or text area will look and function identically on Macintosh, Solaris, Linux, and Windows platforms. This design eliminates the need to test and debug applications on each target platform.

Note: The only exceptions to this are four heavyweight Swing components that are direct subclasses of AWT classes relying on platform-dependent peers: `JApplet`, `JDialog`, `JFrame`, and `JWindow`. See chapter 3.

1.2.3 Swing package overview

`javax.swing`

Contains the most basic Swing components, default component models, and interfaces. (Most of the classes shown in Figure 1.2 are contained in this package.)

`javax.swing.border`

Classes and interfaces used to define specific border styles. Note that borders can be shared by any number of Swing components, as they are not components themselves.

`javax.swing.colorchooser`

Classes and interfaces supporting the `JColorChooser` component, used for color selection. (This package also contains some interesting undocumented private classes.)

`javax.swing.event`

The event package contains all Swing-specific event types and listeners. Swing components also support events and listeners defined in `java.awt.event` and `java.beans`.

`javax.swing.filechooser`

Classes and interfaces supporting the `JFileChooser` component, used for file selection.

`javax.swing.plaf`

Contains the pluggable look-and-feel API used to define custom user interface components. Most of the classes in this package are abstract. They are subclassed and implemented by look-and-feel implementations such as metal, motif, and basic. The classes in this package are intended for use only by developers who, for one reason or another, cannot build on top of existing look-and-feels.

`javax.swing.plaf.basic`

Consists of the Basic look-and-feel implementation which all look-and-feels provided with Swing are built on top of. We are normally expected to subclass the classes in this package if we want to create our own customized look-and-feel.

`javax.swing.plaf.metal`

Metal is the default look-and-feel of Swing components. It is the only look-and-feel that ships with Swing not designed to be consistent with a specific platform.

`javax.swing.plaf.multi`

This is the Multiplexing look-and-feel. This is not a regular look-and-feel implementation in that it does not define the actual look or feel of any components. Rather, it provides the ability to combine several look-and-feels for simultaneous use. A typical example might be using an audio-based look-and-feel in combination with metal or motif. Currently Java 2 does not ship with any multiplexing look-and-feel implemenations (however, rumor has it that the Swing team is working on an audio look-and-feel as we write this).

`javax.swing.table`

Classes and interfaces supporting the `JTable` control. This component is used to manage tabular data in spreadsheet form. It supports a high degree of customization without requiring look-and-feel enhancements.

`javax.swing.text`

Classes and interfaces used by the text components including support for plain and styled documents, the views of those documents, highlighting, caret control and customization, editor actions and keyboard customization.

`javax.swing.text.html`

This extension of the text package contains support for HTML text components. (HTML support was being completely rewritten and expanded upon while we were writing this book. Because of this our coverage of it is regretably limited.)

`javax.swing.text.html.parser`

Support for parsing HTML.

`javax.swing.text.rtf`

Contains support for RTF documents.

`javax.swing.tree`

Classes and interfaces supporting the `JTree` component. This component is used for the display and management of hierarcical data. It supports a high degree of customization without requiring look-and-feel enhancements.

`javax.swing.undo`

The `undo` package contains support for implementing and managing undo/redo functionality.

1.3 MVC architecture

MVC is a well known object-oriented user interface design decomposition that dates back to the late 1970s. Components are broken down into three parts: a model, a view, and a controller. Each Swing component is based on a more modern version of this design. Before we discuss how MVC works in Swing, we need to understand how it was originally designed to work.

Note: The three-way separation described here is only used today by a small number of user interface frameworks, VisualWorks being the most notable.

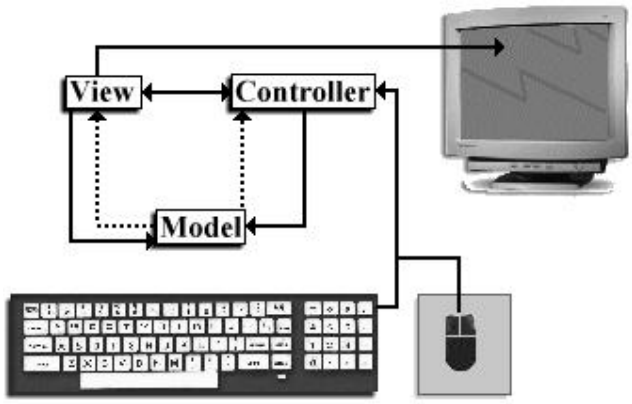


Figure 1.3 Model-view-controller architecture

<<file figure1-3.gif>>

1.3.1 Model

The model is responsible for maintaining all aspects of the component state. This includes, for example, such values as the pressed/unpressed state of a push button, a text component's character data and information about how it is structured, etc. A model may be responsible for *indirect* communication with the with the view and the controller. By indirect we mean that the model does not 'know' its view and controller--it does not maintain or retreive references to them. Instead the model will send out notifications or *broadcasts* (what we know as events). In figure 1.3 this indirect communication is represented by dashed lines.

1.3.2 View

The view determines the visual representation of the component's model. This is a component's "look." For example, the view displays the correct color of a component, whether the component appears raised or lowered (in the case of a button), and the rendering of a desired font. The view is responsible for keeping its on-screen representation updated

and may do so upon receiving indirect messages from the model, or direct messages from the controller.

1.3.3 Controller

The controller is responsible for determining whether the component should react to any input events from input devices such as the keyboard or mouse. The controller is the "feel" of the component, and it determines what actions are performed when the component is used. The controller can receive direct messages from the view, and indirect messages from the model.

For example, suppose we have a checked (selected) checkbox in our interface. If the controller determines that the user has performed a mouse click it may send a message to the view. If the view determines that the click occurred on the checkbox it sends a message to the model. The model then updates itself and broadcasts a message, which will be received by the view(s), to tell it that it should update itself based on the new state of the model. In this way, a model is not bound to a specific view or controller, allowing us to have several views and controller's manipulating a single model.

1.3.4 Custom view and conroller

One of the major advantages MVC architecture provides is the ability to customize the "look" and "feel"of a component without modifying the model. Figure 1.4 shows a group of components using two different user interfaces. The important point to make about this figure is that the components shown are actually the same, but they are shown using two different *look-and-feel* implementations (different views and conrollers -- discussed below).

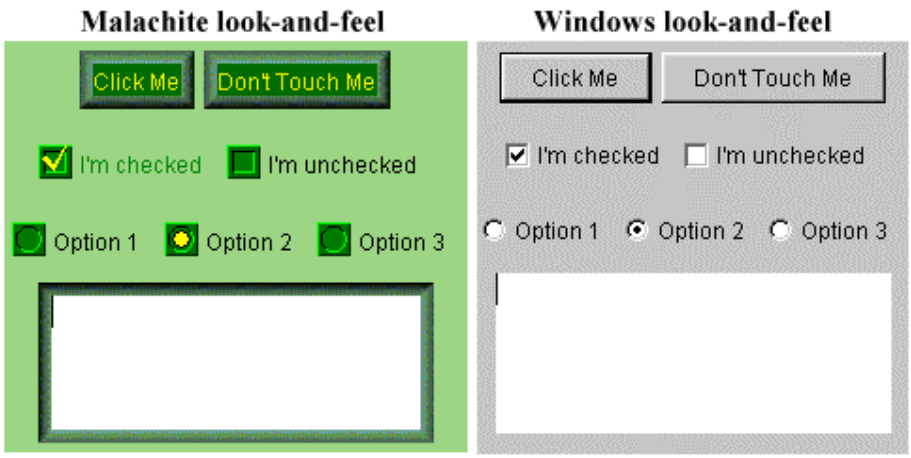


Figure 1.4 Malachite and Windows look-and-feels of the same components

<<file figure1-4.gif>>

Some Swing components also provide the ability to customize specific parts of a component without affecting the model. More specifically, these components allow us to define custom cell renderers and editors used to display and accept specific data respectively. Figure 1.5 shows the columns of a table containing stock market data rendered with custom icons and colors. We will examine how to take advantage of this functionality in our study of Swing combo boxes, lists, tables, and trees.

Sun Microsystems	140 5/8	10.625	SUNW	130 15/16	10	17,734,600
Lucent Technology	64 5/8	9.65	LU	59 15/16	4 11/16	29,856,300
Dell Computers	46 3/16	6.24	DELL	44 1/2	1 11/16	47,310,000
Sony Corp.	96 3/16	1.18	SNE	95 5/8	1 1/8	330,600
Hitachi, Ltd.	78 1/2	1.12	HIT	77 5/8	7/8	49,400
Enamelon Inc.	4 7/8	0.0	ENML	5	-1/8	35,900
AT&T	65 3/16	-0.1	T	66	-13/16	554,000
Intl. Bus. Machines	183	-0.51	IBM	183 1/8	-1/8	4,371,400
Microsoft Corp.	94 1/16	-0.92	MSFT	95 3/16	-1 1/8	19,836,900
Egghead.com	17 1/4	-1.43	EGGS	17 7/16	-3/16	2,146,400
Sprint	104 9/16	-1.82	FON	106 3/8	-1 13/16	1,135,100
Hewlett-Packard	70	-2.01	HWP	71 1/16	-1 7/16	2,410,700
Compaq Computers	30 7/8	-2.18	CPQ	31 1/4	-3/8	11,853,900

Figure 1.5 Custom rendering

<<file figure1-5.gif>>

1.3.5 Custom models

Another major advantage of Swing's MVC architecture is the ability customize and replace a component's data model. For example, we can construct our own text document model that enforces the entry of a date or phone number in a very specific form. We can also associate the same data model with more than one component (as we discussed above in looking at MVC). For instance, two `JTextAreas` can store their textual content in the same document model, while maintaining two different views of that information.

We will design and implement our own data models for `JComboBox`, `JList`, `JTree`, `JTable`, and extensively throughout our coverage of text components. Below we've listed some of Swing's model interface definitions along with a brief description of what data their implementations are designed to store, and what components they are used with:

`BoundedRangeModel`

Used by: `JProgressBar`, `JScrollBar`, `JSlider`.

Stores: 4 integers: value, extent, min, max.

The value and the extent must be between a specified min and max values. The extent is always <= max and >=value.

`ButtonModel`

Used by: All `AbstractButton` subclasses.

Stores: A boolean representing whether the button is selected (armed) or unselected (disarmed).

`ListModel`

Used by: `JList`.

Stores: A collection of objects.

`ComboBoxModel`

Used by: `JComboBox`.

Stores: A collection of objects and a selected object.

`MutableComboBoxModel`

Used by: `JComboBox`.

Stores: A `Vector` (or another mutable collection) of objects and a selected object.

`ListSelectionModel`

Used by: `JList`, `TableColumnModel`.

Stores: One or more indices of selected list or table items. Allows single, single-interval, or multiple-interval selections.

`SingleSelectionModel`

Used by: `JMenuBar`, `JPopupMenu`, `JMenuItem`, `JTabbedPane`.

Stores: The index of the selected element in a collection of objects owned by the implementor.

`ColorSelectionModel`

Used by: `JColorChooser`.

Stores: A `Color`.

`TableModel`

Used by: `JTable`.

Stores: A two dimensional array of objects.

`TableColumnModel`

Used by: `JTable`.

Stores: A collection of `TableColumn` objects, a set of listeners for table column model events, width between each column, total width of all columns, a selection model, and a column selection flag.

`TreeModel`

Used by: `JTree`.

Stores: Objects that can be displayed in a tree. Implementations must be able to distinguish between branch and leaf objects, and the objects must be organized hierarchically.

`TreeSelectionModel`

Used by: `JTree`.

Stores: Selected rows. Allows single, contiguous, and discontiguous selection.

`Document`

Used by: All text components.

Stores: Content. Normally this is text (character data). More complex implementations support styled text, images, and other forms of content (e.g. embedded components).

Not all Swing components have models. Those that act as containers, such as `JApplet`, `JFrame`, `JLayeredPane`, `JDesktopPane`, `JInternalFrame`, etc. do not have models. However, interactive components such as `JButton`, `JTextField`, `JTable`, etc. *do* have models. In fact some Swing components have more than one model (e.g. `JList` uses a model to hold selection information, and another model to store its data). The point is that MVC is not hard and fastened rule in Swing. Simple components, or complex components that don't store lots of information (such as `JDesktopPane`), do not need separate models. The view and controller of each component is, however, almost always separate for each component, as we will see in the next section.

So how does the component itself fit into the MVC picture? The component acts as a mediator between the model(s), the view and the controller. It is neither the M, the V, or the C, although it can take the place of any or all of these parts if we design it to. This will become more clear as we progress through this chapter, and throughout the rest of the book.

1.4 UI delegates and PLAF

Almost all modern user interface frameworks coalesce the view and controller, whether they are based on SmallTalk, C++, and now Java. Examples include MacApp, Smalltalk/V, Interviews, and the X/Motif widgets used in IBM Smalltalk.[1] JFC Swing is the newest addition to this crowd. Swing packages each component's view and controller into an object called a UI delegate. For this reason Swing's underlying architecture is more accurately referred to as *model-delegate* rather than model-view-controller. Ideally communication between both the model and the UI delegate is indirect, allowing more than one model to be associated with one UI delegate, and vice v ersa. Figure 1.6 illustrates.

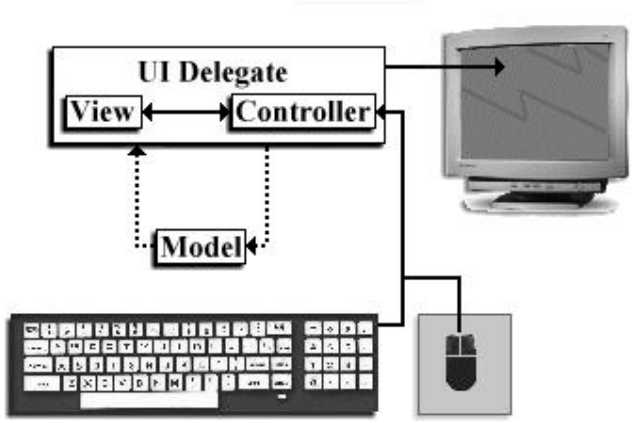


Figure 1.6 Model-delegate architecture

<<file figure1-6.gif>>

1.4.1 The ComponentUI class

Each UI delegate is derived from an abstract class called `ComponentUI`. `ComponentUI` methods describe the fundamentals of how a UI delegate and a component using it will communicate. Note that each method takes a `JComponent` as parameter.

`ComponentUI` methods:

`static ComponentUI CreateUI(JComponent c)`

This is normally implemented to return a shared instance of the UI delegate defined by the defining `ComponentUI` subclass itself. This instance is used for sharing among components of the same type (e.g. All `JButton`s using the Metal look-and-feel share the same static UI delegate instance defined in `javax.swing.plaf.metal.MetalButtonUI` by default.)

`installUI(JComponent c)`

Installs this `ComponentUI` on the specified component. This normally adds listeners to the component and/or its model(s), to notify the UI delegate when changes in state occur that require a view update.

`uninstallUI(JComponent c)`

Removes this `ComponentUI` and any listeners added in `installUI()` from the specified component and/or

[1] Chamond Liu, "Smalltalk, Objects, and Design" Manning Publications Co. 1996.

its model(s).

`update(Graphics g, JComponent c)`

If the component is opaque this should paint its background and then call `paint(Graphics g, JComponent c)`.

`paint(Graphics g, JComponent c)`

Gets all information it needs from the component and possibly its model(s) to render it correctly.

`getPreferredSize(JComponent c)`

Return the preferred size for the specified component based on this `ComponentUI`.

`getMinimumSize(JComponent c)`

Return the minimum size for the specified component based on this `ComponentUI`.

`getMaximumSize(JComponent c)`

Return the maximum size for the specified component based on this `ComponentUI`.

To enforce the use of a specific UI delegate we can call a component's `setUI()` method (note that `setUI()` is declared protected in `JComponent` because it only makes sense in terms of a `JComponent` subclass):

```
JButton m_button = new JButton();
m_button.setUI((MalachiteButtonUI)
  MalachiteButtonUI.createUI(m_button));
```

Most UI delegates are constructed such that they know about a component and its model(s) only while performing painting and other view-controller tasks. Swing normally avoids associating UI delegates on a per-component basis (hence the static instance). However, nothing stops us from assigning our own as the code above demonstrates.

Note: The `JComponent` class defines methods for assigning UI delegates because the method declarations required do not involve component-specific code. However, this is not possible with data models because there is no base interface that all models can be traced back to (i.e. there is no base class such as `ComponentUI` for Swing models). For this reason methods to assign models are defined in subclasses of `JComponent` where necessary.

1.4.2 Pluggable look-and-feel

Swing includes several sets of UI delegates. Each set contains `ComponentUI` implementations for most Swing components and we call each of these sets a *look-and-feel* or a *pluggable look-and-feel* (PLAF) implementation. The `javax.swing.plaf package` consists of abstract classes derived from `ComponentUI`, and the classes in the `javax.swing.plaf.basic` package extend these abstract classes to implement the Basic look-and-feel. This is the set of UI delegates that all other look-and-feel classes are expected to use as a base for building off of. (Note that the Basic look-and-feel cannot be used on its own, as `BasicLookAndFeel` is an abstract class.) There are three pluggable look-and-feel implemenations derived from the Basic look-and-feel:

Windows: `com.sun.java.swing.plaf.windows.WindowsLookAndFeel`
CDE\Motif: `com.sun.java.swing.plaf.motif.MotifLookAndFeel`
Metal (default): `javax.swing.plaf.metal.MetalLookAndFeel`

There is also a `MacLookAndFeel` for simulating Macintosh user interfaces, but this does not ship with Java 2 and must

be downloaded separately. The Windows and Macintosh pluggable look-and-feel libraries are only supported on the corresponding platform.[2]

The multiplexing look-and-feel, `javax.swing.plaf.multi.MultiLookAndFeel`, extends all the abstract classes in `javax.swing.plaf`. It is designed to allow combinations of look-and-feels to be used simultaneously and is intended for, but not limited to, use with Accessibility look-and-feels. The job of each multiplexing UI delegate is to manage each of its child UI delegates.

Each look-and-feel package contains a class derived from the abstract class `javax.swing.LookAndFeel`: `BasicLookAndFeel`, `MetalLookAndFeel`, `WindowsLookAndFeel`, etc. These are the central points of access to each look-and-feel package. We use them when changing the current look-and-feel, and the `UIManager` class (used to manage installed look-and-feels) uses them to access the current look-and-feel's `UIDefaults` table (which contains, among other things, UI delegate class names for that look-and-feel corresponding to each Swing component). To change the current look-and-feel of an application we can simply call the `UIManager`'s `setLookAndFeel()` method, passing it the fully qualified name of the `LookAndFeel` to use. The following code can be used to accomplish this at run-time:

```
try {
  UIManager.setLookAndFeel(
    "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
  SwingUtilities.updateComponentTreeUI(myJFrame);
}
catch (Exception  e) {
  System.err.println("Could not load LookAndFeel");
}
```

`SwingUtilities.updateComponentTreeUI()` informs all children of the specified component that the look-and-feel has changed and they need to discard their UI delegate in exchange for a different one of the specified type.

1.4.3 Where are the UI delegates?[3]

We've discussed `ComponentUI`, and the packages `LookAndFeel` implementations reside in, but we haven't really mentioned anything about the specific UI delegate classes derived from `ComponentUI`. Each abstract class in the `javax.swing.plaf` package extends `ComponentUI` and corresponds to a specific Swing component. The name of each class follows the general scheme of class name (without the "J" prefix) plus a "UI" suffix. For instance `LabelUI` extends `ComponentUI` and is the base delegate used for `JLabel`s.

These classes are extended by concrete implementations such as those in the `basic` and `multi` packages. The names of these subclasses follow the general scheme of look-and-feel name prefix added to the superclass name. For instance,

---

[2] There are some simple ways to get around this but it wouldn't be wise of us to publish them here.

[3] We do not detail the complete functionality and construction of any UI delegate classes in this book. The only reference available at the time of this writing with coverage of the Basic UI delegates is Manning's "Java Foundation Classes: Swing Reference."

`BasicLabelUI` and `MultiLabelUI` both extend `LabelUI` and reside in the `basic` and `multi` packages respectively. Figure 1.7 illustrates the `LabelUI` hierarchy.

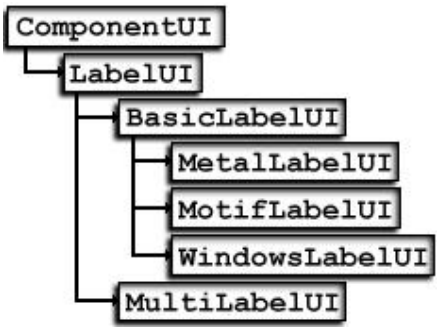


Figure 1.7 LabelUI hierarchy

<<file figure1-7.gif>>

Most look-and-feel implementations are expected to extend the concrete classes defined in the `basic` package, or use them directly. The Metal, Motif, and Windows UI delegates are built on top of Basic versions. The Multi look-and-feel, however, is unique in that each implementation does not extend from Basic, and is merely a shell allowing an arbitrary number of UI delegates to be installed on a given component.

Figure 1.7 should emphasize the fact that Swing supplies a very large number of UI delegate classes. If we were to create an entire pluggable look-and-feel implementation, it is evident that some serious time and effort would be involved. In chapter 21 we will learn all about this process, as well as how to modify and work with the existing look-and-feels.

Chapter 2. Swing Mechanics

In this chapter:

- JComponent properties, sizing, and positioning
- Event handling and dispatching
- Multithreading
- Timers
- AppContext & service classes
- Inside Timers & the TimerQueue
- JavaBeans
- Fonts, Colors, Graphics & text
- Using the Graphics clipping area
- Graphics Debugging
- Painting and Validation

- Focus Management
- Keyboard input, KeyStrokes, and Actions
- SwingUtilities

2.1JComponent properties, size, and positioning

2.1.1 Properties

All Swing components conform to the JavaBeans specification. In section 2.7 we will discuss this in detail. Among the five features a JavaBean is expected to support is a set of properties and associated accessor methods. A *property* is a global variable, and its accessor methods, if any, are normally of the form `setPropertyname()`, `getPropertyname()` or `isPropertyname()`.

A property that has no event firing associated with a change in its value is called a *simple* property. A *bound* property is one for which `PropertyChangeEvent`s are fired after it changes state. We can register `PropertyChangeListener`s to listen for `PropertyChangeEvent`s through `JComponent`'s `addPropertyChangeListener()` method. A *constrained* property is one for which `PropertyChangeEvent`s are fired *before* a change in state occurs. We can register `VetoableChangeListener`s to listen for `PropertyChangeEvent`s through `JComponent`'s `addVetoableChangeListener()` method. A change can be vetoed in the event handling code of a `VetoableChangeListener` by throwing a `PropertyVetoException`. (There is only one Swing class with constrained properties: `JInternalFrame`).

Note: Each of these event and listener classes is defined in the `java.awt.beans` package.

`PropertyChangeEvent`'s carry three pieces of information with them: name of the property, old value, and new value. Beans can use an instance of `PropertyChangeSupport` to manage the dispatching of `PropertyChangeEvent`s corresponding to each bound property, to each registered listener. Similarly, an instance of `VetoableChangeSupport` can be used to manage the sending of all `PropertyChangeEvent`s corresponding to each constrained property.

Swing introduces a new class called `SwingPropertyChangeSupport` (defined in `javax.swing.event`) which is a subclass of, and almost identical to, `PropertyChangeSupport`. The difference is that `SwingPropertyChangeSupport` has been built to be more efficient. It does this by sacrificing thread safety, which, as we will see later in this chapter, is not an issue in Swing if the multithreading guidelines are followed consistently (because all event processing should occur on only one thread--the event-dispatching thread). So if we are confident that our code has been constructed in a thread-safe mannar, we are encouraged to use this more efficent version, rather than `PropertyChangeSupport`.

Note: There is no Swing equivalent of `VetoableChangeSupport` because there are currently only four constrained properties in Swing--all defined in `JInternalFrame`.

Swing also introduces a new type of property which we will call a *change* property, for lack of a given name. We use `ChangeListener`s to listen for `ChangeEvent`s that get fired when these properties change state. A `ChangeEvent`

only carries one piece of information with it: the source of the event. For this reason, change properties are less powerful than bound or constrained properties, but they are more widespread. A `JButton`, for instance, sends change events whenever it is armed (pressed for the first time), pressed, and released (see chapter 5).

Another new property-like feature Swing introduces is the notion of *client properties*. These are basically key/value pairs stored in a `Hashtable` provided by each Swing component. This allows properties to be added and removed at run-time, and is often a convenient place to store data without having to build a separate subclass.

Warning: Client properties may seem like a great way to add property change support for custom components, but we are explicitly advised against this: "The `clientProperty` dictionary is not intended to support large scale extensions to `JComponent` nor should it be considered an alternative to subclassing when designing a new component."[API]

Client properties are bound properties: when a client property changes, a `PropertyChangeEvent` is dispatched to all registered `PropertyChangeListener`s. To add a property to a component's client properties `Hashtable`, we do the following:

```
myComponent.putClientProperty("myname", myValue);
```

To retrieve a client property:

```
myObject = myComponent.getClientProperty("myname");
```

To remove a client propery we can provide a `null` value:

```
myComponent.putClientProperty("myname", null);
```

For example, `JDesktopPane` uses a client property to control the outline dragging mode for `JInternalFrame`s (this will work no matter which L&F is in use):

```
myDesktop.putClientProperty("JDesktopPane.dragMode", "outline");
```

Note: You can always find out which properties have change events associated with them, as well as any other type of event, by referencing to the Swing source code. Unless you are using Swing for simple interfaces, we strongly suggest getting used to this.

Five Swing components have special client properties that only the Metal L&F pays attention to. Briefly these are:

`JTree.lineStyle`
: A `String` used to specify whether node relationships are displayed as angular connecting lines ("Angled"), horizontal lines defining cell boundaries ("Horizontal" -- default), or no lines at all ("None").

`JScrollBar.isFreeStanding`
: A `Boolean` value used to specify whether all sides of a `JScrollbar` will have an etched border (`Boolean.FALSE` -- default) or only the top and left edges (`Boolean.TRUE`).

`JSlider.isFilled`

A `Boolean` value used to specify whether the lower portion of a slider should be filled (`Boolean.TRUE`) or not (`Boolean.FALSE` -- default).

`JToolBar.isRollover`

A `Boolean` value used to specify whether a toolbar button displays an etched border only when the mouse is within its bounds and no border if not (`Boolean.TRUE`), or always use an etched border (`Boolean.FALSE` -- default).

`JInternalFrame.isPalette`

A `Boolean` value used to specify whether a very thin border is used (`Boolean.TRUE`) or the regular border is used (`Boolean.FALSE` -- default). As of Java 2 FCS this property is not used.

2.1.2 Size and positioning

Because `JComponent` extends `java.awt.Container` it inherits all the sizing and positioning functionality we are used to. We are encouraged to manage a component's preferred, minimum, and maximum sizes using the following methods:

`setPreferredSize()`, `getPreferredSize()`

The most comfortable size of a component. Used by most layout managers to size each component.

`setMinimumSize()`, `getMinimumSize()`

Used during layout to act as a lower bounds for a component's dimensions.

`setMaximumSize()`, `getMaximumSize()`

Used during layout to act as an upper bounds for a component's dimensions.

Each `setXX()`/`getXX()` method accepts/returns a `Dimension` instance. We will learn more about what these sizes mean in terms of each layout manager in chapter 4. Whether or not a layout manager pays attention to these sizes is solely based on that layout manager's implementation. It is perfectly feasible to construct a layout manager that simply ignores all of them, or pays attention to only one. The sizing of components in a container is layout-manager specific.

`JComponent`'s `setBounds()` method can be used to assign a component both a size and a position within its parent container. This overloaded method can take either a `Rectangle` parameter (`java.awt.Rectangle`) or four `int` paramaters representing x-coordinate, y-coordinate, width, and height. For example, the following two are equivalent:

```
myComponent.setBounds(120,120,300,300);

Rectangle rec = new Rectangle(120,120,300,300);
myComponent.setBounds(rec);
```

Note that `setBounds()` will not override any layout policies in effect due to a parent container's layout manager. For this reason a call to `setBounds()` may appear to have been ignored in some situations because it tried to do its job and was forced back to its original size by the layout manager (layout managers always have first crack at setting the size of a component).

`setBounds()` is commonly used to manage child components in containers with no layout manager (such as `JLayeredPane`, `JDesktopPane`, and `JComponent` itself). For instance, we normally use `setBounds()` when adding a `JInternalFrame` to a `JDesktopPane`.

A component's size can safely be queried in typical AWT style, such as:

```
int h = myComponent.getHeight();
int w = myComponent.getWidth();
```

Size can also be retrieved as a `Rectangle` or a `Dimension` instance:

```
Rectangle rec2 = myComponent.getBounds();
Dimension dim = myComponent.getSize();
```

`Rectangle` contains four publically accessible properties describing its location and size:

```
int recX = rec2.x;
int recY = rec2.y;
int recWidth = rec2.width;
int recHeight = rec2.height;
```

`Dimension` contains two publically accessible properties describing size:

```
int dimWidth = dim.width;
int dimHeight = dim.height;
```

The coordinates returned in the `Rectangle` instance using `getBounds()` represent a component's location within its parent. These coordinates can also be obtained using the `getX()` and `getY()` methods. Additionaly, we can set a component's position within its container using the `setLocation(int x, int y)` method.

`JComponent` also maintains an alignment. Horizontal and vertical alignments can be specified by float values between 0.0 and 1.0: 0.5 means center, closer to 0.0 means left or top, and closer to 1.0 means right or bottom. The corresponding `JComponent` methods are:

```
setAlignmentX(float f);
setAlignmentY(float f);
```

These values are only used in containers managed by `BoxLayout` and `OverlayLayout`.

2.2 Event handling and dispatching

Events occur anytime a key or mouse button is pressed. The way components receive and process events has not changed from JDK1.1. There are many different types of events that Swing components can generate, including those in `java.awt.event` and even more in `javax.swing.event`. Many of the new Swing event types are component-specific. Each event type is represented by an object that, at the very least, identifies the source of the event, and often carries additional information about what specific kind of event it is, and information about the state of the source before and after the event was generated. Sources of events are most commonly components or models, but there

are also different kinds of objects that can generate events.

As we discussed in the last chapter, in order to receive notification of events, we need to register listeners with the target object. A listener is an implementation of any of the `XXListener` classes (where `XX` is an event type) defined in the `java.awt.event`, `java.beans`, and `javax.swing.event` packages. There is always at least one method defined in each interface that takes a corresponding `XXEvent` as parameter. Classes that support notification of `XXEvent`s generally implement the `XXListener` interface, and have support for registering and unregistering those listeners through the use of `addXXListener()` and `removeXXListener()` methods respectively. Most event targets allow any number of listeners to be registered with them. Similarly, any listener instance can be registered to receive events from any number of event sources. Usually classes that support `XXEvent`s provide protected `fireXX()` methods used for constructing event objects and sending them to the event handlers for processing.

2.2.1 class javax.swing.event.EventListenerList

`EventListenerList` is an array of `XXEvent`/`XXListener` pairs. `JComponent` and each of its decendants use an `EventListenerList` to maintain their listeners. All default models also maintain listeners and an `EventListenerList`. When a listener is added to a Swing component or model, the associated event's `Class` instance (used to identify event type) is added to its `EventListenerList` array, followed by the listener itself. Since these pairs are stored in an array rather than a mutable collection (for efficiency purposes), a new array is created on each addition or removal using the `System.arrayCopy()` method. When events are received, the list is walked through and events are sent to each listener with a matching type. Because the array is ordered in an `XXEvent`, `XXListener`, `YYEvent`, `YYListener`, etc. fashion, a listener corresponding to a given event type is always next in the array. This approach allows very efficient event-dispatching routines (see section 2.7.7). For thread safety the methods for adding and removing listeners from an `EventListenerList` synchronize access to the array when it is manipulated.

`JComponent` defines its `EventListenerList` as a protected field called `listenerList` so that all subclasses inherit it. Swing components manage most of their listeners directly through `listenerList`.

2.2.2 Event-dispatching thread

All events are processed by the listeners that receive them within the event-dispatching thread (an instance of `java.awt.EventDispatchThread`). All painting and component layout is expected to occur within this thread as well. The event-dispatching thread is of primary importance to Swing and AWT, and plays a key role in keeping updates to component state and display in an app under control.

Associated with this thread is a FIFO queue of events -- the system event queue (an instance of `java.awt.EventQueue`). This gets filled up, as any FIFO queue, in a serial fashion. Each request takes its turn executing event handling code, whether this be updating component properties, layout, or repainting. All events are processed serially to avoid such situations as a component's state being modified in the middle of a repaint. Knowing this, we must be careful not to dispatch events *outside* of the event-dispatching thread. For instance, calling a `fireXX()` method directly from a separate thread of execution is unsafe. We must also be sure that event handling code, and painting code can be executed quickly. Otherwise the whole system event queue will be blocked waiting for one event process, repaint, or layout to occur, and our application will appear frozen or locked up.

2.3 Multithreading

To help us in ensuring that all our event handling code gets executed only from within the event-dispatching thread, Swing provides a very helpful class that, among other things, allows us to add `Runnable` objects to the system event queue. This class is called `SwingUtilities` and it contains two methods that we are interested in here: `invokeLater()` and `invokeAndWait()`. The first method adds a `Runnable` to the system event queue and returns immediately. The second method adds a `Runnable` and waits for it to be dispatched, then returns after it finishes. The basic syntax of each follows:

```
Runnable trivialRunnable = new Runnable() {
  public void run() {
    doWork(); // do some work
  }
};
SwingUtilities.invokeLater(trivialRunnable);

try {
  Runnable trivialRunnable2 = new Runnable() {
    public void run() {
      doWork(); // do some work
    }
  };
  SwingUtilities.invokeAndWait(trivialRunnable2);
}
catch (InterruptedException ie) {
  System.out.println("...waiting thread interrupted!");
}
catch (InvocationTargetException ite) {
  System.out.println(
    "...uncaught exception within Runnable's run()");
}
```

Because these `Runnable`s are placed into the system event queue for execution within the event-dispatching thread, we should be just as careful that they execute quickly, as any other event handling code. In the above two examples, if the `doWork()` method did something that takes a long time (like loading a large file) we would find that the application would freeze up until the load finishes. In time-intensive cases such as this, we should use our own separate thread to maintain responsiveness.

The following code shows a typical way to build our own thread to do some time-intensive work. In order to safely update the state of any components from inside this thread, we must use `invokeLater()` or `invokeAndWait()`:

```
Thread workHard = new Thread() {
  public void run() {
    doToughWork(); // do some really time-intensive work
```

```
    SwingUtilities.invokeLater( new Runnable () {
      public void run() {
        updateComponents(); // update the state of component(s)
      }
    });
  }
};
workHard.start();
```

Note: `invokeLater()` should be instead of `invokeAndWait()` whenever possible. If we do have to use `invokeAndWait()`, we should make sure that there are no locks (i.e.synchronized blocks) held by the calling thread that another thread might need during the operation.

This solves the problem of responsiveness, and it does dispatch component-related code to the event-dispatching thread, but it still cannot be considered completely user-friendly. Normally the user should be able to interrupt a time-intensive procedure. If we are waiting to establish a network connection, we certainly don't want to continue waiting indefinitely if the destination no longer exists. In most circumstances the user should have the option to interrupt our thread. The following pseudocode code shows a typical way to accomplish this, where `stopButton` causes the thread to be interrupted, updating component state accordingly:

```
Thread workHarder = new Thread() {
  public void run() {
    doTougherWork();
    SwingUtilities.invokeLater( new Runnable () {
      public void run() {
        updateMyComponents(); // update the state of component(s)
      }
    });
  }
};
workHarder.start();

public void doTougherWork() {
  try {
    // [some sort of loop]
    // ...if, at any point, this involves changing
    // component state we'll have to use invokeLater
    // here because this is a separate thread.
    //
    // We must do at least one of the following:
    // 1. Periodically check Thread.interrupted()
    // 2. Periodically sleep or wait
    if (Thread.interrupted()) {
      throw new InterruptedException();
```

```
      }
      Thread.wait(1000);
    }
    catch (InterruptedException e) {
      // let somebody know we've been interrupted
      // ...if this involves changing component state
      // we'll have to use invokeLater here.
    }
  }

  JButton stopButton = new JButton("Stop");
  ActionListener stopListener = new ActionListener() {
    public void actionPerformed(ActionEvent event) {
      // interrupt the thread and let the user know the
      // thread has been interrupted by disabling the
      // stop button.
      // ...this will occur on the regular event dispatch thread
      workHarder.interrupt();
      stopButton.setEnabled(false);
    }
  };
  stopButton.addActionListener(stopListener);
```

Our `stopButton` interrupts the `workHarder` thread when pressed. There are two ways that `doTougherWork()` will know whether `workHarder`, the thread it is executed in, has been interrupted. If it is currently `sleep`ing or `wait`ing, an `InterruptedException` will be thrown which we can catch and process accordingly. The only other way to detect interruption is to periodically check the interrupted state by calling `Thread.interrupted()`.

This approach is commonly used for constructing and displaying complex dialogs, I/O processes that result in component state changes (such as loading a document into a text component), intensive class loading or calculations, waiting for messages or to establish a network connection, etc.

Reference: Members of the Swing team have written a few articles about using threads with Swing, and have provided a class called `SwingWorker` that makes managing the type of multithreading described here more convenient. See http://java.sun.com/products/jfc/tsc/archive/tech_topics_arch/threads/threads.html

---

2.3.1 Special cases

There are some special cases in which we do not *need* to delegate code affecting the state of components to the event-dispatching thread:

1. Some methods in Swing, although few and far between, are marked as thread-safe and do not need special consideration. Some methods are thread-safe but are not marked as such: `repaint()`, `revalidate()`, and `invalidate()`.

2. A component can be constructed and manipulated in any fashion we like, without regard for threads, as long as it has not yet been *realized* (i.e. its has been displayed or a repaint request has been queued). Top-level containers (`JFrame`, `JDialog`, `JApplet`) are realized after any of `setVisible(true)`, `show()`, or `pack()` have been called on them. Also note that a component is considered realized as soon as it is added to a realized container.

3. When dealing with Swing applets (`JApplet`s) all components can be constructed and manipulated without regard to threads until the `start()` method has been called, which occurs after the `init()` method.

2.3.2 How do we build our own thread-safe methods?

This is quite easy. Here is a thread-safe method template we can use to guarantee this method's code only executes in the event-dispatching thread:

```
public void doThreadSafeWork() {
  if (SwingUtilities.isEventDispatchThread()) {
    //
    // do all work here...
    //
  }
  else {
    Runnable callDoThreadSafeWork =  new Runnable() {
      public void run() {
        doThreadSafeWork();
      }
    };
    SwingUtilities.invokeLater(callDoThreadSafeWork);
  }
}
```

2.3.3 How do invokeLater() and invokeAndWait() work?[4]

class javax.swing.SystemEventQueueUtilities [package private]

When `SwingUtilities` receives a `Runnable` object through `invokeLater()`, it passes it immediately to the `postRunnable()` method of a class called `SystemEventQueueUtilities`. If a `Runnable` is received through `invokeAndWait()`, first the current thread is checked to make sure that it is not the event-dispatching thread. (It would be fatal to allow `invokeAndWait()` to be invoked from the event-dispatch thread itself!) An error is thrown if this is the case. Otherwise, we construct an `Object` to use as the lock on a critical section (i.e. a synchronized block). This block contains two statements. The first sends the `Runnable` to `SystemEventQueueUtilities`' `postRunnable()` method, along with a reference to the lock object. The second waits on the lock object so the calling thread won't proceed until this object is notified--hence "invoke and wait."

The `postRunnable()` method first communicates with the private `SystemEventQueue`, an inner class of

---

[4] This section is particularly advanced and is only of interest to those seeking a low level understanding of how `Runnable`s are dispatched and processed.

SystemEventQueueUtilities, to return a reference to the system event queue. We then wrap the Runnable in an instance of RunnableEvent, another private inner class. The RunnableEvent constructor takes a Runnable and an Object representing the lock object (null if invokeLater() was called) as parameters.

The RunnableEvent class is a subclass of AWTEvent, and defines its own static int event ID -- EVENT_ID. (Note that whenever we define our own event we are expected to use an event ID greater than the value of AWTEvent.RESERVED_ID_MAX.) RunnableEvent's EVENT_ID is AWTEvent.RESERVED_ID_MAX + 1000. RunnableEvent also contains a static instance of a RunnableTarget, yet another private inner class. RunnableTarget is a subclass of Component and its only purpose is to act as the source and target of RunnableEvents.

How does RunnableTarget do this? Its constructor enables events with event ID matching RunnableEvent's ID:

```
enableEvents(RunnableEvent.EVENT_ID);
```

It also overrides Component's protected processEvent() method to receive RunnableEvents. Inside this method it first checks to see if the event passed as parameter is in fact an instance of RunnableEvent. If it is, it is passed to SystemEventQueueUtilities' processRunnableEvent() method (this occurs after the RunnableEvent has been dispatched from the system event queue.)

Now back to RunnableEvent. The RunnableEvent constructor calls its superclass (AWTEvent) constructor passing its static instance of RunnableTarget as the event source, and EVENT_ID as the event ID. It also keeps references to the given Runnable and lock object.

So in short: when invokeLater() or invokeAndWait() is called, the Runnable passed to them is then passed to the SystemEventQueueUtilities.postRunnable() method along with a lock object that the calling thread (if it was invokeAndWait()) is waiting on. This method first tries to gain access to the system event queue and then wraps the Runnable and the lock object in an instance of RunnableEvent.

Once the RunnableEvent instance has been created, the postRunnable() method (which we have been in this whole time) checks to see if it did successfully gain access to the system event queue. This will only occur if we are not running as an applet, because applets do not have direct access to the system event queue. At this point, there are two possible paths depending on whether we are running an applet or an application:

Applications:

Since we have direct access to the AWT Sytstem event queue we just post the RunnableEvent and return. Then the event gets dispatched at some point in the event-dispatching thread by being sent to RunnableTarget's processEvent() method, which then sends it to the processRunnableEvent() method. If there was no lock used (i.e. invokeLater() was called) the Runnable is just executed and we are done. If there was a lock used (i.e. invokeAndWait() was called), we enter a a synchronized block on the lock object so that nothing else can access that object when we execute the Runnable. Remember that this is the same lock object that the calling thread is waiting on from within SwingUtilities.invokeAndWait(). Once the Runnable finishes, we call notify on this object, which then wakes up the calling thread and we are done.

Applets:

`SystemEventQueueUtilities` does some very interesting things to get around the fact that applets do not have direct access to the system event queue. To summarize a quite involved workaround procedure, an invisible `RunnableCanvas` (a private inner class that extends `java.awt.Canvas`) is maintained for each applet and stored in a static `Hashtable` using the calling thread as its key. A `Vector` of `RunnableEvent`s is also maintained and instead of manually posting an event to the system event queue, a `RunnableCanvas` posts a `repaint()` request. Then, when the repaint request is dispatched in the event-dispatching thread, the appropriate `RunnableCanvas`'s `paint()` method is called as expected. This method has been constructed to locate any `RunnableEvent`s (stored in the `Vector`) associated with a given `RunnableCanvas`, and execute them (somewhat of a hack, but it works).

2.4 Timers

class javax.swing.Timer

You can think of the `Timer` as a unique thread conveniently provided by Swing to fire `ActionEvent`s at specified intervals (although this is not exactly how a `Timer` works internally, as we will see in section 2.6). `ActionListener`s can be registered to received these events just as we register them on buttons, and other components. To create a simple `Timer` that fires `ActionEvent`s every second we can do something like the following:

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

class TimerTest
{
  public TimerTest() {
    ActionListener act = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.out.println("Swing is powerful!!");
      }
    };
    Timer tim = new Timer(1000, act);
    tim.start();

    while(true) {};
  }

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

First we set up an `ActionListener` to receive `ActionEvent`s. Then we built a new `Timer` passing the time in milliseconds between events, the *delay*, and an `ActionListener` to send them to. Finally we call the `Timer`'s `start()` method to turn it on. Since there is no GUI running for us the program will immediately exit, so we set up a

loop to let the `Timer` continue to do its job indefinitely (we will explain why this is necessary in section 2.6).

When you run this code you will see "Swing is powerful!!" sent to standard output every second. Note that the `Timer` does not fire an event right when it is started. This is because its *initial delay* time defaults to the delay time passed to the constructor. If we want the `Timer` to fire an event right when it is started we would set the initial delay time to 0 using its `setInitialDelay()` method.

At any point we can call `stop()` to stop the `Timer` and `start()` to restart it (`start()` does nothing if it is already running). We can call `restart()` on a `Timer` to start the whole process over. The `restart()` method is just a shortcut way to call `stop()` and `start()` sequentually.

We can set a `Timer`'s delay using the `setDelay()` method and tell it whether to repeat or not using the `setRepeats()` method. Once a `Timer` has been set to non-repeating it will fire only one action when started (or if it is currently running), and then it will stop.

The `setCoalesce()` method allows several `Timer` event postings to be combined (coalesced) into one. This can be useful under heavy loads when the `TimerQueue` (see below) thread doesn't have enough processing time to handle all its `Timer`s.

`Timer`s are easy to use and can often be used as convenient replacements for building our own threads. However, there is a lot more going on behind the scenes that deserves a bit of revealing. Before we are ready to look at how `Timer`s work under the hood, we'll take a look at Swing's `SecurityContext`-to-`AppContext` service class mapping for applets, as well as how applications manage their service classes (also using `AppContext`). If you are *not* curious about how Swing manages the sharing of service classes behind the scenes, you will want to skip the next section. Although we will refer to `AppContext` from time to time, it is by no means necessary to understand the details.

2.5 AppContext services[5]

class sun.awt.AppContext [platform specific]

Warning: `AppContext` is not meant to be used by *any* developer, as it is not part of the Java 2 core API. We are discussing it here only to facilitate a more thorough understanding of how Swing service classes work behind the scenes.

`AppContext` is an application/applet (we'll say "app" for short) *service* table that is unique to each Java session (applet or application). For applets, a separate `AppContext` exists for each `SecurityContext` which corresponds to an applet's codebase. For instance, if we have two applets on the same page, each using code from a different directory, both of those applets would have distinct `SecurityContext`s associated with them. If, however, they each were loaded from the same codebase, they would necessarily share a `SecurityContext`. Java applications do not have `SecurityContext`s. Rather, they run in namespaces which are distinguished by `ClassLoader`s. We will not go into the details of `SecurityContext`s or `ClassLoader`s here, but it suffices to say that they can be used by `SecurityManager`s to indicate security domains, and the `AppContext` class is designed to take advantage of this by only allowing one instance of itself to exist per security domain. In this way, applets from different codebases cannot

---

[5] This section is particularly advanced and is only of interest to those seeking a low level understanding of how service classes are shared throughout a Java session.

access each other's `AppContext`. So why is this significant? We're getting there...

A *shared instance* is an instance of a class that is normally retreivable using a static method defined in that class. Each `AppContext` maintains a `Hashtable` of shared instances available to the associated security domain, and each instances is referred to as a *service*. When a service is requested for the first time it registers its shared instance with the associated `AppContext`. This consists of creating a new instance of itself and adding it to the `AppContext` key/value mapping.

One reason these shared instances are registered with an `AppContext` instead of being implemented as normal static instances, directly retreivable by the service class, is for security purposes. Services registered with an `AppContext` can only be accessed by trusted apps, whereas classes directly providing static instances of themselves allow these instances to be used on a global basis (requiring us to implement our own security mechanism if we want to limit access to them). Another reason for this is robustness. The less applets interact with each other in undocumented ways, the more robust they can be.[6]

For example, suppose an app tries to access all of the key events on the system `EventQueue` (where all events get queued for processing in the event-dispatching thread) to try and steal passwords. By using distinct `EventQueue`s in each `AppContext`, the only key events that the app would have access to are its own. (There is in fact only one `EventQueue` per `AppContext`.)

So how do we access our `AppContext` to add, remove, and retrieve services? `AppContext` is not meant to be accessed by developers. But we *can* if we really need to, and this would guarantee that our code would never be certified as 100% pure, because `AppContext` is not part of the core API. Nevertheless, here's what is involved: The static `AppContext.getAppContext()` method determines the correct `AppContext` to use depending on whether we are running an applet or application. We can then use the returned `AppletContext`'s `put()`, `get()`, and `remove()` methods to manage shared instances. In order to do this we would need to implement our own methods such as the following:

```
private static Object appContextGet(Object key) {
  return sun.awt.AppContext.getAppContext().get(key);
}

private static void appContextPut(Object key, Object value) {
  sun.awt.AppContext.getAppContext().put(key, value);
}

private static void appContextRemove(Object key) {
  sun.awt.AppContext.getAppContext().remove(key);
}
```

In Swing, this functionality is implemented as three `SwingUtilities` static methods (refer to SwingUtilities.java source code):

---

[6] Tom Ball, Sun Microsystems.

```
static void appContextPut(Object key, Object value)
static void appContextRemove(Object key, Object value)
static Object appContextGet(Object key)
```

However, we cannot access these because they are package private. These are the methods used by Swing's service classes. Some of the Swing service classes that register shared instances with `AppContext` include: `EventQueue`, `TimerQueue`, `ToolTipManager`, `RepaintManager`, `FocusManager` and `UIManager.LAFState` (all of which we will discuss at some point in this book). Interestingly, `SwingUtilities` secretly provides an invisible `Frame` instance registered with `AppContext` to act as the parent to all `JDialog`s and `JWindow`s with `null` owners.

2.6Inside Timers & the TimerQueue

class javax.swing.TimerQueue [package private]

A `Timer` is an object containing a small `Runnable` capable of dispatching `ActionEvent`s to a list of `ActionListener`s (stored in an `EventListenerList`). Each `Timer` instance is managed by the shared `TimerQueue` instance (registered with `AppContext`).

A `TimerQueue` is a service class whose job it is to manage all `Timer` instances in a Java session. The `TimerQueue` class provides the static `sharedInstance()` method to retreive the `TimerQueue` service from `AppContext`. Whenever a new `Timer` is created and `start`ed it is added to the shared `TimerQueue`, which maintains a singly-linked list of `Timer`s sorted by the order in which they will expire (i.e. time to fire the next event).

The `TimerQueue` is a *daemon* thread which is started immediately upon instantiation. This occurs when `TimerQueue.sharedInstance()` is called for the first time (i.e. when the first `Timer` in a Java session is started). It continusouly waits for the `Timer` with the nearest expiration time to expire. Once this occurs it signals that `Timer` to post `ActionEvents` to all its listeners, then assigns a new `Timer` as the head of the list, and finally removes the expired `Timer`. If the expired `Timer`'s repeat mode is set to `true` it is added back into the list at the appropriate place based on its delay time.

Note: The real reason why the `Timer` example from section 2.4 would exit immediately if we didn't build a loop, is because the `TimerQueue` is a *daemon* thread. Daemon threads are service threads and when the Java virtual machine only has daemon threads running it will exit because it assumes that no real work is being done. Normally this behavior is desirable.

A `Timer`'s events are always posted in a thread-safe mannar to the event dispatching thread by sending it's `Runnable` object to `SwingUtilities.invokeLater()`.

2.7JavaBeans architecture

Since we are concerned with creating Swing applications in this book, we need to understand and appreciate the fact that every component in Swing is a JavaBean.

Note: If you are familiar with the JavaBeans component model you may want to skip to the next section.

2.7.1 The JavaBeans component model

The JavaBeans specification identifies five features that each bean is expected to provide. We will review these features here, along with the classes and mechanisms that make them possible. The first thing to do is think of a simple component, such as a button, and apply what we discuss here to this component. Second, we are assuming basic knowledge of the Java Reflection API:

"Instances of `Class` represent classes and interfaces in a running Java application. "[API]

"A `Method` provides information about, and access to, a single method on a class or interface. "[API]

"A `Field` provides information about, and dynamic access to, a single field of a class or an interface. "[API]

2.7.2 Introspection

Introspection is the ability to discover the methods, properties, and events information of a bean. This is accomplished through use of the `java.beans.Introspector` class. `Introspector` provides static methods to generate a `BeanInfo` object containing all discoverable information about a specific bean. This includes information from each of a bean's superclasses, unless we specify which superclass introspection should stop at (i.e. we can specify the 'depth' of an instrospection). The following retrieves all discoverable information of a bean:

```
BeanInfo myJavaBeanInfo =
  Introspector.getBeanInfo(myJavaBean);
```

A `BeanInfo` object partitions all of a bean's information into several groups, some of which are:

- A `BeanDescriptor`: provides general descriptive information such as a display name.
- An array of `EventSetDescriptors`: provides information about a set of events a bean fires. These can be used to, among other things, retrieve that bean's event listener related methods as `Method` instances.
- An array of `MethodDescriptors`: provides information about the methods of a bean that are externally accessible (this would include, for instance, all public methods). This information is used to construct a `Method` instance for each method.
- An array of `PropertyDescriptors`: provides information about each property that a bean maintains which can be accessed through `get`, `set`, and/or `is` methods. These objects can be used to construct `Method` and `Class` instances corresponding to that property's accessor methods and class type respectively.

2.7.3 Properties

As we discussed in section 2.1.1, beans support different types of properties. *Simple* properties are variables such that, when modified, a bean will do nothing. *Bound* and *constrained* properties are variables such that, when modified, a bean will send notification events to any listeners. This notification takes the form of an event object which contains the property name, the old property value, and the new property value. Whenever a bound property changes, the bean should send out a `PropertyChangeEvent`. Whenever a constrained property is about to change, the bean should send out a `PropertyChangeEvent` *before* the change occurs, allowing it to possibly be vetoed. Other objects can listen for these events and process them accordingly (which leads to *communication*).

Associated with properties are a bean's `setXX()`/`getXX()` and `isXX()` methods. If a `setXX()` method is available the associated property is said to be *writeable*. If a `getXX()` or `isXX()` method is available the associated property is said to be *readable*. An `isXX()` method normally corresponds to retrieval of a boolean property (occasionaly `getXX()` methods are used for this as well).

2.7.4 Customization

A bean's properties are exposed through its `setXX()`/`getXX()` and `isXX()` methods, and can be modified at run-time (or design-time). JavaBeans are commonly used in interface development environments where property sheets can be displayed for each bean allowing read/write (depending on the available accessors) property functionality.

2.7.5 Communication

Beans are designed to send events that notify all event listeners registered with that bean, when a bound or constrained property changes value. Apps are constructed by registering listeners from bean to bean. Since we can use introspection to determine event sending and receiving information about any bean, design tools can take advantage of this knowledge to allow more powerful, design-time customization. Communication is the basic glue that holds an interactive GUI together.

2.7.6 Persistency

All JavaBeans must implement the `Serializable` interface (directly or indirectly) to allow serialization of their state into persistent storage (storage that exists beyond program termination). All objects are saved except those declared `transient`. (Note that `JComponent` directly implements this interface.)

Classes which need special processing during serialization need to implement the following private methods:

```
private void writeObject(java.io.ObjectOutputStreamout) and
private void readObject(java.io.ObjectInputStream in )
```

These methods are called to write or read an instance of this class to a stream. Note that the default serialization mechanism will be invoked to serialize all sub-classes because these are private methods. (Refer to the API documentation or Java tutorial for more information about serialization.)

Note: As of the first release of Java 2, `JComponent` implements `readObject()` and `writeObject()` as private. All subclasses need to implement these methods if special processing is desired. Currently long-term persistance is not recommended and is subject to change in future releases. However, there is nothing wrong with implementing Short-term persistance (e.g. for RMI, misc. data transfer, etc.).

Classes that intend to take comple control of their serialization and deserialization should, instead, implement the `Externalizable` interface.

Two methods are defined in the `Externalizable` interface:

```
public void writeExternal(ObjectOutput out)
public void readExternal(ObjectInput in)
```

These methods will be invoked when `writeObject()` and `readObject()` (discussed above) are invoked to handle any serialization/deserialization.

2.7.7 A simple Swing-based JavaBean

The following code demonstrates how to build a Swing-based JavaBean with simple, bound, constrained, and 'change' properties.

The code: BakedBean.java
see \Chapter1\1

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

public class BakedBean extends JComponent implements Externalizable
{
  // Property names (only needed for bound or constrained properties)
  public static final String BEAN_VALUE = "Value";
  public static final String BEAN_COLOR = "Color";

  // Properties
  private Font m_beanFont;         // simple
  private Dimension m_beanDimension; // simple
  private int m_beanValue;         // bound
  private Color m_beanColor;       // constrained
  private String m_beanString;     // change

  // Manages all PropertyChangeListeners
  protected SwingPropertyChangeSupport m_supporter =
    new SwingPropertyChangeSupport(this);

  // Manages all VetoableChangeListeners
  protected VetoableChangeSupport m_vetoer =
    new VetoableChangeSupport(this);

  // Only one ChangeEvent is needed since the event's only
  // state is the source property.  The source of events generated
  // is always "this". You'll see this in lots of Swing source.
  protected transient ChangeEvent m_changeEvent = null;

  // This can manage all types of listeners, as long as we set
  // up the fireXX methods to correctly look through this list.
```

```
// This makes you appreciate the XXSupport classes.
protected EventListenerList m_listenerList =
  new EventListenerList();

public BakedBean() {
  m_beanFont = new Font("SanSerif", Font.BOLD | Font.ITALIC, 12);
  m_beanDimension = new Dimension(150,100);
  m_beanValue = 0;
  m_beanColor = Color.black;
  m_beanString = "BakedBean #";
}

public void paintComponent(Graphics g) {
  super.paintComponent(g);
  g.setColor(m_beanColor);
  g.setFont(m_beanFont);
  g.drawString(m_beanString + m_beanValue,30,30);
}

public void setBeanFont(Font font) {
  m_beanFont = font;
}

public Font getBeanFont() {
  return m_beanFont;
}

public void setBeanValue(int newValue) {
  int oldValue = m_beanValue;
  m_beanValue = newValue;

  // Notify all PropertyChangeListeners
  m_supporter.firePropertyChange(BEAN_VALUE,
    new Integer(oldValue), new Integer(newValue));
}

public int getBeanValue() {
  return m_beanValue;
}

public void setBeanColor(Color newColor)
 throws PropertyVetoException {
  Color oldColor = m_beanColor;
```

```
  // Notify all VetoableChangeListeners before making change
  // ...exception will be thrown here if there is a veto
  // ...if not we continue on and make the change
  m_vetoer.fireVetoableChange(BEAN_COLOR, oldColor, newColor);

  m_beanColor = newColor;
  m_supporter.firePropertyChange(BEAN_COLOR, oldColor, newColor);
}

public Color getBeanColor() {
  return m_beanColor;
}

public void setBeanString(String newString) {
  m_beanString = newString;

  // Notify all ChangeListeners
  fireStateChanged();
}

public String getBeanString() {
  return m_beanString;
}

public void setPreferredSize(Dimension dim) {
  m_beanDimension = dim;
}

public Dimension getPreferredSize() {
  return m_beanDimension;
}

public void setMinimumSize(Dimension dim) {
  m_beanDimension = dim;
}

public Dimension getMinimumSize() {
  return m_beanDimension;
}

public void addPropertyChangeListener(
 PropertyChangeListener l) {
  m_supporter.addPropertyChangeListener(l);
}
```

```
public void removePropertyChangeListener(
 PropertyChangeListener l) {
  m_supporter.removePropertyChangeListener(l);
}

public void addVetoableChangeListener(
 VetoableChangeListener l) {
  m_vetoer.addVetoableChangeListener(l);
}

public void removeVetoableChangeListener(
 VetoableChangeListener l) {
  m_vetoer.removeVetoableChangeListener(l);
}

// Remember that EventListenerList is an array of
// key/value pairs:
//   key = XXListener class reference
//   value = XXListener instance
public void addChangeListener(ChangeListener l) {
  m_listenerList.add(ChangeListener.class, l);
}

public void removeChangeListener(ChangeListener l) {
  m_listenerList.remove(ChangeListener.class, l);
}

// This is typical EventListenerList dispatching code.
// You'll see this in lots of Swing source.
protected void fireStateChanged() {
  Object[] listeners = m_listenerList.getListenerList();
  // Process the listeners last to first, notifying
  // those that are interested in this event
  for (int i = listeners.length-2; i>=0; i-=2) {
    if (listeners[i]==ChangeListener.class) {
     if (m_changeEvent == null)
       m_changeEvent = new ChangeEvent(this);
       ((ChangeListener)listeners[i+1]).stateChanged(m_changeEvent);
    }
  }
}

public void writeExternal(ObjectOutput out) throws IOException {
```

```
    out.writeObject(m_beanFont);
    out.writeObject(m_beanDimension);
    out.writeInt(m_beanValue);
    out.writeObject(m_beanColor);
    out.writeObject(m_beanString);
  }

  public void readExternal(ObjectInput in)
   throws IOException, ClassNotFoundException {
    setBeanFont((Font)in.readObject());
    setPreferredSize((Dimension)in.readObject());
    // Use preferred size for minimum size..
    setMinimumSize(getPreferredSize());
    setBeanValue(in.readInt());
    try {
      setBeanColor((Color)in.readObject());
    }
    catch (PropertyVetoException pve) {
      System.out.println("Color change vetoed..");
    }
    setBeanString((String)in.readObject());
  }

  public static void main(String[] args) {
    JFrame frame = new JFrame("BakedBean");
    frame.getContentPane().add(new BakedBean());
    frame.setVisible(true);
    frame.pack();
  }
}
```

`BakedBean` has a visual representation (not a requirement for a bean). It has *properties*: `m_beanValue`, `m_beanColor`, `m_beanFont`, `m_beanDimension`, and `m_beanString`. It supports *persistency* by implementing the `Externalizable` interface and implementing the `writeExternal()` and `readExternal()` methods to control its own serialization (note that the order in which data is written and read match). `BakedBean` supports *customization* through its `setXX()` and `getXX()` methods, and it supports *communication* by allowing the registration of `PropertyChangeListener`s, `VetoableChangeListener`s, and `ChangeListener`s. And, without having to do anything special, it supports *introspection*.

Attaching a main method to display `BakedBean` in a frame does not get in the way of any JavaBeans functionality. Figure 2.1 shows `BakedBean` when executed as an application.

Figure 2.1 BakedBean in our custom JavaBeans property editor

<<file figure2-1.gif>>

In chapter 18 (section 18.9) we construct a full-featured JavaBeans property editing environment. Figure 2.2 shows a `BakedBean` instance in this environment. The `BakedBean` shown has had its `m_beanDimension`, `m_beanColor`, and `m_beanValue` properties modified with our property editor and was then serialized to disk. What figure 2.2 really shows is an instance of that `BakedBean` after it had been deserialized (loaded from disk). Note that any Swing component can be created, modified, serialized, and deserialized using this environment because they are all JavaBeans compliant!

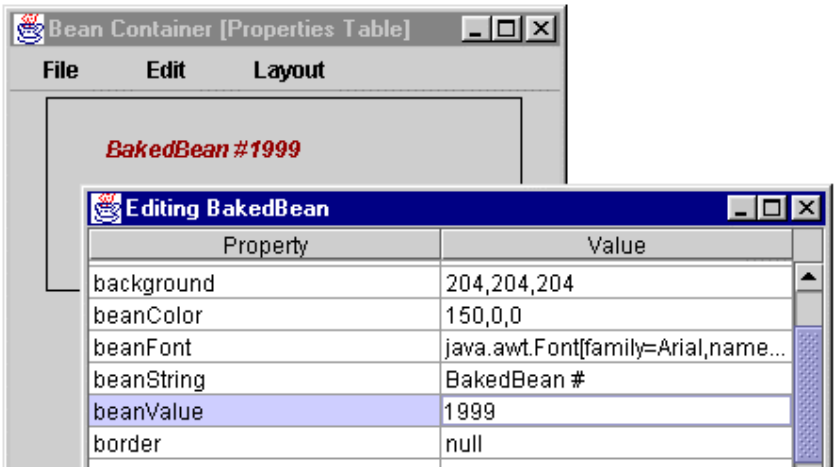


Figure 2.2 BakedBean in our custom JavaBeans property editor

<<file figure2-2.gif>>

2.8 Fonts, Colors, Graphics and text

2.8.1 Fonts

class java.awt.Font, abstract class java.awt.GraphicsEnvironment

As we saw in `BakedBean` above, fonts are quite easy to create:

```
m_beanFont = new Font("SanSerif", Font.BOLD | Font.ITALIC, 12);
```

In this code `"SanSerif"` is the font *name*, `Font.Bold | Font.PLAIN` is the *style* (which in this case is both bold and italic), and 12 is the *size*. The `Font` class defines three static `int` constants to denote font style: `Font.BOLD`, `Font.ITALIC`, `FONT.PLAIN`. We can specify font size as any `int` in the `Font` constructor (as shown above). Using

Java 2, in order to get a list of available font names at run-time we ask the local `GraphicsEnvironment`:

```
GraphicsEnvironment ge = GraphicsEnvironment.
  getLocalGraphicsEnvironment();
String[] fontNames = ge.getAvailableFontFamilyNames();
```

Note: Java 2 introduces a whole new powerful mechanism for communicating with devices that can render graphics, such as screens, printers or image bufferes. These devices are represented as instances of the `GraphicsDevice` class. Interstingly, a `GraphicsDevice` might reside on the local machine, or it might reside on a remote machine. Each `GraphicsDevice` has a set of `GraphicsConfiguration` objects associated with it. A `GraphicsConfiguration` describes specific characteristics of the associated device. Usually each `GraphicsConfiguration` of a `GraphicsDevice` represents a different mode of operation (for instance resolution and number of colors).

Note: In JDK1.1 code, getting a list of font *name*s often looked like this:

```
String[] fontnames = Toolkit.getDefaultToolkit().getFontList();
```

The `Toolkit.getFontList()` method has been deprecated in Java 2 and this code should be updated.

`GraphicsEnvironment` is an abstract class that describes a collection of `GraphicsDevices`. Subclasses of `GraphicsEnvironment` must provide three methods for retreiving arrays of `Font`s and `Font` information:

`Font[] getAllFonts()`: retreives all available `Font`s in one-point size.

`String[] getAvailableFontFamilyNames()`: retreives the names of all font families available.

`String[] getAvailableFontFamilyNames(Locale l)`: retreives the names of all font families available using the specifice `Locale` (internationalization support).

`GraphicsEnvironment` also provides static methods for retrieving `GraphicsDevice`s and the local `GraphicsEnvironment` instance. In order to find out what `Font`s are available to the system our program is running on, we must refer to this local `GraphicsEnvironment` instance, as shown above. It is much more efficient and convenient to retreive the available names and use them to construct `Font`s than it is to retreive an actual array of `Font` objects (no less, in one-point size).

We might think that, given a `Font` object, we can use typical `getXX()`/`setXX()` accessors to alter its name, style, and size. Well, we would be half right. We *can* use `getXX()` methods to retrieve this information from a `Font`:

```
String getName()
int getSize()
float getSize2D()
int getStyle
```

However, we *cannot* use typical `setXX()` methods. Instead we must use one of the following `Font` instance methods to derive a new `Font`:

```
deriveFont(float size)
deriveFont(int style)
deriveFont(int style, float size)
deriveFont(Map attributes)
deriveFont(AffineTransform trans)
deriveFont(int style, AffineTransform trans)
```

Normally we will only be interested in the first three methods.

Note: `AffineTransform`s are used in the world of Java 2D to perform things such as translations, scales, flips, rotations, and shears. A `Map` is an object that maps keys to values (it does not contain the objects involved) and the *attributes* referred to here are key/value pairs as described in the API docs for `java.text.TextAttribute` (this class is defined in the `java.awt.font` package that is new to Java 2, and considered part of Java 2D -- see chapter 23).

2.8.2 Colors

The `Color` class provides several static `Color` instances to be used for convenience (e.g. `Color.blue`, `Color.yellow`, etc.). We can also construct a `Color` using, among others, the following constructors:

```
Color(float r, float g, float b)
Color(int r, int g, int b)
Color(float r, float g, float b, float a)
Color(int r, int g, int b, int a)
```

Normally we use the first two methods, and those familiar with JDK1.1 will most likely recognize them. The first allows red, green, and blue values to be specified as `float`s from 0.0 to 1.0. The second takes these values as `int`s from 0 to 255.

The second two methods are new to Java 2. They each contain a fourth parameter which represents the `Color`'s *alpha* value. The alpha value directly controls transparency. It defaults to 1.0 or 255 which means completely opaque. 0.0 or 0 means completely transparent.

Note that, as with `Font`s, there are plenty of `getXX()` accessors but no `setXX()` accessors. Instead of modifying a `Color` object we are normally expected to create a new one.

Note: The `Color` class does have static `brighter()` and `darker()` methods that return a `Color` brighter or darker than the `Color` specified, but their behavior is unpredicatble due to internal rounding errors and we suggest staying away from them for most practical purposes.

By specifying an alpha value we can use the resulting `Color` as a component's background to make it transparent! This will work for any lightweight component provided by Swing such as labels, text components, internal frames, etc. Of course there will be component-specific issues involved (such as making the borders and title bar of an internal

frame transparent). The next section demonstrates a simple Swing canvas example showing how to use the alpha value to paint some transparent shapes.

Note: A Swing component's opaque property, controlled using `setOpaque()`, is not directly related to `Color` transparency. For instance, if we have an opaque `JLabel` whose background has been set to a transparent green (e.g. `Color(0,255,0,150)`) the label's bounds will be completely filled with this color only because it is opaque. We will be able to see through it only because the color is transparent. If we then turn off opacity the background of the label would not be rendered. Both need to be used together to create transparent components, but they are not directly related.

2.8.3 Graphics and text

abstract class java.awt.Graphics, abstract class java.awt.FontMetrics

Painting is much different in Swing than it is in AWT. In AWT we typically override `Component`'s `paint()` method to do rendering and the `update()` method for things like implementing our own double-buffering or filling the background before `paint()` is called.

With Swing, component rendering is much more complex. Though `JComponent` is a subclass of `Component`, it uses the `update()` and `paint()` methods for different reasons. In fact, the `update()` method is never invoked at all. There are also five additional stages of painting that normally occur from within the `paint()` method. We will discuss this process in section 2.11, but it suffices to say here that any `JComponent` subclass that wants to take control of its own rendering should override the `paintComponent()` method and not the `paint()` method. Additionally, it should always begin its `paintComponent()` method with a call to `super.paintComponent()`.

Knowing this, it is quite easy to build a `JComponent` that acts as our own *lightweight canvas*. All we have to do is subclass it and override the `paintComponent()` method. Inside this method we can do all of our painting. This is how to take control of the rendering of simple custom components. However, this should not be attempted with normal Swing components because UI delegates are in charge of their rendering (we will see how to take customize UI delegate rendering at the end of chapter 6, and throughout chapter 21).

Note: The awt `Canvas` class can be replaced by a simplified version of the `JCanvas` class we define in the following example.

Inside the `paintComponent()` method we have access to that component's `Graphics` object (often referred to as a component's *graphics context*) which we can use to paint shapes and draw lines and text. The `Graphics` class defines many methods used for these purposes and we refer you to the API docs for these. The following code shows how to construct a `JComponent` subclass that paints an `ImageIcon` and some shapes and text using various `Font`s and `Color`s, some completely opaque and some partially transparent (we saw similar, but less interesting, functionality in `BakedBean`). Figure 2.3 illustrates.

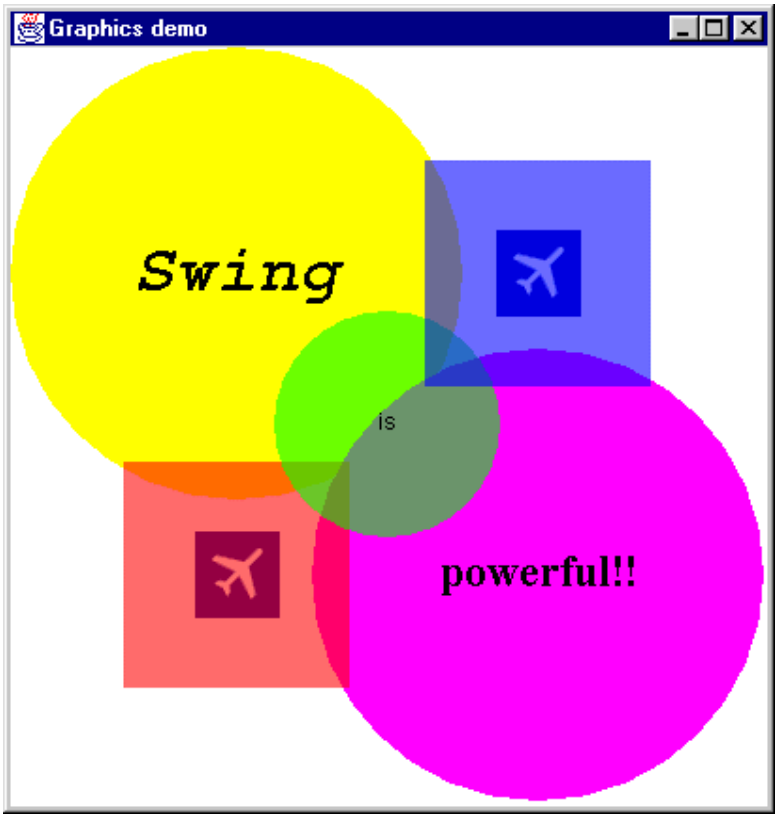


Figure 2.3 Graphics demo in a lightweight canvas.

<<file figure2-3.gif>>

The Code: TestFrame.java

see \Chapter1\2

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

class TestFrame extends JFrame
{
  public TestFrame() {
    super( "Graphics demo" );
    getContentPane().add(new JCanvas());
  }

  public static void main( String args[] ) {
    TestFrame mainFrame = new TestFrame();
    mainFrame.pack();
    mainFrame.setVisible( true );
  }
}
```

```
class JCanvas extends JComponent {
  private static Color m_tRed = new Color(255,0,0,150);
  private static Color m_tGreen = new Color(0,255,0,150);
  private static Color m_tBlue = new Color(0,0,255,150);

  private static Font m_biFont =
    new Font("Monospaced", Font.BOLD | Font.ITALIC, 36);
  private static Font m_pFont =
    new Font("SanSerif", Font.PLAIN, 12);
  private static Font m_bFont = new Font("Serif", Font.BOLD, 24);

  private static ImageIcon m_flight = new ImageIcon("flight.gif");

  public JCanvas() {
    setDoubleBuffered(true);
    setOpaque(true);
  }

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

    // fill entire component white
    g.setColor(Color.white);
    g.fillRect(0,0,getWidth(),getHeight());

    // filled yellow circle
    g.setColor(Color.yellow);
    g.fillOval(0,0,240,240);

    // filled magenta circle
    g.setColor(Color.magenta);
    g.fillOval(160,160,240,240);

    // paint the icon below blue sqaure
    int w = m_flight.getIconWidth();
    int h = m_flight.getIconHeight();
    m_flight.paintIcon(this,g,280-(w/2),120-(h/2));

    // paint the icon below red sqaure
    m_flight.paintIcon(this,g,120-(w/2),280-(h/2));

    // filled transparent red square
    g.setColor(m_tRed);
    g.fillRect(60,220,120,120);
```

```
  // filled transparent green circle
  g.setColor(m_tGreen);
  g.fillOval(140,140,120,120);

  // filled transparent blue square
  g.setColor(m_tBlue);
  g.fillRect(220,60,120,120);

  g.setColor(Color.black);

  // Bold, Italic, 36-point "Swing"
  g.setFont(m_biFont);
  FontMetrics fm = g.getFontMetrics();
  w = fm.stringWidth("Swing");
  h = fm.getAscent();
  g.drawString("Swing",120-(w/2),120+(h/4));

  // Plain, 12-point "is"
  g.setFont(m_pFont);
  fm = g.getFontMetrics();
  w = fm.stringWidth("is");
  h = fm.getAscent();
  g.drawString("is",200-(w/2),200+(h/4));

  // Bold 24-point "powerful!!"
  g.setFont(m_bFont);
  fm = g.getFontMetrics();
  w = fm.stringWidth("powerful!!");
  h = fm.getAscent();
  g.drawString("powerful!!",280-(w/2),280+(h/4));
}

// Most layout managers need this information
public Dimension getPreferredSize() {
  return new Dimension(400,400);
}

public Dimension getMinimumSize() {
  return getPreferredSize();
}

public Dimension getMaximumSize() {
  return getPreferredSize();
```

```
  }
}
```

Note that we override `JComponent`'s `getPreferredSize()`, `getMinimumSize()`, and `getMaximumSize()`, methods so most layout managers can intelligably size this component (otherwise some layout managers will set its size to 0x0). It is always good practice to override these methods when implementing custom components.

The `Graphics` class uses what is called the *clipping area*. Inside a component's `paint()` method, this is the region of that component's view that is being repainted (we often say that the clipping area represents the *damaged* or *dirtied* region of the component's view). Only painting done within the clipping area's bounds will actually be rendered. We can get the size and position of these bounds by calling `getClipBounds()` which will give us back a `Rectangle` instance describing it. The reason a clipping area is used is for efficiency purposes: there is no reason to paint undamaged or invisible regions when we don't have to. (We will show how to extend this example to work with the clipping area for maximum efficiency in the next section).

Note: All Swing components are double buffered by default. If we are building our own lightweight canvas we do not have to worry about double-buffering. This is not the case with an awt `Canvas`.

As we mentioned earlier, `Font`s and `Font` manipulation is very complex under the hood. We are certainly glossing over their structure, but one thing we should discuss is how to obtain useful information about fonts and the text rendered using them. This involves use of the `FontMetrics` class. In the example above, `FontMetrics` allowed us to determine the width and hieght of three `String`s, rendered in the current `Font` associated with the `Graphics` object, so that we could draw them centered in the circles.

Figure 2.4 illustrates some of the most common information that can be retreived from a `FontMetrics` object. The meaning of *baseline*, *ascent*, *descent*, and *height* should be clear from the diagram. The ascent is supposed to be the distance from the baseline to the top of most characters in that font. Note that when we use `g.drawString()` to render text, the coordinates specified represent the position to place the baseline of the first character.

`FontMetrics` provides several methods for retrieving this and more detailed information, such as the width of a `String` rendered in the associated `Font`.

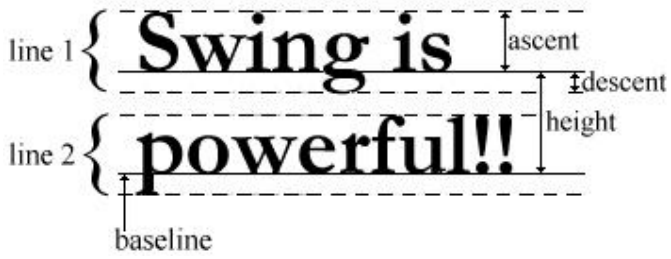


Figure 2.4 Using FontMetrics

<<file figure2-4.gif>>

In order to get a `FontMetrics` instance we first tell our `Graphics` object to use the `Font` we are intersted in

examining using the `setFont()` method. Then we create the `FontMetrics` instance by calling `getFontMetrics()` on our `Graphics` object:

```
g.setFont(m_biFont);
FontMetrics fm = g.getFontMetrics();
```

A typical operation when rendering text is to center it on a given point. Suppose we want to center the text "Swing" on 200,200. Here is the code we would use (assuming we have retrieved the `FontMetrics` object, `fm`, as shown above):

```
int w = fm.stringWidth("Swing");
int h = fm.getAscent();
g.drawString("Swing",200-(w/2),200+(h/4));
```

We get the width of "Swing" in the current font, divide it by two, and subtract it from 200 to center the text horizontally. To center it vertically we get the ascent of the current font, divide it by four, and add 200. The reason we divide the ascent by four is probably NOT so clear.

It is now time to address a common mistake that has arisen with Java 2. Figure 2.4 is not an accurate way to document `FontMetrics`. This is the way we have seen things documented in the Java tutorial and just about everywhere else that we have referenced. However, there appears to be a few problems with `FontMetrics` as of Java 2 FCS. Here we'll write a simple program that demonstrates these problems. Our program will draw the text "Swing" in a 36-point bold, monospaced font. We draw lines where its ascent, ascent/2, ascent/4, baseline, and descent lie. Figure 2.5 illustrates.

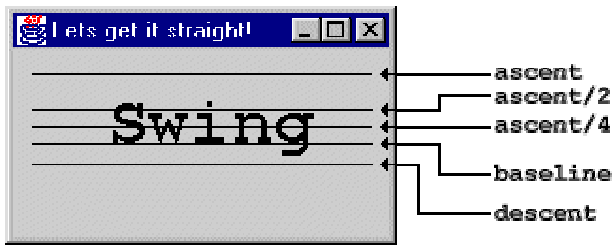


Figure 2.5 The realdealwith FontMetrics in Java 2

<<file figure2-5.gif>>

The Code: TestFrame.java

See \Chapter1\2\fontmetrics

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

class TestFrame extends JFrame
{
  public TestFrame() {
    super( "Lets get it straight!" );
```

```
    getContentPane().add(new JCanvas());
  }

  public static void main( String args[] ) {
    TestFrame mainFrame = new TestFrame();
    mainFrame.pack();
    mainFrame.setVisible( true );
  }
}

class JCanvas extends JComponent
{
  private static Font m_biFont = new Font("Monospaced", Font.BOLD, 36);

  public void paintComponent(Graphics g) {
    g.setColor(Color.black);

    // Bold 36-point "Swing"
    g.setFont(m_biFont);
    FontMetrics fm = g.getFontMetrics();
    int h = fm.getAscent();

    g.drawString("Swing",50,50); // Try these as well: Ñ Ö Ü ^

    // draw Ascent line
    g.drawLine(10,50-h,190,50-h);

    // draw Ascent/2 line
    g.drawLine(10,50-(h/2),190,50-(h/2));

    // draw Ascent/4 line
    g.drawLine(10,50-(h/4),190,50-(h/4));

    // draw baseline line
    g.drawLine(10,50,190,50);

    // draw Descent line
    g.drawLine(10,50+fm.getDescent(),190,50+fm.getDescent());
  }

  public Dimension getPreferredSize() {
    return new Dimension(200,100);
  }
}
```

We encourage you to try this demo program with various different fonts, font sizes, and even characters with diacritical marks such as Ñ, Ö, or Ü. You will find that the ascent is always much higher than it is typically documented to be, and the descent is always lower. The most reliable means of vertically centering text we found turned out to be baseline + ascent/4. However, baseline + descent might also be used and, depending on the font in use, may provide more accurate centering.

The point is that there is no correct way to perform this task due to the current state of `FontMetrics` in Java 2.You may experience very different results if not using the first release of Java 2. It is a good idea to run this program and verify whether or not results similar to those shown in figure 2.5 are produced on your system. If not you will want to use a different centering mechanism for your text which should be fairly simple to determine through experimentation with this application.

Note: In JDK1.1 code, getting a `FontMetrics` instance often looked like this:

```
FontMetrics fm = Toolkit.getDefaultToolkit().getFontMetrics(myfont);
```

The `Toolkit.getFontMetrics` method has been deprecated in Java 2 and this code should be updated.

2.9 Using the Graphics clipping area

We can use the clipping area to optimize component rendering. This may not noticably improve rendering speed for simple components such as our `JCanvas` above, but it is important to understand how to implement such functionality, as Swing's whole painting system is based on this concept (we will find out more about this in the next section).

We now modify `JCanvas` so that each of our shapes, strings, and images is only painted if the clipping area intersects its bounding rectangular region. (These intersections are fairly simple to compute, and it may be helpful for you to work through, and verify each one.) Additionaly, we maintain a local counter that is incremented each time one of our items is painted. At the end of the `paintComponent()` method we display the total number of items that were painted. Below is our optimized `JCanvas` `paintComponent()` method (with counter):

The Code: JCanvas.java
see \Chapter1\3

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

  // counter
  int c = 0;

  // for use below
  int w = 0;
  int h = 0;
  int d = 0;
```

```
// get damaged region
Rectangle r = g.getClipBounds();
int clipx = r.x;
int clipy = r.y;
int clipw = r.width;
int cliph = r.height;

// fill only damaged region only
g.setColor(Color.white);
g.fillRect(clipx,clipy,clipw,cliph);

// filled yellow circle if bounding region has been damaged
if (clipx <= 240 && clipy <= 240) {
  g.setColor(Color.yellow);
  g.fillOval(0,0,240,240); c++;
}

// filled magenta circle if bounding region has been damaged
if (clipx + clipw >= 160 && clipx <= 400
    && clipy + cliph >= 160 && clipy <= 400) {
  g.setColor(Color.magenta);
  g.fillOval(160,160,240,240); c++;
}

w = m_flight.getIconWidth();
h = m_flight.getIconHeight();
// paint the icon below blue sqaure if bounding region damaged
if (clipx + clipw >= 280-(w/2) && clipx <= (280+(w/2))
    && clipy + cliph >= 120-(h/2) && clipy <= (120+(h/2))) {
  m_flight.paintIcon(this,g,280-(w/2),120-(h/2)); c++;
}

// paint the icon below red sqaure if bounding region damaged
if (clipx + clipw >= 120-(w/2) && clipx <= (120+(w/2))
    && clipy + cliph >= 280-(h/2) && clipy <= (280+(h/2))) {
  m_flight.paintIcon(this,g,120-(w/2),280-(h/2)); c++;
}

// filled transparent red square if bounding region damaged
if (clipx + clipw >= 60 && clipx <= 180
    && clipy + cliph >= 220 && clipy <= 340) {
  g.setColor(m_tRed);
  g.fillRect(60,220,120,120); c++;
}
```

```
// filled transparent green circle if bounding region damaged
if (clipx + clipw > 140 && clipx < 260
    && clipy + cliph > 140 && clipy < 260) {
  g.setColor(m_tGreen);
  g.fillOval(140,140,120,120); c++;
}

// filled transparent blue square if bounding region damaged
if (clipx + clipw > 220 && clipx < 380
    && clipy + cliph > 60 && clipy < 180) {
  g.setColor(m_tBlue);
  g.fillRect(220,60,120,120); c++;
}

g.setColor(Color.black);

g.setFont(m_biFont);
FontMetrics fm = g.getFontMetrics();
w = fm.stringWidth("Swing");
h = fm.getAscent();
d = fm.getDescent();
// Bold, Italic, 36-point "Swing" if bounding regiondamaged
if (clipx + clipw > 120-(w/2) && clipx < (120+(w/2))
    && clipy + cliph > (120+(h/4))-h && clipy < (120+(h/4))+d)
{
  g.drawString("Swing",120-(w/2),120+(h/4)); c++;
}

g.setFont(m_pFont);
fm = g.getFontMetrics();
w = fm.stringWidth("is");
h = fm.getAscent();
d = fm.getDescent();
// Plain, 12-point "is" if bounding region damaged
if (clipx + clipw > 200-(w/2) && clipx < (200+(w/2))
    && clipy + cliph > (200+(h/4))-h && clipy < (200+(h/4))+d)
{
  g.drawString("is",200-(w/2),200+(h/4)); c++;
}

g.setFont(m_bFont);
fm = g.getFontMetrics();
w = fm.stringWidth("powerful!!");
```

```
    h = fm.getAscent();
    d = fm.getDescent();
    // Bold 24-point "powerful!!" if bounding region damaged
    if (clipx + clipw > 280-(w/2) && clipx < (280+(w/2))
        && clipy + cliph > (280+(h/4))-h && clipy < (280+(h/4))+d)
    {
      g.drawString("powerful!!",280-(w/2),280+(h/4)); c++;
    }

    System.out.println("# items repainted = " + c + "/10");
  }
```

Try running this example and dragging another window in your desktop over parts of the `JCanvas`. Keep your console in view so that you can monitor how many items are painted during each repaint. Your output should be displayed something like the following (of course you'll probably see different numbers):

```
# items repainted = 4/10
# items repainted = 0/10
# items repainted = 2/10
# items repainted = 2/10
# items repainted = 1/10
# items repainted = 2/10
# items repainted = 10/10
# items repainted = 10/10
# items repainted = 8/10
# items repainted = 4/10
```

Optimizing this canvas wasn't that bad, but imagine how tough it would be to optimize a container with a variable number of children, possibly overlapping, with double-buffering options and transparency. This is what `JComponent` does, and it does it quite efficiently. We will learn a little more about how this is done in section 2.11. But first we'll finish our high level overview of graphics by introducing a very powerful and well-met feature new to Swing: graphics debugging.

2.10 Graphics debugging

Graphics debugging provides the ability to observe each painting operation that occurs during the rendering of a component and all of its children. This is done in slow-motion, using distinct flashes to indicate the region being painted. It is intended to help find problems with rendering, layouts, and container hierarchies -- just about anything display related. If graphics debugging is enabled, the `Graphics` object used in painting is actually an instance of `DebugGraphics` (a subclass of `Graphics`). `JComponent`, and thus all Swing components, support graphics debugging and it can be turned on/off with `JComponent`'s `setDebugGraphicsOptions()` method. This method takes an `int` parameter which is normally one of (or a bitmask combination -- using the bitwise | operator) four static values defined in `DebugGraphics`.

2.10.1 Graphics debugging options

1. `DebugGraphics.FLASH_OPTION`: Each paint operation flashes a specified number of times, in a specified flash color, with a specified flash interval. The default values are: 250ms flash interval, 4 flashes, and red flash color. These values can be set with the following `DebugGraphics` static methods:

```
setFlashTime(int flashTime)
setFlashCount(int flashCount)
setFlashColor(Color flashColor)
```

If we don't disable double-buffering in the `RepaintManager` (discussed in the next section) we will not see the painting as it occurs:

```
RepaintManager.currentManager(null).
  setDoubleBufferingEnabled(false);
```

Note: Turning off buffering in the `RepaintManager` has the effect of ignoring *every* component's `doubleBuffered` property.

---

2. `DebugGraphics.LOG_OPTION`: This sends messages describing each paint operation as they occur. By default these messages are directed to standard output (the console -- `System.out`). However, we can change the log destination with `DebugGraphics`' static `setLogStream()` method. This method takes a `PrintStream` parameter. To send output to a file we would do something like the following:

```
PrintStream debugStream = null;
try {
  debugStream = new PrintStream(
    new FileOutputStream("JCDebug.txt"));
}
catch (Exception e) {
  System.out.println("can't open JCDebug.txt..");
}
DebugGraphics.setLogStream(debugStream);
```

If at some point we need to change the log stream back to standard output:

```
DebugGraphics.setLogStream(System.out);
```

We can insert any string into the log by retreiving it with `DebugGraphics`' static `logStream()` method, and then printing into it:

```
PrintStream ps = DebugGraphics.logStream();
ps.println("\n===> paintComponent ENTERED <===");
```

Warning: Writing a log to a file will overwrite that file each time we reset the stream.

Each operation is printed with the following syntax:

```
"Graphics" + (isDrawingBuffer() ? "<B>" : "") +
  "(" + graphicsID + "-" +  debugOptions + ")"
```

Each line starts with "Graphics." The `isDrawingBuffer()` method tells us whether buffering is enabled. If it is, a "<B>" is appended. The `graphicsID` and `debugOptions` values are then placed in parenthesis, and separated by a "-". The `graphicsID` value represents the number of `DebugGraphics` instances that have been created during the application's lifetime (i.e. it's a static `int` counter). The `debugOption`s value represents the current debugging mode:

```
LOG_OPTION = 1
LOG_OPTION and FLASH_OPTION = 3
LOG_OPTION and BUFFERED_OPTION = 5
LOG_OPTION, FLASH_OPTION, and BUFFERED_OPTION = 7
```

For example, with logging and flashing enabled, we see output similar to this for each operation:

```
Graphics(1-3) Setting color: java.awt.Color[r=0,g=255,b=0]
```

Calls to each `Graphics` method will get logged when this option is enabled. The above line was generated when a call to `setColor()`was made.

3. `DebugGraphics.BUFFERED_OPTION`: This is supposed to pop up a frame showing rendering as it occurs in the offscreen buffer if double-buffereing is enabled. As of the Java 2 FCS this option is not functional.

4. `DebugGraphics.NONE_OPTION`: This nullifies graphics debugging settings and basically shuts it off.

2.10.2 Graphics debugging caveats

There are several issues to be aware of when using graphics debugging:

1. Graphics debugging will not work for any component whose UI is `null`. Thus, if you have created a direct `JComponent` subclass without a UI delegate, as we did with `JCanvas` above, graphics debugging will simply do nothing. The simplest way to work around this is to define a trivial (empty) UI delegate. We'll show how to do this in the example below.

2. `DebugGraphics` does not properly clean up after itself. By default, a solid red flash color is used. When a region is flashed, that region is filled in with this red flash color and it does not get erased (it just gets painted over). This presents a problem because transparent rendering will not show up transparent. Instead, it will be alpha-blended with the red below (or whatever the flash color happens to be set to). This is not necessarily a design flaw because there is nothing stopping us from using a completely transparent flash color. With an alpha value of 0 the flash color will never be seen. The only downside is that we don't see any flashing. However, in most cases it is easy to follow what is being drawn if we set the `flashTime` and `flashCount` to wait long enough between operations.

2.10.3 Using graphics debugging

We now enable graphics debugging in our `JCanvas` example from the last two sections. Because we must have a non-null UI delegate, we define a trivial extension of `ComponentUI` and implement its `createUI()` method to return a static instance of itself:

```
class EmptyUI extends ComponentUI
{
  private static final EmptyUI sharedInstance = new EmptyUI();

  public static ComponentUI createUI(JComponent c) {
    return sharedInstance;
  }
}
```

In order to properly associate this UI delegate with `JCanvas` we simply call `super.setUI(EmptyUI.createUI(this))` from the `JCanvas` constructor. We also set up a `PrintStream` variable in `JCanvas` and use it to add a few of our own lines to the log stream during the `paintComponent` method (to log when the method starts and finishes). Other than this, no changes have been made to the `JCanvas`'s `paintComponent()` code.

In our test application, `TestFrame`, we create an instance of `JCanvas` and enable graphics debugging with the `LOG_OPTION` and `FLASH_OPTION` options. We disable buffering in the `RepaintManager`, set the flash time to 100ms, set the flash count to 2, and use a completely transparent flash color.

The Code: TestFrame.java
see \Chapter1\4

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

class TestFrame extends JFrame
{
  public TestFrame() {
    super( "Graphics demo" );
    JCanvas jc = new JCanvas();
    RepaintManager.currentManager(jc).
      setDoubleBufferingEnabled(false);
    jc.setDebugGraphicsOptions(DebugGraphics.LOG_OPTION |
      DebugGraphics.FLASH_OPTION);
    DebugGraphics.setFlashTime( 100 );
    DebugGraphics.setFlashCount( 2 );
    DebugGraphics.setFlashColor(new Color(0,0,0,0));
```

```
    getContentPane().add(jc);
  }

  public static void main( String args[] ) {
    TestFrame mainFrame = new TestFrame();
    mainFrame.pack();
    mainFrame.setVisible( true );
  }
}

class JCanvas extends JComponent
{
  // Unchanged code from section 2.9

  private PrintStream ps;

  public JCanvas() {
    super.setUI(EmptyUI.createUI(this));
  }

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

    ps = DebugGraphics.logStream();
    ps.println("\n===> paintComponent ENTERED <===");

    // All painting code unchanged

    ps.println("\n# items repainted = " + c + "/10");
    ps.println("===> paintComponent FINISHED <===\n");
  }

  // Unchanged code from section 2.9
}

class EmptyUI extends ComponentUI
{
  private static final EmptyUI sharedInstance = new EmptyUI();

  public static ComponentUI createUI(JComponent c) {
    return sharedInstance;
  }
}
```

By setting the `LOG_OPTION`, graphics debugging provides us with a more informative way of checking how well our clipping area optimization (from the last section) works. When this example is run the following output should be seen in your console (assuming you don't obscure `JCanvas`'s visible region as it is painted for the first time):

```
Graphics(0-3) Enabling debug
Graphics(0-3) Setting color:
    javax.swing.plaf.ColorUIResource[r=0,g=0,b=0]
Graphics(0-3) Setting font:
    javax.swing.plaf.FontUIResource[family=dialog,name=Dialog,
    style=plain,size=12]

===> paintComponent ENTERED <===
Graphics(1-3) Setting color: java.awt.Color[r=255,g=255,b=255]
Graphics(1-3) Filling rect: java.awt.Rectangle[x=0,y=0,
    width=400,height=400]
Graphics(1-3) Setting color: java.awt.Color[r=255,g=255,b=0]
Graphics(1-3) Filling oval: java.awt.Rectangle[x=0,y=0,
    width=240,height=240]
Graphics(1-3) Setting color: java.awt.Color[r=255,g=0,b=255]
Graphics(1-3) Filling oval:
    java.awt.Rectangle[x=160,y=160,width=240,height=240]
Graphics(1-3) Drawing image: sun.awt.windows.WImage@32a5625a at:
    java.awt.Point[x=258,y=97]
Graphics(1-3) Drawing image: sun.awt.windows.WImage@32a5625a at:
    java.awt.Point[x=98,y=257]
Graphics(1-3) Setting color: java.awt.Color[r=255,g=0,b=0]
Graphics(1-3) Filling rect:
    java.awt.Rectangle[x=60,y=220,width=120,height=120]
Graphics(1-3) Setting color: java.awt.Color[r=0,g=255,b=0]
Graphics(1-3) Filling oval:
    java.awt.Rectangle[x=140,y=140,width=120,height=120]
Graphics(1-3) Setting color: java.awt.Color[r=0,g=0,b=255]
Graphics(1-3) Filling rect:
    java.awt.Rectangle[x=220,y=60,width=120,height=120]
Graphics(1-3) Setting color: java.awt.Color[r=0,g=0,b=0]
Graphics(1-3) Setting font:
    java.awt.Font[family=monospaced.bolditalic,name=Mono
    spaced,style=bolditalic,size=36]
Graphics(1-3) Drawing string: "Swing" at:
    java.awt.Point[x=65,y=129]
Graphics(1-3) Setting font:
    java.awt.Font[family=Arial,name=SanSerif,style=plain,size=12]
Graphics(1-3) Drawing string: "is" at:
    java.awt.Point[x=195,y=203]
```

```
Graphics(1-3) Setting font:
    java.awt.Font[family=serif.bold,name=Serif,style=bold,size=24]
Graphics(1-3) Drawing string: "powerful!!" at:
    java.awt.Point[x=228,y=286]

# items repainted = 10/10
===> paintComponent FINISHED <===
```

2.11 Painting and validation[7]

At the heart of `JComponent`'s painting and validation mechanism lies a service class called `RepaintManager`. It is the `RepaintManager` that is responsible for sending painting and validation requests to the AWT system event queue for dispatching. To summarize, it does this by intercepting `repaint()` and `revalidate()` requests, coalescing any requests where possible, wrapping them in `Runnable` objects, and sending them to `invokeLater()`. There are a few issues we have encountered in this chapter that deserve more attention here before we actually discuss details of the painting and validation processes.

Note: This section contains a relatively exhaustive explanation of the most complex mechanism underlying Swing. If you are relatively new to Java or Swing we encourage you to skim this section now, but to come back at a later time for a more complete reading. If you are just looking for information on how to override and use your own painting methods, see section 2.8. For customizing UI delegate rendering see chapter 21.

2.11.1 Double buffering

We've mentioned double-buffering, how to disable it in the `RepaintManager`, and how to specify the double-buffering of individual components with `JComponent`'s `setDoubleBuffered()` method. But how does it work?

Double buffering is the technique of painting into an off-screen image rather than painting directly to a visible component. In the end, the resulting image is painted to the screen (which occurs relatively quickly). Using awt components, developers were required to implement their own double-buffering to reduce flashing. It was clear that double-buffering should be a built-in feature because of its widespread use. Thus, it is not much of a surprise to find this feature in all Swing components.

Behind the scenes, double-buffering consists of creating an `Image` and retrieving its `Graphics` object to use in all painting methods. If the component being repainted has children, this `Graphics` object will be passed down to them to use for painting, and so on. So if we are using double-buffering for a component, all its children will also be using double-buffering (whether or not they have double-buffering enabled or not) because they will be rendering into the same `Graphics` object. Note that there is only one off-screen image per `RepaintManager`, and there is normally only one `RepaintManager` instance per applet or application (`RepaintManager` is a service class that registers a shared instance of itself with `AppContext`--see section 2.5).

---

[7] For a higher level summary of the painting process, see the Swing Connection article: "Painting in AWT and Swing." http://java.sun.com/products/jfc/tsc/special_report/Painting/painting.html

As we discuss in chapter 3, `JRootPane` is the top-level Swing component in any window (this includes `JInternalFrame` -- although it isn't really a window). By enabling double-buffering on `JRootPane`, all of its children will also be painted using double-buffering. As we saw in the last section, `RepaintManager` also provides global control over all component double-buffering. So another way to guarantee that all components will use double-buffering is to call:

```
RepaintManager.currentManager(null).setDoubleBufferingEnabled(true);
```

2.11.2 Optimized drawing

We didn't really discuss the fact that components can overlap each other in Swing yet, but they can. `JLayeredPane`, for example, is a container allowing any number of components to overlap each other. Repainting such a container is much more complex than repainting a container we know does not allow overlapping, mainly because of the ability for components to be transparent.

What does it mean for a component to be transparent? Technically this means its `isOpaque()` method returns `false`. We can set this property by calling `setOpaque()`. What opacity means, in this context, is that a component will paint every pixel within its bounds. If this is set to false, we are not guaranteed that this will happen. This is generally set to true, but we will see that when it is set to `false` it increases the workload of the whole painting mechanism. Unless we are constructing a component that must not fill its entire rectangular region (as we will do in chapter 5 with custom polygonal buttons), we should always leave this set to `true`, as it is by default for most components. (This value is normally set by a component's UI delegate.)

`JComponent`'s `isOptimizedDrawingEnabled()` method is overriden to return `true` for almost all `JComponent` subclasses except: `JLayeredPane`, `JViewport`, and `JDesktopPane` (a subclass of `JLayeredpane`). Basically, calling this method is equivalent to asking a component: is it possible that any of your child components can overlap each other? If it is then there is a lot more repainting work to do to take into account that fact that any number of components, from virtually anywhere in our container hierarchy, can overlap each other. Additionaly, since components can be transparent, components layered completely behind others may still show through. Such components are not necessarily siblings (i.e. in the same container) because we could conceivably have several non-opaque containers layered one on top of another. In situations like this, we must do a whole lot of 'tree walking' to figure out which components need to be refreshed. If `isOptimizedDrawingEnabled()` is overriden to return `true`, then we assume we do not have to consider any situations like this. Thus, painting becomes more efficient, or 'optimized.'

2.11.3 Root validation

A `revalidate()` request is generated when a component needs to be laid out again. When a request is received from a certail component there must be some way of determining whether laying that component out will affect anything else/ `JComponent`'s `isValidateRoot()` method returns `false` for most components. Basically, calling this method is equivalent to asking: if I lay your contents out again, can you guarantee that none of your parents or siblings will be adversely affected (i.e. need to be laid out again)? By default only `JRootPane`, `JScrollPane`, and `JTextField` return true. This seems surprizing at first, but apparently it is true that these components are the only Swing components whose contents can be successfully laid out, in any situation (assuming no heavyweight components), without affecting parents or siblings. No matter how big we make anything *inside* a `JRootPane`, `JScrollPane`, or `JTextField`, they will not change size or location unless some *outside* influence comes into play

(i.e. a sibling or parent). To help convince you of this, try adding a multi-line text component to a container *without* placing it in a scroll pane. You may notice that creating new lines will change its size (depending on the layout). The point is not that it rarely happens or that it can be prevented, but that it *can* happen. This is the type of thing that `isValidateRoot()` is supposed to warn us about. So where is this method used?

A component or its parent is normally revalidated when a property value changes and that component's size, location, or internal layout has been affected. By recursively calling `isValidateRoot()` on a Swing component's parent until we obtain `true`, we will end with the closest ancestor of that component that guarantees us its validation will not affect its siblings or parents. We will see that `RepaintManager` relies on this method for dispatching validation requests.

Note: By siblings we mean components in the same container. By parents we mean parent containers.

2.11.4 RepaintManager

As we know there is usually only one instance of a service class in use per applet or application. So unless we specifically create our own instance of `RepaintManager`, which we will almost never need to do, all repainting is managed by the shared instance which is registered with `AppContext`. We normally retreive it using `RepaintManager`'s static `currentManager()` method:

```
myRepaintManager = RepaintManager.currentManager(null);
```

This method takes a `Component` as its parameter. However, it doesn't matter what we pass it. In fact the component passed to this method is not used anywhere inside the method at all (see the RepaintManager.java source code), so a value of `null` can safely be used here. (This definition exists for sub-classes to use if they want to work with more than one `RepaintManager`, possibly on a per-component basis.)

`RepaintManager` exists for two purposes: to provide efficient revalidation and repainting. It intercepts all `repaint()` and `revalidate()` requests. This class also handles all double-buffering in Swing and maintains a single `Image` used for this purpose. This `Image`'s maximum size is, by default, the size of the screen. However, we can set its size manually using `RepaintManager`'s `setDoubleBufferMaximumSize()` method. (All other `RepaintManager` functionality will be discussed throughout this section where applicable.)

Note: Cell renderers used in components such as `JList`, `JTree`, and `JTable` are special in that they are wrapped in instances of `CellRendererPane` and all validation and repainting requests do not propogate up the containment hierarchy. See chapter 17 for more about `CellRendererPane` and why this behvaior exists. It suffices to say here that cell renderers do not follow the painting and validation scheme discussed in this section.

2.11.5 Revalidation

`RepaintManager` maintains a `Vector` of components that need to be validated. Whenever a `revalidate` request is intercepted, the source component is sent to the `addInvalidComponent()` method and its "validateRoot" property is checked using `isValidateRoot()`. This occurs recursively on that component's parent until `isValidateRoot()` returns true. The resulting component, if any, is then checked for visibility. If any one of its parent containers is not

visible there is no reason to validate it. Otherwise `RepaintManager` 'walks down its tree' until it reaches the 'root' component, a `Window` or `Applet`. `RepaintManager` then checks the invalid components `Vector` and if the component isn't already there it is added. After being successfully added, `RepaintManager` then passes the 'root' to the `SystemEventQueueUtilities`' `queueComponentWorkRequest()` method (we saw this class in section 2.3). This method checks to see if there is a `ComponentWorkRequest` (this is a private static class in `SystemEventQueueUtilities` that implements `Runnable`) corresponding to that 'root' already stored in the work requests table. If there isn't one, a new one is created. If there is one already, we just grab a reference to it. Then we synchronize access to that `ComponentWorkRequest`, place it in the work requests table if it is a new one, and check if it is pending (i.e. it has been added to the AWT system event queue). If it isn't pending, we send it to `SwingUtilities.invokeLater()`. It is then marked as pending and we leave the synchronized block. When it is finally run from the event-dispatching thread it notifies `RepaintManager` to execute `validateInvalidComponents()`, followed by `paintDirtyRegions()`.

The `validateInvalidComponents()` method basically checks `RepaintManager`'s `Vector` containing the components in need of validation, and calls `validate()` on each one. (This method is actually a bit more careful than we describe here, as it synchronizes access to prevent the addition of invalid components while executing).

Note: Remember that `validateInvalidComponents()` should only be called from within the event-dispatching thread. Never call this method from any other thread. The same rules apply for `paintDirtyRegions()`.

The `paintDirtyRegions()` method is much more complicated, and we'll discuss *some* of its details below. For now, it suffices to say that this method paints all damaged regions of each component maintained by `RepaintManager`.

2.11.6 Repainting

`JComponent` defines two `repaint()` methods, and the no-argument version of `repaint()` is inherited from `java.awt.Container`:

```
public void repaint(long tm, int x, int y, int width, int height)
public void repaint(Rectangle r)
public repaint() // inherited from java.awt.Container
```

If you call the no-argument version the whole component is repainted. For small, simple components this is fine. But for larger, more complex components this is certainly not efficient. The other two methods take the bounding region to be repainted (the *dirtied* region) as parameters. The first method's `int` parameters correspond to the x-coordinate, y-coordinate, width, and height of that region. The second method takes the same information encapsulated in a `Rectange` instance. The second `repaint()` method shown above just sends its traffic to the first. The first method sends the dirtied region's parameters to `RepaintManager`'s `addDirtyRegion()` method.

Note: The long parameter in the first `repaint()` method represents absolutely nothing and is not used at all. It does not matter what value you use for this. The only reason this is here is to override the correct `repaint()` method from `java.awt.Component`.

`RepaintManager` maintains a `Hashtable` of dirty regions. Each component will have at most one dirty region in

this table at any time. When a dirty region is added, using `addDirtyRegion()`, the size of the region and the component are checked. On the off chance that either has a width or height <= 0 the method returns and nothing happens. If it is bigger than 0x0, the source component's visibility is then tested along with each of its ancestors, and, if they are all visible, its 'root' component, a `Window` or `Applet`, is located by 'walking down its tree' (similar to what occurs in `addInvalidateComponent()`). The dirty regions `Hashtable` is then asked if it already has a dirty region of our component stored. If it does it returns its value, a `Rectangle`, and the handy `SwingUtilities.computeUnion()` method is used to combine the new dirty region with the old one. Finally, `RepaintManager` passes the 'root' to the `SystemEventQueueUtilities' queueComponentWorkRequest()` method. What happens from here on is identical to what we saw for revalidation (see above).

Now we can talk a bit about the `paintDirtyRegions()` method we summarized earlier. (Remember that this should only be called from within the event-dispatching thread.) This method starts out by creating a local reference to `RepaintManger`'s dirty regions `Hashtable`, and redirecting `Repaintmanager`'s dirty regions `Hashtable` reference to a different, empty one. This is all done in a critical section so that no dirty regions can be added while the swap occurs. The remainder of this method is fairly long and complicated so we'll conclude with a summary of the most significant code (see RepaintManager.java for details).

The `paintDirtyRegions()` method continues by iterating through an `Enumeration` of the dirty components, calling `RepaintManager`'s `collectDirtyComponents()` method for each. This method looks at all the ancestors of the specified dirty component and checks each one for any overlap with its dirty region using the `SwingUtilities.computeIntersection()` method. In this way each dirty region's bouds are minimized so that only its visible region remains. (Note that `collectDirtyComponents()` *does* take transparency into account.) Once this has been done for each dirty component, the `paintDirtyRegions()` method enters a loop. This loop computes the final intersection of each dirty component and its dirty region. At the end of each iteration `paintImmediately()` is called on the associated dirty component, which actually paints each minimized dirty region in its correct location (we'll discuss this below). This completes the `paintDirtyRegions()` method, but we still have the most significant feature of the whole process left to discuss: painting.

2.11.7 Painting

`JComponent` includes an `update()` method which simply calls `paint()`. The `update()` method is never actually used by any Swing components at all, but is provided for backward compatibility. The `JComponent` `paint()` method, unlike typical AWT `paint()` implementations, does not handle all of a component's painting. In fact it very rarely handles *any* of it directly. The only rendering work `JComponent`'s `paint()` method is really responsible for is working with clipping areas, translations, and painting pieces of the `Image` used by `RepaintManager` for double-buffering. The rest of the work is delegated to several other methods. We will briefly discuss each of these methods and the order in which painting operations occur. But first we need to discuss how `paint()` is actually invoked.

As we know from our discussion of the repainting process above, `RepaintManager` is responsible for invoking a method called `paintImmediately()` on each component to paint its dirty region (remember there is always just one dirty region per component because they are intelligently coalesced by `RepaintManager`). This method, and the private ones it calls, make an intelligently crafted repainting process even more impressive. It first checks to see if the target component is visible, as it could have been moved, hidden, or disposed since the original request was made. Then it recursively searches the component's non-opaque (using `isOpaque()`) parents and increases the bounds of the

region to repaint accordingly until it reaches an opaque parent:

1. If the parent reached is a `JComponent` subclass, the private `_paintImmediately()` method is called and passed the newly computed region. This method queries the `isOptimizedDrawing()` method, checks whether double-buffering is enabled (if so it uses the off-screen `Graphics` object associated with `RepaintManager`'s buffered `Image`), and continues working with `isOpaque()` to determine the final parent component and bounds to invoke `paint()` on.

   A. If double-buffering *is* enabled, it calls `paintWithBuffer()(`another private method). This method works with the off-screen `Graphics` object and its clipping area to generate many calls to the parent's `paint()` method (passing it the off-screen `Graphics` object using a specific clipping area each time). After each call to `paint()`, it uses the resulting off-screen `Graphics` object to draw directly to the visible component. (In this specific case, the `paint()` method will not use any buffers internally because it knows, by checking certain flags we will not discuss, that the buffering is being taken care of elsewhere.)

   B. If double-buffering is *not* enabled, a single call to `paint()` is made on the parent.

2. If the parent is not a `JComponent`, the region's bounds are sent to that parent's `repaint()` method, which will normally invoke the `java.awt.Component` `paint()` method. This method will then forward traffic to each of its lightweight children's `paint()` method. However, before doing this it makes sure that each lightweight child it notifies is not completely covered by the current clipping area of the `Graphics` object that was passed in.

In all cases we have *finally* reached `JComponent`'s `paint()` method!

Inside `JComponent`'s `paint()` method, if graphics debugging is enabled a `DebugGraphics` instance will be used for all rendering. Interestingly, a quick look at `JComponent`'s painting code shows heavy use of a class called `SwingGraphics`. This isn't in the API docs because its package private. It appears to be a very slick class for handling custom translations, clipping area management, and a `Stack` of `Graphics` objects used for caching, recyclability, and undo-type operations. `SwingGraphics` actually acts as a wrapper for all `Graphics` instances used during the painting process. It can only be instantiated by passing it an existing `Graphics` object. This functionality is made even more explicit, by the fact that it implements an interface called `GraphicsWrapper`, which is also package private.

The `paint()` method checks whether double-buffering is enabled and whether this method was called by `paintWithBuffer()` (see above):

1. If `paint()` was called from `paintWithBuffer()` or if double-buffering is not enabled, `paint()` checks whether the clipping area of the current `Graphics` object is completely obscured by any child components. If it isn't, `paintComponent()`, `paintBorder()`, and `paintChildren()` are called in that order. If it is completely obscured, then only `paintChildren()` needs to be called. (We will see what these three methods do shortly.)

2. If double-buffering is enabled and this method was not called from `paintWithBuffer()`, it will use the off-screen `Graphics` object associated with `RepaintManager`'s buffered `Image` throughout the remainder of this method. Then it will check whether the clipping area of the current `Graphics` object is completely obscured

by any child components. If it isn't, `paintComponent()`, `paintBorder()`, and `paintChildren()` will be called in that order. If it is completely obscured, only `paintChildren()` needs to be called.

A. The `paintComponent()` method checks if the component has a UI delegate installed. If it doesn't it just exits. If it does, it simply calls `update()` on that UI delegate and then exits. The `update()` method of a UI delegate is normally responsible for painting a component's background, if it is opaque, and then calling `paint()`. A UI delegate's `paint()` method is what actually paints the corresponding component's content. (We will see how to customize UI delegates extensively throughout this text.)

B. The `paintBorder()` method simply paints the component's border if it has one.

C. The `paintChildren()` method is a bit more involved. To summarize, it searches through all child components and determines whether `paint()` should be invoked on them using the current `Graphics` clipping area, the `isOpaque()` method, and the `isOptimizedDrawingEnabled()` method. The `paint()` method called on each child will essentially start that child's painting process from part 2 above, and this process will repeat until either no more children exist or need to be painted.

The bottom line: When building or extending lightweight Swing components it is normally expected that if we want to do any painting within the component itself (vs. in the UI delegate where it normally should be done) we will override the `paintComponent()` method and immediately call `super.paintComponent()`. In this way the UI delegate will be given a chance to render the component first. Overriding the `paint()` method, or any of the other methods mentioned above should rarely be necessary, and it is always good practice to avoid doing so.

2.12 Focus Management

When Swing components are placed in a Swing container, the route of keyboard focus is, by default, left to right and top to bottom. This route is referred to as a *focus cycle*, and moving focus from one component to the next in a cycle is accomplished using the TAB key or CTRL-TAB. To move in the reverse direction through a cycle we use SHIFT-TAB or CTRL-SHIFT-TAB. This cycle is controlled by an instance of the abstract `FocusManager` class.

`FocusManager` relies on five `JComponent` properties of each component to tell it how to treat each component when the current focus reaches or leaves it:

`focusCycleRoot`: this specifies whether or not the component contains a focus cycle of its own. If it contains a focus cycle, focus will enter that component and loop through its focus cycle until it is manually or programatically moved out of that component. By default this property is `false` (for most components), and it cannot be assigned with a typical `setXX()` accessor. It can only be changed through overriding the `isFocusCycleRoot()` method and returning the appropriate boolean value.

`managingFocus`: this specifies whether or not `KeyEvent`s corresponding to a focus change will be sent to the component itself or intercepted and devoured by the `FocusManager`. By default this property is `false` (for most components), and it cannot be assigned with a typical `setXX()` accessor. It can only be changed through overriding the `isManagingFocus()` method and returning the appropriate boolean value.

`focusTraversable`: this specifies whether or not focus can be transfered to the component by the `FocusManager` due to a focus shift in the focus cycle. By default this property is `true` (for most components), and it cannot be assigned with a typical `setXX()` accessor. It can only be changed through overriding the `isFocusTraversable()` method and returning the appropriate boolean value. (Note that when focus reaches a

component through a mouse click its `requestFocus()` method is called. By overriding `requestFocus()` we can respond to focus requests on a component-specific basis.)

`requestFocusEnabled`: this specifies whether or not a mouse click will give focus to that component. This does not affect how the FocusManager works, which will continue to transfer focus to the component as part of the focus cycle. By default this property is `true` (for most components), and it can be assigned with `JComponent`'s `setRequestFocusEnabled()` method.

`nextFocusableComponent`: this specifies the component to transfer focus to when the TAB key is pressed. By default this is set to null, as focus traversal is handled for us by a default `FocusManager` service. Assigning a component as the `nextFocusableComponent` will overpower `FocusManager`'s focus traversal mechanism. This is done by passing the component to `JComponent`'s `setNextFocusableComponent()` method.

2.12.1 FocusManager

abstract class javax.swing.FocusManager

This abstract class defines the responsibility of determining how focus moves from one component to another. `FocusManager` is a service class whose shared instance is stored in `AppContext`'s service table (see 2.5). To access the `FocusManager` we use its static `getCurrentManager()` method. To assign a new `FocusManager` we us the static `setCurrentManager()` method. We can disable the current `FocusManager` service using the static `disableFocusManager()` method, and we can check whether it is enabled or not at any given point using the static `isFocusManagerEnabled()` method.

The following three abstract methods must be defined by sub-classes:

`focusNextComponent(Component aComponent)`: should be called to shift focus to the next component in the focus cycle whose `focusTraversable` property is `true`.

`focusPreviousComponent(Component aComponent)`: should be called to shift focus to the previous focusable component in the focus cycle whoe `focusTraversable` property is `true`.

`processKeyEvent(Component focusedComponent, KeyEvent anEvent)`: should be called to either consume a `KeyEvent` sent to the given component, or allow it to pass through and be processed by that component itself. This method is normally used to determine whether a key press corresponds to a shift in focus. If this is determined to be the case, the `KeyEvent` is normally consumed and focus is moved forward or backward using the `focusNextComponent()` or `focusPreviousComponent()` methods respectively.

Note: "FocusManager will receive `KEY_PRESSED`, `KEY_RELEASED` and `KEY_TYPED` key events. If one event is consumed, all other events should be consumed."[API]

2.12.2 DefaultFocusManager

class javax.swing.DefaultFocusManager

`DefaultFocusManager` extends `FocusManager` and defines the three required methods as well as several additional methods. The most significant method in this class is `compareTabOrder()`, which takes two `Components` as parameters and determines first which component is located closer to the top of the container acting as their focus cycle root. If they are both located at the same height this method will determine which is left-most. A value of `true` will be returned if the first component passed in should be given focus before the second. Otherwise `false` will be

returned.

The `focusNextComponent()` and `focusPreviousComponent()` methods shift focus as expected, and the `getComponentBefore()` and `getComponentAfter()` methods are defined to return the previous or next component, respectively, that will receive the focus after a given component in the focus cycle. The `getFirstComponent()` and `getLastComponent()` methods return the first and last component to receive focus in a given container's focus cycle.

The `processKeyEvent()` method intercepts `KeyEvent`s sent to the currently focused component. If these events correspond to a shift in focus (i.e. TAB, CTRL-TAB, SHIFT-TAB, and SHIFT-CTRL-TAB) they are consumed and the focus is changed accordingly. Otherwise these events are sent to the component for processing (see section 2.13). Note that the `FocusManager` always has first crack at keyboard events.

Note: By default, CTRL-TAB and SHIFT-CTRL-TAB can be used to shift focus out of text components. TAB and SHIFT-TAB will move the caret instead (see chapters 11 and 19).

2.12.3 Listening for focus changes

As with AWT components, we can listen for focus changes on a component by attaching an instance of the `java.awt.FocusListener` interface. `FocusListener` defines two methods, each of which take a `java.awt.FocusEvent` instance as parameter:

- `focusGained(FocusEvent e)`: this method receives a `FocusEvent` when focus is given to a component this listener is attached to.
- `focusLost(FocusEvent e)`: this method receives a `FocusEvent` when focus is removed from a component this listener is attached to.

`FocusEvent` extends `java.awt.ComponentEvent` and defines, among others, the `FOCUS_LOST` and `FOCUS_GAINED` ids to distinguish between its two event types. A `FOCUS_LOST` event will occur corresponding to the *temporary* or *permanent* loss of focus. Temporary loss occurs when another app or window is given focus. When focus returns to this window, the component that originally lost the focus will once again gain the current focus, and a `FOCUS_GAINED` event will be dispatched at that time. Permanent focus loss occurs when the focus is moved by either clicking on another component in the same window, programmatically invoking `requestFocus()` on another component, or dispatching of any `KeyEvent`s that cause a focus change when sent to the current `FocusManager`'s `processKeyEvent()` method. As expected, we can attach and remove `FocusListener` implementations to any Swing component using `Component`'s `addFocusListener()` and `removeFocusListener()` methods respectively.

2.13 Keyboard input, KeyStrokes, and Actions

2.13.1 Listening for keyboard input

`KeyEvent`s are fired by a component whenever that component has the current focus and the user presses a key. To listen for these events on a particular component we can attach `KeyListener`s using the `addKeyListener()` method. `KeyEvent` extends `InputEvent` and, contrary to most events, `KeyEvent`s are dispatched before the corresponding operation takes place (e.g. in a text field the operation might be adding a specific character to the document content). We can devour these events using the `consume()` method before they are handled further by key

bindings or other listeners (below we'll discuss exactly who gets notification of keyboard input, and what order this occurs in).

There are three `KeyEvent` event types, each of which normally occurs at least once per keyboard activation (i.e. a press and release of a single keyboard key):

- `KEY_PRESSED`: this type of key event is generated whenever a keyboard key is pressed. The key that is pressed is specified by the `keyCode` property and a *virtual key code* representing it can be retrieved with `KeyEvent`'s `getKeyCode()` method. A virtual key code is used to report the exact keyboard key that caused the event, such as `KeyEvent.VK_ENTER`. `KeyEvent` defines numerous static `int` constants each starting with prefix "VK," meaning *Virtual Key* (see `KeyEvent` API docs for a complete list). For example, if CTRL-C is typed, two `KEY_PRESSED` events will be fired. The `int` returned by `getKeyCode()` corresponding to pressing CTRL will be a value matching `KeyEvent.VK_CTRL`. Similarly, the `int` returned by `getKeyCode()` corresponding to pressing the "C" key will be a value matching `KeyEvent.VK_C`. (Note that the order in which these are fired depends on the order in which they are pressed.) `KeyEvent` also maintains a `keyChar` property which specifies the Unicode representation of the character pressed (if there is no Unicode representation `KeyEvent.CHAR_UNDEFINED` is used--e.g. the function keys on a typical PC keyboard). We can retrieve the `keyChar` character corresponding to any `KeyEvent` using the `getKeyChar()` method. For example, the character returned by `getKeyChar()` corresponding to pressing the "C" key will be 'c'. If SHIFT was pressed and held while the "C" key was pressed, the character returned by `getKeyChar()` corresponding to the "C" key press would be 'C'. (Note that distinct `keyChar`s are returned for upper and lower case characters, wheras the same `keyCode` is used in both situations--e.g. the value of `VK_C` will be returned by `getKeyCode()` regardless of whether SHIFT is held down when the "C" key is pressed or not. Also note that there is no `keyChar` associated with keys such as CTRL, and `getKeyChar()` will simply return '' in this case.)
- `KEY_RELEASED`: this type of key event is generated whenever a keyboard key is released. Other than this difference, `KEY_RELEASED` events are identical to `KEY_PRESSED` events (however, as we will discuss below, they occur much less frequently).
- `KEY_TYPED`: this type of event is fired somewhere between a `KEY_PRESSED` and `KEY_RELEASED` event. It never carries a `keyCode` property corresponding to the actual key pressed, and `0` will be returned whenever `getKeyCode()` is called on an event of this type. Note that for keys with no Unicode representation (such as PAGE UP, PRINT SCREEN, etc.), no `KEY_TYPED` event will be generated at all.

Most keys with Unicode representations, when held down for longer than a few moments, repeatedly generate `KEY_PRESSED` and `KEY_TYPED` events (in this order). The set of keys that exhibit this behavior, and the rate at which they do so, cannot be controlled and is platform-specific.

Each `KeyEvent` maintains a set of modifiers which specifies the state of the SHIFT, CTRL, ALT, and META keys. This is an `int` value that is the result of the bit-wise OR of `InputEvent.SHIFT_MASK`, `InputEvent.CTRL_MASK`, `InputEvent.ALT_MASK`, and `InputEvent.META_MASK` (depending on which keys are pressed at the time of the event). We can retrieve this value with `getModifiers()`, and we can query specifically whether any of these keys was pressed at the time the event was fired using `isShiftDown()`, `isControlDown()`, `isAltDown()`, and `isMetaDown()`.

`KeyEvent` also maintains the boolean `actionKey` property which specifies whether the invoking keyboard key corresponds to an action that should be performed by that app (`true`) vs. data that is normally used for such things as

addition to a text component's document content (`false`). We can use `KeyEvent`'s `isActionKey()` method to retrieve the value of this property.

2.13.2 KeyStrokes

Using `KeyListener`s to handle all keyboard input on a component-by-component basis was required pre-Java 2. Because of this, a significant, and often tedious, amount of time needed to spent planning and debugging keyboard operations. The Swing team recognized this, and thankfully included functionality for key event interception regardless of which component currently has the focus. This functionality is implemented by binding instances of the `javax.swing.KeyStroke` class with `ActionListener`s (normally instances of `javax.swing.Action`).

Note: Registered keyboard actions are also commonly referred to as keyboard accelerators.

Each `KeyStroke` instance encapsulates a `KeyEvent keyCode` (see above), a `modifiers` value (identical to that of `KeyEvent` -- see above), and a boolean property specifying whether it should be activated on a key press (`false` -- default) or on a key release (`true`). The `KeyStroke` class provides five[8] static methods for creating `KeyStroke` objects (note that all `KeyStroke`s are cached, and it is not necessarily the case that these methods will always return a brand new instance):

```
getKeyStroke(char keyChar)
getKeyStroke(int keyCode, int modifiers)
getKeyStroke(int keyCode, int modifiers, boolean onKeyRelease)
getKeyStroke(String representation)[9]
getKeyStroke(KeyEvent anEvent)
```

The last method will return a `KeyStroke` with properties corresponding to the given `KeyEvent`'s attributes. The `keyCode`, `keyChar`, and `modifiers` properties are taken from the `KeyEvent` and the `onKeyRelease` property is set to `true` if the event is of type `KEY_RELEASED` and `false` otherwise.

To register a `KeyStroke`/`ActionListener` combination with any `JComponent` we can use its `registerKeyBoardAction(ActionListener action, KeyStroke stroke, int condition)` method. The `ActionListener` parameter is expected to be defined such that its `actionPerformed()` method performs the necessary operations when keyboard input corresponding to the `KeyStroke` parameter is intercepted. The `int` parameter specifies under what conditions the given `KeyStroke` is considered to be valid:

- `JComponent.WHEN_FOCUSED`: the corresponding `ActionListener` will only be invoked if the component this `KeyStroke` is registered with has the current focus.
- `JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT`: the corresponding `ActionListener` will only be invoked if the component this `KeyStroke` is registered with is the ancestor of (i.e. it contains) the component with the current focus.
- `JComponent.WHEN_IN_FOCUSED_WINDOW`: the corresponding `ActionListener` will be invoked if the

---

[8] Actually `KeyStroke` provides six static methods for creating `KeyStroke`s, but `getKeyStroke(char keyChar, boolean onKeyRelease)` has been deprecated.

[9] This method is not implemented as of Java 2 FCS, and will always return `null`.

component this KeyStroke is registered with is anywhere within the peer-level window (i.e. JFrame, JDialog, JWindow, JApplet, or any other heavyweight component) that has the current focus. Note that keyboard actions registered with this condition are handled in an instance of the private KeyBoardManager service class (see 2.13.4) rather than the component itself.

For example, to associate the invocation of an ActionListener corresponding to ALT-H no matter what component has the focus in a given JFrame, we can do the following:

```
KeyStroke myKeyStroke =
  KeyStroke.getKeyStroke(KeyEvent.VK_H,
    InputEvent.ALT_MASK, false);

myJFrame.getRootPane().registerKeyBoardAction(
  myActionListener, myKeyStroke,
  JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
```

Each JComponent maintains a Hashtable client property containing all bound KeyStrokes. Whenever a KeyStroke is registered using the registerKeyboardAction() method, it is added to this structure. Only one ActionListener can be registered corresponding to each KeyStroke, and if there is already an ActionListener mapped to a particular KeyStroke, the new one will effectively overwrite the previous one. We can retrieve an array of KeyStrokes corresponding to the current bindings stored in this Hashtable using JComponent's getRegisteredKeyStrokes() method, and we can wipe out all bindings by with the resetKeyboardActions() method. Given a KeyStroke object we can retrieve the corresponding ActionListener with JComponent's getActionForKeyStroke() method, and we can retrive the corresponding condition property with the getConditionForKeyStroke() method.

2.13.3 Actions

An Action instance is basically a convenient ActionListener implementation that encapsulates a Hashtable of bound properties similar JComponent's client properties (see chapter 12 for details about working with Action implementations and their properties). We often use Action instances when registering keyboard actions.

Note: Text components are special in that they use hierarchically resolving KeyMaps. A KeyMap is a list of Action/KeyStroke bindings and JTextComponent supports multiple levels of such mappings. See chapters 11 and 19.

2.13.4 The flow of keyboard input

Each KeyEvent is first dispatched to the focused component. The FocusManager gets first crack at processing it. If the FocusManager doesn't want it, then the focused JComponent it is sent to calls super.processKeyEvent() which allows any KeyListeners a chance to process the event. If the listeners don't consume it and the focused component is a JTextComponent, the KeyMap hierarchy is traversed (see chapters 11 and 19 for more about KeyMaps). If the event is not consumed by this time the key bindings registered with the focused component get a shot. First, KeyStrokes defined with the WHEN_FOCUSED condition get a chance. If none of these handle the event, then the component walks though it's parent containers (up until a JRootPane is reached) looking for KeyStrokes

defined with the WHEN_ANCESTOR_OF_FOCUSED_COMPONENT condition. If the event hasn't been handled after the top-most container is reached, it is sent to KeyboardManager, a package private service class (note that unlike most service classes in Swing, KeyboardManager does not register its shared instance with AppContext -- see section 2.5). KeyboardManager looks for components with registered KeyStrokes with the WHEN_IN_FOCUSED_WINDOW condition and sends the event to them. If none of these are found then KeyboardManager passes the event to any JMenuBars in the current window and lets their accelerators have a crack at it. If the event is still not handled, we check if the current focus resides in a JInternalFrame (because it is the only RootPaneContainer that can be contained inside another Swing component). If this is the case, we move up to the JInternalFrame's parent. This process continues until either the event is consumed or the top-level window is reached.

2.14 SwingUtilities

class javax.swing.SwingUtilities

In section 2.3 we discussed two methods in the SwingUtilities class used for executing code in the event-dispatching thread. These are just 2 of the 36 generic utility methods defined SwingUtilities, which break down logically into seven groups: computational methods, conversion methods, accessibility methods, retrieval methods, multithreading/event-related methods, mouse button methods, and layout/rendering/UI methods. Each of these methods is static, and they are described very briefly in this section (for a more thorough understanding see the SwingUtilities.java source code).

2.14.1 Computational methods

`Rectangle[] computeDifference(Rectangle rectA, Rectangle rectB)`: returns those rectangular regions representing the portion of `rectA` that do not intersect with `rectB`.

`Rectangle computeIntersection(int x, int y, int width, int height, Rectangle dest)`: returns the intersection of two rectangular areas. The first region is defined by the `int` parameters and the second by the `Rectangle` parameter. The `Rectangle` parameter is altered and returned as the result of the computation so that a new `Rectangle` does not have to be instantiated.

`Rectangle computeUnion(int x, inty, int width, int height, Rectangle dest)`: returns the union of two rectangular areas. The first region is defined by the `int` parameters and the second by the `Rectangle` parameter. The `Rectangle` parameter is altered and returned as the result of the computation so that a new `Rectangle` does not have to be instantiated.

`isRectangleContainingRectangle(Rectangle a, Rectangle b)`: returns `true` if `Rectangle b` is completely contained in `Rectangle a`.

`computeStringWidth(FontMetrics fm, String str)`: returns the width of the given `String` according to the given `FontMetrics` object (see 2.8.3).

2.14.2 Conversion methods

`MouseEvent convertMouseEvent(Component source, MouseEvent sourceEvent, Component destination)`: returns a new `MouseEvent` with `destination` as its source and x,y-coordinates converted to the corrdinate system of `destination` (both assuming destination is not null). If `destination` is `null` the coordinates are converted to the coordinate system of `source`, and `source` is set as the source of the event. If both are `null` the `MouseEvent` returned is identical to the event passed in.

`Point convertPoint(Component source, Point aPoint, Component destination)`: returns a `Point` representing `aPoint` converted to coordinate system of the `destination` component as if it were originating in

the `source` component. If either component is `null` the coordinate system of the other is used, and if both are `null` the `Point` returned is identical to the `Point` passed in.

`Point convertPoint(Component source, int x, int y, Component destination)`: this method acts identically to the above `convertPoint()` method except that it takes `int` parameters representing the `Point` to convert rather than a `Point` instance.

`Rectangle convertRectangle(Component source, Rectangle aRectangle, Component destination)`: returns a `Rectangle` converted from then source component's coordinate system to the `destination` component's coordinate system. This method behaves similarly to `convertPoint()`.

`void convertPointFromScreen(Point p, Component c)`: converts a given `Point` in screen coordinates to the coordinate system of the given `Component`.

`void convertPointToScreen(Point p, Component c)`: converts a given `Point` in the given `Component`'s coordinate system to the coordinate system of the screen.

2.14.3 Accessibility methods

`Accessible getAccessibleAt(Component c, Point p)`: returns the `Accessible` component at the given `Point` in the coordinate system of the given `Component` (`null` will be returned if none is found). Note that an `Accessible` component is one that implements the `javax.accessibility.Accessible` interface.

`Accessible getAccessibleChild(Component c, int i)`: returns the i`th` `Accessible` child of the given `Component`.

`int getAccessibleChildrenCount(Component c)`: returns the number of `Accessible` children contained in the given `Component`.

`int getAccessibleIndexInParent(Component c)`: returns the index of the given `Component` in its parent disregarding all contained components that do not implement the `Accessible` interface. `-1` will be returned if the parent is `null` or does not implement `Accessible`, or the given `Component` does not implement `Accessible`.

`AccessibleStateSet getAccessibleStateSet(Component c)`: returns the set of `AccessibleState`s that are active for the given `Component`.

2.14.4 Retrieval methods

`Component findFocusOwner(Component c)`: returns the component contained within the given `Component` (or the given `Component` itself) that has the current focus. If there is no such component `null` is returned.

`Container getAncestorNamed(String name, Component comp)`: returns the closest ancestor of the given `Component` with the given name. Otherwise `null` is returned. (Note that each `Component` has a `name` property which can assigned and retrieved using `setName()` and `getName()` methods respectively.)

`Container getAncestorOfClass(Class c, Component comp)`: returns the closest ancestor of the given `Component` that is an instance of `c`. Otherwise `null` is returned.

`Component getDeepestComponentAt(Component parent, int x, int y)`: returns the most 'contained' child of the given `Component` containing the point (x,y) in terms of the coordinate system of the given `Component`. If the given `Component` is not a `Container` this method simply returns it immediately.

`Rectangle getLocalBounds(Component c)`: returns a `Rectagle` representing the bounds of a given `Component` in terms of its own coordinate system (thus it always starts at 0,0).

`Component getRoot(Component c)`: returns the first ancestor of c that is a `Window`. Otherwise this method returns the last ancestor that is an `Applet`.

`JRootPane getRootPane(Component c)`: returns the first `JRootPane` parent of `c`, or `c` itself if it is a `JRootPane`.

`Window windowForComponent(Component c)`: return the first ancestor of `c` that is a `Window`. Otherwise this method returns `null`.

`boolean isDescendingFrom(Component allegedDescendent, Component allegedAncestor)`: returns `true` if `allegedDescendent` is contained in `allegedAncestor`.

2.14.5 Multithreading/event-related methods

See section 2.3 for more about these methods.

`void invokeAndWait(Runnable obj)`: sends the given `Runnable` to the event-dispatching queue and blocks on the current thread.

`void invokeLater(Runnable obj)`: sends the given `Runnable` to the event-dispatching queue and continues.

`boolean isEventDispatchThread()`: returns `true` if the current thread is the event-dispatching thread.

2.14.6 Mouse button methods

`boolean isLeftMouseButton(MouseEvent)`: returns true if the given `MouseEvent` corresponds to left mouse button activation.

`boolean isMiddleMouseButton(MouseEvent)`: returns true if the given `MouseEvent` corresponds to middle mouse button activation.

`boolean isRightMouseButton(MouseEvent)`: returns true if the given `MouseEvent` corresponds to right mouse button activation.

2.14.7 Layout/rendering/UI methods

`String layoutCompoundLabel(FontMetrics fm, String text, icon icon, int verticalAlignment, int horizontalAlignment, int verticalTextPosition, int horizontalTextPosition, Rectangle viewR, Rectangle iconR, Rectangle textR, int textIconGap)`: this method is normally used by `JLabel`'s UI delegate to lay out text and/or an icon using the given `FontMetrics` object, alignment settings and text positions within the `viewR Rectangle`. If it is determined that the label text will not fit within this `Rectangle`, an elipsis ("...") is used in place of the text that will not fit. The `textR` and `iconR Rectangle`s are modified to reflect the new layout, and the `String` that results from this layout is returned.

`String layoutCompoundLabel(JComponent c, FontMetrics fm, String text, icon icon, int verticalAlignment, int horizontalAlignment, int verticalTextPosition, int horizontalTextPosition, Rectangle viewR, Rectangle iconR, Rectangle textR, int textIconGap)`: this method is identical to the above method, but takes target component to check whether or not text orientation should play a role (see the "Component Orientation in Swing: How JFC Components support BIDI text" article at the Swing Connection for more about orientation: http://java.sun.com/products/jfc/tsc/tech_topics/bidi/bidi.html).

`void paintComponent(Graphics g, Component c, Container p, int x, int y, int w, int h)`: paints the given `Component` in the given graphical context, using the rectangle defined by the four `int` parameters as the clipping area. The given `Container` is used to act as the `Component`'s parent so that any validation and repaint requests that occur on that component do not propogate up the ancestry tree of the component owning the given graphial context. This is the same methodology used by component renderers of

`JList`, `JTree`, and `JTable` to properly exhibit "rubber stamp" behavior. This behavior is accomplished through the use of a `CellRendererPane` (see chapter 17 for more about this class and why it is used to wrap renderers).

`void paintComponent(Graphics g, Component c, Container p, Rectangle r)`: functions identical to the above method, but takes a `Rectangle` parameter rather than four `int`s.

`void updateComponentTreeUI(Component c)`: notifies all components contained in `c`, and `c` itself, to update their UI delegate to match the current `UIManager` and `UIDefaults` settings (see chapter 21).

Part II - The Basics

Part II consists of twelve chapters containing discussion and examples of the basic Swing components. Chapter 3 introduces frames, panels and borders, including an example showing how to create a custom rounded-edge border. Chapter 4 is devoted to layout managers with a comparison of the most commonly used layouts, a contributed section on the use of `GridBagLayout`, the construction of several custom layouts, and the beginnings of a JavaBeans property editing environment with the ability to change the layout manager dynamically. Chapter 5 covers labels and buttons, and presents the construction of a custom transparent polygonal button designed for use in applets, as well as a custom tooltip manager to provide proper tooltip functionality for these polygonal buttons. Chapter 6 is about using and customizing tabbed panes, including an example showing how to customize `JTabbedPane` and its UI delegate to build a tabbed pane which uses background images. Chapter 7 discusses scrollpanes and how to customize scrolling functionality. Examples show how to use the row and column headers for tracking scroll position, how to change the speed of scrolling through implementation of the `Scrollable` interface, how to implement grab-and-drag scrolling, and how to programmatically invoke scrolling. Chapter 8 takes a brief look at split panes with an example showing how to base programmatic actions on the position of the divider (a gas model simulation). Chapter 9 covers combo boxes with examples showing how to build custom combo box models and cell renderers, add functionlity to the default combo box editor, and serialize a combo box model for later use. Chapter 10 is about list boxes with examples of building a custom tab-based cell renderer, adding keyboard search functionality for quick item selection, and constructing a custom checkbox cell renderer. Chapter 11 introduces the text components and undo/redo functionality with basic examples and discussions of each (text package coverage continues in chapters 19 and 20). Chapter 12 is devoted to menu bars, menus, menu items, toolbars and actions. Examples include the construction of a basic text editor with floatable toolbar, custom toolbar buttons, and a custom color chooser menu item. Chapter 13 discusses progress bars, sliders and scrollbars, including a custom scrollpane, a slider-based date chooser, a JPEG image quality editor, and an FTP client application. Chapter 14 covers dialogs, option panes, and file and color choosers. Examples demonstrate the basics of custom dialog creation and the use of `JOptionPane`, as well as how to add a custom component to `JColorChooser`, and how to customize `JFileChooser` to allow multiple file selection and the addition of a custom component (a ZIP/JAR archive creation, extraction and preview tool).

Chapter 3. Frames, Panels, and Borders

In this chapter:

- Frames and Panels overview
- Borders
- Creating a custom border

3.1 Frames and panels overview

3.1.1 JFrame

class javax.swing.JFrame

The main container for a Swing-based application is `JFrame`. All objects associated with a `JFrame` are managed by its only child, an instance of `JRootPane`. `JRootPane` is a simple container for several child panes. When we add components to a `JFrame` we don't directly add them to the `JFrame` as we did with an AWT `Frame`. Instead we have to specify exactly which pane of the `JFrame`'s `JRootPane` we want the component to be placed in. In most cases components are added to the `contentPane` by calling:

```
myJFrame.getContentPane().add(myComponent);
```

Similarly when setting a layout for a `JFrame`'s contents we usually just want to set the layout for the `contentPane`:

```
myJFrame.getContentPane().setLayout(new FlowLayout());
```

Each `JFrame` contains a `JRootPane` as protected field `rootPane`. Figure 3.1 illustrates the hierarchy of a `JFrame` and its `JRootPane`. The lines in this diagram extend downward representing the "has a" relationship of each container. We will see `JFrame` in action soon enough.

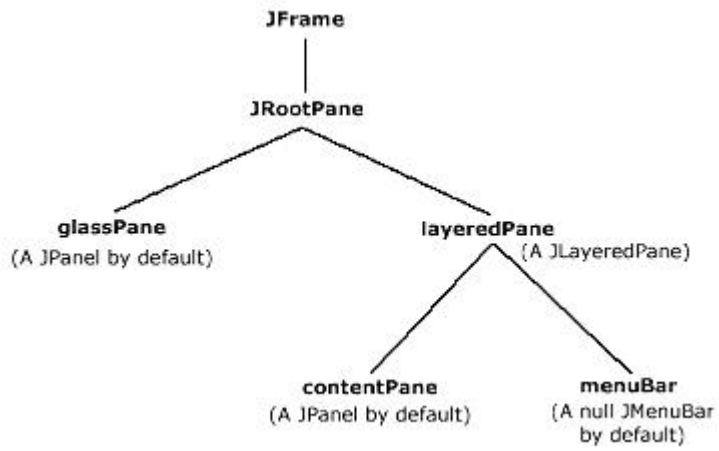


Figure 3.1 Default JFrame and JRootPane "has a" relationship
<<file figure3-1.gif>>

3.1.2 JRootPane

class javax.swing.JRootPane

Each `JRootPane` contains several components referred to here by variable name: `glassPane` (a `JPanel` by default), `layeredPane` (a `JLayeredPanel`), `contentPane` (a `JPanel` by default) and `menuBar` (a `JMenuBar`).

Note: `glassPane` and `contentPane` are just variable names used by `JRootPane`. They are not unique Swing classes, as some explanations might lead you to believe.

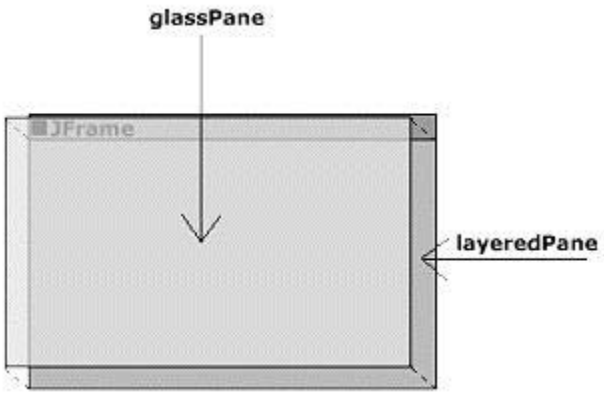


Figure 3.2 glassPane
<<file figure3-2.gif>>

The glassPane is initialized as a non-opaque JPanel that sits on top of the JLayeredPane as illustrated in Figure 3.3. This component is very useful in situations where we need to intercept mouse events to display a certain cursor over the whole frame or redirect the current application focus. The glassPane can be any component but is a JPanel by default. To change the glassPane from a JPanel to another component a call to the setGlassPane() method must be made:

```
setGlassPane(myComponent);
```

Though the glassPane does sit on top of the layeredPane it is, by default, not visible. It can be set visible (i.e. show itself) by calling:

```
getGlassPane().setVisible(true);
```

The glassPane allows you to display components in front of an existing JFrame's contents. In chapter 15 we will find that can be useful as an invisible panel for detecting internal frame focus changes.

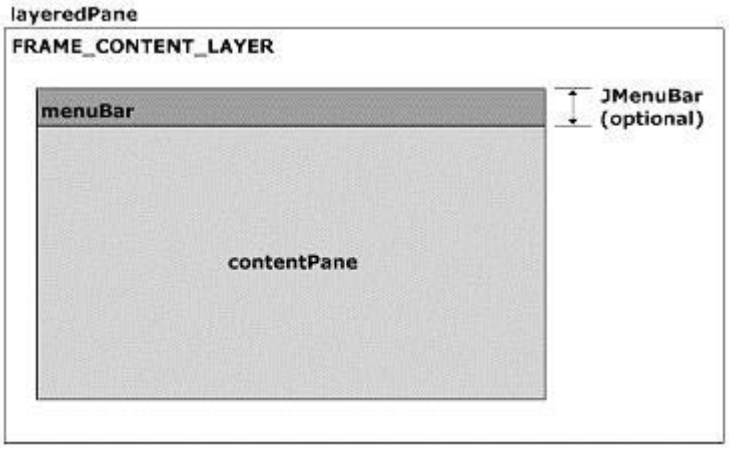


Figure 3.3 Default JFrame contents of the JLayeredPane FRAME_CONTENT_LAYER
<<file figure3-3.gif>>

The contentPane, and optional menuBar, are contained within JRootPane's layeredPane at the FRAME_CONTENT_LAYER (this is layer -30000; see chapter 15). The menuBar does not exist by default but can be set by calling the setJMenuBar() method:

```
JMenuBar menu = new JMenuBar();
setJMenuBar(menu);
```

When the JMenuBar is set it is automatically positioned at the top of the FRAME_CONTENT_LAYER. The rest

of the layer is occupied by the `contentPane` as illustrated in Figure 3.3.

The `contentPane` is, by default, an opaque `JPanel`. It can be set to any other component by calling:

```
setContentPane(myComponent);
```

Note: The default layout for the `contentPane` is `BorderLayout`. The default layout for any other `JPanel` is `FlowLayout`. Be careful not to set the layout of a `JFrame` directly. This will generate an exception. Setting the layout of the `rootPane` is also something that should be avoided because every `JRootPane` uses its own custom layout manager called `RootLayout`. We will discuss layout managers more in chapter 4.

3.1.3 RootLayout

class javax.swing.JRootPane.RootLayout

`RootLayout` is a layout manager built specifically to manage `JRootPane`'s `layeredPane`, `glassPane`, and `menuBar`. If it is replaced by another layout manager that manager must be able to handle the positioning of these components. `RootLayout` is an inner class defined within `JRootPane` and as such, is not intended to have any use outside of this class. Thus it is not discussed in this text.

3.1.4 The RootPaneContainer interface

abstract interface javax.swing.RootPaneContainer

The purpose of the `RootPaneContainer` interface is to organize a group of methods that should be used to access a container's `JRootPane` and its different panes (see API docs). Because `JFrame`'s main container is a `JRootPane`, it implements this interface (as does `JFrame` and `JDialog`). If we were to build a new component which uses a `JRootPane` as its main container we would most likely implement the `RootPaneContainer` interface. (Note that this interface exists for convenience, consistency, and organizational purposes. We are encouraged to, but certainly not required to, use it in our own container implementations.)

3.1.5 The WindowConstants interface

abstract interface javax.swing.WindowConstants

We can specify how a `JFrame, JInternalFrame, or JDialog` acts in response to the user closing it through use of the `setDefaultCloseOperation()` method. There are three possible settings, as defined by `WindowConstants` interface fields:

```
WindowConstants.DISPOSE_ON_CLOSE
WindowConstants.DO_NOTHING_ON_CLOSE
WindowConstants.HIDE_ON_CLOSE
```

The names are self-explanatory. `DISPOSE_ON_CLOSE` disposes of the `JFrame` and its contents, `DO_NOTHING_ON_CLOSE` renders the close button useless, and `HIDE_ON_CLOSE` just removes the container from view. (`HIDE_ON_CLOSE` may be useful if we may need the conatiner, or something it contains, at a later time but do not want it to be visible until then. `DO_NOTHING_ON_CLOSE` is very useful as you will see below.)

3.1.6 The WindowListener interface

abstract interface java.awt.event.WindowListener

Classes that want explicit notification of window events (such as window closing, iconification, etc.) need to implement this interface. Normally the `WindowAdapter` class is extended instead. "When the window's

status changes by virtue of being opened, closed, activated or deactivated, iconified or deiconified, the relevant method in the listener object is invoked, and the `WindowEvent` is passed to it."[API]

The methods any implementation of this interface must define are:

```
void windowActivated(WindowEvent e)
void windowClosed(WindowEvent e)
void windowClosing(WindowEvent e)
void windowDeactivated(WindowEvent e)
void windowDeiconified(WindowEvent e)
void windowIconified(WindowEvent e)
void windowOpened(WindowEvent e)
```

3.1.7 WindowEvent

class java.awt.event.WindowEvent

The type of event used to indicate that a window has changed state. This event is passed to every `WindowListener` or `WindowAdapter` object which is registered on the source window to receive such events. Method `getWindow()` returns the window that generated the event. Method `paramString()` retreives a `String` describing the event type and its source, among other things.

There are six types of `WindowEvents` that can be generated; each is represented by the following static `WindowEvent` fields: `WINDOW_ACTIVATED`, `WINDOW_CLOSED`, `WINDOW_CLOSING`, `WINDOW_DEACTIVATED`,`WINDOW_DEICONIFIED`,`WINDOW_ICONIFIED`,`WINDOW_OPENED`.

3.1.8 WindowAdapter

abstract class java.awt.event.WindowAdapter

This is an abstract implementation of the `WindowListener` interface. It is normally more convenient to extend this class than to implement `WindowListener` directly.

Notice that none of the `WindowConstants` close operations actually terminate program execution. This can be accomplished by extending `WindowAdapter` and overriding the methods we are interested in handling (in this case just the `windowClosed()` method). We can create an instance of this extended class, cast it to a `WindowListener` object, and register this listener with the `JFrame` using the `addWindowListener()` method. This can easily be added to any application as follows:

```
WindowListener l = new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
};
myJFrame.addWindowListener(l);
```

The best method of all is to combine `WindowAdapter` and values from the `WindowConstants` interface to present the user with an exit confirmation dialog as follows:

```
myJFrame.setDefaultCloseOperation(
  WindowConstants.DO_NOTHING_ON_CLOSE);
WindowListener l = new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    int confirm = JOptionPane.showOptionDialog(myJFrame,
      "Really Exit?", "Exit Confirmation",
      JOptionPane.YES_NO_OPTION,
      JOptionPane.QUESTION_MESSAGE,
      null, null, null);
```

```
        if (confirm == 0) {
          myJFrame.dispose();
          System.exit(0);
        }
      }
    };
    myJFrame.addWindowListener(l);
```

Note: This can also be done for `JDialog`. However, to do the same thing for a `JInternalFrame` we must build a custom `JInternalFrame` subclass and implement the `PropertyChangeListener` interface. See chapter 16 for details.

Inserting this code into your application will always display the dialog shown in figure 3.4 when the `JFrame` close button is pressed. Note that this functionality is not used until later chapters in which we work with applications where closing may cause a loss of unsaved material.

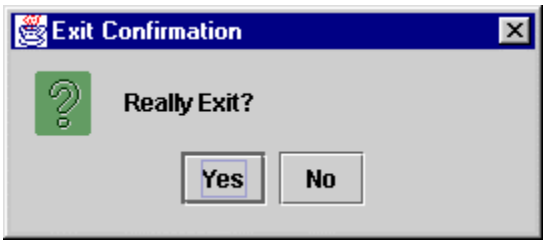


Figure 3.4 Application exit confirmation dialog
<<file figure3-4.gif>>

Reference: Dialogs and `JOptionPane` are discussed in Chapter 14.

3.1.9 Custom frame icons

Often we want to use a custom icon to represent our application frame. Because `JFrame` is a subclass of `awt.Frame` we can set its icon using the `setIconImage()` method. This method is intended to set the minimized icon to use for this frame. On some platforms this icon is also used for a title bar image (e.g. Windows).

UI Guideline: Brand Identity. Use the frame icon to establish and reinforce your brand identity. Pick a simple image which can be effective in the small space and re-used throughout the application and any accompanying material. Fig 3-5 shows the Sun Coffee Cup which was itilise as a brand mark for Java.

```
    ImageIcon image = new ImageIcon("spiral.gif");
    myFrame.setIconImage(image.getImage());
```

There is no limit to the size of the icon that can be used. A `JFrame` will resize any image passed to `setIconImage()` to fit the bound it needs. Figure 3.5 shows the top of a `JFrame` with a custom icon.

Figure 3.5 JFrame custom Icon
<<file figure3-5.gif>>

3.1.10 Centering a frame on the screen

By default a `JFrame` displays itself in the upper left-hand corner of the screen. Often it is desirable to place it in the center of the screen. Using the `getToolkit()` method of the `Window` class (of which `JFrame` is a second level subclass), we can communicate with the operating system and query the size of the screen. (The `Toolkit` methods make up the bridge between Java components and their native, operating-system-specific, peer components.)

The `getScreenSize()` method gives us the information we need:

```
Dimension dim = getToolkit().getScreenSize();
```

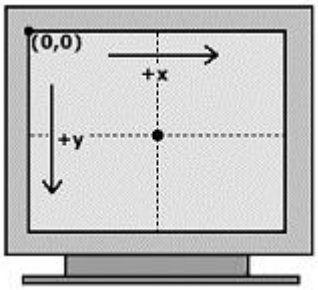


Figure 3.6 Screen coordinates
<<file figure3-6.gif>>

When setting the location of the `JFrame`, the upper left hand corner of the frame is the relevant coordinate. So to center a `JFrame` on the screen we need to subtract half its width and half its height from the center-of-screen coordinate:

```
myJFrame.setLocation(dim.width/2 - myJFrame.getWidth()/2,
  dim.height/2 - myJFrame.getHeight()/2);
```

3.1.11 JApplet

class javax.swing.JApplet

`JApplet` is the Swing equivalent of the AWT `Applet` class. Like `JFrame`, `JApplet`'s main child component is a `JRootPane` and its structure is the same. `JApplet` acts just like `Applet` and we will not go into detail about how applets work.

Reference: We suggest that readers unfamiliar with applets refer to the Java tutorial to learn more: http://java.sun.com/docs/books/tutorial/

Several examples in later chapters are constructed as Swing applets and we will see `JApplet` in action soon enough. Appendix A contains information about deploying Swing-based applets for use in a browser, Java 2 applet security issues, and working with JAR archives.

3.1.12 JWindow

class javax.swing.JWindow

`JWindow` is very similar to `JFrame` except that it has no title bar, and is not resizable, minimizable, maximizable, or closable. Thus it cannot be dragged without writing custom code to do so (e.g. `JToolBar`'s UI delegate provides this functionality for docking and undocking). We normally use `JWindow` to display a

temporary message or splash screen logo. Since `JWindow` is a `RootPaneContainer`, we can treat it just like `JFrame` or `JApplet` when manipulating its contents.

Note: We will use `JWindow` to display a splash screen in several complex examples later to provide simple feedback to the user when potentially long startup times are encountered.

3.1.12 JPanel

class javax.swing.JPanel

This is the simple container component commonly used to organize a group or groups of child components inside another container. `JPanel` is an integral part of `JRootPane`, as we discussed above, and is used in each example throughout this book. Each `JPanel`'s child components are managed by a layout manager. A layout manager controls the size and location of each child in a container. `JPanel`'s default layout manager is `FlowLayout` (we will discuss this more in chapter 4). The only exception to this is `JRootPane`'s `contentPane`, which is managed by a `BorderLayout` by default.

3.2 Borders

package javax.swing.border

The `border` package provides us with the following border classes which can be applied to any Swing component:

`BevelBorder`
: A 3D border with a raised or lowered appearance.

`CompoundBorder`
: A combination of two borders: an inside border and an outside border.

`EmptyBorder`
: A transparent border used to define empty space (often referred to as white space) around a component.

`EtchedBorder`
: A border with an etched line appearance.

`LineBorder`
: A flat border with a specified thickness and color.

`MatteBorder`
: A border consisting of either a flat color or tiled image.

`SoftBevelBorder`
: A 3D border with a raised or lowered appearance, and rounded edges.

`TitledBorder`
: A border allowing a `String` title in a specific location and position. We can set the title font, color, justification, and position of the title text using `TitleBorder` methods and constants where necessary (see API docs).

To set the border of a Swing component we simply call `JComponent`'s `setBorder()` method. There is also a convenience class called `BorderFactory`, contained in the `javax.swing` package (not the `javax.swing.border` package as you might think), which contains a group of static methods used for constructing borders quickly. For example, to create an `EtchedBorder` we can use `BorderFactory` as follows:

```
myComponent.setBorder(BorderFactory.createEtchedBorder());
```

The border classes do not provide methods set their dimensions, colors, etc. Instead of modifying an existing border we are normally expected to create a new instance to replace the old one.

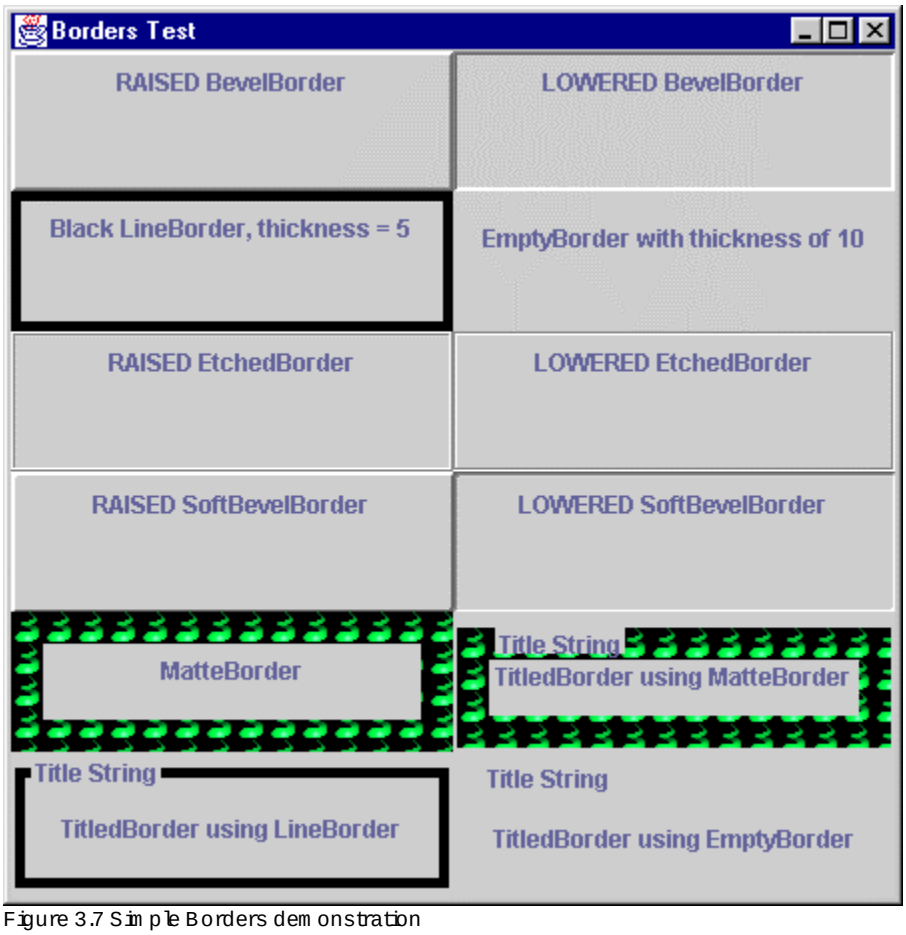


Figure 3.7 Simple Borders demonstration

<<file figure3-7.gif>>

The following code creates a `JFrame` containing twelve `JPanels` using borders of all types. Figure 3.7 illustrates:

The Code: BorderTest.java
see \Chapter3\1

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

class BorderTest extends JFrame
{
  public BorderTest() {
    setTitle("Border Test");
    setSize(450, 450);

    JPanel content = (JPanel) getContentPane();
    content.setLayout(new GridLayout(6,2));

    JPanel p = new JPanel();
    p.setBorder(new BevelBorder (BevelBorder.RAISED));
    p.add(new JLabel("RAISED BevelBorder"));
    content.add(p);

    p = new JPanel();
```

```
    p.setBorder(new BevelBorder (BevelBorder.LOWERED));
    p.add(new JLabel("LOWERED BevelBorder"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new LineBorder (Color.black, 5));
    p.add(new JLabel("Black LineBorder, thickness = 5"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new EmptyBorder (10,10,10,10));
    p.add(new JLabel("EmptyBorder with thickness of 10"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new EtchedBorder (EtchedBorder.RAISED));
    p.add(new JLabel("RAISED EtchedBorder"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new EtchedBorder (EtchedBorder.LOWERED));
    p.add(new JLabel("LOWERED EtchedBorder"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new SoftBevelBorder (SoftBevelBorder.RAISED));
    p.add(new JLabel("RAISED SoftBevelBorder"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new SoftBevelBorder (SoftBevelBorder.LOWERED));
    p.add(new JLabel("LOWERED SoftBevelBorder"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new MatteBorder (new ImageIcon("spiral.gif")));
    p.add(new JLabel("MatteBorder"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new TitledBorder (
      new MatteBorder (new ImageIcon("spiral.gif")),
      "Title String"));
    p.add(new JLabel("TitledBorder using MatteBorder"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new TitledBorder (
      new LineBorder (Color.black, 5),
      "Title String"));
    p.add(new JLabel("TitledBorder using LineBorder"));
    content.add(p);

    p = new JPanel();
    p.setBorder(new TitledBorder (
      new EmptyBorder (10,10,10,10),
      "Title String"));
    p.add(new JLabel("TitledBorder using EmptyBorder"));
    content.add(p);

    setVisible(true);
  }
```

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

UIGuideline:

Borders for VisualLayering. Use borders to create a visual association between components on a view. Bevelled borders are graphically very strong and can be used to strongly associate items. Windows Look & Feel does this. For example buttons use a RAISED BevelBorder and data fields use a LOWERED BevelBorder. If you want to visually associate components or draw attention to a component then you can create a visual layer by careful use of BevelBorder. If you want to draw attention to a particular button or group of buttons, you might consider thickening the RAISED bevel using BorderInsets discussed in 3.2.2

Borders for VisualGrouping. Use borders to create Group Boxes. EtchedBorder and LineBorder are particularly effective for this, as they are graphically weaker then BevelBorder. EmptyBorder is also very useful for grouping. It uses the power of negative (white) space to visually associate the contained components and draw the viewer eye to the group.

You may wish to create a visual grouping of attributes or simply signify the bounds of a set of choices. Grouping related Radio Buttons and Checkboxes is particularly useful.

Achieving Visual Integration and Balance using Negative Space. Use a compound border including an EmptyBorder to increase the Negative (white) Space around a Component or Panel. Visually, a Border sets what is known as a Ground (or area) for a Figure. The Figure is what is contained within the Border. It is important to keep Figure and Ground in balance. This is done by providing adequate white space around the Figure. The stronger the border, the more white space will be required e.g. a BevelBorder will require more white space than an EtchedBorder. Ref. Mullet 95 (see Appendix B).

Border for VisualGrouping with Layering. Doubly compounded borders can be used to group information and communicate hierarchy using VisualLayering. Consider the following implementation (fig 3-8). Here we are indicating a common belonging for the attributes within the border. They are both attributes of Customer. Because we have indicated the label Customer (top LHS) in the border title, we do not need to repeat the label for each field. We are further communicating the type of the Customer with the VIP label (bottom RHS).

VisualLayering of the hierachy involved is achieved by position and font.

(i) Position: In western cultures, the eye is trained to scan from top left to bottom right. Thus something located top left has a visual higher rank than something located bottom right.

(ii) Font: By bolding the Customer, we are clearly communicating it as the highest ranking detail. What we are displaying is a Customer of type VIP. Not a VIP of type Customer. The positioning and re-inforcing with heavier font, clearly communicate this message

---

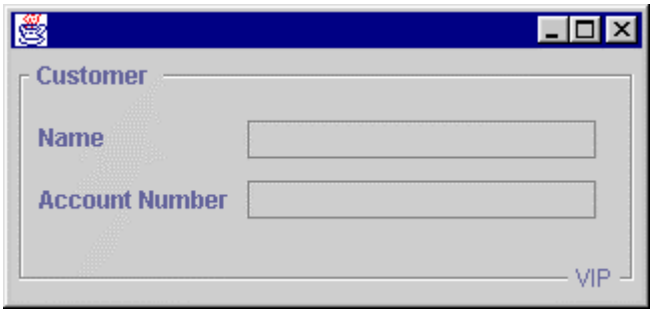


Figure 3.8 VisualGrouping with Layering

<<file figure3-8.gif>>

3.2.2 Inside Borders

It is important to understand that borders are not components. In fact `AbstractBorder`, the abstract class all border classes are derived from, directly extends `Object`. Thus we cannot attach action and mouse listeners, set tooltips, etc.

Note: This has certain side effects, one of which is that borders are much less efficient in painting themselves. There is no optimization support like there is in `JComponent`. We can do intersting things like use a very thick `MatteBorder` to tile a panel with an image, but this is an inefficient (and also unreliable) solution. In general don't use really large borders for anything. If you need an extremely large border consider simulating one using `JLabels` and a container managed by `BorderLayout`.

One major benefit of `Borders` not being components is that we can use a single `Border` instance with an arbitrary number of components. In large scale apps this can reduce a significant amount of overhead.

When a Swing component is assigned a border its `Insets` are defined by that border's width and height settings. When layout managers lay out `JComponents`, as we will see in the next chapter, they take into account their `Insets` and normally use `JComponent`'s `getInsets()` method to obtain this information. Inside the `getInsets()` method, the current border is asked to provide its `Insets` using the `getBorderInsets()` method.

The `Insets` class consists of four publicly accessible `int` values: `bottom`, `left`, `right`, `top`. `TitleBorder` must compute its `Insets` based on its current font and text position which will not effect every side and requires handling for many different cases--without a doubt this is the most complex border provided by Swing. In the case of `CompoundBorder`, both its outer and inner `Insets` are retrieved through calls to `getBorderInsets()` and then summed. A `MatteBorder`'s `Insets` are determined by the width and height of its image. `BevelBorder` and `EtchedBorder`, have `Insets` values: 2, 2, 2, 2. `SoftBevelBorder` has Insets values: 3, 3, 3, 3. `EmptyBorder`'s `Insets` are simply the values that were passed in to the constructor. Each of `LineBorder`'s `Insets` values equals the thickness that was specified in the constructor (or 1 as the default).

`Borders` get painted last in the `JComponent` rendering pipeline to ensure that they always appear on top of their associated component. `AbstractBorder` defines methods to get a `Rectangle` representing the interior region of the component a border is attached to: `getInteriorRectangle()`. Any `JComponent` subclass implementing its own painting methods may be interested in this area. Combined with the `Graphics` clipping area, components may use this information to minimize their rendering work (refer back to chapter 2).

3.3 Creating a custom border

To create a custom border we can implement the `javax.swing.Border` interface and define the following three methods:

- `void paintBorder(Component c, Graphics g)`: perform the border rendering--only paint within the `Insets` region.
- `Insets getBorderInsets(Component c)`: return an `Insets` instance representing the top, bottom, left, and right thicknesses.
- `boolean isBorderOpaque()`: return whether or not the border is opaque or transparent.

The following class is a simple implementation of a custom rounded-rectangle border which we call `OvalBorder`.

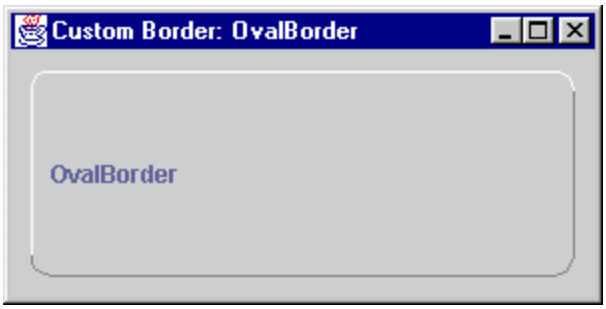


Figure 3.9 A custom rounded-corner Border implementation.
<<file figure3-9.gif>>

The Code: OvalBorder.java
see \Chapter3\2

```
import java.awt.*;

import javax.swing.*;
import javax.swing.border.*;

public class OvalBorder implements Border
{
  protected int m_w=6;
  protected int m_h=6;
  protected Color m_topColor = Color.white;
  protected Color m_bottomColor = Color.gray;

  public OvalBorder() {
    m_w=6;
    m_h=6;
  }

  public OvalBorder(int w, int h) {
    m_w=w;
    m_h=h;
  }

  public OvalBorder(int w, int h, Color topColor,
   Color bottomColor) {
    m_w=w;
    m_h=h;
    m_topColor = topColor;
    m_bottomColor = bottomColor;
  }

  public Insets getBorderInsets(Component c) {
    return new Insets(m_h, m_w, m_h, m_w);
  }

  public  boolean isBorderOpaque() { return true; }

  public void paintBorder(Component c, Graphics g,
   int x, int y, int w, int h) {
    w--;
    h--;
    g.setColor(m_topColor);
    g.drawLine(x, y+h-m_h, x, y+m_h);
    g.drawArc(x, y, 2*m_w, 2*m_h, 180, -90);
    g.drawLine(x+m_w, y, x+w-m_w, y);
```

```
    g.drawArc(x+w-2*m_w, y, 2*m_w, 2*m_h, 90, -90);

    g.setColor(m_bottomColor);
    g.drawLine(x+w, y+m_h, x+w, y+h-m_h);
    g.drawArc(x+w-2*m_w, y+h-2*m_h, 2*m_w, 2*m_h, 0, -90);
    g.drawLine(x+m_w, y+h, x+w-m_w, y+h);
    g.drawArc(x, y+h-2*m_h, 2*m_w, 2*m_h, -90, -90);
  }

  public static void main(String[] args) {
    JFrame frame = new JFrame("Custom Border: OvalBorder");
    JLabel label = new JLabel("OvalBorder");
    ((JPanel) frame.getContentPane()).setBorder(new CompoundBorder(
      new EmptyBorder(10,10,10,10), new OvalBorder(10,10)));
    frame.getContentPane().add(label);
    frame.setBounds(0,0,300,150);
    frame.setVisible(true);
  }
}
```

Understanding the Code

This border consists of a raised shadowed rectangle with rounded corners. Instance variables:

`int m_w`: left and right inset value.

`int m_h`: top and bottom inset value.

`Color m_topColor`: non-shadow color.

`Color m_bottomColor`: shadow color.

Three constructors are provided to allow optional specification of the width and height of left/right and top/bottom inset values respectively. We can also specify the shadow color (bottom color) and non-shadow color (top color). The inset values default to 6, the top color defaults to white, and the shadow color defaults to gray.

The `isBorderOpaque()` method always returns true to signify that this border's region will always be completely filled. `getBorderInsests()` simpley returns an Insets instance made up of the left/right and top/bottom inset values specified in a constructor.

The `paintBorder()` method is responsible of rendering our border and simply paints a sequence of four lines and arcs in the approproate colors. By simply reversing the use of `bottomColor` and `topColor` we can switch from a raised look to a lowered look (a more flexible implementation might include a raised/lowered flag, and an additional constructor parameter used to specify this).

The `main()` method creates a `JFrame` with content pane surrounded by a `CompoundBorder`. The outer border is an `EmptyBorder` to provide white space, and the inner border is an instance of our `OvalBorder` class with inset values of `10`.

Running the Code

Figure 3.9 illustrates. Try running this class and resize the parent frame. Note that with a very small width or height the border does not render itself pleasantly. A more professional implementation would take this into account in the `paintBorder()` routine.

Chapter 4. Layout Managers

In this chapter:

- Layouts overview
- Comparing common layout managers
- Using GridBagLayout, by James Tan
- Choosing the right layout
- Custom layout manager: part I - Label/field pairs
- Custom layout manager: part II - Common interfaces
- Dynamic layout in a JavaBeans container

4.1 Layouts overview

In this chapter we present several examples showing how to use various layouts to satisfy specific goals, and how to create two custom layout managers that simplify the construction of many common interfaces. We also show how to construct a basic container for JavaBeans which must be able to manage a dynamic number of components. But before we present these examples it is helpful to understand the big picture of layouts, which classes use their own custom layouts, and exactly what it means to be a layout manager.

All layout managers implement one of two interfaces defined in the `java.awt` package: `LayoutManager` or its subclass, `LayoutManager2`. `LayoutManager` declares a set of methods that are intended to provide a straight-forward, organized means of managing component positions and sizes in a container. Each implementation of `LayoutManager` defines these methods in different ways accoring to its specific needs. `LayoutManager2` enhances this by adding methods intended to aid in managing component postitions and sizes using constraints-based objects. Constraints-based objects usually store position and sizing information about one component and implementations of `LayoutManager2` normally store one contraints-based object per component. For instance, `GridBagLayout` uses a `Hashtable` to map each `Component` it manages to its own `GridBagConstraints` object.

Figure 4.1 shows all the classes implementing `LayoutManager` and `LayoutManager2`. Note that there are several UI classes implementing these interfaces to provide custom layout functionality for themselves. The other classes--the classes we are most familar and concerned with--are built solely to provide help in laying out containers they are assigned to.

Each container should be assigned one layout manager, and no layout manager should be used to manage more than one container.

Note: In the case of several UI components shown in figure 4.1, the container and the layout manager are the same object. Normally, however, the container and the layout manager are separate objects that communicate heavily with each other.

Figure 4.1 LayoutManager and LayoutManager2 implementations
<<file figure4-1.gif>>

4.1.1 LayoutManager

abstract interface java.awt.LayoutManager

This interface must be implemented by any layout manager. Two methods are especially noteworthy:

`layoutContainer(Container parent)` calculates and sets the bounds for all components in the given container.

`preferredLayoutSize(Container parent)` calculates the preferred size requirements to lay out components in the given container and returns a `Dimension` instance representing this size.

4.1.2 LayoutManager2

abstract interface java.awt.LayoutManager2

This interface extends `LayoutManager` to provide a framework for those layout managers that use constraints-based layouts. Method `addLayoutComponent(Component comp, Object constraints)` adds a new component associated with a constraints-based object which carries information about how to lay out this component.

A typical implementation is `BorderLayout` which requires a direction (north, east, etc.) to position a component. In this case the constraint objects used are static `Strings` such as `BorderLayout.NORTH`, `BorderLayout.EAST`, etc. We are normally blind to the fact that `BorderLayout` is constraints-based because we are never required to manipulate the constraint objects at all. This is not the case with layouts such as `GridBagLayout`, where we must work directly with the contraint objects (instances of `GridBagConstraints`).

4.1.3 BoxLayout

class javax.swing.BoxLayout

`BoxLayout` organizes the components it manages along either the x-axis or y-axis of the owner panel. The only constructor, `BoxLayout(Container target, int axis)`, takes a reference to the `Container` component it will manage and a direction (`BoxLayout.X_AXIS` or `BoxLayout.Y_AXIS`). Components are laid out according to their preferred sizes and not wrapped, even if the container does not provide enough space.

4.1.4 Box

class javax.swing.Box

To make using the `BoxLayout` manager easier, Swing also provides a class named `Box` which is a container with an automatically assigned `BoxLayout` manager. To create an instance of this container we simply pass the desired alignment to its constructor. The `Box` class also supports the insertion of invisible blocks (instances of `Box.Filler`--see below) allowing regions of unused space to be specified. These blocks are basically lightweight components with bounds (position and size) but no view.

4.1.5 Filler

static class javax.swing.Box.Filler

This static inner class defines invisible components that affect a container's layout. The `Box` class provides convenient static methods for the creation of three different variations: glue, struts, and rigid areas.

- `createHorizontalGlue()`, `createVerticalGlue()`: returns a component which fills the space between its neighboring components, pushing them aside to occupy all available space (this functionality is more analagous to a spring than it is to glue).
- `createHorizontalStrut(int width)`, `createVerticalStrut(int height)`: returns a fixed-width (height) component which provides a fixed gap between its neighbors.
- `createRigidArea(Dimension d)`: returns an invisible component of fixed width and height.

Note: All relevant `Box` methods are static and, as such, they can be applied to any container managed by a `BoxLayout`, not just instances of `Box`. `Box` should be thought of as utilities class as much as it is a container.

4.1.6 FlowLayout

class java.awt.FlowLayout

This is a simple layout which places components from left to right in a row using the preferred component sizes (i.e. size returned by `getPreferredSize()`) until no space in the container is available. When no space is available a new row is started. Because this placement depends on the current size of the container we cannot always guarantee in advance which row a component will be placed in.

`FlowLayout` is too simple to rely on in serious applications where we want to be sure, for instance, that a set of buttons will reside at the bottom of a dialog and not on it's right side. However, it can be useful as a pad for a single component to ensure that this component will be placed in the center of a container. Note that `FlowLayout` is the default layout for all `JPanels` (the only exception is the content pane of a `JRootPane` which is always initialized with a `BorderLayout`).

4.1.7 GridLayout

class java.awt.GridLayout

This layout places components in a rectangular grid. There are three constructors:

`GridLayout()`: creates a layout with one column per component. Only one row is used.

`GridLayout(int rows, int cols)`: creates a layout with the given number of rows and columns.

`GridLayout(int rows, int cols, int hgap, int vgap)`: creates a layout with the given number of rows and columns, and the given size of horizontal and vertical gaps between each row and column.

`GridLayout` places components from left to right and from top to bottom assigning the same size to each. It forces occupation of all available container space and shares this space evenly between components. When not used carefully this can lead to undesirable component sizing, such as text boxes three times higher than expected.

4.1.8 GridBagLayout

class java.awt.GridBagLayout, class java.awt.GridBagConstraints

This layout extends the capabilities of `GridLayout` to become constraints-based. It breaks the container's space into equal rectangular pieces (like bricks in a wall) and places each component in one or more of these pieces. You need to create and fill a `GridBagConstraints` object for each component to inform `GridBagLayout` how to place and size that component.

`GridBagLayout` can be effectively used for placement of components, if no special behavior is required on resizing. However, due to it's complexity it usually requires some helper methods or classes to handle all necessary constraints information. James Tan, a usability expert and `GridBagLayout` extraordinaire, gives a comprehensive overview of this manager in section 4.3. He also presents a helper class to ease the burden of dealing with `GridBagConstraints`.

4.1.9 BorderLayout

class java.awt.BorderLayout

This layout divides a container into five regions: center, north, south, east, and west. To specify the region to place a component in we use `String`s of the form "Center," "North," etc., or the static `String` fields defined in `BorderLayout`: `BorderLayout.CENTER`, `BorderLayout.NORTH`, etc. During the layout process, components in the north and south regions will first be allotted their preferred height (if possible) and the width of the container. Once south and north components have been assigned sizes, components in the east and west regions will attempt to occupy their preferred width and any remaining height between the north and south components. A component in the center region will occupy all remaining available space. `BorderLayout` is very useful, especially in conjunction with other layouts as we will see in this and future chapters.

4.1.10 CardLayout

class java.awt.CardLayout

`CardLayout` treats all components similar to cards of equal size overlapping one another. Only one card component is visible at any given time (figure 4.2 illustrates). Methods `first()`, `last()`, `next()`, `previous()`, and `show()` can be called to switch between components in the parent `Container`.

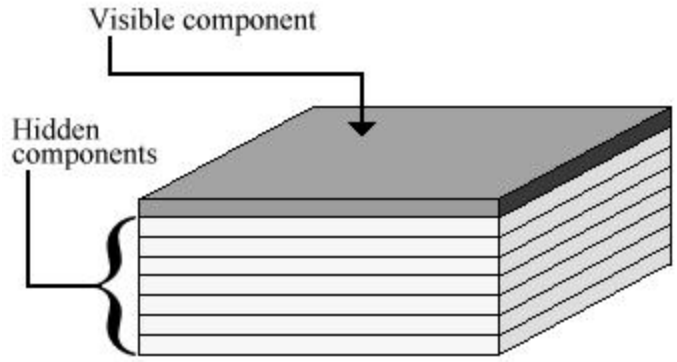


Figure 4.2 CardLayout
<<file figure4-2.gif>>

In a stack of several cards, only the top-most card is visible. The following code a simple `CardLayout` demo, which endlessly flips through four cards containing buttons.

4.1.11 JPanel

class javax.swing.JPanel

This class represents a generic lightweight container. It works in close cooperation with layout managers. The default constructor creates a `JPanel` with `FlowLayout`, but different layouts can be specified in the constructor or assigned using the `setLayout()` method (see chapter 3 for more about `JPanel`).

Note: The content pane of a `JRootPane` container is a `JPanel` which, by default, is assigned a `BorderLayout`, not a `FlowLayout`.

Note: We have purposely omitted the discussion of several layout managers here (e.g. `ViewportLayout`, `ScrollPaneLayout`, `JRootPane.RootPaneLayout`, etc.) because they are rarely used by developers and are more appropriately discussed in terms of the components that rely on them. For instance, we discuss `ViewportLayout` and `ScrollPaneLayout` in chapter 7.

4.2 Comparing common layout managers

The following example demonstrates the most commonly used AWT and Swing layout managers. It shows a set of `JInternalFrames` containing identical sets of components, each using a different layout. The purpose of this example is to allow direct simultaneous layout manager comparisons using resizable containers.

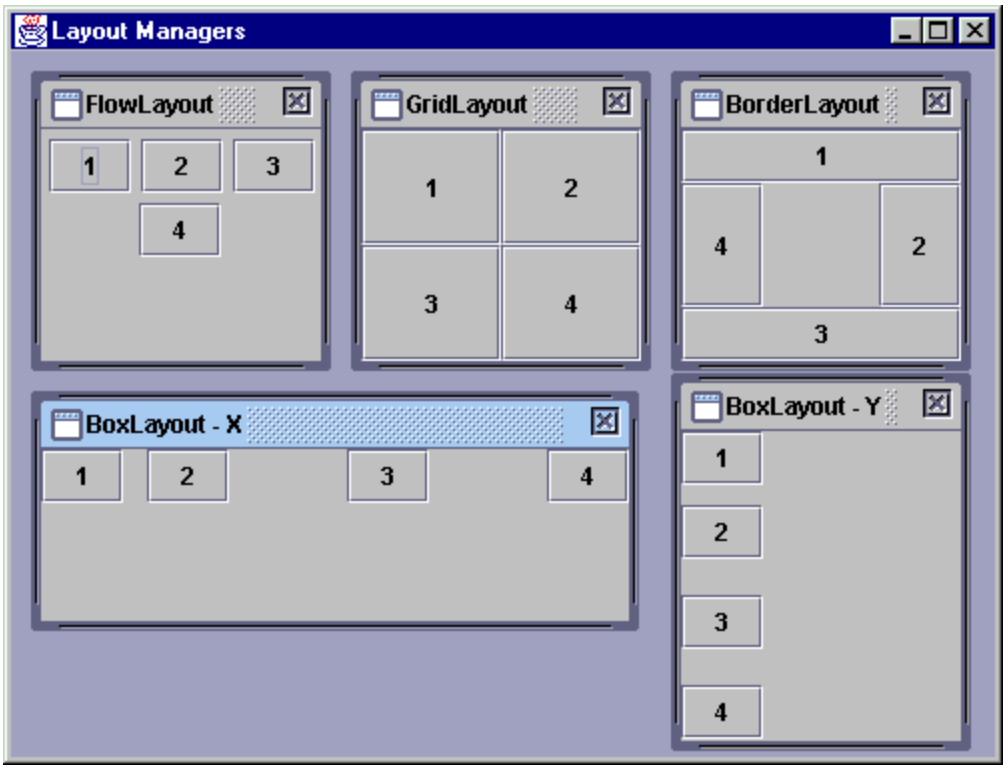


Figure 4.3 Comparing common layouts

<<file figure4-3.gif>>

The Code: CommonLayouts.java

see \Chapter4\1

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class CommonLayouts extends JFrame
{
  public CommonLayouts() {
    super("Common Layout Managers");
    setSize(500, 380);

    JDesktopPane desktop = new JDesktopPane();
    getContentPane().add(desktop);

    JInternalFrame fr1 =
      new JInternalFrame("FlowLayout", true, true);
    fr1.setBounds(10, 10, 150, 150);
    Container c = fr1.getContentPane();
    c.setLayout(new FlowLayout());
    c.add(new JButton("1"));
    c.add(new JButton("2"));
    c.add(new JButton("3"));
    c.add(new JButton("4"));
    desktop.add(fr1, 0);

    JInternalFrame fr2 =
```

```
    new JInternalFrame("GridLayout", true, true);
  fr2.setBounds(170, 10, 150, 150);
  c = fr2.getContentPane();
  c.setLayout(new GridLayout(2, 2));
  c.add(new JButton("1"));
  c.add(new JButton("2"));
  c.add(new JButton("3"));
  c.add(new JButton("4"));
  desktop.add(fr2, 0);

  JInternalFrame fr3 =
    new JInternalFrame("BorderLayout", true, true);
  fr3.setBounds(330, 10, 150, 150);
  c = fr3.getContentPane();
  c.add(new JButton("1"), BorderLayout.NORTH);
  c.add(new JButton("2"), BorderLayout.EAST);
  c.add(new JButton("3"), BorderLayout.SOUTH);
  c.add(new JButton("4"), BorderLayout.WEST);
  desktop.add(fr3, 0);

  JInternalFrame fr4 = new JInternalFrame("BoxLayout - X",
    true, true);
  fr4.setBounds(10, 170, 250, 120);
  c = fr4.getContentPane();
  c.setLayout(new BoxLayout(c, BoxLayout.X_AXIS));
  c.add(new JButton("1"));
  c.add(Box.createHorizontalStrut(12));
  c.add(new JButton("2"));
  c.add(Box.createGlue());
  c.add(new JButton("3"));
  c.add(Box.createHorizontalGlue());
  c.add(new JButton("4"));
  desktop.add(fr4, 0);

  JInternalFrame fr5 = new JInternalFrame("BoxLayout - Y",
    true, true);
  fr5.setBounds(330, 170, 150, 180);
  c = fr5.getContentPane();
  c.setLayout(new BoxLayout(c, BoxLayout.Y_AXIS));
  c.add(new JButton("1"));
  c.add(Box.createVerticalStrut(10));
  c.add(new JButton("2"));
  c.add(Box.createGlue());
  c.add(new JButton("3"));
  c.add(Box.createVerticalGlue());
  c.add(new JButton("4"));
  desktop.add(fr5, 0);

  try {
    fr1.setSelected(true);
  }
  catch (java.beans.PropertyVetoException ex) {}

  WindowListener wndCloser = new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  };
  addWindowListener(wndCloser);

  setVisible(true);
}
```

```
  public static void main(String argv[]) {
    new CommonLayouts();
  }
}
```

Understanding the Code

Class CommonLayouts

The `CommonLayouts` constructor creates five `JInternalFrames` and places them in a `JDesktopPane`. Each of these frames contains four `JButtons` labeled "1," "2," "3" and "4." Each frame is assigned a unique layout manager: a `FlowLayout`, a 2x2 `GridLayout`, a `BorderLayout`, an x-oriented `BoxLayout`, and a y-oriented `BoxLayout` respectively. Note that the internal frames using `BoxLayout` also use strut and glue filler components to demonstrate their behavior.

Running the Code

Figure 4.3 shows `CommonLayouts` in action. Note the differences in each frame's content as it changes size:

- `FlowLayout` places components in one or more rows depending on the width of the container.
- `GridLayout` assigns an equal size to all components and fills all container space.
- `BorderLayout` places components along the sides of the container.
- x-oriented `BoxLayout` always places components in a row. The distance between the first and second components is 12 pixels (determined by the horizontal strut component). Distances between the second, third, and fourth components are equalized and take up all remaining width (determined by the two glue filler components).
- y-oriented `BoxLayout` always places components in a column. The distance between the first and second components is 10 pixels (determined by the vertical strut component). Distances between the second, third, and fourth components are equalized and take up all available height (determined by the two glue filler components).

4.3 Using GridBagLayout

...by James Tan, Systems Analyst, United Overseas Bank Singapore, jamestan@earthling.net

Of all the layouts included with Swing and AWT, `GridBagLayout` is by far the most complex. In this section, we will walk through the various constraints attributes it relies on, along with several short examples showing how to use them. We follow up this discussion with a comprehensive input dialog example putting together all these attributes. We then conclude this section with the construction and demonstration of a helper class designed to make using `GridBagLayout` more convenient.

4.3.1 Default behavior of GridBagLayout

By simply setting a container's layout to a `GridBagLayout` and adding `Components` to it, the result will be a row of components, each set to their preferred size, tightly packed and placed in the center of the container. Unlike `FlowLayout`, `GridBagLayout` will allow components to be clipped by the edge of the managing container, and it will not move child components down into a new row. The following code demonstrates, and figure 4.4 shows the result:

```
  JInternalFrame fr1 = new JInternalFrame(
    "Example 1", true, true );
```

```
fr1.setBounds( 5, 5, 270, 100 );
cn = fr1.getContentPane();
cn.setLayout( new GridBagLayout() );
cn.add( new JButton( "Wonderful" ) );
cn.add( new JButton( "World" ) );
cn.add( new JButton( "Of" ) );
cn.add( new JButton( "Swing !!!" ) );
desktop.add( fr1, 0 );
```

Figure 4.4 Default GridBagLayout behaviour

<<figure4-4.gif>>

4.3.2 Introducing GridBagConstraint

When a component is added to a container which has been assigned a `GridBagLayout`, a default `GridBagConstraints` object is used by the layout manager to place the component accordingly, as in the above example. By creating and setting a `GridBagConstraints`' attributes and passing it in as an additional parameter in the `add()` method, we can flexibly manage the placement of our components.

Below are the various attributes we can set in a `GridBagConstraints` object along with their default values. The behaviour of these attributes will be explained in the examples that follow.

```
public int gridx = GridBagConstraints.RELATIVE;
public int gridy = GridBagConstraints.RELATIVE;
public int gridwidth = 1;
public int gridheight = 1;
public double weightx = 0.0;
public double weighty = 0.0;
public int anchor = GridBagConstraints.CENTER;
public int fill = GridBagConstraints.NONE;
public Insets insets = new Insets( 0, 0, 0, 0 );
public int ipadx = 0;
public int ipady = 0;
```

4.3.3 Using the gridx, gridy, insets, ipadx and ipady constraints

The `gridx` and `gridy` constraints (or column and row constraints) are used to specify the exact grid cell location where we want our component to be placed. Components placement starts from the upper left corner of the container, and `gridx` and `gridy` begin with values of `0`. Specifying negative values for either of these attributes is equivalent to setting them to `GridBagConstraints.RELATIVE`, which means that the next component added will be placed directly after the previous `gridx` or `gridy` location.

The `insets` constraint adds an invisible exterior padding around the associated component. Negative values can be used which will force the component to be sized larger than the cell it is contained in.

The `ipadx` and `ipady` constraints add an interior padding which increases the preferred size of the associated component. Specifically, it adds `ipadx * 2` pixels to the preferred width and `ipady * 2` pixels to the preferred height (`* 2` because this padding applies to both sides of the component).

In this example we place the "Wonderful" and "World" buttons in the first row and the other two buttons in the second row. We also associate insets with each button so they don't look too cluttered, and they vary in both height and width.

```
JInternalFrame fr2 = new JInternalFrame("Example 2", true, true );
fr2.setBounds( 5, 110, 270, 140 );
cn = fr2.getContentPane();
cn.setLayout( new GridBagLayout() );

c = new GridBagConstraints();
c.insets = new Insets( 2, 2, 2, 2 );
c.gridx = 0;   // column 0
c.gridy = 0;   // row 0
c.ipadx = 5;   // increases component width by 10 pixels
c.ipady = 5;   // increases component height by 10 pixels
cn.add( new JButton( "Wonderful" ), c );

c.gridx = 1;   // column 1
c.ipadx = 0;   // reset the padding to 0
c.ipady = 0;
cn.add( new JButton( "World" ), c );

c.gridx = 0;   // column 0
c.gridy = 1;   // row 1
cn.add( new JButton( "Of" ), c );

c.gridx = 1;   // column 1
cn.add( new JButton( "Swing !!!" ), c );

desktop.add( fr2, 0 );
```

We begin by creating a `GridBagConstraints` object to set the constraints for the first button component. We pass it in together with the button in the `add()` method. We reuse this same constraints object by changing the relevant attributes and passing in again for each remaining component. This conserves memory and also relieves us of having to reassign a whole new group of attributes. Figure 4.5 shows the result.

Figure 4.5 Using the gridx, gridy, insets, ipadx and ipady constraints.
<<figure4-5.gif>>

4.3.4 Using weightx and weighty constraints

When the container in the above example is resized, the components respect the constraints we have assigned, but the whole group remains in the center of the container. Why don't the buttons grow to occupy a proportional amount of the increased space surrounding them? The answer lies in the use of the `weightx` and `weighty` constraints, which both default to zero when `GridBagConstraints` is instantiated.

These two constraints specify how any extra space in a container should be distributed among each component's cell. The `weightx` attribute is used to specify the fraction of extra horizontal space to occupy. Similarly, `weighty` is used to specify the fraction of extra vertical space to occupy. Both constraints can be assigned values ranging from `0.0` to `1.0`.

For example, lets say we have two buttons, A and B, placed in columns 0 and 1 of row 0 respectively. If we specify `weightx = 1.0` for the first button and `weightx = 0` for the second button, when we resize the container, all extra space will be distributed to the first button's cell -- 50% on the left of the button and 50% on the right. The other button will be pushed to the right of the container as far as possible. Figure 4.6 illustrates.

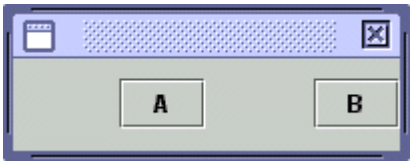


Figure 4.6 weightx and weighty constraints.
<<figure4-6.gif>>

Getting back to our "Wonderful" "World" "Of" "Swing !!!" example, we now modify all button cells to share any extra container space equally as the container is resized. Specifying `weightx = 1.0` and `weighty = 1.0`, and keeping these attributes constant as each component is added, will tell `GridBagLayout` to use all available space for each cell.

```
JInternalFrame fr3 = new JInternalFrame("Example 3", true, true );
fr3.setBounds( 5, 255, 270, 140 );
cn = fr3.getContentPane();
cn.setLayout( new GridBagLayout() );

c = new GridBagConstraints();
c.insets = new Insets( 2, 2, 2, 2 );
c.weighty = 1.0;
c.weightx = 1.0;
c.gridx = 0;
c.gridy = 0;
cn.add( new JButton( "Wonderful" ), c );

c.gridx = 1;
cn.add( new JButton( "World" ), c );

c.gridx = 0;
c.gridy = 1;
cn.add( new JButton( "Of" ), c );

c.gridx = 1;
cn.add( new JButton( "Swing !!!" ), c );

desktop.add( fr3, 0 );
```

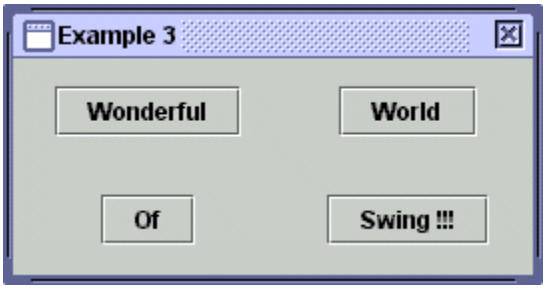
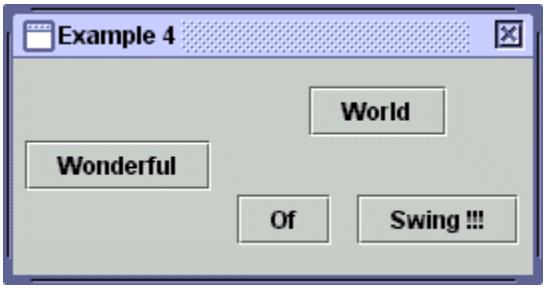


Figure 4.7 Using weightx and weighty constraints.
<<figure4-7.gif>>

4.3.5 Using gridwidth and gridheight constraints

`GridBagLayout` also allows us to span components across multiple cell using the `gridwidth` and `gridheight` constraints. To demonstrate we modify our example to force the "Wonderful" button to occupy 2 rows and the "World" button to occupy 2 columns. Figure 4.8 illustrates. Note that occupying more cells forces more rows and/or columns to be created based on current container size.

```
JInternalFrame fr4 = new JInternalFrame("Example 4", true, true );
fr4.setBounds( 280, 5, 270, 140 );
cn = fr4.getContentPane();
cn.setLayout( new GridBagLayout() );

c = new GridBagConstraints();
c.insets = new Insets( 2, 2, 2, 2 );
c.weighty = 1.0;
c.weightx = 1.0;
c.gridx = 0;
c.gridy = 0;
c.gridheight = 2; // span across 2 rows
cn.add( new JButton( "Wonderful" ), c );

c.gridx = 1;
c.gridheight = 1; // remember to set back to 1 row
c.gridwidth = 2; // span across 2 columns
cn.add( new JButton( "World" ), c );

c.gridy = 1;
c.gridwidth = 1; // remember to set back to 1 column
cn.add( new JButton( "Of" ), c );

c.gridx = 2;
cn.add( new JButton( "Swing !!!" ), c );

desktop.add( fr4, 0 );
```



Figures 4.8 Using gridwidth and gridheight constraints.
<<figure4-8.gif>>

4.3.6 Using anchor constraints

We can control how a component is aligned within its cell(s) by setting the `anchor` constraint. By default this is set to `GridBagConstraints.CENTER`, which forces the component to be centered within its occupied cell(s). We can choose from the following `anchor` settings:

```
GridBagConstraints.NORTH
GridBagConstraints.SOUTH
GridBagConstraints.EAST
GridBagConstraints.WEST
GridBagConstraints.NORTHEAST
GridBagConstraints.NORTHWEST
```

```
GridBagConstraints.SOUTHEAST
GridBagConstraints.SOUTHWEST
GridBagConstraints.CENTER
```

Below we've modified our example to anchor the "Wonderful" button NORTH and the "World" button SOUTHWEST. The "Of" and "Swing !!!" buttons are achored in the CENTER of their cells. Figure 4.9 illustrates.

```
JInternalFrame fr5 = new JInternalFrame("Example 5", true, true );
fr5.setBounds( 280, 150, 270, 140 );
cn = fr5.getContentPane();
cn.setLayout( new GridBagLayout() );

c = new GridBagConstraints();
c.insets = new Insets( 2, 2, 2, 2 );
c.weighty = 1.0;
c.weightx = 1.0;
c.gridx = 0;
c.gridy = 0;
c.gridheight = 2;
c.anchor = GridBagConstraints.NORTH;
cn.add( new JButton( "Wonderful" ), c );

c.gridx = 1;
c.gridheight = 1;
c.gridwidth = 2;
c.anchor = GridBagConstraints.SOUTHWEST;
cn.add( new JButton( "World" ), c );

c.gridy = 1;
c.gridwidth = 1;
c.anchor = GridBagConstraints.CENTER;
cn.add( new JButton( "Of" ), c );

c.gridx = 2;
cn.add( new JButton( "Swing !!!" ), c );

desktop.add( fr5, 0 );
```

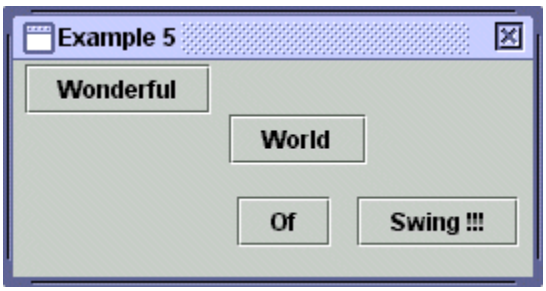


Figure 4.9 Using gridwidth and gridheight constraints.
<<figure4-9.gif>>

4.3.7 Using fill constraints

The most common reason for spanning multiple cells is because we want the component contained in that cell to occupy this enlarged space. To do this we use the gridheight/gridwidth constraints as described above, as well as the fill constraint. The fill constraint can be assigned any of the following values:

```
GridBagConstraints.NONE
```

```
GridBagConstraints.HORIZONTAL
GridBagConstraints.VERTICAL
GridBagConstraints.BOTH
```

Below we modify our example to force the "Wonderful" button to occupy all available cell space, both vertically and horizontally. The "World" button now occupies all available horizontal cell space, but continues to use its preferred vertical size. The "Of" button does not make use of the fill constraint and simply uses its preferred size. The "Swing !!!" button occupies all available vertical cell space, but uses its preferred horizontal size. Figure 4.10 illustrates.

```
JInternalFrame fr6 = new JInternalFrame("Example 6", true, true );
fr6.setBounds( 280, 295, 270, 140 );
cn = fr6.getContentPane();
cn.setLayout( new GridBagLayout() );

c = new GridBagConstraints();
c.insets = new Insets( 2, 2, 2, 2 );
c.weighty = 1.0;
c.weightx = 1.0;
c.gridx = 0;
c.gridy = 0;
c.gridheight = 2;
c.fill = GridBagConstraints.BOTH;
cn.add( new JButton( "Wonderful" ), c );

c.gridx = 1;
c.gridheight = 1;
c.gridwidth = 2;
c.fill = GridBagConstraints.HORIZONTAL;
cn.add( new JButton( "World" ), c );

c.gridy = 1;
c.gridwidth = 1;
c.fill = GridBagConstraints.NONE;
cn.add( new JButton( "Of" ), c );

c.gridx = 2;
c.fill = GridBagConstraints.VERTICAL;
cn.add( new JButton( "Swing !!!" ), c );

desktop.add( fr6, 0 );
```

Figure 4.10 Using fill constraints.

4.3.8 Putting it all together: Constructing a complaints dialog

Figure 4.11 shows a sketch of a generic complaints dialog that can be used for various forms of user feedback. This sketch shows clearly how we plan to lay out the various components, and the columns and rows in which they will be placed. In order to set the constraints correctly so that the components will be laid out as shown, we must do the following:

For the "Short Description" text field, we set the `gridwidth` constraint to `3` and the `fill` constraint to

`GridBagConstraints.HORIZONTAL`. In order to make this field occupy all the horizontal space available, we also need to set the `weightx` constraints to `1.0`.

For the "Description" text area, we set the `gridwidth` constraint to `3`, `gridheight` to `2`, and the `fill` constraint to `GridBagConstraint.BOTH`. In order to make this field occupy all available horizontal and vertical space, we set the `weightx` and `weighty` constraints to `1.0`.

For the "Name," "Telephone," "Sex" and "ID Number" input fields, we want each to use their preferred width. Since widths each exceed the width of one cell, we set `gridwidth` to `3`, and set `weightx` to `0.0` so that they have enough space to fit but they will not use any additional available horizontal space.

For the "Help" button, we set the `anchor` constraint to `GridBagConstraint.NORTH` so that it will stick together with the upper two buttons, "Submit" and "Cancel." The `fill` constraint is set to `HORIZONTAL` to force each of these buttons to occupy all available horizontal cell space.

All labels use their preferred sizes, and each component in this dialog is anchored `WEST`.

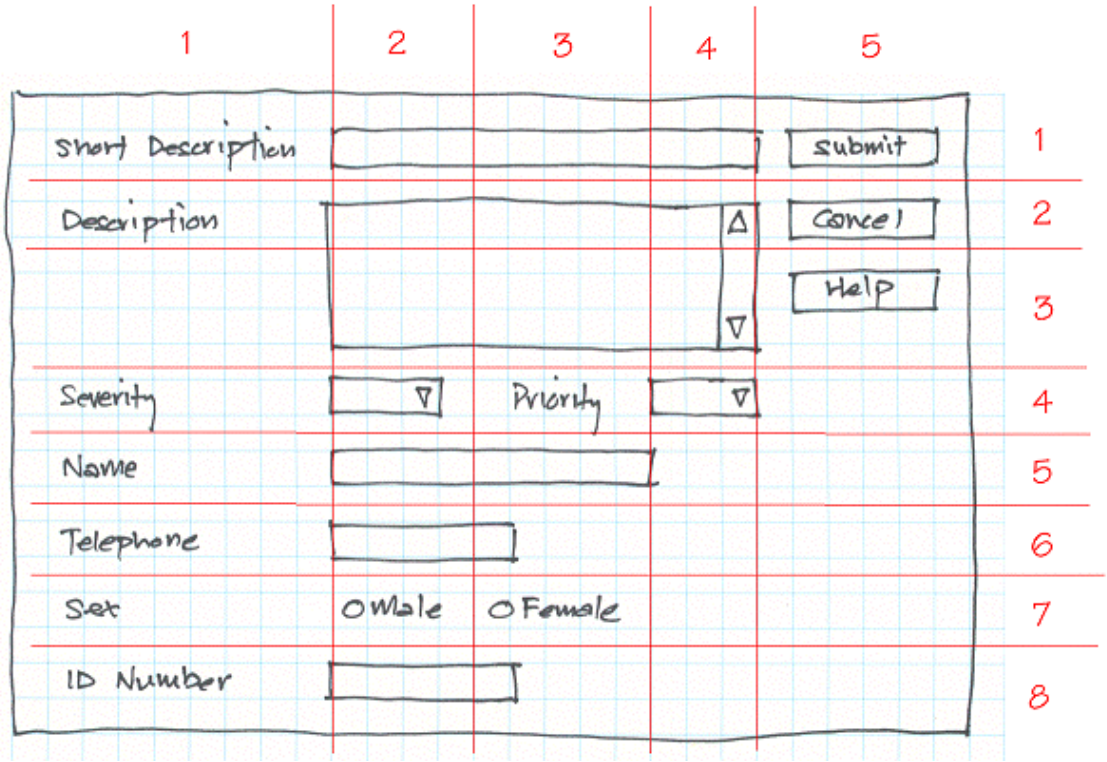


Figure 4.11 Sketch of a generic complaints dialog.
<<figure4-11.gif>>

Our implementation follows, and figure 4.12 shows the resulting dialog.

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

public class ComplaintsDialog extends JDialog
{
  public ComplaintsDialog( JFrame frame ) {
    super( frame, true );
    setTitle( "Simple Complaints Dialog" );
    setSize( 500, 300 );
```

```
// Creates a panel to hold all components
JPanel panel = new JPanel( new BorderLayout() );
panel.setLayout( new GridBagLayout() );

// give the panel a border gap of 5 pixels
panel.setBorder( new EmptyBorder( new Insets( 5, 5, 5, 5 ) ) );
getContentPane().add( BorderLayout.CENTER, panel );

GridBagConstraints c = new GridBagConstraints();

// Define preferred sizes for input fields
Dimension shortField = new Dimension( 40, 20 );
Dimension mediumField = new Dimension( 120, 20 );
Dimension longField = new Dimension( 240, 20 );
Dimension hugeField = new Dimension( 240, 80 );

// Spacing between label and field
EmptyBorder border = new EmptyBorder( new Insets( 0, 0, 0, 10 ) );
EmptyBorder border1 = new EmptyBorder( new Insets( 0, 20, 0, 10 ) );

// add space around all components to avoid clutter
c.insets = new Insets( 2, 2, 2, 2 );

// anchor all components WEST
c.anchor = GridBagConstraints.WEST;

JLabel lbl1 = new JLabel( "Short Description" );
lbl1.setBorder( border ); // add some space to the right
panel.add( lbl1, c );
JTextField txt1 = new JTextField();
txt1.setPreferredSize( longField );
c.gridx = 1;
c.weightx = 1.0; // use all available horizontal space
c.gridwidth = 3; // spans across 3 columns
c.fill = GridBagConstraints.HORIZONTAL; // fills the 3 columns
panel.add( txt1, c );

JLabel lbl2 = new JLabel( "Description" );
lbl2.setBorder( border );
c.gridwidth = 1;
c.gridx = 0;
c.gridy = 1;;
c.weightx = 0.0; // do not use any extra horizontal space
panel.add( lbl2, c );
JTextArea area1 = new JTextArea();
JScrollPane scroll = new JScrollPane( area1 );
scroll.setPreferredSize( hugeField );
c.gridx = 1;
c.weightx = 1.0; // use all available horizontal space
c.weighty = 1.0; // use all available vertical space
c.gridwidth = 3; // span across 3 columns
c.gridheight = 2; // span across 2 rows
c.fill = GridBagConstraints.BOTH; // fills the cols & rows
panel.add( scroll, c );

JLabel lbl3 = new JLabel( "Severity" );
lbl3.setBorder( border );
c.gridx = 0;
c.gridy = 3;
c.gridwidth = 1;
c.gridheight = 1;
```

```
c.weightx = 0.0;
c.weighty = 0.0;
c.fill = GridBagConstraints.NONE;
panel.add( lbl3, c );
JComboBox combo3 = new JComboBox();
combo3.addItem( "A" );
combo3.addItem( "B" );
combo3.addItem( "C" );
combo3.addItem( "D" );
combo3.addItem( "E" );
combo3.setPreferredSize( shortField );
c.gridx = 1;
panel.add( combo3, c );

JLabel lbl4 = new JLabel( "Priority" );
lbl4.setBorder( border1 );
c.gridx = 2;
panel.add( lbl4, c );
JComboBox combo4 = new JComboBox();
combo4.addItem( "1" );
combo4.addItem( "2" );
combo4.addItem( "3" );
combo4.addItem( "4" );
combo4.addItem( "5" );
combo4.setPreferredSize( shortField );
c.gridx = 3;
panel.add( combo4, c );

JLabel lbl5 = new JLabel( "Name" );
lbl5.setBorder( border );
c.gridx = 0;
c.gridy = 4;
panel.add( lbl5, c );
JTextField txt5 = new JTextField();
txt5.setPreferredSize( longField );
c.gridx = 1;
c.gridwidth = 3;
panel.add( txt5, c );

JLabel lbl6 = new JLabel( "Telephone" );
lbl6.setBorder( border );
c.gridx = 0;
c.gridy = 5;
panel.add( lbl6, c );
JTextField txt6 = new JTextField();
txt6.setPreferredSize( mediumField );
c.gridx = 1;
c.gridwidth = 3;
panel.add( txt6, c );

JLabel lbl7 = new JLabel( "Sex" );
lbl7.setBorder( border );
c.gridx = 0;
c.gridy = 6;
panel.add( lbl7, c );
JPanel radioPanel = new JPanel();

// Create a FlowLayout JPanel with 5 pixel horizontal gaps
// and no vertical gaps
radioPanel.setLayout( new FlowLayout( FlowLayout.LEFT, 5, 0 ) );
ButtonGroup group = new ButtonGroup();
JRadioButton radio1 = new JRadioButton( "Male" );
```

```
    radio1.setSelected( true );
    group.add( radio1 );
    JRadioButton radio2 = new JRadioButton( "Female" );
    group.add( radio2 );
    radioPanel.add( radio1 );
    radioPanel.add( radio2 );
    c.gridx = 1;
    c.gridwidth = 3;
    panel.add( radioPanel, c);

    JLabel lbl8 = new JLabel( "ID Number" );
    lbl8.setBorder( border );
    c.gridx = 0;
    c.gridy = 7;
    c.gridwidth = 1;
    panel.add( lbl8, c );
    JTextField txt8 = new JTextField();
    txt8.setPreferredSize( mediumField );
    c.gridx = 1;
    c.gridwidth = 3;
    panel.add( txt8, c );

    JButton submitBtn = new JButton( "Submit" );
    c.gridx = 4;
    c.gridy = 0;
    c.gridwidth = 1;
    c.fill = GridBagConstraints.HORIZONTAL;
    panel.add( submitBtn, c );

    JButton cancelBtn = new JButton( "Cancel" );
    c.gridy = 1;
    panel.add( cancelBtn, c );

    JButton helpBtn = new JButton( "Help" );
    c.gridy = 2;
    c.anchor = GridBagConstraints.NORTH; // anchor north
    panel.add( helpBtn, c );

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener( wndCloser );

    setVisible( true );
  }

  public static void main( String[] args ) {
    new ComplaintsDialog( new JFrame() );
  }
}
```

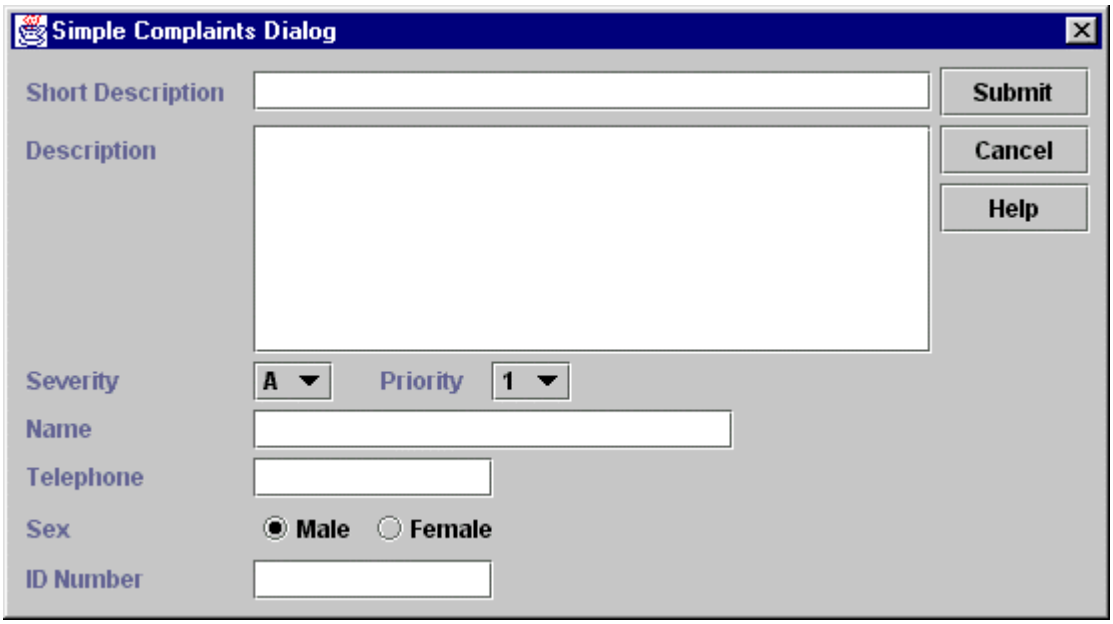


Figure 4.12 The Complaints Dialog.
<<figure4-12.gif>>

4.3.9 Simple Helper class example

As we can see from the code above, constructing dialogs with more than a few components easily becomes a very tedious task and reduces source code legibility as well as organization. One way to make the use of `GridBagLayout` cleaner and easier is to create a helper class that manages all constraints for us, and provides self-explanitory method names and predefined parameters.

Below is the source code of a simple helper class we have constructed for this purpose. The method names used are easier to understand and laying out our components using row and column parameters is more intuitive than `gridx` and `gridy`. Methods implemented in this class are each a variation of one of the following:

`addComponent`: used to add a component that needs to adhere to its preferred size.

`addAnchoredComponent`: used to add a component that needs to be anchored.

`addFilledComponent`: used to add a component that will fill the entire cell space allocated to it.

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

public class GriddedPanel extends JPanel
{
  private GridBagConstraints constraints;

  // default constraints value definitions
  private static final int C_HORZ = GridBagConstraints.HORIZONTAL;
  private static final int C_NONE = GridBagConstraints.NONE;
  private static final int C_WEST = GridBagConstraints.WEST;
  private static final int C_WIDTH = 1;
  private static final int C_HEIGHT = 1;

  // Create a GridBagLayout panel using a default insets constraint.
  public GriddedPanel() {
    this(new Insets(2, 2, 2, 2));
```

```
  }

  // Create a GridBagLayout panel using the specified insets
  // constraint.
  public GriddedPanel(Insets insets) {
    super(new GridBagLayout());
    constraints = new GridBagConstraints();
    constraints.anchor = GridBagConstraints.WEST;
    constraints.insets = insets;
  }

  // Add a component to specified row and col.
  public void addComponent(JComponent component, int row, int col) {
    addComponent(component, row, col, C_WIDTH,
      C_HEIGHT, C_WEST, C_NONE);
  }

  // Add component to specified row and col, spanning across
  // a specified number of columns and rows.
  public void addComponent(JComponent component, int row, int col,
   int width, int height ) {
    addComponent(component, row, col, width,
      height, C_WEST, C_NONE);
  }

  // Add component to specified row and col, using a specified
  // anchor constraint
  public void addAnchoredComponent(JComponent component, int row,
   int col, int anchor ) {
    addComponent(component, row, col, C_WIDTH,
     C_HEIGHT, anchor, C_NONE);
  }

  // Add component to specified row and col, spanning across
  // a specified number of columns and rows, using a specified
  // anchor constraint
  public void addAnchoredComponent(JComponent component,
   int row, int col, int width, int height, int anchor) {
     addComponent(component, row, col, width,
       height, anchor, C_NONE);
  }

  // Add component to specified row and col
  // filling the column horizontally.
  public void addFilledComponent(JComponent component,
   int row, int col) {
    addComponent(component, row, col, C_WIDTH,
     C_HEIGHT, C_WEST, C_HORZ);
  }

  // Add component to the specified row and col
  // with the specified fill constraint.
  public void addFilledComponent(JComponent component,
   int row, int col, int fill) {
    addComponent(component, row, col, C_WIDTH,
     C_HEIGHT, C_WEST, fill);
  }

  // Add component to the specified row and col,
  // spanning a specified number of columns and rows,
  // with specified fill constraint
  public void addFilledComponent(JComponent component,
```

```
   int row, int col, int width, int height, int fill) {
    addComponent(component, row, col, width, height, C_WEST, fill);
  }

  // Add component to the specified row and col,
  // spanning specified number of columns and rows, with
  // specified fill and anchor constraints
  public void addComponent(JComponent component,
   int row, int col, int width, int height, int anchor, int fill) {
    constraints.gridx = col;
    constraints.gridy = row;
    constraints.gridwidth = width;
    constraints.gridheight = height;
    constraints.anchor = anchor;
    double weightx = 0.0;
    double weighty = 0.0;

    // only use extra horizontal or vertical space if component
    // spans more than one column and/or row.
    if(width > 1)
      weightx = 1.0;
    if(height > 1)
      weighty = 1.0;

    switch(fill)
    {
      case GridBagConstraints.HORIZONTAL:
        constraints.weightx = weightx;
        constraints.weighty = 0.0;
        break;
      case GridBagConstraints.VERTICAL:
        constraints.weighty = weighty;
        constraints.weightx = 0.0;
        break;
      case GridBagConstraints.BOTH:
        constraints.weightx = weightx;
        constraints.weighty = weighty;
        break;
      case GridBagConstraints.NONE:
        constraints.weightx = 0.0;
        constraints.weighty = 0.0;
        break;
        default:
        break;
    }
    constraints.fill = fill;
    add(component, constraints);
  }
}
```

Below is the source code used to construct the same complaints dialog as above, using our helper class methods instead of manipulating the constraints directly. Note that the size of the code has been reduced and the readability improved. Also note that we add components starting at row 1 and column 1, rather than row 0 and column 0, as this is the most common numbering scheme for rows and columns (see figure 4.11).

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

public class ComplaintsDialog2 extends JDialog
```

```
{
  public ComplaintsDialog2( JFrame frame ) {
    super( frame, true );
    setTitle( "Simple Complaints Dialog" );
    setSize( 500, 300 );

    GriddedPanel panel = new GriddedPanel();
    panel.setBorder(new EmptyBorder(new Insets(5, 5, 5, 5)));
    getContentPane().add(BorderLayout.CENTER, panel);

    // Input field dimensions
    Dimension shortField = new Dimension( 40, 20 );
    Dimension mediumField = new Dimension( 120, 20 );
    Dimension longField = new Dimension( 240, 20 );
    Dimension hugeField = new Dimension( 240, 80 );

    // Spacing between labels and fields
    EmptyBorder border = new EmptyBorder(
      new Insets( 0, 0, 0, 10 ));
    EmptyBorder border1 = new EmptyBorder(
      new Insets( 0, 20, 0, 10 ));

    JLabel lbl1 = new JLabel( "Short Description" );
    lbl1.setBorder( border );
    panel.addComponent( lbl1, 1, 1 );
    JTextField txt1 = new JTextField();
    txt1.setPreferredSize( longField );
    panel.addFilledComponent( txt1, 1, 2, 3, 1,
      GridBagConstraints.HORIZONTAL );

    JLabel lbl2 = new JLabel( "Description" );
    lbl2.setBorder( border );
    panel.addComponent( lbl2, 2, 1 );
    JTextArea area1 = new JTextArea();
    JScrollPane scroll = new JScrollPane( area1 );
    scroll.setPreferredSize( hugeField );
    panel.addFilledComponent( scroll, 2, 2, 3, 2,
      GridBagConstraints.BOTH );

    JLabel lbl3 = new JLabel( "Severity" );
    lbl3.setBorder( border );
    panel.addComponent( lbl3, 4, 1 );
    JComboBox combo3 = new JComboBox();
    combo3.addItem( "A" );
    combo3.addItem( "B" );
    combo3.addItem( "C" );
    combo3.addItem( "D" );
    combo3.addItem( "E" );
    combo3.setPreferredSize( shortField );
    panel.addComponent( combo3, 4, 2 );

    JLabel lbl4 = new JLabel( "Priority" );
    lbl4.setBorder( border1 );
    panel.addComponent( lbl4, 4, 3 );
    JComboBox combo4 = new JComboBox();
    combo4.addItem( "1" );
    combo4.addItem( "2" );
    combo4.addItem( "3" );
    combo4.addItem( "4" );
    combo4.addItem( "5" );
    combo4.setPreferredSize( shortField );
    panel.addComponent( combo4, 4, 4 );
```

```
    JLabel lbl5 = new JLabel( "Name" );
    lbl5.setBorder( border );
    panel.addComponent( lbl5, 5, 1 );
    JTextField txt5 = new JTextField();
    txt5.setPreferredSize( longField );
    panel.addComponent( txt5, 5, 2, 3, 1 );

    JLabel lbl6 = new JLabel( "Telephone" );
    lbl6.setBorder( border );
    panel.addComponent( lbl6, 6, 1 );
    JTextField txt6 = new JTextField();
    txt6.setPreferredSize( mediumField );
    panel.addComponent( txt6, 6, 2, 3, 1 );

    JLabel lbl7 = new JLabel( "Sex" );
    lbl7.setBorder( border );
    panel.addComponent( lbl7, 7, 1 );
    JPanel radioPanel = new JPanel();
    radioPanel.setLayout(new FlowLayout(FlowLayout.LEFT, 5, 0));
    ButtonGroup group = new ButtonGroup();
    JRadioButton radio1 = new JRadioButton( "Male" );
    radio1.setSelected( true );
    group.add( radio1 );
    JRadioButton radio2 = new JRadioButton( "Female" );
    group.add( radio2 );
    radioPanel.add( radio1 );
    radioPanel.add( radio2 );
    panel.addComponent( radioPanel, 7, 2, 3, 1 );

    JLabel lbl8 = new JLabel( "ID Number" );
    lbl8.setBorder( border );
    panel.addComponent( lbl8, 8, 1 );
    JTextField txt8 = new JTextField();
    txt8.setPreferredSize( mediumField );
    panel.addComponent( txt8, 8, 2, 3, 1 );

    JButton submitBtn = new JButton( "Submit" );
    panel.addFilledComponent( submitBtn, 1, 5 );

    JButton cancelBtn = new JButton( "Cancel" );
    panel.addFilledComponent( cancelBtn, 2, 5 );

    JButton helpBtn = new JButton( "Help" );
    panel.addComponent(helpBtn, 3, 5, 1, 1,
      GridBagConstraints.NORTH, GridBagConstraints.HORIZONTAL);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener( wndCloser );

    setVisible( true );
  }

  public static void main( String[] args ) {
    new ComplaintsDialog2( new JFrame() );
  }
}
```

4.4 Choosing the right layout

In this section we'll show how to choose the right combination of layouts and intermediate containers to satisfy a pre-defined program specification. Consider a sample application which makes airplane ticket reservations. The following specification describes which components should be included and how they should be placed in the application frame:

1. A text field labeled "Date:", a combo box labeled "From:", and a combo box labeled "To:" must reside at the top of frame. Labels must be placed to the left side of their corresponding component. The text field and combo boxes must be of equal size, reside in a column, and occupy all available width.
2. A group of radio buttons titled "Options" must reside in the top right corner of the frame. This group must include "First class", "Business", and "Coach" radio buttons.
3. A list component titled "Available Flights" must occupy the central part of the frame and it should grow or shrink when the size of the frame changes.
4. Three buttons titled "Search", "Purchase", and "Exit" must reside at the bottom of the frame. They must form a row, have equal sizes, and be center-aligned.

Our `FlightReservation` example demonstrates how to fulfill these requirements. We do not process any input from these controls and do not attempt to put them to work; we just display them on the screen in the correct position and size. (Three variants are shown to accomplish the layout of the text fields, combo boxes, and their associated labels. Two are commented out, and a discussion of each is given below.)

Note: A similar control placement assignment is part of Sun's Java Developer certification exam.

Figure 4.13 FlightReservation layout-Variant1

<<file figure4-13.gif>>

Figure 4.14 FlightReservation layout-Variant2

<<file figure4-14.gif>>

Figure 4.15 FlightReservation layout-Variant3

<<file figure4-15.gif>>

The Code: FlightReservation.java

see \Chapter4\3

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class FlightReservation extends JFrame
{
  public FlightReservation() {
```

```
    super("Flight Reservation Dialog");
    setSize(400, 300);

    JPanel p1 = new JPanel();
    p1.setLayout(new BoxLayout(p1, BoxLayout.X_AXIS));

    JPanel p1r = new JPanel();
    p1r.setBorder(new EmptyBorder(10, 10, 10, 10));

    // Variant 1
    p1r.setLayout(new GridLayout(3, 2, 5, 5));

    p1r.add(new JLabel("Date:"));
    p1r.add(new JTextField());

    p1r.add(new JLabel("From:"));
    JComboBox cb1 = new JComboBox();
    cb1.addItem("New York");
    p1r.add(cb1);

    p1r.add(new JLabel("To:"));
    JComboBox cb2 = new JComboBox();
    cb2.addItem("London");
    p1r.add(cb2);

    p1.add(p1r);

    ///////////////
    // Variant 2 //
    ///////////////
    // JPanel p11 = new JPanel();
    // p11.setLayout(new BoxLayout(p11, BoxLayout.Y_AXIS));
    //
    // JPanel p12 = new JPanel();
    // p12.setLayout(new BoxLayout(p12, BoxLayout.Y_AXIS));
    //
    // p11.add(new JLabel("Date:"));
    // p12.add(new JTextField());
    //
    // p11.add(new JLabel("From:"));
    // JComboBox cb1 = new JComboBox();
    // cb1.addItem("New York");
    // p12.add(cb1);
    //
    // p11.add(new JLabel("To:"));
    // JComboBox cb2 = new JComboBox();
    // cb2.addItem("London");
    // p12.add(cb2);
    //
    // p1.add(p11);
    // p1.add(Box.createHorizontalStrut(10));
    // p1.add(p12);

    ///////////////
    // Variant 3 //
    ///////////////
    // JPanel p11 = new JPanel();
    // p11.setLayout(new GridLayout(3, 1, 5, 5));
    //
    // JPanel p12 = new JPanel();
    // p12.setLayout(new GridLayout(3, 1, 5, 5));
    //
```

```
    // p11.add(new JLabel("Date:"));
    // p12.add(new JTextField());
    //
    // p11.add(new JLabel("From:"));
    // JComboBox cb1 = new JComboBox();
    // cb1.addItem("New York");
    // p12.add(cb1);
    //
    // p11.add(new JLabel("To:"));
    // JComboBox cb2 = new JComboBox();
    // cb2.addItem("London");
    // p12.add(cb2);
    //
    // p1r.setLayout(new BorderLayout());
    // p1r.add(p11, BorderLayout.WEST);
    // p1r.add(p12, BorderLayout.CENTER);
    // p1.add(p1r);

    JPanel p3 = new JPanel();
    p3.setLayout(new BoxLayout(p3, BoxLayout.Y_AXIS));
    p3.setBorder(new TitledBorder(new EtchedBorder(),
      "Options"));

    ButtonGroup group = new ButtonGroup();
    JRadioButton r1 = new JRadioButton("First class");
    group.add(r1);
    p3.add(r1);

    JRadioButton r2 = new JRadioButton("Business");
    group.add(r2);
    p3.add(r2);

    JRadioButton r3 = new JRadioButton("Coach");
    group.add(r3);
    p3.add(r3);

    p1.add(p3);

    getContentPane().add(p1, BorderLayout.NORTH);

    JPanel p2 = new JPanel(new BorderLayout());
    p2.setBorder(new TitledBorder(new EtchedBorder(),
      "Available Flights"));
    JList list = new JList();
    JScrollPane ps = new JScrollPane(list);
    p2.add(ps, BorderLayout.CENTER);
    getContentPane().add(p2, BorderLayout.CENTER);

    JPanel p4 = new JPanel();
    JPanel p4c = new JPanel();
    p4c.setLayout(new GridLayout(1, 3, 5, 5));

    JButton b1 = new JButton("Search");
    p4c.add(b1);

    JButton b2 = new JButton("Purchase");
    p4c.add(b2);

    JButton b3 = new JButton("Exit");
    p4c.add(b3);

    p4.add(p4c);
```

```
    getContentPane().add(p4, BorderLayout.SOUTH);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public static void main(String argv[]) {
    new FlightReservation();
  }
}
```

Understanding the Code

Class FlightReservation

The constructor of the `FlightReservation` class creates and positions all necessary GUI components. We will explain step by step how we've chosen intermediate containers and their layouts to fulfill the requirements listed at the beginning of this section.

The frame (more specifically it's `contentPane`) is managed by a `BorderLayout` by default. A text field, and the combo boxes and associated labels are added in a separate container to the north along with the radio buttons; push buttons in the south; and the list component is placed in the center. This guarantees that top and bottom (north and south) containers will receive their natural height, and that the central component (the list) will occupy all remaining space.

The intermediate container, `JPanel p1r`, holds the text field, combo boxes, and their associated labels and is placed in panel `p1` which is managed by a horizontally aligned `BoxLayout`. The `p1r` panel is surrounded by an `EmptyBorder` to provide typical surrounding white space.

This example offers three variants of managing `p1r` and its six child components. The first variant uses a 3x2 `GridLayout`. This places labels and boxes in two columns opposite one another. Since this panel resides in the north region of the `BorderLayout`, it receives its natural (preferable) height. In the horizontal direction this layout works satisfactory: it resizes boxes and labels to occupy all available space. The only remaining problem is that `GridLayout` assigns too much space to the labels (see figure 4.13). We do not need to make labels equal in size to their corresponding input boxes. We need only allow them to occupy their preferred width.

The second variant uses two vertical `BoxLayouts` so one can hold labels and the other can hold the corresponding text field and combo boxes. If you try recompiling and running the code with this variant you'll find that the labels now occupy only their necessary width, and the boxes occupy all remaining space (see figure 4.14). This is good, but another problem arises: now the labels are aligned exactly opposite with their corresponding components. Instead, they are shifted in the vertical direction!

The third variant offers the best solution. It places the labels and their corresponding components in two columns, but uses 3x1 `GridLayouts` instead of `BoxLayouts`. This places all components evenly in the vertical direction. To provide only the minimum width to the labels (the first column) and assign all remaining space to the boxes (the second column) we place these two containers into another intermediate container managed by a `BorderLayout`: labels in the west, and corresponding components in the center. This solves our problem (see figure 4.15). The only downside to this solution is that it requires the construction of three intermediate containers with different layouts. In the next section we'll show how to build a custom layout manager that simplifies this relatively common layout task.

Now let's return to the remaining components. A group of `JRadioButtons` seems to be the simplest part of our design. They're placed into an intermediate container, `JPanel p3`, with a `TitledBorder` containing the required title: "Options". A vertical `BoxLayout` is used to place these components in a column and a `ButtonGroup` is used to coordinate their selection. This container is then added to panel `p1` (managed by a horizontal `BoxLayout`) to sit on the eastern side of panel `p1r`.

The `JList` component is added to a `JScrollPane` to provide scrolling capabilities. It is then placed in an intermediate container, `JPanel p2`, with a `TitledBorder` containing the required title "Available Flights".

Note: We do not want to assign a `TitledBorder` to the `JScrollPane` itself because this would substitute its natural border, resulting in a quite awkward scroll pane view. So we nest the `JScrollPane` in its own `JPanel` with a `TitledBorder`.

Since the list grows and shrinks when the frame is resized and the group of radio buttons (residing to the right of the list) must occupy only the necessary width, it only makes sense to placed it in the center of the `BorderLayout`. We can then use the south region for the three remaining buttons.

Since all three buttons must be equal in size, they're added to a `JPanel, p4c`, with a 1x3 `GridLayout`. However, this `GridLayout` will occupy all available width (fortunately it's limited in the vertical direction by parent `BorderLayout`). This is not exactly the behavior we are looking for. To resolve this problem we use another intermediate container, `JPanel p4`, with a `FlowLayout`. This sizes the only added component, `p4c`, based on its preferred size and centers it both vertically and horizontally.

Running the Code

Figure 4.13, 4.14, and 4.15 show the resulting placement of our components in the parent frame using the first, second, and third variants described above. Note that variant 3's placement satisfies our specification. Note also that components are resized as expected when the frame container is resized.

When the frame is stretched in the horizontal direction, the text field, combo boxes, and list component consume additional space, and the buttons at the bottom are shifted to the center. When the frame is stretched in the vertical direction, the list component and the panel containing the radio buttons consume all additional space and all other components remain unchanged.

UI Guideline: Harnessing the Power of Java Layouts

Layout managers are powerful but awkward to use. In order to maximize the effectiveness of the visual communication we must make extra effort with the code. Making a bad choice of layout or making sloppy use of default settings may lead to designs which look poor or communicate badly.

In this example, we have shown three alternative designs for the same basic specification. Each exhibit pros and cons and highlight the design trade-offs which can be made, reflecting the principles which were discussed in chapter 2.

A sense of balance.

This occurs when there is sufficient white space used to balance the size of the components. An unbalanced panel can be fixed by bordering the components with a compound border including an empty border.

A sense of scale

Balance can be further affected by the extraordinary size of some components such as the combo boxes shown in Fig 4.14. The combo boxes are bit too big for the purpose intended. This affects the sense of scale as well as the balance of the design. Its important to size comboboxes appropriately. Layout managers have a tendency to stretch components to be larger than might be desirable.

4.5 Custom layout manager: part I - Label/field pairs

This example is intended to familiarize you with developing custom layouts. You may find this knowledge useful in cases where the traditional layouts are not satisfactory or are too complex. In developing large scale applications it is often more convenient to build custom layouts, such as the one we develop here, to help with specific tasks. This often provides increased consistency, and may save a significant amount of coding in the long run.

The example in the previous section highlighted a problem: what is the best way to lay out input field components (e.g. text fields, combo boxes, etc.) and their corresponding labels? We have seen that it can be done using a combination of several intermediate containers and layouts. This section shows how we can simplify the process by using a custom-built layout manager. The goal is to construct a layout manager that knows how to lay out labels and their associated input fields in two columns, allocating the minimum required space to the column containing the labels, and using the remainder for the column containing the input fields.

First we need to clearly state our design goals for this layout manager, which we will appropriately call `DialogLayout`. It is always a good idea to reserve plenty of time for thinking about your design. Well-defined design specifications can save you tremendous amounts of time in the long run, and can help pinpoint flaws and oversights before they arise in the code. (We strongly recommend that adopting a design specification stage become part of your development regimin.)

`DialogLayout` specification:

1. This layout manager will be applied to a container that has all the necessary components added to it in the following order: `label1`, `field1`, `label2`, `field2`, etc. (Note that when components are added to a container they are tracked in a list. If no index is specified when a component is added to a container it will be added to the end of the list using the next available index. As usual this indexing starts from `0`. A component can be retreived by index using the `getComponent(int index)` method.) If the labels and fields are added correctly, all even numbered components in the container will correspond to labels, and all odd numbered components will correspond to input fields.
2. The components must be placed in pairs forming two vertical columns.
3. Components making up each pair must be placed opposite one another (i.e. `label1` `field1`). Each pair's label and field must receive the same preferable height, which should be the preferred height of the field.
4. Each left component (labels) must receive the same width. This width should be the maximum preferable width of all left components.
5. Each right component (input fields) must also receive the same width. This width should occupy all the remaining space left over from that taken by the left components column.

The code below introduces our custom `DialogLayout` class which satisfies the above design specification. This class is placed in its own package named `dl`. The code used to construct the GUI is almost identical to that of the previous example. However, we now revert back to variant 1 and use an instance of `DialogLayout` instead of a `GridLayout` to manage the `p1r JPanel`.

Figure 4.16 Using DialogLayout-custom layout manager part I
<<file figure4-16.gif>>

The Code: FlightReservation.java
see \Chapter4\4

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

import dl.*;

public class FlightReservation extends JFrame
{
  public FlightReservation() {
    super("Flight Reservation Dialog [Custom Layout]");

    // Unchanged code from section 4.4

    // Variant 1
    JPanel p1r = new JPanel();
    p1r.setBorder(new EmptyBorder(10, 10, 10, 10));
    p1r.setLayout(new DialogLayout(20, 5));

    p1r.add(new JLabel("Date:"));
    p1r.add(new JTextField());

    p1r.add(new JLabel("From:"));
    JComboBox cb1 = new JComboBox();
    cb1.addItem("New York");
    p1r.add(cb1);

    p1r.add(new JLabel("To:"));
    JComboBox cb2 = new JComboBox();
    cb2.addItem("London");
    p1r.add(cb2);
```

```
    p1.add(p1r);
    getContentPane().add(p1, BorderLayout.NORTH);
    // end Variant 1

// All remaining code unchanged from section 4.4
```

DialogLayout.java
see \Chapter4\4\dl

```
package dl;

import java.awt.*;
import java.util.*;

public class DialogLayout implements LayoutManager
{
  protected int m_divider = -1;
  protected int m_hGap = 10;
  protected int m_vGap = 5;

  public DialogLayout() {}

  public DialogLayout(int hGap, int vGap) {
    m_hGap = hGap;
    m_vGap = vGap;
  }

  public void addLayoutComponent(String name, Component comp) {}

  public void removeLayoutComponent(Component comp) {}

  public Dimension preferredLayoutSize(Container parent) {
    int divider = getDivider(parent);

    int w = 0;
    int h = 0;
    for (int k=1 ; k<parent.getComponentCount(); k+=2) {
      Component comp = parent.getComponent(k);
      Dimension d = comp.getPreferredSize();
      w = Math.max(w, d.width);
      h += d.height + m_vGap;
    }
    h -= m_vGap;

    Insets insets = parent.getInsets();
    return new Dimension(divider+w+insets.left+insets.right,
      h+insets.top+insets.bottom);
  }

  public Dimension minimumLayoutSize(Container parent) {
    return preferredLayoutSize(parent);
  }

  public void layoutContainer(Container parent) {
    int divider = getDivider(parent);

    Insets insets = parent.getInsets();
    int w = parent.getWidth() - insets.left
      - insets.right - divider;
    int x = insets.left;
    int y = insets.top;
```

```
    for (int k=1 ; k<parent.getComponentCount(); k+=2) {
      Component comp1 = parent.getComponent(k-1);
      Component comp2 = parent.getComponent(k);
      Dimension d = comp2.getPreferredSize();

      comp1.setBounds(x, y, divider-m_hGap, d.height);
      comp2.setBounds(x+divider, y, w, d.height);
      y += d.height + m_vGap;
    }
  }

  public int getHGap() { return m_hGap; }

  public int getVGap() { return m_vGap; }

  public void setDivider(int divider) {
    if (divider > 0)
      m_divider = divider;
  }

  public int getDivider() { return m_divider; }

  protected int getDivider(Container parent) {
    if (m_divider > 0)
      return m_divider;

    int divider = 0;
    for (int k=0 ; k<parent.getComponentCount(); k+=2) {
      Component comp = parent.getComponent(k);
      Dimension d = comp.getPreferredSize();
      divider = Math.max(divider, d.width);
    }
    divider += m_hGap;
    return divider;
  }

  public String toString() {
    return getClass().getName() + "[hgap=" + m_hGap + ",vgap="
      + m_vGap + ",divider=" + m_divider + "]";
  }
}
```

Understanding the Code

Class FlightReservation

This class now imports package `dl` and sets that layout for `JPanel p1r` (which contains the labels and input fields). Package `dl` contains our custom layout, `DialogLayout`.

Class DialogLayout

This class implements the `LayoutManager` interface to serve as our custom layout manager. Three instance variables are needed:

`int m_divider`: width of the left components. This can be calculated or set to some mandatory value.

`int m_hGap`: horizontal gap between components.

`int m_vGap`: vertical gap between components.

Two constructors are available to create a `DialogLayout`: a no-argument default constructor and a

constructor which takes horizontal and vertical gap sizes as parameters. The rest of the code implements methods from the `LayoutManager` interface.

Methods `addLayoutComponent()` and `removeLayoutComponent()` are not used in this class and receive empty implementations. We do not support an internal collection of the components to be managed. Rather, we refer to these component directly from the container which is being managed.

The purpose of the `preferredLayoutSize()` method is to return the preferable container size required to lay out the components in the given container according to the rules used in this layout. In our implementation we first determine the `divider` size (the width of the first column plus the horizontal gap, `m_hGap`) by calling the `getDivider()` custom method.

```
        int divider = getDivider(parent);
```

If no positive divider size has been specified using the `setDivider()` method (see below), the `getDivider()` method looks at each even indexed component in the container (this should be all the labels if the components were added to the container in the correct order) and returns the largest preferred width found plus the horizontal gap value, `m_hGap` (which defaults to 10 if the default constructor is used):

```
    if (m_divider > 0)
      return m_divider;

    int divider = 0;
    for (int k=0 ; k<parent.getComponentCount(); k+=2) {
      Component comp = parent.getComponent(k);
      Dimension d = comp.getPreferredSize();
      divider = Math.max(divider, d.width);
    }
    divider += m_hGap;
    return divider;
```

Now, back to the `preferredLayoutSize()` method. Once `getDivider` returns we then examine all components in the container with odd indices (this should be all the input fields) and determine the maximum width, `w`. This is found by checking the preferred width of each input field. While we are determining this maximum width, we are also continuing to accumulate the height, `h`, of the whole input fields column by summing each field's preferred height (not forgetting to add the vertical gap size, `m_vGap`, each time; notice that `m_vGap` is subtracted from the height at the end because there is no vertical gap for the last field). (Remember that `m_vGap` defaults to 5 if the the default constructor is used.)

```
    int w = 0;
    int h = 0;
    for (int k=1 ; k<parent.getComponentCount(); k+=2) {
      Component comp = parent.getComponent(k);
      Dimension d = comp.getPreferredSize();
      w = Math.max(w, d.width);
      h += d.height + m_vGap;
    }
    h -= m_vGap;
```

So at this point we have determined the width of the labels column (including the space between columns), `divider`, and the preferred hieght, `h`, and width, `w`, of the input fields column. So `divider+w` gives us the preferred width of the container, and `h` gives us the total preferred height. Not forgetting to take into account any `Insets` that might have been applied to the container, we can now return the correct preferred size:

```
    Insets insets = parent.getInsets();
    return new Dimension(divider+w+insets.left+insets.right,
      h+insets.top+insets.bottom);
```

The purpose of the `minimumLayoutSize()` method is to return the minimum size required to lay out the components in the given container according to the rules used in this layout. We return `preferredLayoutSize()` in this method, because we chose not to make a distinction between minimum and preferable sizes (to avoid over-complication).

`layoutContainer()` is the most important method in any layout manager. This method is responsible for actually assigning the bounds (position and size) for the components in the container being managed. First it determines the size of the `divider` (as discussed above), which represents the width of the labels column plus an additional `m_hGap`. From this it determines the width, `w`, of the fields column by subtracting the container's left and right insets and `divider` from the width of the whole container:

```
    int divider = getDivider(parent);

    Insets insets = parent.getInsets();
    int w = parent.getWidth() - insets.left
      - insets.right - divider;
    int x = insets.left;
    int y = insets.top;
```

Now all pairs of components are examined in turn. Each left component receives a width equal to `divider - m_hGap`, and all right components receive a width of `w`. Both left and right components receive the preferred height of the right component (which should be the input field).

Coordinates of the left components are assigned starting with the container's `Insets`, `x` and `y`. Notice that `y` is continually incremented based on the preferred height of each right component plus the vertical gap, `m_vGap`. The right components are assigned a y-coordinate identical to their left component counterpart, and an x-coordinate of `x+divider` (remember that `divider` includes the horizontal gap, `m_hGap`):

```
    for (int k=1 ; k<parent.getComponentCount(); k+=2) {
      Component comp1 = parent.getComponent(k-1);
      Component comp2 = parent.getComponent(k);
      Dimension d = comp2.getPreferredSize();

      comp1.setBounds(x, y, divider-m_hGap, d.height);
      comp2.setBounds(x+divider, y, w, d.height);
      y += d.height + m_vGap;
    }
```

Method `setDivider()` allows us to manually set the size of the left column. The `int` value passed as parameter gets stored in the `m_divider` instance variable. Whenever `m_divider` is greater than `0` the calculations of `divider` size are overridden in the `getDivider()` method and this value is returned instead.

The `toString` method provides typical class name and instance variable information. (It is always a good idea to implement informative `toString()` methods for each class. Although we don't consistently do this throughout this text, we feel that production code should always include this functionality.)

Running the Code

At this point you can compile and execute this example. Figure 4.16 shows the sample interface introduced in the previous section now using `DialogLayout` to manage the layout of the input fields (text field and two combo boxes) and their corresponding labels. Note that the labels occupy only their preferred space and do not resize when the frame resizes. The gap between labels and boxes can be managed easily by manually setting the `divider` size with the `setDivider()` method (discussed above). The input fields form the right column and occupy all remaining space.

Using `DialogLayout`, all that is required is adding the labels and input fields in the correct order. We can now use this layout manager each time we encounter label/input field pairs without worrying about intermediate containers. In the next section we build upon `DialogLayout` to create an even more general layout manager that can be used to create complete dialog interfaces very easily.

UI Guideline: Alignment across controls as well as within. It is a common mistake in UI Design to achieve good alignment with a control or component but fail to achieve this across a whole screen, panel or dialog. Unfortunately, the architecture of Swing lends itself to this problem. For example, if you have 4 custom components which inherit from a JPanel, each has its own LayoutManager and each is functional in its own right. Then you wish to build a composite component which requires all four. So you create a new Component with a GridLayout for example, then add each of your 4 components in turn. The result can be very messy. The fields within each component will align e.g. 3 radio buttons, but those radio buttons will not align with say 3 TextFields in the next component. Why not? The answer is simple. With Swing, there is no way for the layout manager within each component to negotiate with the others. So alignment cannot be achieved across the components. The answer to this problem is that you must flatten out the design into a single panel, as DialogLayout achieves.

4.6 Custom layout manager: part II - Common interfaces

In section 4.4 we saw how to choose intermediate containers and appropriate layouts for placing components according to a given specification. This required the use of several intermediate containers and several variants were developed in a search for the best solution. This raises the question: can we somehow just add components one after another to a container which is intelligent enough to lay them out as we would typically expect? The answer is yes, to a certain extent.

In practice the contents of many Java frames and dialogs are constructed using a scheme similar to the following (we realize that this is a big generalization, but you will see these situations arise in many examples throughout this text):

1. Groups (or panels) of controls are laid out in the vertical direction.
2. Labels and their corresponding input fields form two-column structures as described in the previous section.
3. Large components (e.g. lists, tables, text areas, trees, etc.) are usually placed in scroll panes and occupy all space in the horizontal direction.
4. Groups of buttons, including check boxes and radio buttons, are centered in an intermediate container and laid out in the horizontal direction. (In this example we purposefully avoid the vertical placement of buttons for simplicity.)

The example in this section shows how to build a layout manager that places components according to this specification. Its purpose is to further demonstrate that layout managers can be built to define template-like pluggable containers. By adhering to intelligently designed specifications, such templates can be developed to help maximize code reuse and increase productivity. Additionally, in the case of large-scale applications, several different interface designers may consider sharing customized layout managers to enforce interface consistency.

The code below introduces our new custom layout manager, `DialogLayout2`, which builds upon `DialogLayout`. To provide boundaries between control groupings, we construct a new component, `DialogSeparator`. `DialogSeparator` is simply a label containing text and a horizontal bar that is drawn across the container. Both `DialogLayout2` and `DialogSeparator` are added to our `dl` package. The

FlightReservation class now shows how to construct the sample airline ticket reservation interface we have been working with since section 4.4 using DialogLayout2 and DialogSeparator. In order to comply with our new layout scheme we are forced to place the radio buttons in a row above the list component. The main things to note are that the code involved to build this interface is done with little regard for the existence of a layout manager, and absolutely no intermediate containers are need to be created!

Note: Constructing custom layout managers for use in a single application is not recommended. Only build them when you know that they will be reused again and again to perform common layout tasks. In general, custom layout manager classes belong within custom packages or embedded as inner classes in custom components. They normally do not belong being defined in applications themselves.

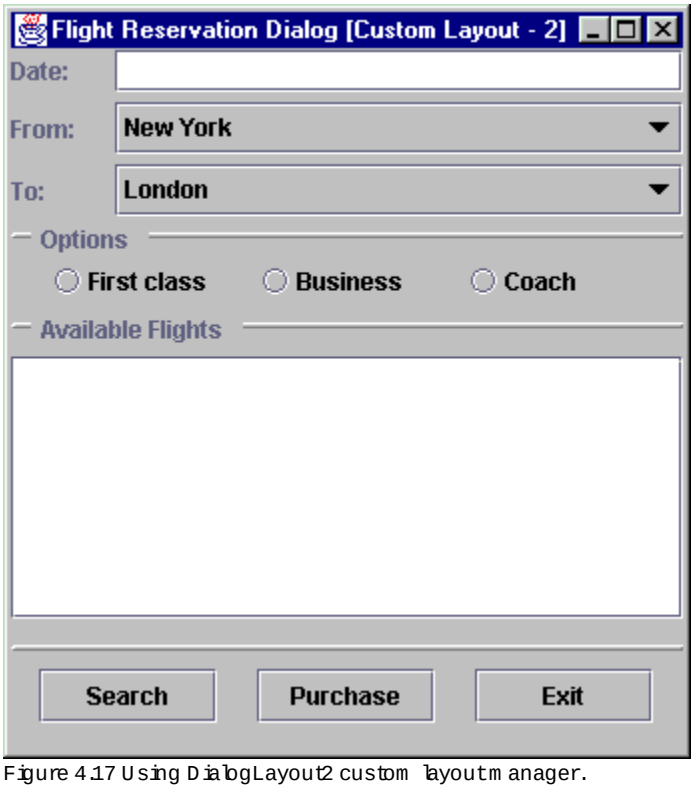


Figure 4.17 Using DialogLayout2 custom layout manager.
<<file figure4-17.gif>>

The Code: FlightReservation.java
see \Chapter4\5

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

import dl.*;

public class FlightReservation extends JFrame
{
  public FlightReservation() {
    super("Flight Reservation Dialog [Custom Layout - 2]");

    Container c = getContentPane();
    c.setLayout(new DialogLayout2(20, 5));
```

```
    c.add(new JLabel("Date:"));
    c.add(new JTextField());

    c.add(new JLabel("From:"));
    JComboBox cb1 = new JComboBox();
    cb1.addItem("New York");
    c.add(cb1);

    c.add(new JLabel("To:"));
    JComboBox cb2 = new JComboBox();
    cb2.addItem("London");
    c.add(cb2);

    c.add(new DialogSeparator("Available Flights"));
    JList list = new JList();
    JScrollPane ps = new JScrollPane(list);
    c.add(ps);

    c.add(new DialogSeparator("Options"));

    ButtonGroup group = new ButtonGroup();
    JRadioButton r1 = new JRadioButton("First class");
    group.add(r1);
    c.add(r1);

    JRadioButton r2 = new JRadioButton("Business");
    group.add(r2);
    c.add(r2);

    JRadioButton r3 = new JRadioButton("Coach");
    group.add(r3);
    c.add(r3);

    c.add(new DialogSeparator());

    JButton b1 = new JButton("Search");
    c.add(b1);

    JButton b2 = new JButton("Purchase");
    c.add(b2);

    JButton b3 = new JButton("Exit");
    c.add(b3);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public static void main(String argv[]) {
    new FlightReservation();
  }
}
```

The Code: DialogLayout2.java

see \Chapter4\5\dl

```
package dl;

import java.awt.*;
import java.util.*;

import javax.swing.*;

public class DialogLayout2 implements LayoutManager
{
  protected static final int COMP_TWO_COL = 0;
  protected static final int COMP_BIG = 1;
  protected static final int COMP_BUTTON = 2;

  protected int m_divider = -1;
  protected int m_hGap = 10;
  protected int m_vGap = 5;
  protected Vector m_v = new Vector();

  public DialogLayout2() {}

  public DialogLayout2(int hGap, int vGap) {
    m_hGap = hGap;
    m_vGap = vGap;
  }

  public void addLayoutComponent(String name, Component comp) {}

  public void removeLayoutComponent(Component comp) {}

  public Dimension preferredLayoutSize(Container parent) {
    m_v.removeAllElements();
    int w = 0;
    int h = 0;

    int type = -1;
    for (int k=0 ; k<parent.getComponentCount(); k++) {
      Component comp = parent.getComponent(k);
      int newType = getLayoutType(comp);
      if (k == 0)
        type = newType;

      if (type != newType) {
        Dimension d = preferredLayoutSize(m_v, type);
        w = Math.max(w, d.width);
        h += d.height + m_vGap;
        m_v.removeAllElements();
        type = newType;
      }

      m_v.addElement(comp);
    }

    Dimension d = preferredLayoutSize(m_v, type);
    w = Math.max(w, d.width);
    h += d.height + m_vGap;

    h -= m_vGap;

    Insets insets = parent.getInsets();
    return new Dimension(w+insets.left+insets.right,
```

```
    h+insets.top+insets.bottom);
  }

  protected Dimension preferredLayoutSize(Vector v, int type) {
    int w = 0;
    int h = 0;
    switch (type)
    {
      case COMP_TWO_COL:
        int divider = getDivider(v);
        for (int k=1 ; k<v.size(); k+=2) {
          Component comp = (Component)v.elementAt(k);
          Dimension d = comp.getPreferredSize();
          w = Math.max(w, d.width);
          h += d.height + m_vGap;
        }
        h -= m_vGap;
        return new Dimension(divider+w, h);
      case COMP_BIG:
        for (int k=0 ; k<v.size(); k++) {
          Component comp = (Component)v.elementAt(k);
          Dimension d = comp.getPreferredSize();
          w = Math.max(w, d.width);
          h += d.height + m_vGap;
        }
        h -= m_vGap;
        return new Dimension(w, h);
      case COMP_BUTTON:
        Dimension d = getMaxDimension(v);
        w = d.width + m_hGap;
        h = d.height;
        return new Dimension(w*v.size()-m_hGap, h);
    }
    throw new IllegalArgumentException("Illegal type "+type);
  }

  public Dimension minimumLayoutSize(Container parent) {
    return preferredLayoutSize(parent);
  }

  public void layoutContainer(Container parent) {
    m_v.removeAllElements();
    int type = -1;
    Insets insets = parent.getInsets();
    int w = parent.getWidth() - insets.left - insets.right;
    int x = insets.left;
    int y = insets.top;
    for (int k=0 ; k<parent.getComponentCount(); k++) {
      Component comp = parent.getComponent(k);
      int newType = getLayoutType(comp);
      if (k == 0)
        type = newType;
      if (type != newType) {
        y = layoutComponents(m_v, type, x, y, w);
        m_v.removeAllElements();
        type = newType;
      }
      m_v.addElement(comp);
    }
    y = layoutComponents(m_v, type, x, y, w);
    m_v.removeAllElements();
  }
```

```
protected int layoutComponents(Vector v, int type,
 int x, int y, int w)
{
  switch (type)
  {
    case COMP_TWO_COL:
      int divider = getDivider(v);
      for (int k=1 ; k<v.size(); k+=2) {
        Component comp1 = (Component)v.elementAt(k-1);
        Component comp2 = (Component)v.elementAt(k);
        Dimension d = comp2.getPreferredSize();
        comp1.setBounds(x, y, divider-m_hGap, d.height);
        comp2.setBounds(x+divider, y, w-divider, d.height);
        y += d.height + m_vGap;
      }
      return y;
    case COMP_BIG:
      for (int k=0 ; k<v.size(); k++) {
        Component comp = (Component)v.elementAt(k);
        Dimension d = comp.getPreferredSize();
        comp.setBounds(x, y, w, d.height);
        y += d.height + m_vGap;
      }
      return y;
    case COMP_BUTTON:
      Dimension d = getMaxDimension(v);
      int ww = d.width*v.size() + m_hGap*(v.size()-1);
      int xx = x + Math.max(0, (w - ww)/2);
      for (int k=0 ; k<v.size(); k++) {
        Component comp = (Component)v.elementAt(k);
        comp.setBounds(xx, y, d.width, d.height);
        xx += d.width + m_hGap;
      }
      return y + d.height;
  }
  throw new IllegalArgumentException("Illegal type "+type);
}

public int getHGap() { return m_hGap; }

public int getVGap() { return m_vGap; }

public void setDivider(int divider) {
  if (divider > 0)
    m_divider = divider;
}

public int getDivider() { return m_divider; }

protected int getDivider(Vector v) {
  if (m_divider > 0)
    return m_divider;
  int divider = 0;
  for (int k=0 ; k<v.size(); k+=2) {
    Component comp = (Component)v.elementAt(k);
    Dimension d = comp.getPreferredSize();
    divider = Math.max(divider, d.width);
  }
  divider += m_hGap;
  return divider;
}
```

```
  protected Dimension getMaxDimension(Vector v) {
    int w = 0;
    int h = 0;
    for (int k=0 ; k<v.size(); k++) {
      Component comp = (Component)v.elementAt(k);
      Dimension d = comp.getPreferredSize();
      w = Math.max(w, d.width);
      h = Math.max(h, d.height);
    }
    return new Dimension(w, h);
  }

  protected int getLayoutType(Component comp) {
    if (comp instanceof AbstractButton)
      return COMP_BUTTON;
    else if (comp instanceof JPanel ||
     comp instanceof JScrollPane ||
     comp instanceof DialogSeparator)
      return COMP_BIG;
    else
      return COMP_TWO_COL;
  }

  public String toString() {
    return getClass().getName() + "[hgap=" + m_hGap + ",vgap="
      + m_vGap + ",divider=" + m_divider + "]";
  }
}
```

The Code: DialogSeparator.java
see \Chapter4\5\dl

```
package dl;

import java.awt.*;

import javax.swing.*;

public class DialogSeparator extends JLabel
{
  public static final int OFFSET = 15;

  public DialogSeparator() {}

  public DialogSeparator(String text) { super(text); }

  public Dimension getPreferredSize() {
    return new Dimension(getParent().getWidth(), 20);
  }
  public Dimension getMinimumSize() { return getPreferredSize(); }
  public Dimension getMaximumSize() { return getPreferredSize(); }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(getBackground());
    g.fillRect(0, 0, getWidth(), getHeight());

    Dimension d = getSize();
    int y = (d.height-3)/2;
    g.setColor(Color.white);
```

```
    g.drawLine(1, y, d.width-1, y);
    y++;
    g.drawLine(0, y, 1, y);
    g.setColor(Color.gray);
    g.drawLine(d.width-1, y, d.width, y);
    y++;
    g.drawLine(1, y, d.width-1, y);

    String text = getText();
    if (text.length()==0)
      return;

    g.setFont(getFont());
    FontMetrics fm = g.getFontMetrics();
    y = (d.height + fm.getAscent())/2;
    int l = fm.stringWidth(text);

    g.setColor(getBackground());
    g.fillRect(OFFSET-5, 0, OFFSET+l, d.height);

    g.setColor(getForeground());
    g.drawString(text, OFFSET, y);
  }
}
```

Understanding the Code

Class FlightReservation

This variant of our airplane ticket reservation sample application uses an instance of `DialogLayout2` as a layout for the whole content pane. Note that no other `JPanel`s are used, and no other layouts are involved. All components are added directly to the content pane and managed by the new layout. This incredibly simplifies the creation of the user interface. Note, however, that we still need to add the label/input field pairs in the correct order because `DialogLayout2` manages these pairs identically to `DialogLayout`.

Note that instances of our `DialogSeparator` class are used to provide borders between groups of components.

Class DialogLayout2

This class implements the `LayoutManager` interface to serve as a custom layout manager. It builds on features from `DialogLayout` to manage all components in its associated container. Three constants declared at the top of the class correspond to the three types of components which are recognized by this layout:

- `int COMP_TWO_COL`: text fields, comboboxes, and their associated labels which must be laid out in two columns using a `DialogLayout`.
- `int COMP_BIG`: wide components (instances of `JPanel`, `JScrollPane`, or `DialogSeparator`) which must occupy the maximum horizontal container space wherever they are placed.
- `int COMP_BUTTON`: button components (instances of `AbstractButton`) which must all be given an equal size, laid out in a single row, and centered in the container.

The instance variables used in `DialogLayout2` are the same as those used in `DialogLayout` with one addition: we declare `Vector m_v` to be used as a temporary collection of components.

To lay out components in a given container we need to determine, for each component, which category it falls under with regard to our `DialogLayout2.COMP_XX` constants. All components of the same type added in a contiguous sequence must be processed according to the specific rules described above.

Method `preferredLayoutSize()` steps through the list of components in a given container and determines their type with our custom `getLayoutType()` method (see below) and stores it in the `newType` local variable. Local variable `type` holds the type of the previous component in the sequence. For the first component in the container, `type` receives the same value as `newType`.

```
  public Dimension preferredLayoutSize(Container parent) {
    m_v.removeAllElements();
    int w = 0;
    int h = 0;

    int type = -1;
    for (int k=0 ; k<parent.getComponentCount(); k++) {
      Component comp = parent.getComponent(k);
      int newType = getLayoutType(comp);
      if (k == 0)
        type = newType;
```

If we find a break in the sequence of types this triggers a call to the overloaded `preferredLayoutSize(Vector v, int type)` method (discussed below) which determines the preferred size for a temporary collection of the components stored in the `Vector m_v`. Then `w` and `h` local variables, which are accumulating the total preferred width and height for this layout, are adjusted, and the temporary collection, `m_v` is cleared. The newly processed component is then added to `m_v`.

```
      if (type != newType) {
        Dimension d = preferredLayoutSize(m_v, type);
        w = Math.max(w, d.width);
        h += d.height + m_vGap;
        m_v.removeAllElements();
        type = newType;
      }

      m_v.addElement(comp);
    }
```

Once our loop finishes we make the unconditional call to `preferredLayoutSize()` to take into account the last (unprocessed) sequence of components and update `h` and `w` accordingly (just as we did in the loop). We then subtract the vertical gap value, `m_vGap`, from `h` because we know that we have just processed the last set of components and therefore there is no vertical gap necessary. Taking into account any `Insets` set on the container, we can now return the computed preferred size as a `Dimension` instance:

```
    Dimension d = preferredLayoutSize(m_v, type);
    w = Math.max(w, d.width);
    h += d.height + m_vGap;

    h -= m_vGap;

    Insets insets = parent.getInsets();
    return new Dimension(w+insets.left+insets.right,
      h+insets.top+insets.bottom);
  }
```

The overloaded method `preferredLayoutSize(Vector v, int type)` computes the preferred size to lay out a collection of components of a given type. This size is accumulated in `w` and `h` local variables. For a collection of type `COMP_TWO_COL` this method invokes a mechanism that should be familiar (see section 4.5). For a collection of type `COMP_BIG` this method adjusts the preferable width and increments the height for each component, since these components will be placed in a column:

```
      case COMP_BIG:
```

```
        for (int k=0 ; k<v.size(); k++) {
          Component comp = (Component)v.elementAt(k);
          Dimension d = comp.getPreferredSize();
          w = Math.max(w, d.width);
          h += d.height + m_vGap;
        }
        h -= m_vGap;
        return new Dimension(w, h);
```

For a collection of type `COMP_BUTTON` this method invokes our `getMaxDimension` method (see below) to calculate the desired size of a single component. Since all components of this type will have an equal size and be contained in one single row, the resulting width for this collection is calculated through multiplication by the number of components, `v.size()`:

```
      case COMP_BUTTON:
        Dimension d = getMaxDimension(v);
        w = d.width + m_hGap;
        h = d.height;
        return new Dimension(w*v.size()-m_hGap, h);
```

Method `layoutContainer(Container parent)` assigns bounds to the components in the given container. (Remember that this is the method that actually performs the layout of its associated container.) It processes an array of components similar to the `preferredLayoutSize()` method. First it steps through the components in the given container, forms a temporarily collection from contiguous components of the same type, and calls the overloaded `layoutComponents(Vector v, int type, int x, int y, int w)` method to lay out that collection.

Method `layoutContainer(Vector v, int type, int x, int y, int w)` lays out components from the temporary collection of a given type, starting from the given coordinates `x` and `y`, and using the specified width, `w`, of the container. It returns an adjusted y-coordinate which may be used to lay out a new set of components.

For a collection of type `COMP_TWO_COL` this method lays out components in two columns identical to how `DialogLayout` did this (see section 4.5). For a collection of type `COMP_BIG` the method assigns all available width to each component:

```
      case COMP_BIG:
        for (int k=0 ; k<v.size(); k++) {
          Component comp = (Component)v.elementAt(k);
          Dimension d = comp.getPreferredSize();
          comp.setBounds(x, y, w, d.height);
          y += d.height + m_vGap;
        }
        return y;
```

For a collection of type `COMP_BUTTON` this method assigns an equal size to each component and places them in the center arranged horizontally:

```
      case COMP_BUTTON:
        Dimension d = getMaxDimension(v);
        int ww = d.width*v.size() + m_hGap*(v.size()-1);
        int xx = x + Math.max(0, (w - ww)/2);
        for (int k=0 ; k<v.size(); k++) {
          Component comp = (Component)v.elementAt(k);
          comp.setBounds(xx, y, d.width, d.height);
          xx += d.width + m_hGap;
        }
        return y + d.height;
```

Note that a more sophisticated implementation might split a sequence of buttons up into several rows if not enough space is available. We do not do that here to avoid over-complication. This might be an interesting exercise to give you more practice at customizing layout managers.

The remainder of the `DialogLayout2` class contains methods which were either explained already, or are simple enough to be considered self-explanatory.

Class DialogSeparator

This class implements a component used to separate two groups of components placed in a column. It extends `JLabel` to inherit all its default characteristics (font, foreground, etc.). Two available constructors allow the creation of a `DialogSeparator` with or without a text label.

Method `getPreferredSize` returns a fixed height, and a width equal to the width of the container. Methods `getMinimumSize()` and `getMaximumSize()` simply delegate calls to the `getPreferredSize()` method.

The `paintComponent()` method draws a separating bar with a raised appearance across the available component space, and draws the title text (if any) at the left-most side taking into account a pre-defined offset, 15.

Running the Code

At this point you can compile and execute this example. Figure 4.17 shows our sample application which now uses `DialogLayout2` to manage the layout of all components. You can see that we have the same set of components placed and sized in accordance with our general layout scheme presented in the beginning of this section. The most important thing to note is that we did not have to use any intermediate containers and layouts to achieve this: all components are added directly to the frame's content pane which is intelligently managed by `DialogLayout2`.

UI Guideline: Consistency of Button Placement

It is important to be consistent with the placement of buttons in Dialogs and Option Panes. In the example shown here, a symmetrical approach to button placement has been adopted. This is a good safe choice. It ensures balance. With Data Entry Dialogs it is also common to use an asymmetrical layout such as Bottom RHS of the dialog.

In addition to achieving balance with the layout, by being consistent with your placement you allow the User to rely on directional memory to find a specific button location. Directional Memory is strong. Once the User learns where you have placed buttons, they will quickly be able to locate the correct button in many dialog and option situations. It is therefore, vital that you place buttons in a consistent order e.g. OK, Cancel, always and never Cancel, OK.

As a general rule, always use symmetrical layout with option dialogs and be consistent with whatever you decide to use for data entry dialogs.

It makes sense to develop custom components such as JOKCancelButtons and JYesNoButtons. You can then re-use these components every time you need a such a set of buttons. This encapsulates the placement and ensures consistency.

4.7 Dynamic layout in a JavaBeans container

In this section we will use different layouts to manage JavaBeans in a simple container application. This will help us to further understand the role of layouts in dynamically managing containers with a variable number of components. This example also sets up the framework for a powerful bean editor environment developed in

chapter 18 using JTables. By allowing modification of component properties we can use this environment to experiment with preferred, maximum, and minimum sizes, and observe the behavior different layout managers exhibit in various situations. This provides us with the ability to learn much more about each layout manager, and allows us to prototype simple interfaces without actually implementing them.

This example consists of a frame container that allows the creation, loading, and saving of JavaBeans using serialization. Beans can be added and removed from this container and we implement a focus mechanism to visually identify the currently selected bean. Most importantly, the layout manager of this container can be changed at run-time. (You may want to review the JavaBeans material in chapter 2 before attempting to work through this example.)

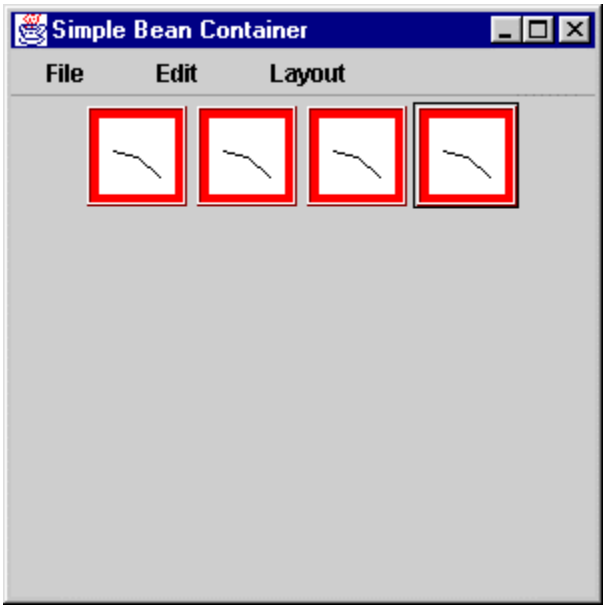


Figure 4.18 BeanContainer displaying 4 Clock components using a FlowLayout
<<file figure4-18.gif>>

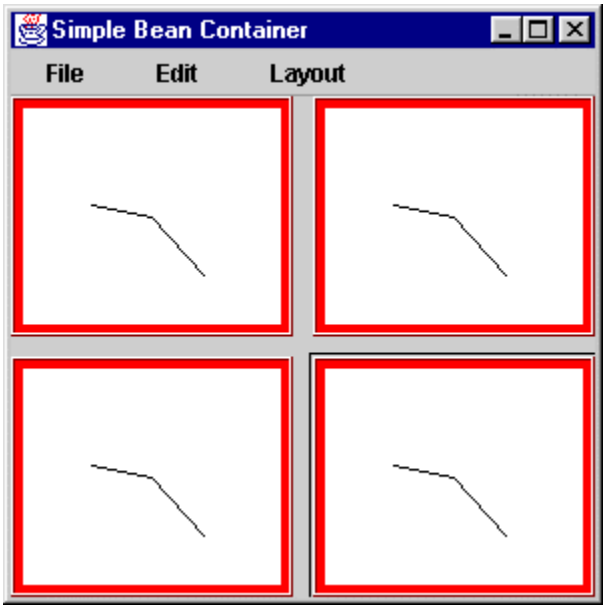


Figure 4.19 BeanContainer displaying 4 Clock components using a GridLayout
<<file figure4-19.gif>>

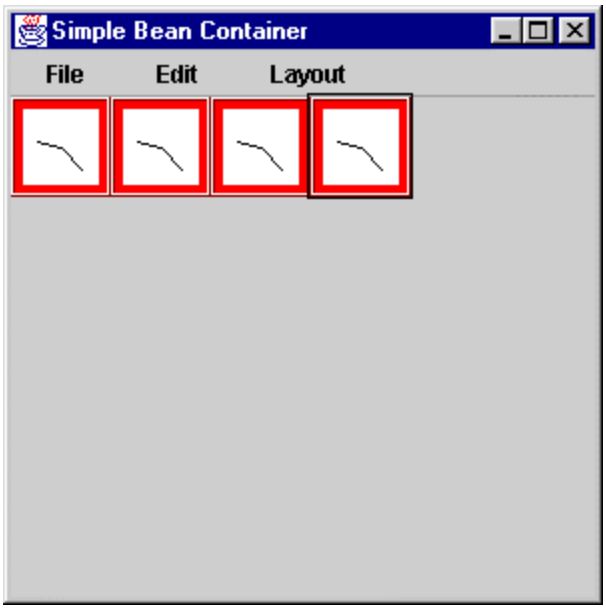


Figure 4.20 BeanContainer displaying 4 Clock components using a horizontalBoxLayout

<<file figure4-20.gif>>

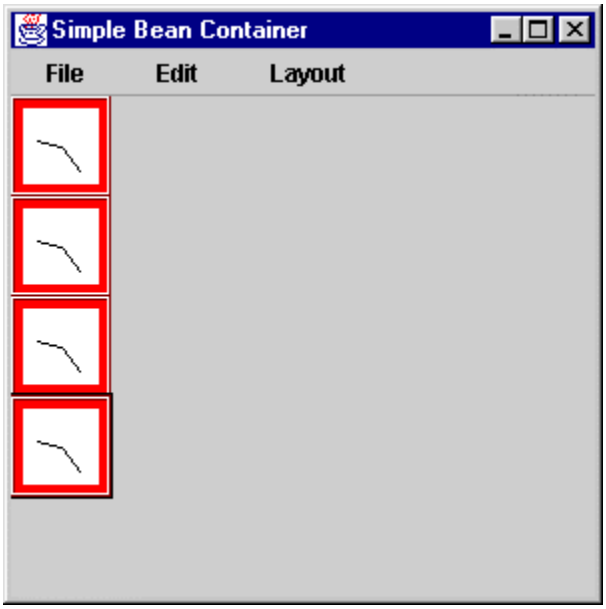


Figure 4.21 BeanContainer displaying 4 Clock components using a verticalBoxLayout

<<file figure4-21.gif>>

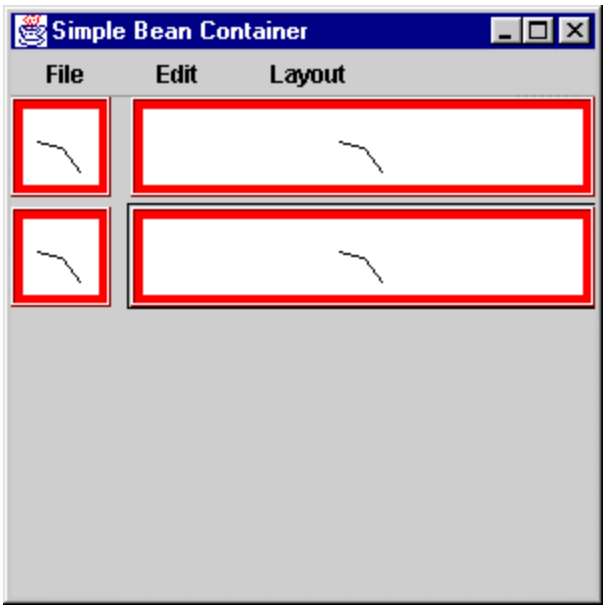


Figure 4.22 BeanContainer displaying 4 Clock components using a DialogLayout

<<file figure4-22.gif>>

Figure 4.23 BeanContainer displaying button/input field pairs using DialogLayout

<<file figure4-23.gif>>

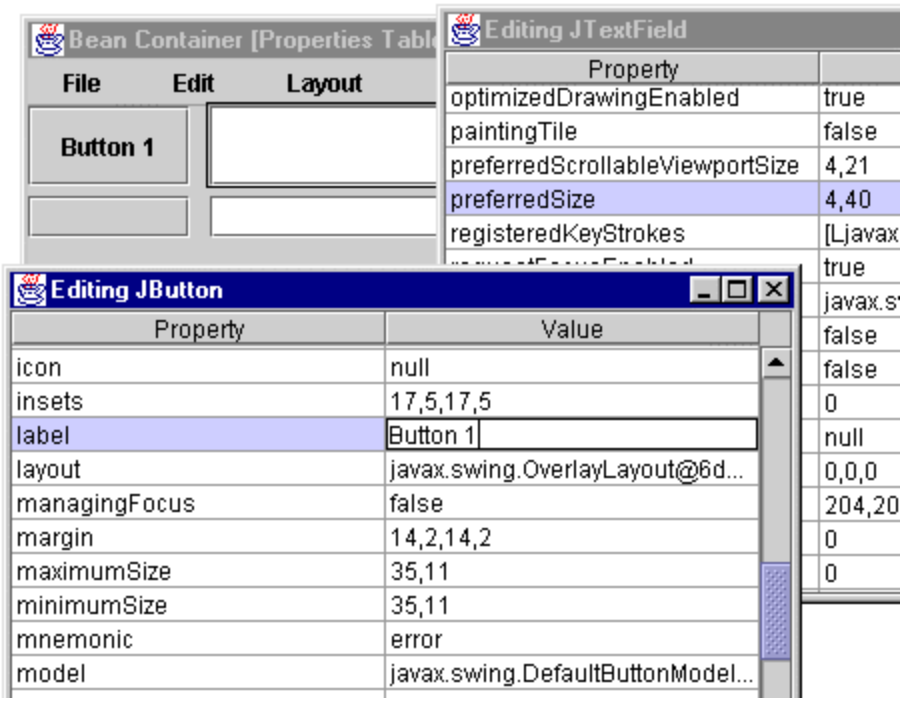


Figure 4.24 BeanContainer displaying button/input field pairs using DialogLayout
<<file figure4-24.gif>>

The Code: BeanContainer.java
see \Chapter4\6

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.beans.*;
import java.lang.reflect.*;

import javax.swing.*;

import dl.*;

public class BeanContainer extends JFrame implements FocusListener
{
  protected File m_currentDir = new File(".");
  protected Component m_activeBean;
  protected String m_className = "clock.Clock";
  protected JFileChooser m_chooser = new JFileChooser();

  public BeanContainer() {
    super("Simple Bean Container");
    getContentPane().setLayout(new FlowLayout());

    setSize(300, 300);

    JPopupMenu.setDefaultLightWeightPopupEnabled(false);

    JMenuBar menuBar = createMenuBar();
    setJMenuBar(menuBar);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
```

```
    }
  };
  addWindowListener(wndCloser);

  setVisible(true);
}

protected JMenuBar createMenuBar() {
  JMenuBar menuBar = new JMenuBar();

  JMenu mFile = new JMenu("File");

  JMenuItem mItem = new JMenuItem("New...");
  ActionListener lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      Thread newthread = new Thread() {
        public void run() {
          String result = (String)JOptionPane.showInputDialog(
            BeanContainer.this,
            "Please enter class name to create a new bean",
            "Input", JOptionPane.INFORMATION_MESSAGE, null,
            null, m_className);
          repaint();
          if (result==null)
            return;
          try {
            m_className = result;
            Class cls = Class.forName(result);
            Object obj = cls.newInstance();
            if (obj instanceof Component) {
              m_activeBean = (Component)obj;
              m_activeBean.addFocusListener(
                BeanContainer.this);
              m_activeBean.requestFocus();
              getContentPane().add(m_activeBean);
            }
            validate();
          }
          catch (Exception ex) {
            ex.printStackTrace();
            JOptionPane.showMessageDialog(
              BeanContainer.this, "Error: "+ex.toString(),
              "Warning", JOptionPane.WARNING_MESSAGE);
          }
        }
      };
      newthread.start();
    }
  };
  mItem.addActionListener(lst);
  mFile.add(mItem);

  mItem = new JMenuItem("Load...");
  lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      Thread newthread = new Thread() {
        public void run() {
          m_chooser.setCurrentDirectory(m_currentDir);
          m_chooser.setDialogTitle(
            "Please select file with serialized bean");
          int result = m_chooser.showOpenDialog(
            BeanContainer.this);
```

```
            repaint();
            if (result != JFileChooser.APPROVE_OPTION)
              return;
            m_currentDir = m_chooser.getCurrentDirectory();
            File fChoosen = m_chooser.getSelectedFile();
            try {
              FileInputStream fStream =
                new FileInputStream(fChoosen);
              ObjectInput  stream  =
                new ObjectInputStream(fStream);
              Object obj = stream.readObject();
              if (obj instanceof Component) {
                m_activeBean = (Component)obj;
                m_activeBean.addFocusListener(
                  BeanContainer.this);
                m_activeBean.requestFocus();
                getContentPane().add(m_activeBean);
              }
              stream.close();
              fStream.close();
              validate();
            }
            catch (Exception ex) {
              ex.printStackTrace();
              JOptionPane.showMessageDialog(
                BeanContainer.this, "Error: "+ex.toString(),
                "Warning", JOptionPane.WARNING_MESSAGE);
            }
            repaint();
          }
        };
        newthread.start();
      }
    };
    mItem.addActionListener(lst);
    mFile.add(mItem);

    mItem = new JMenuItem("Save...");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Thread newthread = new Thread() {
          public void run() {
            if (m_activeBean == null)
              return;
            m_chooser.setDialogTitle(
              "Please choose file to serialize bean");
            m_chooser.setCurrentDirectory(m_currentDir);
            int result = m_chooser.showSaveDialog(
              BeanContainer.this);
            repaint();
            if (result != JFileChooser.APPROVE_OPTION)
              return;
            m_currentDir = m_chooser.getCurrentDirectory();
            File fChoosen = m_chooser.getSelectedFile();
            try {
              FileOutputStream fStream =
                new FileOutputStream(fChoosen);
              ObjectOutput stream  =
                new ObjectOutputStream(fStream);
              stream.writeObject(m_activeBean);
              stream.close();
              fStream.close();
```

```
            }
            catch (Exception ex) {
              ex.printStackTrace();
            JOptionPane.showMessageDialog(
              BeanContainer.this, "Error: "+ex.toString(),
              "Warning", JOptionPane.WARNING_MESSAGE);
            }
          }
        };
        newthread.start();
      }
    };
    mItem.addActionListener(lst);
    mFile.add(mItem);

    mFile.addSeparator();

    mItem = new JMenuItem("Exit");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    };
    mItem.addActionListener(lst);
    mFile.add(mItem);
    menuBar.add(mFile);

    JMenu mEdit = new JMenu("Edit");

    mItem = new JMenuItem("Delete");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if (m_activeBean == null)
          return;
        getContentPane().remove(m_activeBean);
        m_activeBean = null;
        validate();
        repaint();
      }
    };
    mItem.addActionListener(lst);
    mEdit.add(mItem);
    menuBar.add(mEdit);

    JMenu mLayout = new JMenu("Layout");
    ButtonGroup group = new ButtonGroup();

    mItem = new JRadioButtonMenuItem("FlowLayout");
    mItem.setSelected(true);
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e){
        getContentPane().setLayout(new FlowLayout());
        validate();
        repaint();
      }
    };
    mItem.addActionListener(lst);
    group.add(mItem);
    mLayout.add(mItem);

    mItem = new JRadioButtonMenuItem("GridLayout");
    lst = new ActionListener() {
```

```
      public void actionPerformed(ActionEvent e){
        int col = 3;
        int row = (int)Math.ceil(getContentPane().
          getComponentCount()/(double)col);
        getContentPane().setLayout(new GridLayout(row, col, 10, 10));
        validate();
        repaint();
      }
    };
    mItem.addActionListener(lst);
    group.add(mItem);
    mLayout.add(mItem);

    mItem = new JRadioButtonMenuItem("BoxLayout - X");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        getContentPane().setLayout(new BoxLayout(
          getContentPane(), BoxLayout.X_AXIS));
        validate();
        repaint();
      }
    };
    mItem.addActionListener(lst);
    group.add(mItem);
    mLayout.add(mItem);

    mItem = new JRadioButtonMenuItem("BoxLayout - Y");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        getContentPane().setLayout(new BoxLayout(
          getContentPane(), BoxLayout.Y_AXIS));
        validate();
        repaint();
      }
    };
    mItem.addActionListener(lst);
    group.add(mItem);
    mLayout.add(mItem);

    mItem = new JRadioButtonMenuItem("DialogLayout");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        getContentPane().setLayout(new DialogLayout());
        validate();
        repaint();
      }
    };
    mItem.addActionListener(lst);
    group.add(mItem);
    mLayout.add(mItem);

    menuBar.add(mLayout);

    return menuBar;
  }

  public void focusGained(FocusEvent e) {
    m_activeBean = e.getComponent();
    repaint();
  }

  public void focusLost(FocusEvent e) {}
```

```
  // This is a heavyweight component so we override paint
  // instead of paintComponent. super.paint(g) will
  // paint all child components first, and then we
  // simply draw over top of them.
  public void paint(Graphics g) {
    super.paint(g);

    if (m_activeBean == null)
      return;

    Point pt = getLocationOnScreen();
    Point pt1 = m_activeBean.getLocationOnScreen();
    int x = pt1.x - pt.x - 2;
    int y = pt1.y - pt.y - 2;
    int w = m_activeBean.getWidth() + 2;
    int h = m_activeBean.getHeight() + 2;

    g.setColor(Color.black);
    g.drawRect(x, y, w, h);
  }

  public static void main(String argv[]) {
    new BeanContainer();
  }
}

The Code: Clock.java
see \Chapter4\6\clock

package clock;

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;

public class Clock extends JButton
 implements Customizer, Externalizable, Runnable
{
  protected PropertyChangeSupport m_helper;
  protected boolean  m_digital = false;
  protected Calendar m_calendar;
  protected Dimension m_preffSize;

  public Clock() {
    m_calendar = Calendar.getInstance();
    m_helper = new PropertyChangeSupport(this);

    Border br1 = new EtchedBorder(EtchedBorder.RAISED,
      Color.white, new Color(128, 0, 0));
    Border br2 = new MatteBorder(4, 4, 4, 4, Color.red);
    setBorder(new CompoundBorder(br1, br2));

    setBackground(Color.white);
    setForeground(Color.black);
```

```
    (new Thread(this)).start();
  }

  public void writeExternal(ObjectOutput out) throws IOException {
    out.writeBoolean(m_digital);
    out.writeObject(getBackground());
    out.writeObject(getForeground());
    out.writeObject(getPreferredSize());
  }

  public void readExternal(ObjectInput in)
   throws IOException, ClassNotFoundException {
    setDigital(in.readBoolean());
    setBackground((Color)in.readObject());
    setForeground((Color)in.readObject());
    setPreferredSize((Dimension)in.readObject());
  }

  public Dimension getPreferredSize() {
    if (m_preffSize != null)
      return m_preffSize;
    else
      return new Dimension(50, 50);
  }

  public void setPreferredSize(Dimension preffSize) {
    m_preffSize = preffSize;
  }

  public Dimension getMinimumSize() {
    return getPreferredSize();
  }

  public Dimension getMaximumSize() {
    return getPreferredSize();
  }

  public void setDigital(boolean digital) {
    m_helper.firePropertyChange("digital",
      new Boolean(m_digital),
      new Boolean(digital));
    m_digital = digital;
    repaint();
  }

  public boolean getDigital() { return m_digital; }

  public void addPropertyChangeListener(
   PropertyChangeListener lst) {
    if (m_helper != null)
      m_helper.addPropertyChangeListener(lst);
  }

  public void removePropertyChangeListener(
   PropertyChangeListener lst) {
    if (m_helper != null)
      m_helper.removePropertyChangeListener(lst);
  }

  public void setObject(Object bean) {}

  public void paintComponent(Graphics g) {
```

```
    super.paintComponent(g);

    g.setColor(getBackground());
    g.fillRect(0, 0, getWidth(), getHeight());
    getBorder().paintBorder(this, g, 0, 0, getWidth(), getHeight());

    m_calendar.setTime(new Date());  // get current time
    int hrs = m_calendar.get(Calendar.HOUR_OF_DAY);
    int min = m_calendar.get(Calendar.MINUTE);

    g.setColor(getForeground());
    if (m_digital) {
      String time = ""+hrs+":"+min;
      g.setFont(getFont());
      FontMetrics fm = g.getFontMetrics();
      int y = (getHeight() + fm.getAscent())/2;
      int x = (getWidth() - fm.stringWidth(time))/2;
      g.drawString(time, x, y);
    }
    else {
      int x = getWidth()/2;
      int y = getHeight()/2;
      int rh = getHeight()/4;
      int rm = getHeight()/3;

      double ah = ((double)hrs+min/60.0)/6.0*Math.PI;
      double am = min/30.0*Math.PI;

      g.drawLine(x, y, (int)(x+rh*Math.sin(ah)),
        (int)(y-rh*Math.cos(ah)));
      g.drawLine(x, y, (int)(x+rm*Math.sin(am)),
        (int)(y-rm*Math.cos(am)));
    }

    g.setColor(colorRetainer);
  }

  public void run() {
    while (true) {
      repaint();
      try {
        Thread.sleep(30*1000);
      }
      catch(InterruptedException ex) { break; }
    }
  }
}
```

Understanding the Code

Class BeanContainer

This class extends `JFrame` to provide the frame for this application. It also implements the `FocusListener` interface to manage focus transfer between beans in the container. This class declares four instance variables:

`File m_currentDir`: the most recent directory used to load and save beans.

`Component m_activeBean`: a bean component which currently has the focus.

`String m_className`: fully qualified class name of our custom `Clock` bean,

`JFileChooser m_chooser`: used for saving and loading beans.

The only GUI provided by the container itself is the menu bar. The `createMenuBar()` method creates the menu bar, its items, and their corresponding action listeners. Three menus are added to the menu bar: "File," "Edit," and "Layout."

Note: All code corresponding to "New ...," "Load ..., and "Save..." in the "File" menu is wrapped in a separate thread to avoid unnecesary load on the event-dispatching thread. See chapter 2 for more about multithreading.

Menu item "New ..." in the "File" menu displays an input dialog (using the `JOptionPane.showInputDialog()` method) to enter the class name of a new bean to be added to the container. Once a name has been entered, the program attempts to load that class, create a new class instance using a default constructor, and add that new object to the container. The newly created component requests the focus and receives a `this` reference to `BeanContainer` as a `FocusListener`. Note that any caught exceptions will be displayed in a message box.

Menu item "Load..." from the "File" menu displays a `JFileChooser` dialog to select a file containing a previously serialized bean component. If this succeeds, the program opens an input stream on this file and reads the first stored object. If this object is derived from the `java.awt.Component` class it is added to the container. The loaded component requests the focus and receives a `this` reference to `BeanContainer` as a `FocusListener`. Note that any caught exceptions will be displayed in a message box.

Menu item "Save..." from the "File" menu displays a `JFileChooser` dialog to select a file destination for serializing the bean component which currently has the focus. If this succeeds, the program opens an output stream on that file and writes the currently active component to that stream. Note that any caught exceptions will be displayed in a message box.

Menu item "Exit" simply quits and closes the application with `System.exit(0)`.

The "Edit" menu contains a single item titled "Delete" which removes the currently active bean from the container:

```
getContentPane().remove(m_activeBean);
m_activeBean = null;
validate();
repaint();
```

Menu "Layout" contains several `JRadioButtonMenuItems` managed with a `ButtonGroup group`. These items are titled "FlowLayout," "GridLayout," "BoxLayout-X," "BoxLayout-Y," and "DialogLayout." Each item receives an `ActionListener` which sets the corresponding layout manager of the application frame's content pane, calls `validate` to lay out the container again, and the `repaints` it. For example:

```
getContentPane().setLayout(new DialogLayout());
validate();
repaint();
```

Method `focusGained` stores a reference to the component which currently has the focus into instance variable `m_activebean`. Method `paint()` is implemented to draw a rectangle around the component which currently has the focus. It is important to note here the static `JPopupMenu` method called in the `BeanContainer` constructor:

```
JPopupMenu.setDefaultLightWeightPopupEnabled(false);
```

This method forces all popup menus (which menu bars use to display their contents) to use heavyweight popups rather than lightweight popups. By default popup menus are lightweight unless they cannot fit within their parent container's bounds. The reason we disable this is because our `paint()` method will render the

bean selection rectangle over top of the lightweight popups otherwise.

Class Clock

This class is a sample bean clock component which can be used in a bean container just as any other bean. This class extends the `JButton` component to inherit it's focus grabbing functionality. This class also implements three interfaces: `Customizer` to handle property listeners, `Externalizable` to completely manage its own serialization, and `Runnable` to be run by a thread. Four instance variables are declared:

`PropertyChangeSupport m_helper`: an object to manage `PropertyChangeListeners`.

`boolean m_digital`: a custom property for this component which manages the display state of the clock (digital or arrow-based).

`Calendar m_calendar`: helper object to handle Java's time objects (instances of `Date`).

`Dimension m_preffSize`: a preferred size for this component which may be set using the `setPreferredSize` method.

The constructor of the `Clock` class creates the helper objects and sets the border for this component as a `CompoundBorder` containing an `EtchedBorder` and a `MatteBorder` imitating the border of a real clock. It then sets the background and foreground colors and starts a new `Thread` to run the clock.

Method `writeExternal()` writes the current state of a `Clock` object into an `ObjectOutput` stream. Four properties are written: `m_digital`, `Background`, `Foreground`, and `PreferredSize`. Method `readExternal()` reads the previously saved state of a `Clock` object from an `ObjectInput` stream. It reads these four properties and applies them to the object previously created with the default constructor. These methods are called from the "Save" and "Load" menu bar action listener code in `BeanContainer`. Specifically, they are called when `writeObject` and `readObject` are invoked.

Note: The serialization mechanism in Swing has not yet matured. You can easily find that both lightweight and heavyweight components throw exceptions during the process of serialization. This is the reason we implement the `Externalizable` interface to take complete control over the serialization of the `Clock` bean. Another reason is that the default serialization mechanism tends to serialize a substantial amount of unnecessary information, whereas our custom implementation stores only the necessities.

The rest of this class need not be explained here, as it does not relate directly to the topic of this chapter and represents a simple example of a bean component. If you're interested, take note of the `paintComponent()` method which, depending on whether the clock is in digital mode or not (determined by `m_digital`), either computes the current position of the clock's arrows and draws them, or renders the time as a drawn `String`.

Running the Code

This application provides a framework for experimenting with any available JavaBeans, as well as with both lightweight (Swing) and heavyweight (AWT) components: we can create, serialize, delete, and restore them.

Note that we can apply several layouts to manage these components dynamically. Figures 4.18-4.23 show `BeanContainer` using five different layout managers to arrange four `Clock` beans. To create a bean choose "New" from the "File" menu and type the fully qualifies name of the class. For instance, to create a `Clock` you need to type "clock.Clock" in the input dialog.

Once you've experimented with `Clock` beans try loading some Swing JavaBeans. Figure 4.24 shows `BeanDialog` with two `JButtons`, and two `JTextFields`. They were created in the following order (and thus have corresponding container indices): `JButton`, `JTextField`, `JButton`, `JTextField`. Try doing this and remember that you need to specify fully qualified class names such as "javax.swing.JButton" when

adding a new bean. Note that this ordering adhere's to our `DialogLayout` label/input field pairs scheme, except that here we are using buttons in place of labels. So when we set `BeanContainer`'s layout to `DialogLayout` we know what to expect.

Note: You will notice selection problems with components such as `JComboBox`, `JSplitPane` and `JLabel` (which has no selection mechanism). Because of `JComboBox` is actually a container containing a button, it is impossible to give it the current focus after it has been added to `BeanContainer`. A more complete version of `BeanContainer` would take this into account and implement more robust focus requesting behavior.

Later in this book, after a discussion of tables, we add powerful functionality to this example allowing the manipulation of bean properties. It is highly suggested that you skip ahead for a moment and run this example: (see \Chapter18\8).

Start the chapter 18 example and create `JButton` and `JTextField` beans exactly as you did above. Select `DialogLayout` from the "Layout" menu and then click on the top-most JButton to give it the focus. Now select "Properties" from the "Edit" menu. A separate frame will pop up with a `JTable` containing all of the `JButton`'s properties. Navigate to the "label" property and change it to "Button 1" (by double clicking on its "Value" field). Now select the corresponding top-most `JTextField` and change its "preferredSize" property to "4,40". Figure 4.24 illustrates what you should see.

By changing the preferred, maximum, and minimum sizes, as well as other component properties, we can directly examine the behavior different layout managers impose on our container. Experimenting with this example is a very convenient way to learn more about how the layout managers behave. It also forms the foundation for an interface development environment (IDE), which many developers use to simplify interface design.

Chapter 5. Labels and Buttons

In this chapter:

- Labels and buttons overview
- Custom buttons: part I - Transparent buttons
- Custom buttons: part II - Polygonal buttons
- Custom buttons: part III - Tooltip management

5.1 Labels and buttons overview

5.1.1 JLabel

class javax.swing.JLabel

`JLabel` is one of the simplest Swing components, and is most often used to identify other components. `JLabel` can display text, an icon, or both in any combination of positions (note that text will always overlap the icon). The following code creates four different `JLabels` and places them in a `GridLayout`, and figure 5.1 illustrates.

Figure 5.1 JLabeldemo
<<file figure5-1.gif>>

The Code: LabelDemo.java
see \Chapter5\6

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

class LabelDemo extends JFrame
{
  public LabelDemo() {
    super("JLabel Demo");
    setSize(600, 100);

    JPanel content = (JPanel) getContentPane();
    content.setLayout(new GridLayout(1, 4, 4, 4));

    JLabel label = new JLabel();
    label.setText("JLabel");
    label.setBackground(Color.white);
    content.add(label);

    label = new JLabel("JLabel",
      SwingConstants.CENTER);
    label.setOpaque(true);
    label.setBackground(Color.white);
    content.add(label);

    label = new JLabel("JLabel");
    label.setFont(new Font("Helvetica", Font.BOLD, 18));
    label.setOpaque(true);
    label.setBackground(Color.white);
    content.add(label);

    ImageIcon image = new ImageIcon("flight.gif");
    label = new JLabel("JLabel", image,
      SwingConstants.RIGHT);
    label.setVerticalTextPosition(SwingConstants.TOP);
    label.setOpaque(true);
    label.setBackground(Color.white);
    content.add(label);

    setVisible(true);
  }

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

The first label is created with the default constructor and its text is set using the `setText()` method. We then set its background to white, but when we run this program the background of this label shows up as light gray!

The reason this happens is because we didn't force this label to be opaque. In chapter 2 we learned that Swing components support transparency, which means that a component does not have to paint every pixel within its bounds. So when a component is not opaque it will not fill its background, and a `JLabel` (as with most components) is non-opaque by default.

Also note that we can set the font and foreground color of a `JLabel` using `JComponent`'s `setFont()` and `setForeground()` methods. Refer back to chapter 2 for information about working the the `Font` and `Color` classes.

The default horizontal alignment of `JLabel` is `LEFT` if only text is used, and `CENTER` if an image or an image and text are used. An image will appear to the left of the text by default, and every `JLabel` is initialized with a centered vertical alignment. Each of these default behaviors can easily be adjusted as we will see below.

5.1.2 Text Alignment

To specify alignment or position in many Swing components we make us of the `javax.swing.SwingConstants` interface. This defines several constant strings, five of which are applicable to `JLabel`'s text alignment settings:

```
SwingConstants.LEFT
SwingConstants.CENTER
SwingConstants.RIGHT
SwingConstants.TOP
SwingConstants.BOTTOM
```

Alignment of both a label's text and icon can be specified in the constructor or through the use of `setHorizontalAlignment()` and `setVerticalAlignment()` methods. The text can be aligned both vertically or horizontally, independent of the icon (text will overlap the icon when necessary) using the `setHorizontalTextAlignment()` and `setVerticalTextAlignment()` methods. Figure 5.2 shows where a `JLabel`'s text will be placed corresponding to each possible combination of vertical and horizontal text alignment settings.

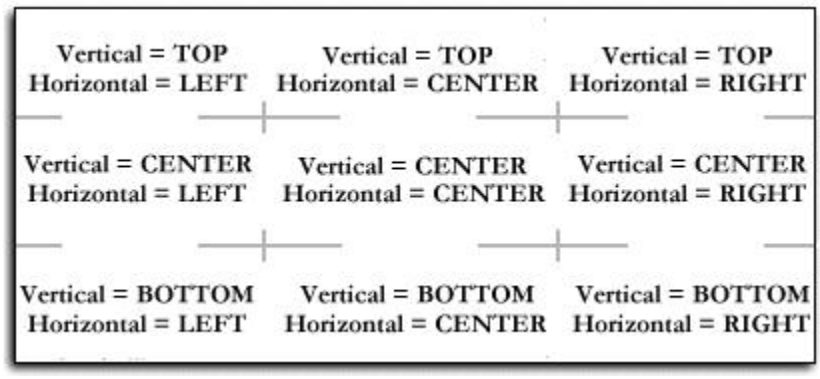


Figure 5.2 `JLabel` text alignment
<<file figure5-2.gif>>

5.1.3 Icons and Icon Alignment

The simple example above included a label with an image of an airplane by reading a GIF file in as an `ImageIcon` and passing it to a `JLabel` constructor:

```
    ImageIcon image = new ImageIcon("flight.gif");
    label = new JLabel("JLabel", image,
      SwingConstants.RIGHT);
```

An image can also be set or replaced at any time using the `setIcon()` method (passing `null` will remove the current icon, if any). `JLabel` also supports a disabled icon for use when a label is in the disabled state. To assign a disabled icon we use the `setDisabledIcon()` method.

Note: Animated GIFs can be used with `ImageIcon`'s and labels just as any static GIF, and require no additional code. `ImageIcon` also supports JPGs.

5.1.4 GrayFilter

class javax.swing.GrayFilter

The `GrayFilter` class can be used to create disabled images using its static `createDisabledImage()` method:

```
    ImageIcon disabledImage = new ImageIcon(
      GrayFilter.createDisabledImage(image.getImage()));
```

Figure 5.3 shows the fourth label in `LabelDemo` now using a disabled icon generated by `GrayFilter`. Note that `JLabel` only displays the disabled icon when it has been disabled using `JComponent`'s `setEnabled()` method.

Figure 5.3 Demonstrating a disabled icon using GrayFilter
<<file figure5-3.gif>>

5.1.5 The labelFor and the displayedMnemonic properties

`JLabel` maintains a `labelFor` property and a `displayedMnemonic` property. The displayed mnemonic is a character that, when pressed in syncronization with `ALT` (e.g. `ALT+R`), will call `JComponent`'s `requestFocus()` method on the component referenced by the `labelFor` property. The first instance of displayed mnemonic character (if any) in a label's text will be underlined. We can access these properties using typical get/set accessors (see API docs).

5.1.6 AbstractButton

abstract class javax.swing.AbstractButton

`AbstractButton` is the template class from which all buttons are defined. This includes push buttons, toggle buttons, check boxes, radio buttons, menu items, and menus themselves. Its direct subclasses are `JButton`, `JToggleButton`, and `JMenuItem`. There are no subclasses of `JButton` in Swing.

JToggleButton has two subclasses: JCheckBox, JRadioButton. JMenuItem has three subclasses: JCheckBoxMenuItem, JRadioButtonMenuItem, and JMenu. The remainder of this chapter will focus on JButton and the JToggleButton family. (See chapter 12 for more about menus and menu items).

5.1.7 The ButtonModel interface

abstract interface javax.swing.ButtonModel

Each button class uses a model to store its state. We can access any button's model with the AbstractButton getModel() and setModel() methods. The ButtonModel interface is the template interface from which all button models are defined. JButton uses the DefaultButtonModel implementation. JToggleButton defines an inner class extension of DefaultButtonModel, JToggleButton.ToggleButtonModel, which is used by itself, as well as both its subclasses.

The following boolean property values represent the state of a button, and have associated isXX() and setXX() accessors in DefaultButtonModel:

- selected: Switches state on each click (only relevant for JToggleButtons).
- pressed: True when the button is held down with the mouse.
- rollover: True when the mouse is hovering over the button.
- armed: This state stops events from being fired when we press a button with the mouse, and then release the mouse when its cursor is outside that button's bounds.
- enabled: True when the button is active. None of the other properties can normally be changed when this is false.

A button's keyboard mnemonic is also stored in its model, as well as the ButtonGroup it belongs to, if any. (We'll discuss the ButtonGroup class while we discuss JToggleButtons, as it only applies to this family of buttons.)

5.1.8 JButton

class javax.swing.JButton

JButton is a basic push-button, one of the simplest Swing components. Almost everything we know about JLabel also applies to JButton. We can add images, specify text and image alignment, set foreground and background colors (remember to call setOpaque(true)), set fonts, etc. Additionally, we can add ActionListeners, ChangeListeners, and ItemListeners to receive ActionEvents, ChangeEvents, and ItemEvents respectively when any properties in its model change value. We will see how to build a custom, non-rectangular subclass of JButton in the chapter examples.

In most application dialogs we expect to find a button which initially has the focus and will capture an Enter key press, regardless of the current keyboard focus (unless focus is within a multi-line text component). This is referred to as the default button. Any JRootPane container can define a default button using JRootPane's setDefaultButton() method (passing null will disable this feature). For instance, to make a button the default button for a JFrame we would do the following:

```
myJFrame.getRootPane().setDefaultButton(myButton);
```

The `isDefaultButton()` method returns a boolean value indicating whether or not the button instance it was called on is a default button for a `JRootPane`.

We most often register an `ActionListener` with a button to receive `ActionEvents` from that button whenever it is clicked (Note that if a button has the focus, pressing the space-bar will also fire an `ActionEvent`). `ActionEvents` carry with them information about the event that occurred including, most importantly, which component they came from. (Note that an `ActionListener` can be registered to receive events from any number of components. See chapter 2 for an overview of event handling.)

To create an `ActionListener` we need to create a class that implements the `ActionListener` interface, which requires the definition of its `actionPerformed()` method. Once we have built an `ActionListener` we can register it with a button using the `JComponent`'s `addActionListener()` method. The following is a typical inner class implementation. When an `ActionEvent` is intercepted, "Swing is powerful!" is printed to standard output.

```
JButton myButton = new JButton();
ActionListener act = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Swing is powerful!!");
  }
};
myButton.addActionListener(act);
```

We primarily use this method throughout this book to attach listeners to components. However, some developers prefer to implement the `ActionListener` interface in the class that owns the button instance. In the case of classes with several registered components, this is not as efficient as using a separate listener class and can require writing common code in several places. Specifically, implementing `ActionListener` for use with multiple event sources requires us to check the source of each `ActionEvent` (using `ActionEvent`'s `getSource()` method) before processing.

An icon can be assigned to a `JButton` instance via the constructor or with the `setIcon()` method. We can optionally assign individual icons for the normal, selected, pressed, rollover, and disabled states. See the API docs for more detail on the following methods:

```
setDisabledSelectedIcon()
setPressedIcon()
setRolloverIcon()
setRolloverSelectedIcon()
setSelectedIcon()
```

A button can also be disabled and enabled the same way as a `JLabel`, using `setEnabled()`. As we would expect, a disabled button will not respond to any user actions.

A button's keyboard mnemonic provides an alternative way of activation. To add a keyboard mnemonic to a button we use the `setMnemonic()` method:

```
button.setMnemonic('R');
```

We can then activate a button (equivalent to clicking it) by pressing ALT and its mnemonic key simultaneously. (e.g. ALT+R) The first appearance of the assigned mnemonic character, if any, in the button

text will be underlined to indicate which key activates it. There is no distinction made between upper and lower case characters. Avoid duplicating mnemonics for components sharing a common ancestor.

5.1.9 JToggleButton

class javax.swing.JToggleButton

`JToggleButton` provides a selected state mechanism which extends to its children, `JCheckBox` and `JRadioButton`, and corresponds to the `selected` property we discussed in section 5.2. We can test whether a toggle button is selected using `AbstractButton`'s `isSelected()` method, and we can set this property with its `setSelected()` method.

5.1.10 ButtonGroup

class javax.swing.ButtonGroup

`JToggleButtons` are often used in `ButtonGroups`. A `ButtonGroup` manages a set of buttons by guaranteeing that only one button within that group can be selected at any given time. Thus, only `JToggleButton` and its subclasses are useful in a `ButtonGroup` (because a `JButton` does not maintain a selected state). The following code constructs three `JToggleButtons` and places them in a single `ButtonGroup`.

Figure 5.4 JToggleButtons in a ButtonGroup

<<file figure5-4.gif>>

The Code: ToggleButtonDemo.java

see \Chapter5\6

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

class ToggleButtonDemo extends JFrame
{
  public ToggleButtonDemo () {
    super("ToggleButton/ButtonGroup Demo");
    getContentPane().setLayout(new FlowLayout());

    JToggleButton button1 = new JToggleButton("Button 1",true);
    getContentPane().add(button1);
    JToggleButton button2 = new JToggleButton("Button 2",false);
    getContentPane().add(button2);
    JToggleButton button3 = new JToggleButton("Button 3",false);
    getContentPane().add(button3);

    ButtonGroup buttonGroup = new ButtonGroup();
    buttonGroup.add(button1);
    buttonGroup.add(button2);
    buttonGroup.add(button3);

    pack();
```

```
    setVisible(true);
  }

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

5.1.11 JCheckBox and JRadioButton

class javax.swing.JCheckBox, class javax.swing.JRadioButton

`JCheckBox` and `JRadioButton` both inherit all `JToggleButton` functionality. In fact the only significant differences between all three components is their UI delegate views (i.e. how they are rendered). Both button types are normally used to select the mode of a particular application function. Figures 5.5 and 5.6 show the previous example running with `JCheckBoxes` and `JRadioButtons` as replacements for the `JToggleButtons`.

Figure 5.5 JCheckBoxes in a ButtonGroup
<<file figure5-5.gif>>

Figure 5.6 JRadioButtons in a ButtonGroup
<<file figure5-6.gif>>

5.1.12 JToolTip & ToolTipManager

class javax.swing.JToolTip, class javax.swing.ToolTipManager

A `JToolTip` is a small popup window designed to contain informative text about a component when the mouse moves over it. We don't generally create instances of these components ourselves. Rather, we call `setToolTipText()` on any `JComponent` subclass and pass it a descriptive `String`. This `String` is then stored as a client property within that component's client properties `Hashtable`, and that component is then registered with the `ToolTipManager` (using `ToolTipManager's registerComponent()` method). The `ToolTipManager` adds a `MouseListener` to each component that registers with it.

To unregister a component we can pass `null` to that component's `setToolTipText()` method. This invokes `ToolTipManager's unregisterComponent()` method which removes its `MouseListener` from that component. Figure 5.7 shows a `JToggleButton` with simple tooltip text.

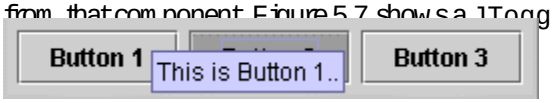


Figure 5.7 JToggleButton with tooltip text
<<file figure5-7.gif>>

The `ToolTipManager` is a service class that maintains a shared instance registered with `AppContext` (see chapter 2). We can access the `ToolTipManager` directly by calling its static `sharedInstance()` method:

```
ToolTipManager toolTipManager = ToolTipManager.sharedInstance();
```

Internally this class uses three non-repeating `Timers` with delay times defaulting to 750, 500, and 4000. `ToolTipManager` uses these `Timer`'s in coordination with mouse listeners to determine if and when to display a `JToolTip` with a component's specified tooltip text. When the mouse enters a components bounds `ToolTipManager` will detect this and wait 750ms until displaying a `JToolTip` for that component. This is referred to as the *initial* delay time. A `JToolTip` will stay visible for 4000ms or until we move the mouse outside of that component's bounds, whichever comes first. This is referred to as the *dismiss* delay time. The 500ms `Timer` represents the *reshow* delay time which specifies how soon the `JToolTip` we have just seen will appear again when this component is re-entered. Each of these delay times can be set using `ToolTipManager`'s `setDismissDelay()`, `setInitialDelay()`, and `setReshowDelay()` methods.

`ToolTipManager` is a very nice service to have implemented for us, but it does have significant limitations. When we construct our polygonal buttons in section 5.6 below, we will find that it is not robust enough to support non-rectangular components. It is also the case that `JToolTips` are only designed to allow a single line of text.

5.1.13 Labels and buttons with HTML text

The Swing 1.1.1 Beta 1 release (February 1999) offers a particularly interesting new feature (among several bug fixes). Now we can use HTML text in `JButton` and `JLabel` components (as well as for tooltip text). We don't have to learn any new methods to use this functionality, and the UI delegate handles the HTML rendering for us. If a button/label's text starts with `<HTML>`, Swing knows to lightweight render the text in HTML format. We can use normal paragraph tags (`<P>`, `</P>`), line break tags (`<BR>`), etc. For instance, we can assign a multiple-line tooltip to any component like so:

```
myComponent.setToolTipText("<html>Multi-line tooltips<br>" +
  "are easy!");
```

The `<br>` tag specifies a line break. The following example demonstrates this functionality.

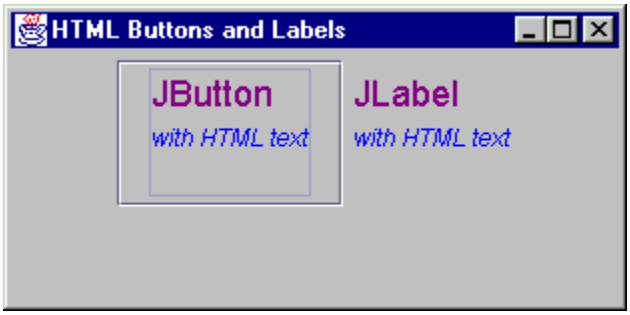


Figure 5.12 A `JButton` and `JLabel` with HTML text.
<<file figure5-12.gif>>

The Code: HtmlButtons.java
see \Chapter5\3

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

public class HtmlButtons extends JFrame
{
  public HtmlButtons() {
```

```
    super("HTML Buttons and Labels");
    setSize(400, 300);

    getContentPane().setLayout(new FlowLayout());

    String htmlText =
      "<html><p><font color=\"#800080\" "+
      "size=\"4\" face=\"Verdana\">JButton</font> </p>"+
      "<address><font size=\"2\"><em>"+
      "with HTML text</em></font>"+
      "</address>";
    JButton btn = new JButton(htmlText);
    getContentPane().add(btn);

    htmlText =
      "<html><p><font color=\"#800080\" "+
      "size=\"4\" face=\"Verdana\">JLabel</font> </p>"+
      "<address><font size=\"2\"><em>"+
      "with HTML text</em></font>"+
      "</address>";
    JLabel lbl = new JLabel(htmlText);
    getContentPane().add(lbl);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);
    setVisible(true);
  }

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

5.2 Custom buttons: part I - Transparent buttons

Buttons in Swing can adopt almost any presentation we can think of. Of course some presentations are tougher to implement than others. In the remainder of this chapter we will deal directly with these issues. The example in this section shows how to construct invisible buttons which only appear when the user moves the mouse cursor over them. Specifically, a border will be painted, and tooltip text will be activated in the default manner.

Such button's can be useful in applets for pre-defined hyperlink navigation, and we will design our invisible button class with this in mind. Thus, we will show how to create an applet that reads a set of parameters from the HTML page it is embedded in, and loads a corresponding set of invisible buttons. For each button, the designer of the HTML page must provide three parameters: the desired hyperlink URL, the button's bounds (positions and size), and the button's tooltip text. Additionaly our sample applet will require a background image parameter. Our button's bounds are intended to directly correspond to an 'active' region of this background image--much like the venerable HTML image mapping functionality.

Figure 5.8 Transparent rectangular buttons in an applet
<<file figure5-8.gif>>

The Code: ButtonApplet.java
see \Chapter5\4

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class ButtonApplet extends JApplet
{
  public ButtonApplet() {}

  public synchronized void init() {
    String imageName = getParameter("image");
    if (imageName == null) {
      System.err.println("Need \"image\" parameter");
      return;
    }
    URL imageUrl = null;
    try {
      imageUrl = new URL(getDocumentBase(), imageName);
    }
    catch (MalformedURLException ex) {
      ex.printStackTrace();
      return;
    }
    ImageIcon bigImage = new ImageIcon(imageUrl);
```

```
    JLabel bigLabel = new JLabel(bigImage);
    bigLabel.setLayout(null);

    int index = 1;
    int[] q = new int[4];
    while(true) {
      String paramSize = getParameter("button"+index);
      String paramName = getParameter("name"+index);
      String paramUrl = getParameter("url"+index);
      if (paramSize==null || paramName==null || paramUrl==null)
        break;

      try {
        StringTokenizer tokenizer = new StringTokenizer(
          paramSize, ",");
        for (int k=0; k<4; k++) {
          String str = tokenizer.nextToken().trim();
          q[k] = Integer.parseInt(str);
        }
      }
      catch (Exception ex) { break; }

      NavigateButton btn = new NavigateButton(this,
        paramName, paramUrl);
      bigLabel.add(btn);
      btn.setBounds(q[0], q[1], q[2], q[3]);

      index++;
    }

    getContentPane().setLayout(null);
    getContentPane().add(bigLabel);
    bigLabel.setBounds(0, 0, bigImage.getIconWidth(),
      bigImage.getIconHeight());
  }

  public String getAppletInfo() {
    return "Sample applet with NavigateButtons";
  }

  public String[][] getParameterInfo() {
    String pinfo[][] = {
      {"image",  "string",  "base image file name"},
      {"buttonX","x,y,w,h", "button's bounds"},
      {"nameX",  "string",  "tooltip text"},
      {"urlX",   "url",     "link URL"} };
    return pinfo;
  }
}

class NavigateButton extends JButton implements ActionListener
{
  protected Border m_activeBorder;
  protected Border m_inactiveBorder;

  protected Applet m_parent;
  protected String m_text;
  protected String m_sUrl;
  protected URL    m_url;

  public NavigateButton(Applet parent, String text, String sUrl) {
    m_parent = parent;
```

```
    setText(text);
    m_sUrl = sUrl;
    try {
      m_url = new URL(sUrl);
    }
    catch(Exception ex) { m_url = null; }

    setOpaque(false);
    enableEvents(AWTEvent.MOUSE_EVENT_MASK);

    m_activeBorder = new MatteBorder(1, 1, 1, 1, Color.yellow);
    m_inactiveBorder = new EmptyBorder(1, 1, 1, 1);
    setBorder(m_inactiveBorder);

    addActionListener(this);
  }

  public void setText(String text) {
    m_text = text;
    setToolTipText(text);
  }

  public String getText() {
    return m_text;
  }

  protected void processMouseEvent(MouseEvent evt) {
    switch (evt.getID()) {
      case MouseEvent.MOUSE_ENTERED:
        setBorder(m_activeBorder);
        setCursor(Cursor.getPredefinedCursor(
          Cursor.HAND_CURSOR));
        m_parent.showStatus(m_sUrl);
        break;
      case MouseEvent.MOUSE_EXITED:
        setBorder(m_inactiveBorder);
        setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
        m_parent.showStatus("");
        break;
    }
    super.processMouseEvent(evt);
  }

  public void actionPerformed(ActionEvent e) {
    if (m_url != null) {
      AppletContext context = m_parent.getAppletContext();
      if (context != null)
        context.showDocument(m_url);
    }
  }

  public void paintComponent(Graphics g) {
    paintBorder(g);
  }
}
```

Understanding the Code

Class ButtonApplet

This class extends `JApplet` to provide web page functionality. The `init()` method creates and initializes all GUI components. It starts by reading the applet's "image" parameter which is then used along with the applet's codebase to construct a `URL`:

```
imageUrl = new URL(getDocumentBase(), imageName);
```

This `URL` points to the image file which is used to create our `bigLabel` label which is used as the applet's background image.

The applet can be configured to hold several invisible buttons for navigating to pre-defined `URL`s. For each button, three applet parameters must be provided:

- `buttonN`: holds four comma-delimited numbers for `x`, `y`, `width`, and `height` of button `N`.
- `nameN`: tooltip text for button `N`.
- `urlN`: `URL` to re-direct the browser when the user clicks mouse over button `N`.

As soon as these parameters can be read and parsed for a given `N`, a new button is created and added to `bigLabel`:

```
NavigateButton btn = new NavigateButton(this,
  paramName, paramUrl);
bigLabel.add(btn);
btn.setBounds(q[0], q[1], q[2], q[3]);
```

Finally the `bigLabel` component is added to the applet's content pane. It receives a fixed size to avoid any repositioning if the applet's pane is somehow resized.

The `getAppletInfo()` method returns a `String` description of this applet. The `getParameterInfo()` method returns a 2-dimensional `String` array describing the parameters accepted by this applet. Both are strongly recommended constituents of any applet, but are not required for raw functionality.

Class NavigateButton

This class extends `JButton` to provide our custom implementation of an invisible button. It implements the `ActionListener` interface, eliminating the need to add an external listener, and shows how we can enable mouse events without implementing the `MouseListener` interface.

Several parameters are declared in this class:

- `Border m_activeBorder`: the border which will be used when the button is active (when the mouse cursor is moved over the button).
- `Border m_inactiveBorder`: the border which will be used when the button is inactive (when no mouse cursor is over the button). Usually this will not be visible.
- `Applet m_parent`: a reference to the parent applet.
- `String m_text`: tooltip text for this button.
- `String m_sUrl`: string representation of the `URL` (for display in the browser's status bar).

`URL m_url`: the actual URL to redirect the browser to when a mouse click occurs.

The constructor of the `NavigateButton` class takes three parameters: a reference to the parent applet, tooltip text, and a `String` representation of a URL. It assigns all instance variables and creates a URL from the given `String`. Note that if the URL address cannot be resolved, it is set to `null` (this will disable navigation). The `Opaque` property is set to `false` because this component is supposed to be transparent. Note also that this component processes its own `MouseEvents`, enabled with the `enableEvents()` method. This button will also receive `ActionEvents` by virtue of implementing `ActionListener` and adding itself as a listener.

The `setText()` and `getText()` methods manage the `m_text` (tooltip text) property. They also serve to override the corresponding methods inherited from the `JButton` class.

The `processMouseEvent()` method will be called for notification about mouse events on this component. We want to process only two kinds of events: `MOUSE_ENTERED` and `MOUSE_EXITED`. When the mouse enters the button's bounds, we set the border to `m_activeBorder`, change the mouse cursor to the hand cursor, display the `String` description of the URL in the browser's status bar. When the mouse exits the button's bounds we perform the opposite actions: set the border to `m_inactiveBorder`, set the mouse cursor to the default cursor, and clear the browser's status bar.

The `actionPerformed()` method will be called when the user presses this button (note that we use the inherited `JButton` processing for both mouse clicks and the keyboard mnemonic). If both the URL and `AppletContext` instances are not `null`, the `showDocument()` method is called to redirect the browser to the button's URL.

Note: Do not confuse `AppletContext` with the `AppContext` class we discussed in section 2.5. `AppletContext` is an interface for describing an applet's environment, including information about the document it is contained in, as well as information about other appets that might also be contained in that document.

The `paintComponent()` method used for this button has a very simple implementation. We just draw the button's border by calling the `paintBorder()`. Because this component does not have a UI delegate we do not need to call `super.paintComponent()` from this method.

Running the Code

To run it in the web browser we have constructed the following HTML file:

```
<html>

<head>
<title></title>
</head>

<body>

<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 563 HEIGHT = 275
codebase="http://java.sun.com/products/plugin/1.2/jinstall-12-
win32.cab#Version=1,2,0,0">
```

```
<PARAM NAME = "CODE" VALUE = "ButtonApplet.class" >
<PARAM NAME = "type" VALUE ="application/x-java-applet;version=1.2">
  <param name="button1" value="49, 134, 161, 22">
  <param name="button2" value="49, 156, 161, 22">
  <param name="button3" value="16, 178, 194, 22">
  <param name="button4" value="85, 200, 125, 22">
  <param name="button5" value="85, 222, 125, 22">
  <param name="image" value="nasa.gif">
  <param name="name1" value="What is Earth Science?">
  <param name="name2" value="Earth Science Missions">
  <param name="name3" value="Science of the Earth System">
  <param name="name4" value="Image Galery">
  <param name="name5" value="For Kids Only">
  <param name="url1"
   value="http://www.earth.nasa.gov/whatis/index.html">
  <param name="url2"
   value="http://www.earth.nasa.gov/missions/index.html">
  <param name="url3"
   value="http://www.earth.nasa.gov/science/index.html">
  <param name="url4"
   value="http://www.earth.nasa.gov/gallery/index.html">
  <param name="url5"
   value="http://kids.mtpe.hq.nasa.gov/">

<COMMENT>
<EMBED type="application/x-java-applet;version=1.2" CODE =
"ButtonApplet.class"
  WIDTH = "563" HEIGHT = "275"
  codebase="./"
  button1="49, 134, 161, 22"
  button2="49, 156, 161, 22"
  button3="16, 178, 194, 22"
  button4="85, 200, 125, 22"
  button5="85, 222, 125, 22"
  image="nasa.gif"
  name1="What is Earth Science?"
  name2="Earth Science Missions"
  name3="Science of the Earth System"
  name4="Image Galery"
  name5="For Kids Only"
  url1="http://www.earth.nasa.gov/whatis/index.html"
  url2="http://www.earth.nasa.gov/missions/index.html"
  url3="http://www.earth.nasa.gov/science/index.html"
  url4="http://www.earth.nasa.gov/gallery/index.html"
  url5="http://kids.mtpe.hq.nasa.gov/"
  pluginspage=
    "http://java.sun.com/products/plugin/1.2/plugin-install.html">
<NOEMBED>
</COMMENT>
alt="Your browser understands the &lt;APPLET&gt; tag but isn't
running the applet, for some reason."
Your browser is completely ignoring the &lt;APPLET&gt; tag!
</NOEMBED>
</EMBED>
</OBJECT>
</p>

<p> </p>
</body>
</html>
```

Note: The HTML file above works with appletviewer, Netscape Navigator 4.0 and Microsoft Internet Explorer 4.0. This compatibility is achieved due to Java Plug-in technology. See http://www.javasoft.com/products/plugin/1.2/docs/tags.html for more details on how to write Plug-in compatible HTML files. The downside is that we need to include all applet parameters two times for each web browser.

Reference: For additional information about the Java Plug-in and Plug-in HTML converter (a convenient utility to generate Plug-in compliant HTML), see Swing Connection's "Swinging on the Web" article at: http://java.sun.com/products/jfc/tsc/java_plug-in/java_plug-in.html.

Figure 5.8 shows `ButtonApplet` running in Netscape Navigator 4.05 using the Java Plug-in. Note how invisible buttons react when the mouse cursor moves over them. Click a button and navigate to one of the NASA sites.

5.3 Custom buttons: part II - Polygonal buttons

The approach described in the previous section assumes that all navigational buttons have a rectangular shape. This can be too restrictive for complex active regions needed in the navigation of images such as geographical maps. In this example we will show how to extend the idea of transparent buttons developed in the previous example, to transparent non-rectangular buttons.

The `java.awt.Polygon` class is extremely helpful for this purpose, especially its following two related methods (see API docs for more info):

`Polygon.contains(int x, int y)`: returns `true` if a point with the given coordinates is contained inside the `Polygon`.

`Graphics.drawPolygon(Polygon polygon)`: draws an outline of a `Polygon` using given `Graphics` object.

The first method will be used in this example to verify that the mouse cursor is located inside the given polygon. The second will be used to actually draw a polygon representing the bounds of a non-rectangular button.

This seems fairly basic, but there is one significant complication that exists. All Swing components are encapsulated in rectangular bounds and nothing can be done about this. If some component receives a mouse event which occurs in it's rectangular bounds, the overlapped underlying components do not have a chance to receive this event. Figure 5.9 illustrates two overlapping non-rectangular buttons. A part of Button B lying under the rectangle of Button A will never receive mouse events and cannot be clicked.

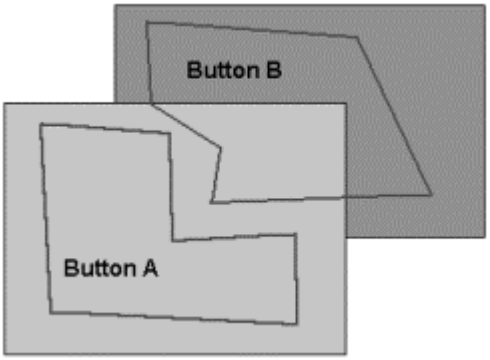


Figure 5.9 Illustration of two overlapping non-rectangular buttons.
<<file figure5-9.gif>>

To resolve this situation we can skip any mouse event processing in our non-rectangular components. Instead, all mouse events can be directed to the parent container. All buttons can then register themselves as `MouseListeners` and `MouseMotionListeners` with that container. In this way, mouse events can be received without regard to overlapping! By doing this all buttons will receive notification about all events without any preliminary filtering. To minimize the resulting impact on the system's performance we need to provide a quick discard of events lying outside the button's basic rectangle.

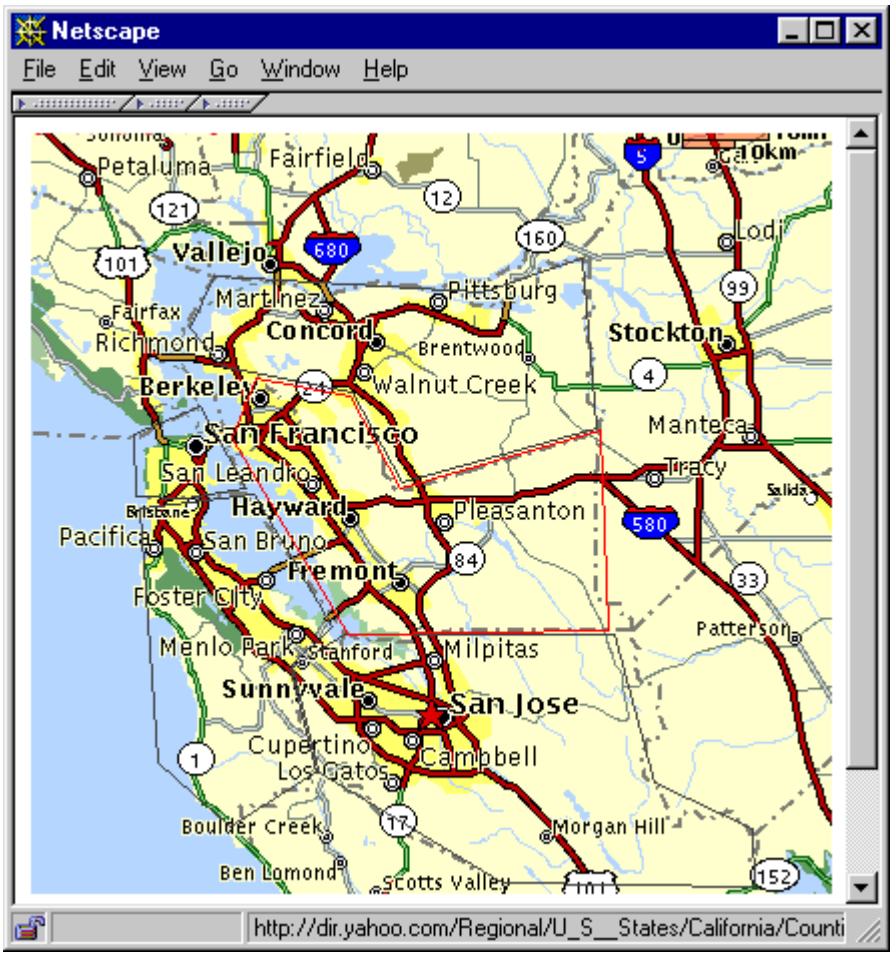


Figure 5.10 Polygonal buttons in an applet.
<<file figure5-10.gif>>

The Code: ButtonApplet2.java
see \Chapter5\5

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;

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

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

public class ButtonApplet2 extends JApplet
{
  public ButtonApplet2() {}

  public synchronized void init() {
    // Unchanged code from section 5.2

    int index = 1;
    while(true) {
      String paramSize = getParameter("button"+index);
      String paramName = getParameter("name"+index);
      String paramUrl = getParameter("url"+index);
      if (paramSize==null || paramName==null || paramUrl==null)
        break;

      Polygon p = new Polygon();
      try {
        StringTokenizer tokenizer = new StringTokenizer(
          paramSize, ",");
        while (tokenizer.hasMoreTokens()) {
          String str = tokenizer.nextToken().trim();
          int x = Integer.parseInt(str);
          str = tokenizer.nextToken().trim();
          int y = Integer.parseInt(str);
          p.addPoint(x, y);
        }
      }
      catch (Exception ex) { break; }

      PolygonButton btn = new PolygonButton(this, p,
        paramName, paramUrl);
      bigLabel.add(btn);

      index++;
    }

    getContentPane().setLayout(null);
    getContentPane().add(bigLabel);
    bigLabel.setBounds(0, 0, bigImage.getIconWidth(),
    bigImage.getIconHeight());
  }

  public String getAppletInfo() {
    return "Sample applet with PolygonButtons";
  }

  public String[][] getParameterInfo() {
    String pinfo[][] = {
      {"image",  "string",  "base image file name"},
      {"buttonX","x1,y1, x2,y2, ...", "button's bounds"},
      {"nameX",  "string",  "tooltip text"},
      {"urlX",   "url",     "link URL"} };
    return pinfo;
  }
}

class PolygonButton extends JComponent
 implements MouseListener, MouseMotionListener
{
  static public Color ACTIVE_COLOR = Color.red;
```

```
static public Color INACTIVE_COLOR = Color. darkGray;

protected JApplet m_parent;
protected String m_text;
protected String m_sUrl;
protected URL    m_url;

protected Polygon m_polygon;
protected Rectangle m_rc;
protected boolean m_active;

protected static PolygonButton m_currentButton;

public PolygonButton(JApplet parent, Polygon p,
 String text, String sUrl)
{
  m_parent = parent;
  m_polygon = p;
  setText(text);
  m_sUrl = sUrl;
  try {
    m_url = new URL(sUrl);
  }
  catch(Exception ex) { m_url = null; }

  setOpaque(false);

  m_parent.addMouseListener(this);
  m_parent.addMouseMotionListener(this);

  m_rc = new Rectangle(m_polygon.getBounds()); // Bug alert!
  m_rc.grow(1, 1);

  setBounds(m_rc);
  m_polygon.translate(-m_rc.x, -m_rc.y);
}

public void setText(String text) { m_text = text; }

public String getText() { return m_text; }

public void mouseMoved(MouseEvent e) {
  if (!m_rc.contains(e.getX(), e.getY()) || e.isConsumed()) {
    if (m_active)
      setState(false);
    return; // quickly return, if outside our rectangle
  }
  int x = e.getX() - m_rc.x;
  int y = e.getY() - m_rc.y;
  boolean active = m_polygon.contains(x, y);

  if (m_active != active)
    setState(active);
  if (m_active)
    e.consume();
}

public void mouseDragged(MouseEvent e) {}

protected void setState(boolean active) {
  m_active = active;
  repaint();
```

```
    if (m_active) {
      if (m_currentButton != null)
        m_currentButton.setState(false);
      m_currentButton = this;
      m_parent.setCursor(Cursor.getPredefinedCursor(
        Cursor.HAND_CURSOR));
      m_parent.showStatus(m_sUrl);
    }
    else {
      m_currentButton = null;
      m_parent.setCursor(Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
      m_parent.showStatus("");
    }
  }

  public void mouseClicked(MouseEvent e) {
    if (m_active && m_url != null && !e.isConsumed()) {
      AppletContext context = m_parent.getAppletContext();
      if (context != null)
        context.showDocument(m_url);
      e.consume();
    }
  }
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseExited(MouseEvent e) { mouseMoved(e); }
  public void mouseEntered(MouseEvent e) { mouseMoved(e); }

  public void paint(Graphics g) {
    g.setColor(m_active ? ACTIVE_COLOR : INACTIVE_COLOR);
    g.drawPolygon(m_polygon);
  }
}
```

Understanding the Code

Class ButtonApplet2

This class is a slightly modified version of the `ButtonApplet` class in the previous section to accommodate polygonal button sizes rather than rectangles (the parser has been modified to read in an arbitrary amount of points). Now it creates a `Polygon` instance and parses a data string, which is assumed to contain pairs of comma-separated coordinates, adding each coordinate to the `Polygon` using the the `addPoint()` method. The resulting `Polygon` instance is used to create a new `PolygonButton` component.

Class PolygonButton

This class serves as a replacement for the `NavigateButton` class in the previous example. Note that it extends `JComponent` directly. This is necessary to disassociate any mouse handling inherent in buttons (which is actually built into the button UI delegates). Remember, we want to handle mouse events ourselves, but we want them each to be sent from within the parent's bounds to each `PolygonButton`, not from each `PolygonButton` to the parent.

Note: This is the opposite way of working with mouse listeners that we are used to. The idea may take a few moments to sink in because directing events from child to parent is so much more common, we generally don't think of things the other way around.

So, to be notified of mouse events from the parent, we'll need to implement the `MouseListener` and `MouseMotionListener` interfaces.

Four new instance variables are declared:

- `Polygon m_polygon`: the polygonal region representing this button's bounds.
- `Rectangle m_rc`: this button's bounding rectangle as seen in the coordinate space of the parent.
- `boolean m_active`: flag indicating that this button is active.
- `PolygonButton m_currentButton`: a static reference to the instance of this class which is currently active.

The constructor of the `PolygonButton` class takes four parameters: a reference to the parent applet, the `Polygon` instance representing this component's bounds, tooltip text, and a `String` representation of a `URL`. It assigns all instance variables and instantiates a `URL` using the associate String parameter (similar to what we saw in the last example). Note that this component adds itself to the parent applet as a `MouseListener` and `MouseMotionListener`:

```
m_parent.addMouseListener(this);
m_parent.addMouseMotionListener(this);
```

The bounding rectangle `m_rc` is computed with the `Polygon.getBounds()` method. Note that this method does not create a new instance of the `Rectangle` class, but returns a reference to the an internal `Polygon` instance variable which is subject to change. This is not safe, so we must explicitly create a new `Rectangle` instance from the supplied reference. This `Rectangle`'s bounds are expanded (using its `grow()` method) to take into account border width. Finally the `Rectangle` `m_rc` is set as the button's bounding region, and the `Polygon` is translated into the component's local coordinates by shifting it's origin using its `translate()` method.

The `mouseMoved()` method is invoked when mouse events occur in the parent container. First we quickly check whether the event lies inside our bounding rectangle and is not yet consumed by another component. If this is true, we continue processing this event. Otherwise our method returns. Before we return, however, we first check whether this button is still active for some reason (this can happen if the mouse cursor moves too fast out of this button's bound, and the given component did not receive a `MOUSE_EXITED` `MouseEvent` to deactivate itself). If this is the case, we deactivate it and then exit.

Next we translate the coordinates of the event, manually, into our button's local system (remember that this is an event from the parent container) and check whether this point lies within our polygon. This gives us a boolean result which should indicate whether this component is currently active or inactive. If our button's current activation state (`m_active`) is not equal to this value, we call the `setState()` method to change it so that it is. Finally, if this component is active we consume the given `MouseEvent` to avoid activation of two components simultaneously.

The `setState()` method is called, as described above, to set a new activation state of this component. It takes a boolean value as parameter and stores it in the `m_active` instance variable. Then it repaints the component to reflect a change in state, if any:

1. If the `m_active` flag is set to `true`, this method checks the class reference to the currently active

button stored in the `m_currentButton` static variable. In the case where this reference still points to some other component (again, it potentially can happen if the mouse cursor moves too quickly out of a components rectangular bounds) we force that component to be inactive. Then we store a `this` reference into the `m_currentButton` static variable, letting all other button's know that this button is now the currently active one. We then change the mouse cursor to the hand cursor (as in the previous example) and display our `URL` in the browser's status bar.

2. If the `m_active` flag is set to `false` this method sets the `m_currentButton` static variable to `null`, changes mouse cursor to the default cursor, and clears the browser's status bar.

The `mouseClicked()` method checks whether this component is active (this implies that the mouse cursor is located within our polygon, and not just within the bounding rectangle), the `URL` is resolved, and the mouse event is not consumed. If all three checks are verifiable, this method redirects the browser to the component's associated `URL` and consumes the mouse event to avoid processing by any other components.

The rest of methods, implemented due to the `MouseListener` and `MouseMotionListener` interfaces, receive empty bodies, except for the `mouseExited()` and `mouseEntered()` methods. Both of these methods send all their traffic to the `mouseMoved()` method to notify the component that the cursor has left or has enetered the container.

The `paintComponent()` method simply draws the component's `Polygon` with in gray if inactive, and in red if active.

Note: We've purposefully avoided including tooltip text for these non-rectangular buttons. The reason is that the underlying Swing `ToolTipManager` essentially relies on the rectangular shape of the components it manages. Somehow, invoking the Swing tooltip API destroys our model of processing mouse events. In order to allow tooltips we have to develop our own version of a tooltip manager--this is the subject of the next example.

Running the Code

To run it in the web browser we have constructed the following `HTML` file (see Java Plug-in and Java Plug-in HTML converter references in the previous example):

```
<html>
<head>
<title></title>
</head>
<body>

<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
WIDTH = 400 HEIGHT = 380
codebase="http://java.sun.com/products/plugin/1.2/jinstall-12-
win32.cab#Version=1,2,0,0">
<PARAM NAME = "CODE" VALUE = "ButtonApplet2.class" >
<PARAM NAME = "type"
       VALUE ="application/x-java-applet;version=1.2">
<param name="image" value="bay_area.gif">

<param name="button1"
       value="112,122, 159,131, 184,177, 284,148, 288,248, 158,250, 100,152">
<param name="name1" value="Alameda County">
```

```
<param name="url1"
       value="http://dir.yahoo.com/Regional/U_S__States/
California/Counties_and_Regions/Alameda_County/">

<param name="button2"
       value="84,136, 107,177, 76,182, 52,181, 51,150">
<param name="name2" value="San Francisco County">
<param name="url2"
       value="http://dir.yahoo.com/Regional/U_S__States/
California/Counties_and_Regions/San_Francisco_County/">

<param name="button3"
       value="156,250, 129,267, 142,318, 235,374, 361,376, 360,347, 311,324,
291,250">
<param name="name3" value="Santa Clara County">
<param name="url3"
       value="http://dir.yahoo.com/Regional/U_S__States/
California/Counties_and_Regions/Santa_Clara_County/">

<param name="button4"
       value="54,187, 111,180, 150,246, 130,265, 143,318, 99,346, 63,314">
<param name="name4" value="San Mateo County">
<param name="url4"
       value="http://dir.yahoo.com/Regional/U_S__States/
California/Counties_and_Regions/San_Mateo_County/">

<param name="button5"
       value="91,71, 225,79, 275,62, 282,147, 185,174, 160,129, 95,116,
79,97">
<param name="name5" value="Contra Costa County">
<param name="url5"
       value="http://dir.yahoo.com/Regional/U_S__States/
California/Counties_and_Regions/Contra_Costa_County/">

<COMMENT>
<EMBED type="application/x-java-applet;version=1.2" CODE =
"ButtonApplet2.class"
  WIDTH = "400" HEIGHT = "380"
  codebase="./"
  image="bay_area.gif"
  button1="112,122, 159,131, 184,177, 284,148, 288,248, 158,250, 100,152"
  name1="Alameda County"

url1="http://dir.yahoo.com/Regional/U_S__States/California/Counties_and_Region
s/Alameda_County/"
  button2="84,136, 107,177, 76,182, 52,181, 51,150"
  name2="San Francisco County"

url2="http://dir.yahoo.com/Regional/U_S__States/California/Counties_and_Region
s/San_Francisco_County/"
  button3="156,250, 129,267, 142,318, 235,374, 361,376, 360,347, 311,324,
291,250"
  name3="Santa Clara County"

url3="http://dir.yahoo.com/Regional/U_S__States/California/Counties_and_Region
s/Santa_Clara_County/"
  button4="54,187, 111,180, 150,246, 130,265, 143,318, 99,346, 63,314"
  name4="San Mateo County"

url4="http://dir.yahoo.com/Regional/U_S__States/California/Counties_and_Region
s/San_Mateo_County/"
  button5="91,71, 225,79, 275,62, 282,147, 185,174, 160,129, 95,116, 79,97"
```

```
  name5="Contra Costa County"

url5="http://dir.yahoo.com/Regional/U_S__States/California/Counties_and_Region
s/Contra_Costa_County/"
  pluginspage="http://java.sun.com/products/plugin/1.2/plugin-install.html">
<NOEMBED></COMMENT>
alt="Your browser understands the &lt;APPLET&gt; tag but isn't running the
applet, for some reason."
        Your browser is completely ignoring the &lt;APPLET&gt; tag!
</NOEMBED>
</EMBED>
</OBJECT>
</p>

<p> </p>
</body>
</html>
```

Figure 5.10 shows the `ButtonApplet2` example running in Netscape 4.05 with the Java Plug-in. Our HTML file has been constructed to display an active map of the San Francisco bay area. Five non-rectangular buttons correspond to this area's five counties. Note how the non-rectangular buttons react when the mouse cursor moves in and out of their boundaries. Verify that they behave correctly even if a part of a given button lies under the bounding rectangle of another button (a good place to check is the sharp border between Alameda and Contra Costa counties). Click over the button and note the navigation to one of the Yahoo sites containing information about the selected county.

It is clear that tooltip displays would help to dispel any confusion as to which county is which. The next example shows how to implement this feature.

5.4 Custom buttons: part III - Tooltip management

In this section we'll discuss how to implement custom management of tooltips in a Swing application. If you're completely satisfied with the default `ToolTipManager` provided with Swing, you can skip this section. But there may be situations when this default implementation is not satisfactory, as in our example above using non-rectangular components.

We will construct our own version of a tooltip manager to display a tooltip window if the mouse cursor rests over some point inside the button's area longer than a specified time interval. It will be displayed for a speecified amount of time and, to avoid annoying the user, we will then hide the tooltip window until the mouse cursor moves to a new position. In designing our tooltip manager we will take a different approach than that taken by Swing's default `ToolTipManager` (see section 5.4). Instead of using three different `Timers`, we will use just one. This involves tracking more information, but is slightly more efficient by avoiding the handling of multiple `ActionEvents`.

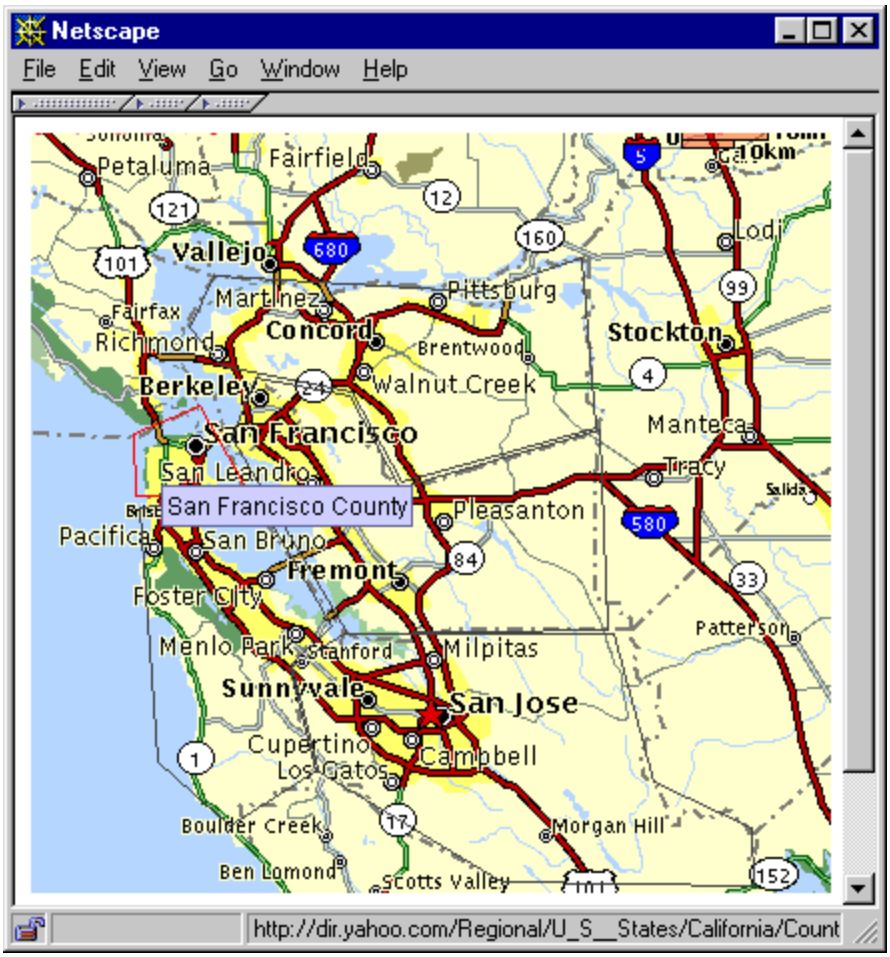


Figure 5.11 Polygonal buttons with custom tooltip manager.
<<file figure5-11.gif>>

The Code: ButtonApplet3.java
see \Chapter5\6

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class ButtonApplet3 extends JApplet
{
  protected ToolTipManager m_manager;

  public ButtonApplet3() {}

  public synchronized void init() {
    // unchanged code from section 5.3

    m_manager = new ToolTipManager(this);
    PolygonButton.m_toolTip = m_manager.m_toolTip;
```

```
    getContentPane().setLayout(null);
    getContentPane().add(bigLabel);
    bigLabel.setBounds(0, 0, bigImage.getIconWidth(),
      bigImage.getIconHeight());
  }

  // unchanged code from section 5.3
}

class PolygonButton extends JComponent
 implements MouseListener, MouseMotionListener
{
  // unchanged code from section 5.3

  public static JToolTip m_toolTip;

  protected void setState(boolean active) {
    m_active = active;
    repaint();
    if (active) {
      if (m_currentButton != null)
        m_currentButton.setState(false);
      m_parent.setCursor(Cursor.getPredefinedCursor(
        Cursor.HAND_CURSOR));
      m_parent.showStatus(m_sUrl);
      if (m_toolTip != null)
        m_toolTip.setTipText(m_text);
    }
    else {
      m_currentButton = null;
      m_parent.setCursor(Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
      m_parent.showStatus("");
      if (m_toolTip != null)
        m_toolTip.setTipText(null);
    }
  }
}

class MyToolTipManager extends MouseMotionAdapter
 implements ActionListener
{
  protected Timer m_timer;
  protected int m_lastX = -1;
  protected int m_lastY = -1;
  protected boolean m_moved = false;
  protected int m_counter = 0;

  public JToolTip m_toolTip = new JToolTip();

  ToolTipManager(JApplet parent) {
    parent.addMouseMotionListener(this);
    m_toolTip.setTipText(null);
    parent.getContentPane().add(m_toolTip);
    m_toolTip.setVisible(false);
    m_timer = new Timer(1000, this);
    m_timer.start();
  }

  public void mouseMoved(MouseEvent e) {
    m_moved = true;
    m_counter = -1;
```

```
    m_lastX = e.getX();
    m_lastY = e.getY();
    if (m_toolTip.isVisible()) {
      m_toolTip.setVisible(false);
      m_toolTip.getParent().repaint();
    }
  }

  public void actionPerformed(ActionEvent e) {
    if (m_moved || m_counter==0 || m_toolTip.getTipText()==null) {
      if (m_toolTip.isVisible())
        m_toolTip.setVisible(false);
      m_moved = false;
      return;
    }

    if (m_counter < 0) {
      m_counter = 4;
      m_toolTip.setVisible(true);
      Dimension d = m_toolTip.getPreferredSize();
      m_toolTip.setBounds(m_lastX, m_lastY+20,
      d.width, d.height);
    }
    m_counter--;
  }
}
```

Understanding the Code

Class ButtonApplet3

This class requires very few modifications from `ButtonApplet2` in the last section. It declares and creates `MyToolTipManager m_manager` and passes a `this` reference to it:

```
        m_manager = new MyToolTipManager(this);
```

As you will see below, our `MyToolTipManager` class manages a publically accessible `JToolTip`, `m_toolTip`. `MyToolTipManager` itself is not intended to provide any meaningful content to this tooltip. Rather, this is to be done by other components, in our case, by the `PolygonButtons`. Thus, our `PolygonButton` class declares a `static` reference to a `JToolTip` component. Whenever a button becomes active, this `JToolTip`'s text will be assigned to that of the active button. So, when we create our instance of `MyToolTipManager`, we assign its publically accessible `JToolTip` as our `Polygon` class's static `JToolTip` (which is also publically accessible):

```
        PolygonButton.m_toolTip = m_manager.m_toolTip;
```

Thus, there will only be one `JToolTip` instance in existance for the lifetime of this applet, and both `MyToolTipManager` and our `PolygonButtons` have control over it.

Class PolygonButton

As we've mentioned above, this class now declares the static variable: `JToolTip m_toolTip`. Class `PolygonButton` does not initialize this reference. However, this reference is checked during `PolygonButton` activation in the `setState()` method. If `m_toolTip` is not `null` (set to point to a valid tooltip window by some outer class--which, in our example, is done in the `ButtonApplet3` `init()` method shown above), the `setTipText()` method is invoked to set the proper text while the mouse cursor hovers over the button.

Class MyToolTipManager

This class represents a custom tooltip manager which is free from assumption of the rectangularity of its child components. It extends the `MouseMotionAdapter` class and implements the `ActionListener` interface to work as both a `MouseMotionListener` and `ActionListener`. Six instance variables are declared:

- `Timer m_timer`: our managing timer.
- `int m_lastX, m_lastY`: the last coordinates of the mouse cursor--reassigned each time the mouse is moved.
- `boolean m_moved`: flag indicating that the mouse cursor has moved.
- `int m_counter`: time ticks counter managing the tooltip's time to live (see below).
- `JToolTip m_toolTip`: the tooltip component to be displayed.

The constructor of the `MyToolTipManager` class takes a reference to the parenting `JApplet` as a parameter and registers itself as a `MouseMotionListener` on this component. Then it creates the `JToolTip m_toolTip` component and adds it to the applet's content pane. `m_tooltip` is set invisible, using `setVisible(false)`, and can then be used by any interested class by repositioning it and calling `setVisible(true)`. Finally a `Timer` with a 1000 ms delay time is created and started.

The `mouseMoved()` method will be invoked when the mouse cursor moves over the applet's pane. It sets the `m_moved` flag to `true`, `m_counter` to -1, and stores the coordinates of the mouse cursor. Then this method hides the tooltip component if is visible.

The `actionPerformed()` method is called when the `Timer` fires events (see 2.6 for details). It implements the logic of displaying/hiding the tooltip window based on two instance variables: `m_moved` and `m_counter`:

```
if (m_moved || m_counter==0 || m_toolTip.getTipText()==null) {
  if (m_toolTip.isVisible())
    m_toolTip.setVisible(false);
  m_moved = false;
  return;
}
```

The above block of code is invoked when any one of the following statements are true:

1. Mouse cursor has been moved since the last time tick.
2. Counter has reached zero.
3. No tooltip text is set.

In any of these cases, the tooltip component is hidden (if previously visible), and the `m_moved` flag is set to `false`. Note that the `m_counter` variable remains unchanged.

```
if (m_counter < 0) {
  m_counter = 4;
  m_toolTip.setVisible(true);
  Dimension d = m_toolTip.getPreferredSize();
  m_toolTip.setBounds(m_lastX, m_lastY+20,
  d.width, d.height);
}
```

This block of code is responsible for displaying the tooltip component. It will be executed only when

m_counter is equal to -1 (set by mouseMoved()), and the m_moved flag is false (cleared by the previous code fragment). m_counter is set to four which determines the amount of time the tooltip will be displayed (4000ms in this example). Then we make the tooltip component visible and place it at the current mouse location with a vertical offset approximately equal to the mouse cursor's height. Note that this construction provides an arbitrary time delay between when mouse motion stops and the tooltip is displayed.

The last line of code in the actionPerformed() method is m_counter--, which decrements the counter each time tick until it reaches 0. As we saw above, once it reaches 0 the tooltip will be hidden.

Note: The actual delay time may vary from 1000ms to 2000ms since the mouse movements and time ticks are not synchronized. A more accurate and complex implementation could start a new timer after each mouse movement, as is done in Swing's ToolTipManager.

The following table illustrates how the m_counter and m_moved variables control this behavior.

Timer tick	m_moved flag	m_counter before	m_counter after	Comment
0	false	0	0	
1	true	-1	-1	Mouse moved between 0-th and 1-st ticks
2	false	-1	4	Tooltip is displayed
3	false	4	3	
4	false	3	2	
5	false	2	1	
6	false	1	0	
7	false	0	0	Tooltip is hidden
8	false	0	0	Waiting for the next mouse move

Running the Code

Figure 5.11 shows ButtonApplet3 running in Netscape Navigator 4.05 with the Java Plug-in. You can use the same HTML file as presented in the previous section. Move the mouse cursor over some non-rectangular component and note how it displays the proper tooltip message. This tooltip disappears after a certain amount of time or when the mouse is moved to a new location.

Chapter 6. Tabbed Panes

In this chapter:

- JTabbedPane
- Dynamically changeable tabbed pane
- Customized JTabbedPane and TabbedPaneUI delegate

6.1 JTabbedPane

class javax.swing.JTabbedPane

`JTabbedPane` is simply a stack of components in selectable layers. Each layer can contain one component which is normally a container. Tab extensions are used to move a given layer to the front of the tabbed pane veiw. These tab extensions are similar to labels in that they can have assigned text, an icon (as well as a disabled icon), background and foreground colors, and a tooltip.

To add a component to a tabbed pane we use one of its overloaded `add()` methods. This creates a new selectable tab and reorganizes the other tab extensions so the new one will fit (if necessary). We can also use the `addTab()` and `insertTab()` methods to do create new selectable layers. The `remove()` method takes a component as a parameter and removes the tab associated with that component, if any.

Tab extensions can reside to the north, south, east, or west of the tabbed pane's content. This can be specified using its `setTabPlacement()` method and passing one of the corresponding `SwingConstants` fields as parameter.

UI Guideline: Vertical or Horizontal Tabs?

When is it best to choose between vertical or horizontal tabs?

There are three possible rules of thumb to help make the decision whether to place tabs horizontally or vertically. Firstly, consider the nature of the data to be displayed, is vertical or horizontal space at a premium within the available display space? If for example you have a list with a single column but 200 entries then clearly vertical space is at a premium. If you have a table with only 10 entries but 15 columns then horizontal space is at a premium. Simply place the tabs where space is cheaper to obtain. In the first example with the long list, place the tabs vertically. When you place the tabs vertically, they use horizontal space which is available. In the second exmaple, place the tabs horizontally. When you place the tabs horizontally, you use vertical space which is available while horizontal space is fully taken by the table columns.

Another possibility is the number and size of the tabs. If you need to display perhaps 12 tabs, each with a long label, then it is unlikely that these will fit across the screen horizontally. In this case you are more likely to fit them by placing them vertically. Using space in these ways when introducing a tabbed pane, should minimise the introduction of scroll panes and maximise ease of use. Finally, consider the layout and mouse movements required for operating the software. If for example, your application uses a toolbar, then it may make sense to align the tabs close to the toolbar, thus minimising mouse movements between the toolbar buttons and the tabs. If you have a horizontal toolbar across the top of the screen, then choose a horizontal set of tabs across the top (North).

We can get/set the selected tab index at any given time using its `getSelectedIndex()` and `setSelectedIndex()` methods respecitively. We can get/set the component associated with the selected tab similarly using the `getSelectedComponent()` and `setSelectedComponent()` methods.

One or more `ChangeListeners` can be added to a `JTabbedPane`, which get registered with its model (an instance of `DefaultSingleSelectionModel` by default -- see chapter 12 for more about `SingleSelectionModel` and `DefaultSingleSelectionModel`). Whenever a new tab is selected the model will send out `ChangeEvents` to all registered `ChangeListeners`. The `stateChanged()` method of each listener is invoked, and in this way we can capture and perform any desired actions when the user selects any tab. `JTabbedPane` also fires `PropertyChangeEvents` whenever its model or tab placement properties change state.

UIGuideline: Transaction Boundaries and Tabbed Panes

If your using a tabbed pane within a dialog then the transaction boundary is normally clear. It will be an OK or CancelButton on the dialog. In this case it is obvious that the OK and Cancel buttons would lie outside the tabbed pane and in the dialog itself. This is an important point. Place action buttons which terminate a transaction outside the tabbed panes. If for example you had a tabbed pane which contained a Save and Cancel button within the first tab, is it clear that the Save and Cancel work across all tabs or only on the first. Actually, its ambiguous! To clearly define the transaction, define the buttons outside the tabbed pane then it should be clear to the User that any changes made to any tab will either be accepted/saved when OK/Save is pressed or discarded when Cancel is pressed. The action buttons apply across the complete set of tabs.

6.2 Dynamically changeable tabbed pane

We now turn to a JTabbedPane example applet demonstrating dynamically reconfigurable tab layout as well as the addition and removal of any number of tabs. A ChangeListener is attached to the tabbed pane to listen for tab selection events and display the currently selected tab index in a status bar. For enhanced feedback, audio clips are played when the tab layout changes and whenever a tab is added and removed.

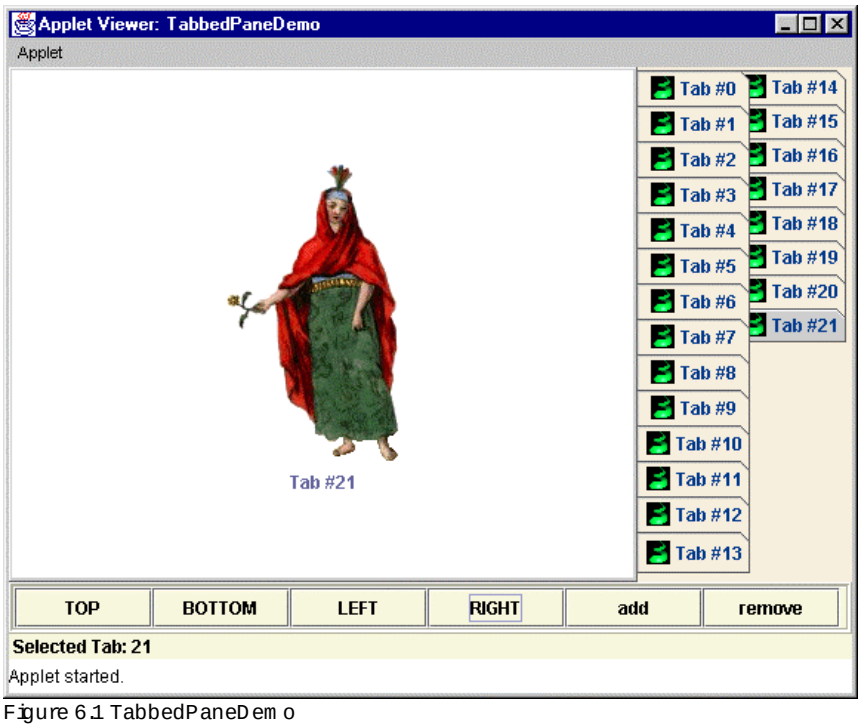


Figure 6.1 TabbedPaneDemo

<<file figure6-1.gif>>

The Code: TabbedPaneDemo.java
see \Chapter6\1

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

```
public class TabbedPaneDemo extends JApplet
 implements ActionListener
{
  private ImageIcon m_tabimage;
  private ImageIcon m_utsguy;
  private ImageIcon m_jfcgirl;
  private ImageIcon m_sbeguy;
  private ImageIcon m_tiger;
  private JTabbedPane m_tabbedPane;
  private JButton m_topButton;
  private JButton m_bottomButton;
  private JButton m_leftButton;
  private JButton m_rightButton;
  private JButton m_addButton;
  private JButton m_removeButton;
  private JLabel m_status;
  private JLabel m_loading;
  private AudioClip m_layoutsound;
  private AudioClip m_tabsound;

  public void init() {
    m_loading = new JLabel("Initializing applet...",
      SwingConstants.CENTER);
    getContentPane().add(m_loading);

    Thread initialize = new Thread() {
      public void run() {
        m_tabimage = new ImageIcon("tabimage.gif");
        m_utsguy = new ImageIcon("utsguy.gif");
        m_jfcgirl = new ImageIcon("jfcgirl.gif");
        m_sbeguy = new ImageIcon("sbeguy.gif");
        m_tiger = new ImageIcon("tiger.gif");
        m_tabbedPane = new JTabbedPane(SwingConstants.TOP);
        m_topButton = new JButton("TOP");
        m_bottomButton = new JButton("BOTTOM");
        m_leftButton = new JButton("LEFT");
        m_rightButton = new JButton("RIGHT");
        m_addButton = new JButton("add");
        m_removeButton = new JButton("remove");
        m_status = new JLabel();

        Color buttonColor = new Color(245,242,219);
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(1,6));
        JPanel lowerPanel = new JPanel();
        lowerPanel.setLayout(new BorderLayout());

        m_topButton.setBackground(buttonColor);
        m_bottomButton.setBackground(buttonColor);
        m_leftButton.setBackground(buttonColor);
        m_rightButton.setBackground(buttonColor);
        m_addButton.setBackground(buttonColor);
        m_removeButton.setBackground(buttonColor);
        m_topButton.addActionListener(TabbedPaneDemo.this);
        m_bottomButton.addActionListener(TabbedPaneDemo.this);
        m_leftButton.addActionListener(TabbedPaneDemo.this);
        m_rightButton.addActionListener(TabbedPaneDemo.this);
        m_addButton.addActionListener(TabbedPaneDemo.this);
        m_removeButton.addActionListener(TabbedPaneDemo.this);

        buttonPanel.add(m_topButton);
        buttonPanel.add(m_bottomButton);
```

```
        buttonPanel.add(m_leftButton);
        buttonPanel.add(m_rightButton);
        buttonPanel.add(m_addButton);
        buttonPanel.add(m_removeButton);
        buttonPanel.setBackground(buttonColor);
        buttonPanel.setOpaque(true);
        buttonPanel.setBorder(new CompoundBorder(
          new EtchedBorder(EtchedBorder.RAISED),
          new EtchedBorder(EtchedBorder.LOWERED)));

        lowerPanel.add("Center", buttonPanel);
        m_status.setHorizontalTextPosition(SwingConstants.LEFT);
        m_status.setOpaque(true);
        m_status.setBackground(buttonColor);
        m_status.setForeground(Color.black);
        lowerPanel.add("South", m_status);

        createTab();
        createTab();
        createTab();
        createTab();

        getContentPane().setLayout(new BorderLayout());
        m_tabbedPane.setBackground(new Color(245,232,219));
        m_tabbedPane.setOpaque(true);
        getContentPane().add("South", lowerPanel);
        getContentPane().add("Center", m_tabbedPane);
        m_tabbedPane.addChangeListener(new MyChangeListener());
        m_layoutsound = getAudioClip(getCodeBase(), "switch.wav");
        m_tabsound = getAudioClip(getCodeBase(), "tab.wav");

        getContentPane().remove(m_loading);
        getRootPane().revalidate();
        getRootPane().repaint();
      }
    };
    initialize.start();
  }

  public void createTab() {
    JLabel label = null;
    switch (m_tabbedPane.getTabCount()%4) {
      case 0:
        label = new JLabel("Tab #" + m_tabbedPane.getTabCount(),
          m_utsguy, SwingConstants.CENTER);
        break;
      case 1:
        label = new JLabel("Tab #" + m_tabbedPane.getTabCount(),
          m_jfcgirl, SwingConstants.CENTER);
        break;
      case 2:
        label = new JLabel("Tab #" + m_tabbedPane.getTabCount(),
          m_sbeguy, SwingConstants.CENTER);
        break;
      case 3:
        label = new JLabel("Tab #" + m_tabbedPane.getTabCount(),
          m_tiger, SwingConstants.CENTER);
        break;
    }
    label.setVerticalTextPosition(SwingConstants.BOTTOM);
    label.setHorizontalTextPosition(SwingConstants.CENTER);
    label.setOpaque(true);
```

```
    label.setBackground(Color.white);
    m_tabbedPane.addTab("Tab #" + m_tabbedPane.getTabCount(),
      m_tabimage, label);
    m_tabbedPane.setBackgroundAt(m_tabbedPane.getTabCount()-1,
      new Color(245,232,219));
    m_tabbedPane.setForegroundAt(m_tabbedPane.getTabCount()-1,
      new Color(7,58,141));
    m_tabbedPane.setSelectedIndex(m_tabbedPane.getTabCount()-1);
    setStatus(m_tabbedPane.getSelectedIndex());
  }

  public void killTab() {
    if (m_tabbedPane.getTabCount() > 0) {
      m_tabbedPane.removeTabAt(m_tabbedPane.getTabCount()-1);
      setStatus(m_tabbedPane.getSelectedIndex());
    }
    else
      setStatus(-1);
  }

  public void setStatus(int index) {
    if (index > -1)
      m_status.setText(" Selected Tab: " + index);
    else
      m_status.setText(" No Tab Selected");
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == m_topButton) {
      m_tabbedPane.setTabPlacement(SwingConstants.TOP);
      m_layoutsound.play();
    }
    else if(e.getSource() == m_bottomButton) {
      m_tabbedPane.setTabPlacement(SwingConstants.BOTTOM);
      m_layoutsound.play();
    }
    else if(e.getSource() == m_leftButton) {
      m_tabbedPane.setTabPlacement(SwingConstants.LEFT);
      m_layoutsound.play();
    }
    else if(e.getSource() == m_rightButton) {
      m_tabbedPane.setTabPlacement(SwingConstants.RIGHT);
      m_layoutsound.play();
    }
    else if(e.getSource() == m_addButton)
      createTab();
    else if(e.getSource() == m_removeButton)
      killTab();
    m_tabbedPane.revalidate();
    m_tabbedPane.repaint();
  }

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

  class MyChangeListener implements ChangeListener
  {
    public void stateChanged(ChangeEvent e) {
      setStatus(
        ((JTabbedPane) e.getSource()).getSelectedIndex());
      m_tabsound.play();
```

```
    }
  }
}
```

Understanding the Code

Class TabbedPaneDemo

`TabbedPaneDemo` extends `JApplet` and implements `ActionListener` (to listen for button events). Several instance variables are used:

`ImageIcon m_tabimage`: image used in each tab extension.

`ImageIcon m_utsguy, m_jfcgirl, m_sbeguy, m_tiger`: images used in tab containers.

`JTabbedPane m_tabbedPane`: the main tabbed pane.

`JButton m_topButton`: TOP tab alignment button.

`JButton m_bottomButton`: BOTTOM tab alignment button.

`JButton m_leftButton`: LEFT tab alignment button.

`JButton m_rightButton`: RIGHT tab alignment button.

`JButton m_addButton`: add tab button.

`JButton m_removeButton`: remove tab button.

`JLabel m_status`: status bar label.

Our `JTabbedPane`, `tabbedPane`, is created with `TOP` tab alignment. (Note: `TOP` is actually the default so this is really not necessary here. The default `JTabbedPane` constructor would do the same thing.)

The `init()` method organizes the buttons inside a `JPanel` using `GridLayout`, and associates `ActionListeners` with each one. We wrap all instantiation and GUI initialization processes in a separate thread and start it in the this method. This is because loading can take several seconds and it is best to allow the interface to be as responsive as possible during this time. We also provide an explicit visual cue to the user that the application is loading by placing an `"Initializing applet..."` label in the content pane where the tabbed pane will be placed once initialized. In this initialization, our `createTab()` method (discussed below) is called four times. We then add the panel containing the tabbed pane controller buttons, and our tabbed pane to the `contentPane`. Finally, an instance of `MyChangeListener` (see below) is attached to our tabbed pane to listen for tab selection changes.

The `createTab()` method is called whenever `m_addButton` is clicked. Based on the current tab count this method chooses between four `ImageIcons`, creates a `JLabel` containing the chosen icon, and adds a new tab containing that label. The `killTab()` method is called whenever `m_removeButton` is clicked to remove the tab with the highest index.

The `setStatus()` method is called each time a different tab is selected. The `m_status JLabel` is updated to reflect which tab is selected at all times.

The `actionPerformed()` method is called whenever any of the buttons are clicked. Clicking `m_topButton`, `m_bottomButton`, `m_leftButton`, or `m_rightButton` causes the tab layout of the `JTabbedPane` to change accordingly using the `setTabPlacement()` method. Each time one of these tab

layout buttons is clicked a WAV file is played. Similarly, when a tab selection change occurs a different WAV file is invoked. These sounds, `m_tabsound` and `m_layoutsound`, are loaded at the end of the `init()` method:

```
m_layoutsound = getAudioClip(getCodeBase(), "switch.wav");
m_tabsound = getAudioClip(getCodeBase(), "tab.wav");
```

Before the `actionPerformed()` method exits it revalidates the `JTabbedPane`. (If this revalidation is omitted we would see that a layout change caused by clicking one of our tab layout buttons will result in incorrect tabbed pane rendering.)

Class TabbedPaneDemo.MyChangeListener

`MyChangeListener` implements the `ChangeListener` interface. There is only one method that must be defined when implementing this interface: `stateChanged()`. This method can process `ChangeEvents` corresponding to when a tabbed pane's selected state changes. In our `stateChanged()` method we update the status bar in `TabbedPaneDemo` and play an appropriate tab switching sound:

```
public void stateChanged(ChangeEvent e) {
  setStatus(
    ((JTabbedPane) e.getSource()).getSelectedIndex());
  m_tabsound.play();
}
```

Running the Code:

Figure 6.1 shows `TabbedPaneDemo` in action. To deploy this applet the following simple HTML file is used (this is not Plug-in compliant):

```
<applet code=TabbedPaneDemo width=570 height=400> </applet>
```

Add and remove some tabs, and play with the tab layout to get a feel for how it works in different situations. Note that you can use your arrow keys to move from tab to tab (if the focus is currently on a tab), and remember to turn your speakers on for the sound effects.

Note: You may have problems with this applet if your system does not support wav files. If so, comment out the audio-specific code and recompile.

6.2.1 Interesting JTabbedPane characteristics

In cases where there is more than one row or column of tabs most of us are used to the following functionality: selecting a tab that is not in the frontmost row or column moves that row or column to the front. This does not occur in a `JTabbedPane` using the default MetalL&F as can be seen in the `TabbedPaneDemo` example above. However, this does occur when using the Windows, Motif, and Basic L&Fs. This feature was purposefully disabled in the MetalL&F (as can be verified in the `MetalTabbedPaneUI` source code).

UI Guideline: Avoid Multiple Rows of Tabs

As a general rule, you should seek to design for no more than a single row or column of tabs. There are three key reasons for this. The first is a cognitive reason: the user has trouble discerning what will happen with the multiple rows of tabs. With Windows L&F for example, the behaviour somewhat mimics the

behaviour of a rolladex filing card system. For some Users this mental model is clear and the behaviour is natural, for others it is simply confusing.

The second reason is a human factors / usability problem. When a rear set of tabs comes to the front, as with Windows Look and Feel, the position of all the other tabs changes. This means that the User has to discern the new position of a tab before visually selecting it and moving the mouse toward it. This has the effect of denying the User the ability to learn the positions of the tabs. Directional memory is a strong attribute and highly productive for usability. Thus it is always better to keep the tabs in the same position. This was the reason that the Sun and Apple designers chose to implement multiple tabs in this fashion! The final reason is a design problem. When a second or subsequent row or column of tabs is introduced, there is a resizing of the tabbed pane itself. Although, the layout manager will cope with this, it may not look visually satisfactory when completed. The size of the tabbed pane becomes dependant on the ability to render the tabs in a given space. Those who remember the OS2 Warp UI will recall that the designers avoided this problem by allowing only a single row of tabs and the ability to scroll them if they didn't fit into the given space. So far no one has implemented a Swing L&F with this style of tabbed pane.

6.3 Customized JTabbedPane and TabbedPaneUI delegate

Although we intend to save most of our discussion of customizing UI delegates for chapter 21, building fancy-looking tabs is too tempting to pass up, and this example may satisfy your UI customization appetite for now. You can use the techniques shown here to implement custom UI delegates for almost any Swing component. However, there will be major differences from component to component, and this will almost always involve referencing the Swing source code.

First we will build a customized `JTabbedPane` with two properties: a background image used for each tab, and a background image used for the tabbed pane itself. We will then build a subclass of `BasicTabbedPaneUI` for use with our customized `JTabbedPane`, and design it so that it paints the tab background image on each tab. This will require modification of its `paint()` method. Our custom delegate will also be responsible for painting the tabbed pane background. As we learned in chapter 2 in our discussion of painting, a UI delegate's `update()` method is used for filling the background of opaque components and then it should pass control to `paint()`. Both images need to be accessible from our customized `JTabbedPane` using `set` and `get` accessors. (In keeping with the concept of UI delegates not being bound to specific component instances, it would not be a good idea to assign these images as UI delegate properties -- unless we were building an image specific L&F.)

`BasicTabbedPaneUI` is the second largest (in terms of source code) and complex UI delegate. It contains, among other things, its own custom layout manager, `TabbedPaneLayout`, and a long rendering routine spread out over several different methods. As complex as it is, understanding its inner workings is not required to build on top of it. Since we are concerned only with how it does its rendering, we can narrow our focus considerably. To start, its `paint()` method calls the `paintTab()` method which is responsible for painting each tab. This method calls several other methods to perform various aspects of this process (see BasicTabbedPaneUI.java). Briefly, and in order, these methods are:

```
paintTabBackground()
paintTabBorder()
layoutLabel()
paintText()
paintIcon()
paintFocusIndicator()
```

By overriding any combination of these methods we can control the tab rendering process however we like.

Our customized TabbedPaneUI, which we'll call ImageTabbedPaneUI, overrides the paintTabBackground() method to construct tabs with background images.

Figure 6.2 TabbedPaneDemo with custom tab rendering, LEFT layout
<<file figure6-2.gif>>

Figure 6.3 TabbedPaneDemo with custom tab rendering, BOTTOM layout
<<file figure6-3.gif>>

The Code: TabbedPaneDemo.java
see \Chapter6\2

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.plaf.TabbedPaneUI;
import javax.swing.plaf.ComponentUI;
import javax.swing.plaf.basic.BasicTabbedPaneUI;

public class TabbedPaneDemo extends JApplet
implements ActionListener
{
  // Unchanged code from section 6.2

  public void init() {
    m_loading = new JLabel("Initializing applet...",
      SwingConstants.CENTER);
    getContentPane().add(m_loading);

    Thread initialize = new Thread() {
      public void run() {
        // Unchanged code from section 6.2

        m_tabbedPane = new ImageTabbedPane(
          new ImageIcon("bloo.gif"),
          new ImageIcon("bubbles.jpg"));
```

```
        // Unchanged code from section 6.2
      }
    };
    initialize.start();
  }

  public void createTab() {
    // Unchanged code from section 6.2

    label.setBackground(Color.white);
    m_tabbedPane.addTab("Tab #" + m_tabbedPane.getTabCount(),
      m_tabimage, label);
    m_tabbedPane.setForegroundAt(m_tabbedPane.getTabCount()-1,
      Color.white);
    m_tabbedPane.setSelectedIndex(m_tabbedPane.getTabCount()-1);
    setStatus(m_tabbedPane.getSelectedIndex());
  }

  // Unchanged code from section 6.2
}

class ImageTabbedPane extends JTabbedPane
{
  // Display properties
  private Image m_tabBackground;
  private Image m_paneBackground;

  public ImageTabbedPane(ImageIcon tabBackground,
   ImageIcon paneBackground) {
    m_tabBackground = tabBackground.getImage();
    m_paneBackground = paneBackground.getImage();
    setUI((ImageTabbedPaneUI) ImageTabbedPaneUI.createUI(this));
  }

  public void setTabBackground(Image i) {
    m_tabBackground = i;
    repaint();
  }

  public void setPaneBackground(Image i) {
    m_paneBackground = i;
    repaint();
  }

  public Image getTabBackground() {
    return m_tabBackground;
  }

  public Image getPaneBackground() {
    return m_paneBackground;
  }
}

class ImageTabbedPaneUI extends BasicTabbedPaneUI
{
  private Image m_image;

  public static ComponentUI createUI(JComponent c) {
    return new ImageTabbedPaneUI();
  }

  public void update(Graphics g, JComponent c) {
```

```
    if (c instanceof ImageTabbedPane) {
      Image paneImage = ((ImageTabbedPane) c).getPaneBackground();
      int w = c.getWidth();
      int h = c.getHeight();
      int iw = paneImage.getWidth(tabPane);
      int ih = paneImage.getHeight(tabPane);
      if (iw > 0 && ih > 0) {
        for (int j=0; j < h; j += ih) {
          for (int i=0; i < w; i += iw) {
            g.drawImage(paneImage,i,j, tabPane);
          }
        }
      }
    }
    paint(g,c);
  }

  public void paint(Graphics g, JComponent c) {
    if (c instanceof ImageTabbedPane)
      m_image = ((ImageTabbedPane) c).getTabBackground();
    super.paint(g,c);
  }


  protected void paintTabBackground(Graphics g, int tabPlacement,
   int tabIndex, int x, int y, int w, int h, boolean isSelected )
  {
    Color tp = tabPane.getBackgroundAt(tabIndex);
    switch(tabPlacement) {
      case LEFT:
        g.drawImage(m_image, x+1, y+1, (w-2)+(x+1), (y+1)+(h-3),
          0, 0, w, h, tp, tabPane);
        break;
      case RIGHT:
        g.drawImage(m_image, x, y+1, (w-2)+(x), (y+1)+(h-3),
          0, 0, w, h, tp, tabPane);
        break;
      case BOTTOM:
        g.drawImage(m_image, x+1, y, (w-3)+(x+1), (y)+(h-1),
          0, 0, w, h, tp, tabPane);
        break;
      case TOP:
        g.drawImage(m_image, x+1, y+1, (w-3)+(x+1), (y+1)+(h-1),
          0, 0, w, h, tp, tabPane);
    }
  }
}
```

Understanding the Code

Class TabbedPaneDemo

We have replaced the `JTabbedPane` instance with an instance of our custom `ImageTabbedPane` class. This class's constructor takes two `ImageIcons` as parameters. The first represents the tab background and the second represents the pane background:

```
        m_tabbedPane = new ImageTabbedPane(
          new ImageIcon("bloo.gif"),
          new ImageIcon("bubbles.jpg"));
```

We've also modified the `createTab()` method to make the tab foreground text white because the tab background image we are using is dark.

Class ImageTabbedPane

This `JTabbedPane` subclass is responsible for keeping two `Image` variables that represent our custom tab background and pane background properties. The constructor takes two `ImageIcons` as parameters, extracts the `Images`, and assigns them to these variables. It calls `setUI()` to enforce the use of our custom `ImageTabbedPaneUI` delegate. This class also defines `set()` and `get()` accessors for both `Image` properties.

Class ImageTabbedPaneUI

This class extends `BasicTabbedPaneUI` and maintains an `Image` variable for use in painting tab backgrounds. The `creatUI()` method is overridden to return an instance of itself (remember that this method was used in the `ImageTabbedPane` constructor):

```
public static ComponentUI createUI(JComponent c) {
  return new ImageTabbedPaneUI();
}
```

The `update()` method, as we discussed above, is responsible for filling the pane background and then calling `paint()`. So we override it to perform a tiling of the pane background image. First we grab a reference to that image check its dimensions. If any are found to be `0` or less we do not go through with painting procedure. Otherwise we tile a region of the pane equal in size to the current size of the `ImageTabbedPane` itself. Note that in order to get the width and height of an `Image` we use the following method calls:

```
int iw = paneImage.getWidth(tabPane);
int ih = paneImage.getHeight(tabPane);
```

(The `tabPane` reference is `protected` in `BasicTabbedPaneUI` and we use it for the `ImageObsever` here--it is a reference to the `JTabbedPane` component being rendered.) Once the tiling is done we call `paint()`.

The `paint()` method simply assigns this class's tab background `Image` variable to that of the `ImageTabbedPane` being painted. It then calls its superclass's `paint()` method which we do not concern ourselves with--let alone fully understand! What we do know is that the superclass's `paint()` method calls its `paintTab()` method, which in turn calls its `paintTabBackground()` method, which is actually responsible for filling the background of each tab. So we overrode this method to use our tab background `Image` instead. This method specifies four cases based on which layout mode the `JTabbedPane` is in: `LEFT`, `RIGHT`, `BOTTOM`, `TOP`. These cases were obtained directly from the `BasicTabbedPaneUI` source code and we did not modify their semantics at all. What we did modify is what each case does.

We use the `drawImage()` method of the `awt.Graphics` class is used to fill a rectangular area defined by the first four `int` parameters (which represent two points). The second four `int` parameters specify an area of the `Image` (passed in as the first parameter) that will be scaled and mapped onto the above rectangular area. The last parameter represents an `ImageObserver` instance.

Note: A more professional implementation would tile the images used in painting the tab background--this implementation assumes that a large enough image will be used to fill a tab's background of any size.

Running the Code:

Figures 6.2 and 6.3 show `TabbedPaneDemo` in action. To deploy this applet you can use the same HTML file that was used in the previous example. Try out all different tab positions.

Note that the images used are only used within the `paint()` and `update()` methods of our UI delegate. In this way we keep with Swing's UI delegate / component separation. One instance of `ImageTabbedPaneUI` could be assigned to multiple `ImageTabbedPane` instances using different `Images`, and there would be no conflicts. Also note that the images are loaded only once (when an instance of `ImageTabbedPaneUI` is created) so the resource overhead is minimal.

`UI Guideline: Custom Tabs Look & Feel`

After reading Chapter 21 on Custom Look and Feel, you may like to reconsider customising the Tabbed Pane styles. There are many possibilities which are worthy of consideration. For example the OS2 Warp style of scrolling tabs. This is particularly good where you cannot afford to give away screen space for an additional row or column of tabs. Another possibility is to fix the size of a tab. Currently tabs size of the width of the label which is largely controlled by the length of text and font selected. This lends greater graphical weight to tabs with long labels. The size of the visual target is greater and therefore tabs with long labels are easier to select than tabs with short labels. You may like to consider a tab which fixes a standard size, thus equalising the size of the target and avoiding biasing the usability toward the longer labelled tabs. Further possibilities are tabs with colour and tabs with icons to indicate useful information such as "updated" or "new" or "mandatory input required".

Chapter 7. Scrolling Panes

In this chapter:

- JScrollPane
- Grab-and-drag scrolling
- Programmatic scrolling

7.1 JScrollPane

class javax.swing.JScrollPane

Using `JScrollPane` is normally very simple. Any component or container can be placed in a `JScrollPane` and scrolled. Simply create a `JScrollPane` by passing its constructor the component you'd like to scroll:

```
JScrollPane jsp = new JScrollPane(myLabel);
```

Normally, our use of `JScrollPane` will not be much more extensive than the one line of code shown above. The following is a simple `JScrollPane` demo application. Figure 7.1 illustrates:

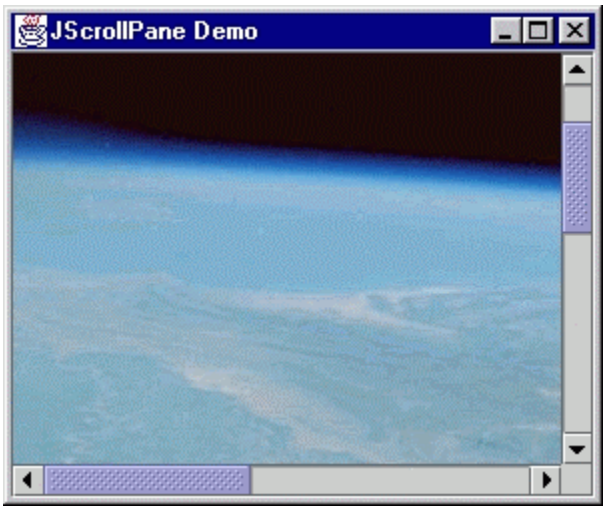


Figure 7.1 JScrollPane demo
<<file figure7-1.gif>>

The Code: ScrollPaneDemo.java
see \Chapter7\1

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

public class ScrollPaneDemo extends JFrame
{
  public ScrollPaneDemo() {
```

```
    super("JScrollPane Demo");
    ImageIcon ii = new ImageIcon("earth.jpg");
    JScrollPane jsp = new JScrollPane(new JLabel(ii));
    getContentPane().add(jsp);
    setSize(300,250);
    setVisible(true);
  }

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

When you run this example try scrolling by pressing or holding down any of the scrollbar buttons. You will find this unacceptably slow because the scrolling occurs one pixel at a time. We will see how to control this shortly.

Many components use a `JScrollPane` internally to display their contents, such as `JComboBox` and `JList`. We are normally expected to place all multi-line text components inside scroll panes (although this is not default behavior).

UI Guideline: Using ScrollPanes

For many applications it is best to avoid the introduction of a scrollpane and concentrate on puting the required data on a screen such that scrolling is unnecessary. However, this is not always possible. When you do need to introduce scrolling put some thought into the type of data and application. If possible try to introduce scrolling in only 1 direction. For example, with text documents, western culture has been used to scrolling vertically since Egyptian times. Usability studies for World Wide Web pages have shown that readers can find data quickly when vertically scrolling. Scrolling horizontally, however, is labourious and difficult with text. Try to avoid it. With visual information, e.g. tables of information, it may be more appropriate for horizontal scrolling, but try to avoid horizontal and vertical scrolling if possible.

We can access a `JScrollPane`'s scroll bars directly with its `getXXScrollBar()` and `setXXScrollBar()` methods, where `XX` is either `HORIZONTAL` or `VERTICAL`.

Reference: In chapter 13 we'll talk more about `JScrollBars`

7.1.1 Scrollbar policies

abstract interface javax.swing.ScrollPaneConstants

We can specify specific policies for when and when not to display a `JScrollPane`'s horizontal and vertical scrollbars. We simply use its `setVerticalScrollBarPolicy()` and `setHorizontalScrollBarPolicy()` methods, provding one of three constants defined in the `ScrollPaneConstants` interface:

```
HORIZONTAL_SCROLLBAR_AS_NEEDED
HORIZONTAL_SCROLLBAR_NEVER
HORIZONTAL_SCROLLBAR_ALWAYS
VERTICAL_SCROLLBAR_AS_NEEDED
VERTICAL_SCROLLBAR_NEVER
VERTICAL_SCROLLBAR_ALWAYS
```

For example, to enforce the display of the vertical scrollbar at all times and always keep the horizontal scrollbar hidden, we could do the following:

```
jsp.setHorizontalScrollBarPolicy(
  ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

jsp.setVerticalScrollBarPolicy(
  ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);
```

7.1.2 JViewport

class javax.swing.JViewport

The `JViewport` class is the container that is really responsible for displaying a specific visible region of the component in `JScrollPane`. We can set/get a viewport's view (the component it contains) using its `setView()` and `getView()` methods. We can control how much of this component `JViewport` displays by setting its extent size to a specified `Dimension` using its `setExtentSize()` method. We can also specify where the origin (upper left corner) of a `JViewport` should begin displaying its contained component by providing specific coordinates (as a `Point`) of the contained component to the `setViewPosition()` method. In fact, when we scroll a component in a `JScrollPane` this view position is constantly being changed by the scrollbars.

Note: `JViewport` enforces a view position that lies within the view component only. We cannot set negative or extremely large view positions (as of JDK 1.2.2 we can set negative view positions). However, since the view position is the upper right hand corner of the viewport, we are still allowed to set the view position such that only part of the viewport is filled. We will show how to watch for this, and stop it from happening, in some of the examples below.

Whenever a change is made to the position or size of the visible portion of the view, `JViewport` fires `ChangeEvents`. We can register `ChangeListeners` to capture these events using `JViewport`'s `addChangeListener()` method. These are the only events that are associated with `JScrollPane` by default. For instance, whenever we scroll using `JScrollPane`'s scrollbars, its main viewport, as well as its row and column header viewports (see below), will each fire `ChangeEvents`.

The visible region of `JViewport`'s view can be retrieved as a `Rectangle` or `Dimension` instance using the `getViewRect()` and `getViewSize()` methods respectively. This will give us the current view position as well as the extent width and hieght. The view position alone can be retrieved with `getViewPosition()`, which returns a `Point` instance. To remove a component from `JViewport` we use its `remove()` method.

We can translate specific `JViewport` coordinates to the coordinates of its contained component by passing a `Point` instance to its `toViewCoordinates()` method. We can do the same for a region by passing `toViewCoordinates()` a `Dimension` instance. We can also manually specify the visible region of the view component by passing a `Dimension` instance to `JViewport`'s `scrollRectToVisible()` method.

We can retreive `JScrollPane`'s main `JViewport` by calling its `getViewport()` method, or assign it a new one using `setViewport()`. We can replace the component in this viewport through `JScrollPane`'s `setViewportView()` method, but there is no `getViewportView()` counterpart. Instead we must first access its `JScrollPane`'s `JViewport` by calling `getViewport()`, and then call `getView()` on that (as discussed above). Typically, to access a `JScrollPane`'s main child component we would do the following:

```
Component myComponent = jsp.getViewport().getView();
```

7.1.3 ScrollPaneLayout

class javax.swing.ScrollPaneLayout

By default `JScrollPane`'s layout is managed by an instance of `ScrollPaneLayout`. `JScrollPane` can contain up to nine components and it is `ScrollPaneLayout`'s job to make sure that they are positioned correctly. These components are:

- A `JViewport` containing the main component to be scrolled.
- A `JViewport` used as the row header. This viewport's view position changes vertically in sync with the main viewport.
- A `JViewport` used as the column header. This viewport's view position changes horizontally in sync with the main viewport.
- Four components for placement in each corner of the scrollpane.
- Two `JScrollBars` for vertical and horizontal scrolling.

The corner components will only be visible if the scrollbars and headers surrounding them are also visible. To assign a component to a corner position we can call `JScrollPane`'s `setCorner()` method. This method takes both a `String` and a component as parameters. The `String` is used to identify which corner this component is to be placed in, and is recognized by `ScrollPaneLayout`. In fact `ScrollPaneLayout` identifies each `JScrollPane` component with a unique `String`. Figure 7.2 illustrates:

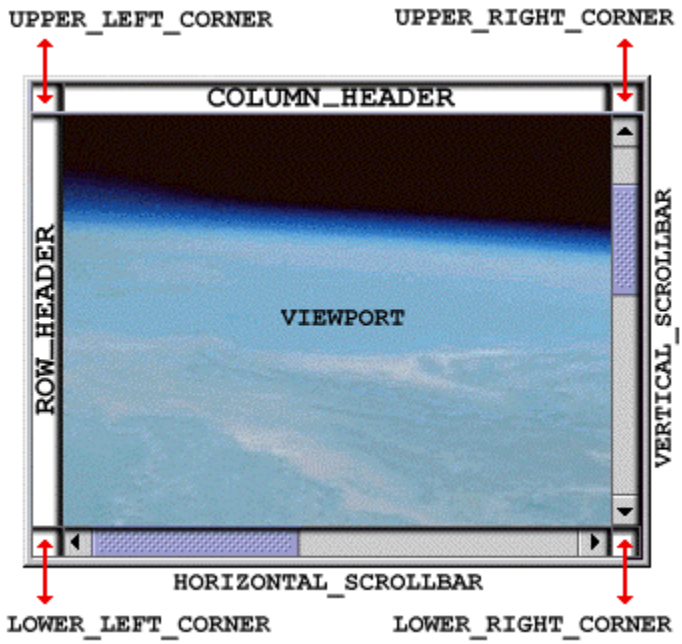


Figure 7.2 JScrollPane components as identified by ScrollPaneLayout
<<file figure7-2.gif>>

To assign `JViewports` as the row and column headers we use `JScrollPane`'s `setRowHeader()` and `setColumnHeader()` methods respectively. We can also avoid the creation of a `JViewport` ourselves by passing the component to be placed in the row or column viewport to `JScrollPane`'s `setRowHeaderView()` or `setColumnHeaderView()` methods.

Because `JScrollPane` is most often used to scroll images, the most obvious use for the row and column

headers is to function as some sort of ruler. Here we present a basic example showing how to populate each corner with a label and create some simple rulers for the row and column headers that display ticks every 30 pixels, and render themselves based on their current viewport position. Figure 7.3 illustrates:

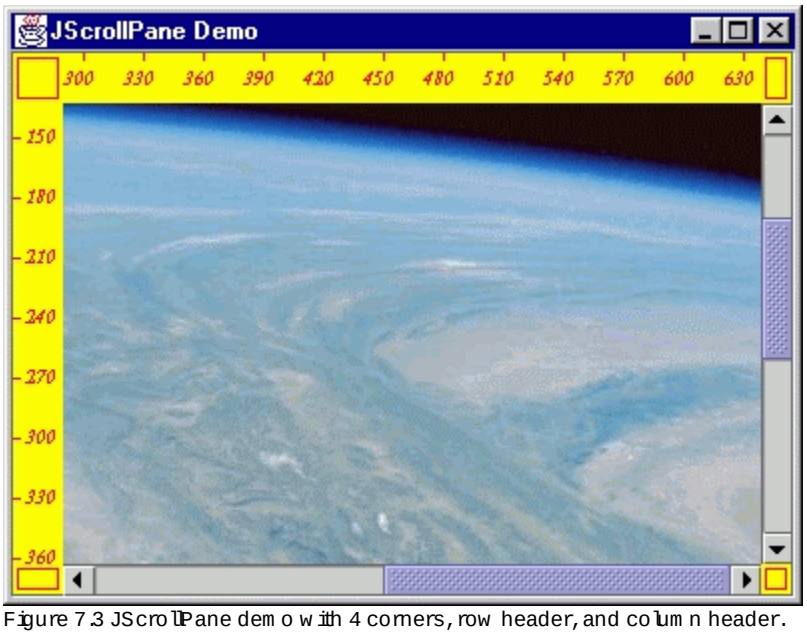


Figure 7.3 JScrollPane demo with 4 corners, row header, and column header.
<<file figure7-3.gif>>

The Code: HeaderDemo.java
see \Chapter7\2

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

public class HeaderDemo extends JFrame
{
  public HeaderDemo() {
    super("JScrollPane Demo");
    ImageIcon ii = new ImageIcon("earth.jpg");
    JScrollPane jsp = new JScrollPane(new JLabel(ii));

    JLabel[] corners = new JLabel[4];
    for(int i=0;i<4;i++) {
      corners[i] = new JLabel();
      corners[i].setBackground(Color.yellow);
      corners[i].setOpaque(true);
      corners[i].setBorder(BorderFactory.createCompoundBorder(
        BorderFactory.createEmptyBorder(2,2,2,2),
        BorderFactory.createLineBorder(Color.red, 1)));
    }

    JLabel rowheader = new JLabel() {
      Font f = new Font("Serif",Font.ITALIC | Font.BOLD,10);
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Rectangle r = g.getClipBounds();
        g.setFont(f);
        g.setColor(Color.red);
        for (int i = 30-(r.y % 30);i<r.height;i+=30) {
          g.drawLine(0, r.y + i, 3, r.y + i);
          g.drawString("" + (r.y + i), 6, r.y + i + 3);
```

```
        }
      }
      public Dimension getPreferredSize() {
        return new Dimension(
         25, (int)label.getPreferredSize().getHeight());
      }
    };
    rowheader.setBackground(Color.yellow);
    rowheader.setOpaque(true);

    JLabel columnheader = new JLabel() {
      Font f = new Font("Serif",Font.ITALIC | Font.BOLD,10);
      public void paintComponent(Graphics g) {
        super.paintComponent(g);
        Rectangle r = g.getClipBounds();
        g.setFont(f);
        g.setColor(Color.red);
        for (int i = 30-(r.x % 30);i<r.width;i+=30) {
          g.drawLine(r.x + i, 0, r.x + i, 3);
          g.drawString("" + (r.x + i), r.x + i - 10, 16);
        }
      }
      public Dimension getPreferredSize() {
        return new Dimension(
         (int)label.getPreferredSize().getWidth(),25);
      }
    };
    columnheader.setBackground(Color.yellow);
    columnheader.setOpaque(true);

    jsp.setRowHeaderView(rowheader);
    jsp.setColumnHeaderView(columnheader);
    jsp.setCorner(JScrollPane.LOWER_LEFT_CORNER, corners[0]);
    jsp.setCorner(JScrollPane.LOWER_RIGHT_CORNER, corners[1]);
    jsp.setCorner(JScrollPane.UPPER_LEFT_CORNER, corners[2]);
    jsp.setCorner(JScrollPane.UPPER_RIGHT_CORNER, corners[3]);

    getContentPane().add(jsp);
    setSize(400,300);
    setVisible(true);
  }

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

Notice that the row and column headers use the graphics clipping area in their `paintComponent()` routine for optimal efficiency. We also override the `getPreferredSize()` method so that the proper width (for the row header) and height (for the column header) will be used by `ScrollPaneLayout`. The other dimensions are obtained by simply grabbing the label's preferred size, as they are completely controlled by `ScrollPaneLayout`.

We are certainly not limited to labels for corners and row headers or within the main viewport itself. As we mentioned in the beginning of this chapter, any component can be placed in a `JViewport` and scrolled in a `JScrollPane`.

7.1.4. The Scrollable interface

The `Scrollable` interface describes five methods that allow us to customize how `JScrollPane` scrolls its

contents. Specifically, by implementing this interface we can specify how many pixels are scrolled when a scrollbar button or scrollbar paging area (the empty region between the scrollbar thumb and the buttons) is pressed. There are two methods that control this functionality: `getScrollableBlockIncrement()` and `getScrollableUnitIncrement()`. The former is used to return the amount to scroll when a scrollbar paging area is pressed, the latter is used when the button is pressed.

Reference: In text components, these two methods are implemented so that scrolling will move one line of text at a time. (`JTextComponent` implements the `Scrollable` interface.)

The other three methods of this interface involve `JScrollPane`'s communication with the main viewport. The `getScrollableTracksViewportWidth()` and `getScrollableTracksHeight()` methods can return true to disable scrolling in the horizontal or vertical direction respecively. Normally these just return `false`. The `getPreferredSize()` method is supposed to return the preferred size of the viewport that will contain this component (the component implementing the `Scrollable` interface). Normally we just return the preferred size of the component for this.

The following code shows how to implement the `Scrollable` interface to create a custom `JLabel` whose unit and block increments will be `10` pixels. As we saw in the example in the beginning of this chapter, scrolling one pixel at a time is tedious at best. Increasing this to a `10` pixel increment provides a more natural feel.

The Code: ScrollableDemo.java
see \Chapter7\3

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

public class ScrollableDemo extends JFrame
{
  public ScrollableDemo() {
    super("JScrollPane Demo");
    ImageIcon ii = new ImageIcon("earth.jpg");
    JScrollPane jsp = new JScrollPane(new MyScrollableLabel(ii));
    getContentPane().add(jsp);
    setSize(300,250);
    setVisible(true);
  }

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

class MyScrollableLabel extends JLabel implements Scrollable
{
  public MyScrollableLabel(ImageIcon i){
    super(i);
  }

  public Dimension getPreferredScrollableViewportSize() {
    return getPreferredSize();
  }

  public int getScrollableBlockIncrement(Rectangle r,
    int orietation, int direction) {
      return 10;
  }
```

```
  public boolean getScrollableTracksViewportHeight() {
    return false;
  }

  public boolean getScrollableTracksViewportWidth() {
    return false;
  }

  public int getScrollableUnitIncrement(Rectangle r,
    int orientation, int direction) {
      return 10;
  }
}
```

7.2 Grab-and-drag scrolling

Many paint programs and document readers (such as Adobe Acrobat) support grab-and-drag scrolling. This is the ability to click on an image and drag it in any direction with the mouse. It is fairly simple to implement, however, we must take care to make the operation smooth without allowing the view to be scrolled past its extremities. `JViewport` takes care of the negative direction for us, as it does not allow the view position coordinates to be less than 0. But it will allow us to change the view position to very large values, which can result in the viewport displaying a portion of the view smaller than the viewport itself.

Note: As of JDK 1.2.2 we are allowed to specify negative view position coordinates.

We've modified the simple example from the beginning of this chapter to support grab-and-drag scrolling.

The Code: GrabAndDragDemo.java
see \Chapter7\4

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

public class GrabAndDragDemo extends JFrame
{
  public GrabAndDragDemo() {
    super("Grab-and-drag Demo");
    ImageIcon ii = new ImageIcon("earth.jpg");
    JScrollPane jsp = new JScrollPane(new GrabAndScrollLabel(ii));
    getContentPane().add(jsp);
    setSize(300,250);
    setVisible(true);

    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(l);
  }

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

```
class GrabAndScrollLabel extends JLabel
{
  public GrabAndScrollLabel(ImageIcon i){
    super(i);
    MouseInputAdapter mia = new MouseInputAdapter() {
      int m_XDifference, m_YDifference;
      Container c;

      public void mouseDragged(MouseEvent e) {
        c = GrabAndScrollLabel.this.getParent();
        if (c instanceof JViewport) {
          JViewport jv = (JViewport) c;
          Point p = jv.getViewPosition();
          int newX = p.x - (e.getX()-m_XDifference);
          int newY = p.y - (e.getY()-m_YDifference);

          int maxX = GrabAndScrollLabel.this.getWidth()
            - jv.getWidth();
          int maxY = GrabAndScrollLabel.this.getHeight()
            - jv.getHeight();
          if (newX < 0)
            newX = 0;
          if (newX > maxX)
            newX = maxX;
          if (newY < 0)
            newY = 0;
          if (newY > maxY)
            newY = maxY;

          jv.setViewPosition(new Point(newX, newY));
        }
      }

      public void mousePressed(MouseEvent e) {
        setCursor(Cursor.getPredefinedCursor(
          Cursor.MOVE_CURSOR));
        m_XDifference = e.getX();
        m_YDifference = e.getY();
      }

      public void mouseReleased(MouseEvent e) {
        setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
      }
    };
    addMouseMotionListener(mia);
    addMouseListener(mia);
  }
}
```

Understanding the Code:

Class GrabAndScrollLabel

This class extends `JLabel` and overrides the `JLabel(Imageicon ii)` constructor. The `GrabAndScrollLabel` constructor starts by calling the super class version and then proceeds to set up a `MouseInputAdapter`. This adapter is the heart of the `GrabAndScrollLabel` class.

The adapter uses three variables:

`int m_XDifference`: x coordinate saved on a mouse press event and used for dragging horizontally

`int m_YDifference`: y coordinate saved on a mouse press event and used for dragging vertically

`Container c`: used to hold a local reference to the parent container in the `mouseDragged()` method.

The `mousePressed()` method changes the cursor to `MOVE_CURSOR`, and stores the event coordinates in variables `m_XDifference` and `m_YDifference` to be use in `mouseDragged()`.

The `mouseDragged()` method first grabs a reference to the parent and checks if it is a `JViewport`. If it isn't we do nothing. If it is we store the current view position and calculate the new view position the drag will bring us into:

```
Point p = jv.getViewPosition();
int newX = p.x - (e.getX()-m_XDifference);
int newY = p.y - (e.getY()-m_YDifference);
```

When dragging components, normally this would be enough (as we will see in future chapters), however we must make sure that we do not move this label in such a way that the viewport is not filled by it. So we calculate the maximum allowable x and y coordinates by subtracting the viewport dimensions from the size of this label (since the view position coordinates are upper-left hand corner):

```
int maxX = GrabAndScrollLabel.this.getWidth()
  - jv.getWidth();
int maxY = GrabAndScrollLabel.this.getHeight()
  - jv.getHeight();
```

The remainder of this method compares the `newX` and `newY` values with the `maxX` and `maxY` values, and adjusts the view position accordingly. If `newX` or `newY` is ever greater than the `maxX` or `maxY` values respectively, we use the max values instead. If `newX` or `newY` is ever less than `0` (which can happen only with JDK 1.2.2), we use `0` instead. This is necessary to allow smooth scrolling in all situations.

7.3 Scrolling programmatically

We are certainly not required to use a `JScrollPane` for scrolling. We can place a component in a `JViewport` and control the scrolling ourselves if we like. This is what `JViewport` was designed for, it just happens to be used by `JScrollPane` as well. We've constructed this example to show how to implement our own scrolling in a `JViewport`. Four buttons are used for scrolling. We enable and disable these buttons based on whether the view component is at any of its extremities. These buttons are assigned keyboard mnemonics which we can use as an alternative to clicking.

This example also shows how to use a `ChangeListener` to capture `ChangeEvents` that are fired when the `JViewport` changes state. The reason we need to capture these events is that when our viewport is resized bigger than its view component child, the scrolling buttons should become disabled. If these buttons are disabled and the viewport is then resized so that it is no longer bigger than its child view component, the buttons should then become enabled. It is quite simple to capture and process these events as you will see below. (As with all of the examples we have presented, it may help if you run this example before stepping through the code.)

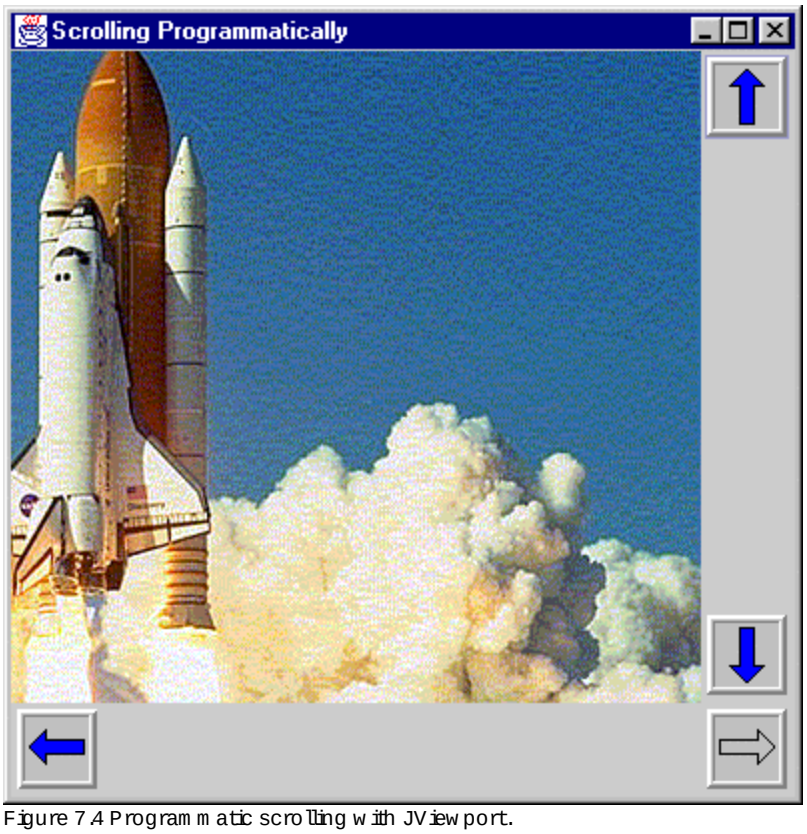


Figure 7.4 Programmatic scrolling with JViewport.

<<file figure7-4.gif>>

The Code: ButtonScroll.java

see \Chapter7\5

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

import javax.swing.*;
import javax.swing.event.*;

public class ButtonScroll extends JFrame
{
  protected JViewport m_viewport;
  protected JButton m_up;
  protected JButton m_down;
  protected JButton m_left;
  protected JButton m_right;

  protected int m_pgVert;
  protected int m_pgHorz;

  public ButtonScroll() {
    super("Scrolling Programmatically");
    setSize(400, 400);
    getContentPane().setLayout(new BorderLayout());

    ImageIcon shuttle = new ImageIcon("shuttle.gif");
    m_pgVert = shuttle.getIconHeight()/5;
    m_pgHorz = shuttle.getIconWidth()/5;
    JLabel lbl = new JLabel(shuttle);
```

```
    m_viewport = new JViewport();
    m_viewport.setView(lbl);
    m_viewport.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent e) {
        enableButtons(
          ButtonScroll.this.m_viewport.getViewPosition());
      }
    });
    getContentPane().add(m_viewport, BorderLayout.CENTER);

    JPanel pv = new JPanel(new BorderLayout());
    m_up = createButton("up", 'u');
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        movePanel(0, -1);
      }
    };
    m_up.addActionListener(lst);
    pv.add(m_up, BorderLayout.NORTH);

    m_down = createButton("down", 'd');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        movePanel(0, 1);
      }
    };
    m_down.addActionListener(lst);
    pv.add(m_down, BorderLayout.SOUTH);
    getContentPane().add(pv, BorderLayout.EAST);

    JPanel ph = new JPanel(new BorderLayout());
    m_left = createButton("left", 'l');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        movePanel(-1, 0);
      }
    };
    m_left.addActionListener(lst);
    ph.add(m_left, BorderLayout.WEST);

    m_right = createButton("right", 'r');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        movePanel(1, 0);
      }
    };
    m_right.addActionListener(lst);
    ph.add(m_right, BorderLayout.EAST);
    getContentPane().add(ph, BorderLayout.SOUTH);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
    movePanel(0, 0);
  }

  protected JButton createButton(String name, char mnemonics) {
```

```
    JButton btn = new JButton(new ImageIcon(name+"1.gif"));
    btn.setPressedIcon(new ImageIcon(name+"2.gif"));
    btn.setDisabledIcon(new ImageIcon(name+"3.gif"));
    btn.setToolTipText("Move "+name);
    btn.setBorderPainted(false);
    btn.setMargin(new Insets(0, 0, 0, 0));
    btn.setContentAreaFilled(false);
    btn.setMnemonic(mnemonics);
    return btn;
  }

  protected void movePanel(int xmove, int ymove) {
    Point pt = m_viewport.getViewPosition();
    pt.x += m_pgHorz*xmove;
    pt.y += m_pgVert*ymove;

    pt.x = Math.max(0, pt.x);
    pt.x = Math.min(getMaxXExtent(), pt.x);
    pt.y = Math.max(0, pt.y);
    pt.y = Math.min(getMaxYExtent(), pt.y);

    m_viewport.setViewPosition(pt);
    enableButtons(pt);
  }

  protected void enableButtons(Point pt) {
    if (pt.x == 0)
      enableComponent(m_left, false);
    else enableComponent(m_left, true);

    if (pt.x >= getMaxXExtent())
      enableComponent(m_right, false);
    else enableComponent(m_right, true);

    if (pt.y == 0)
      enableComponent(m_up, false);
    else enableComponent(m_up, true);

    if (pt.y >= getMaxYExtent())
      enableComponent(m_down, false);
    else enableComponent(m_down, true);
  }

  protected void enableComponent(JComponent c, boolean b) {
    if (c.isEnabled() != b)
      c.setEnabled(b);
  }

  protected int getMaxXExtent() {
    return m_viewport.getView().getWidth()-m_viewport.getWidth();
  }

  protected int getMaxYExtent() {
    return m_viewport.getView().getHeight()-m_viewport.getHeight();
  }

  public static void main(String argv[])  {
    new ButtonScroll();
  }
}
```

Understanding the Code

Class ButtonScroll

Several instance variables are declared:

- `JViewport m_viewport`: viewport to display a large image.
- `JButton m_up`: push button to scroll up programmatically.
- `JButton m_down`: push button to scroll down programmatically.
- `JButton m_left`: push button to scroll left programmatically.
- `JButton m_right`: push button to scroll right programmatically.
- `int m_pgVert`: number of pixels for a vertical scroll.
- `int m_pgHorz`: number of pixels for a horizontal scroll.

The constructor of the `ButtonScroll` class creates and initializes the GUI components for this example. A `BorderLayout` is used to manage the components in this frame's content pane. `JLabel lbl` holding a large image is placed in the viewport, `m_viewport`, to provide programmatic viewing capabilities. This `JViewport` is added to the center of our frame.

As we mentioned above, we need to capture the `ChangeEvents` that are fired when our `JViewport` changes size so that we can enable and disable our buttons accordingly. We do this by simply attaching a `ChangeListener` to our viewport and call our `enableButtons()` method (see below) from `stateChanged()`:

```
m_viewport.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent e) {
    enableButtons(
      ButtonScroll.this.m_viewport.getViewPosition());
  }
});
```

Two buttons `m_up` and `m_down` are created for scrolling in the vertical direction. Method `createButton()` is used to create a new `JButton` component and set a group of properties for it (see below). Each of the new buttons receives an `ActionListener` which calls the `movePanel()` method in response to a mouse click. These two buttons are added to the intermediate container, `JPanel pv`, which is added to the `EAST` side of our frame's content pane. Similarly, two buttons, `m_left` and `m_right`, are created for scrolling in the horizontal direction and added to the `SOUTH` region of the content pane.

Method `createButton()` creates a new `JButton` component and sets a group of properties for it. This method takes two parameters: the name of the scrolling direction as a `String` and the button's mnemonic as a `char`. This method assumes that three image files are prepared:

- name1.gif: the default icon.
- name2.gif: the pressed icon.
- name3.gif: the disabled icon.

These images are loaded as `ImageIcons` and attached to the button with the associated `setXX()` method:

```
JButton btn = new JButton(new ImageIcon(name+"1.gif"));
btn.setPressedIcon(new ImageIcon(name+"2.gif"));
btn.setDisabledIcon(new ImageIcon(name+"3.gif"));
btn.setToolTipText("Moves "+name);
btn.setBorderPainted(false);
btn.setMargin(new Insets(0, 0, 0, 0));
```

```
        btn.setContentAreaFilled(false);
        btn.setMnemonic(mnemonic);
        return btn;
```

Then we remove any border or content area painting, so the presentation of our button is completely determined by our icons. Finally we set the tooltip text and mnemonic and return that component instance.

Method `movePanel()` programmatically scrolls the image in the viewport in the direction determined by two parameters: `xmove` and `ymove`. These parameters can have values -1, 0, or 1. To determine the actual amount of scrolling we multiply these parameters by `m_pgHorz` (`m_pgVert`). Local variable `Point pt` determines a new viewport position. It is limited so the resulting view will not display any empty space (not belonging to the displaying image), similar to how we enforce the viewport view position in the grab-and-drag scrolling example above. Finally, method `setViewPosition()` is called to scroll to the new position and `enableButtons()` enables/disables buttons according to the new position:

```
    Point pt = m_viewport.getViewPosition();
    pt.x += m_pgHorz*xmove;
    pt.y += m_pgVert*ymove;

    pt.x = Math.max(0, pt.x);
    pt.x = Math.min(getMaxXExtent(), pt.x);
    pt.y = Math.max(0, pt.y);
    pt.y = Math.min(getMaxYExtent(), pt.y);

    m_viewport.setViewPosition(pt);
    enableButtons(pt);
```

Method `enableButtons()` disables a button if scrolling in the corresponding direction is not possible and enables it otherwise. For example, if the viewport position's x coordinate is `0` we can disable the scroll left button (remember that the view position will never be negative, as enforced by `JViewport`):

```
    if (pt.x <= 0)
      enableComponent(m_left, false);
    else enableComponent(m_left, true);
```

..Similarly, if the viewport position's x coordinate is greater than or equal to our maximum allowable x position (determined by `getMaxXExtent()`) we disable the scroll right button:

```
    if (pt.x >= getMaxXExtent())
      enableComponent(m_right, false);
    else enableComponent(m_right, true);
```

Methods `getMaxXExtent()` and `getMaxYExtent()` return the maximum coordinates available for scrolling in the horizontal (vertical direction) by subtracting the appropriate viewport dimension from the appropriate dimension of the child component.

Running the Code

Note: The shuttle image for this example was found at http://shuttle.nasa.gov/sts-95/images/esc/

Note how different images completely determine the presentation of our buttons. Press the buttons and note how the image is scrolled programmatically. Use the keyboard mnemonic as an alternative way to press the buttons, and note how this mnemonic is displayed in the tooltip text. Note how a button is disabled when scrolling in the corresponding direction is no longer available, and are enabled otherwise. Now try resizing the frame and note how the buttons will change state depending on whether the viewport is bigger or smaller than its child component.

Chapter 8. Split Panes

In this chapter:

- JSplitPane
- Basic splitpane example
- Gas model simulation using a splitpane

8.1 JSplitPane

class javax.swing.JSplitPane

Split panes allow the user to dynamically change the size of two or more components displayed side-by-side (within a window or another panel). Special dividers can be dragged with the mouse to increase space for one component and decrease the display space for another. Note that the total display area does not change. This gives applications a more modern and sophisticated view. A familiar example is the combination of a tree and a table separated by a horizontal divider (e.g. file explorer-like applications). The Swing framework for split panes consists only of `JSplitPane`.

`JSplitPane` can hold two components separated by a horizontal or vertical divider. The components on either side of a `JSplitPane` can be added either in one of the constructors, or with the proper `setXXComponent()` methods (where `XX` is substituted by `Left`, `Right`, `Top`, or `Bottom`). We can also set the orientation at run-time using its `setOrientation()` method.

The divider between the side components represents the only visible part of `JSplitPane`. It's size can be managed with the `setDividerSize()` method, and its position by the two overloaded `setDividerLocation()` methods (which take an absolute location in pixels or proportional location as a double). The divider location methods have no affect until `JSplitPane` is displayed. `JSplitPane` also maintains a `oneTouchExpandable` property which, when `true`, places two small arrows inside the divider that will move the divider to its extremities when clicked.

UI Guideline: Resizable Panelled Display

Split pane becomes really useful when your design has panelled the display for ease of use but you (as designer) have no control over the actual window size. The Netscape e-mail reader is a good example of this. A split pane is introduced to let the user vary the size of the message header panel against the size of the message text panel.

An interesting feature of the `JSplitPane` component is that you can specify whether or not to repaint side components during the divider's motion using the `setContinuousLayout()` method. If you can repaint components fast enough, resizing will have a more natural view with this setting. Otherwise, this flag should be set to `false`, in which case side components will be repainted only when the divider's new location is chosen. In this latter case, a divider line will be shown as the divider location is dragged to illustrate the new position.

`JSplitPane` will not size any of its constituent components smaller than their minimum sizes. If the minimum size of each component is larger than the size of the splitpane the divider will be effectively disabled (unmovable). Also note that we can call its `resetToPreferredSize()` method to resize its children to their preferred sizes, if possible.

UI Guideline: Use SplitPane in conjunction with ScrollPane

It's important to use a ScrollPane on the panels which are being split with the SplitPane. The scrollbar will then invoke automatically as required when data is obscured as the split pane is dragged back and forth. The introduction of the scrollpane gives the viewer a clear indication that there is some hidden data. They can then choose to scroll with the scrollbar or uncover the data using the splitpane.

8.2 Basic splitpane example

This basic introductory demo shows `JSplitPane` at work. We can manipulate four simple custom panels placed in three `JSplitPanes`:

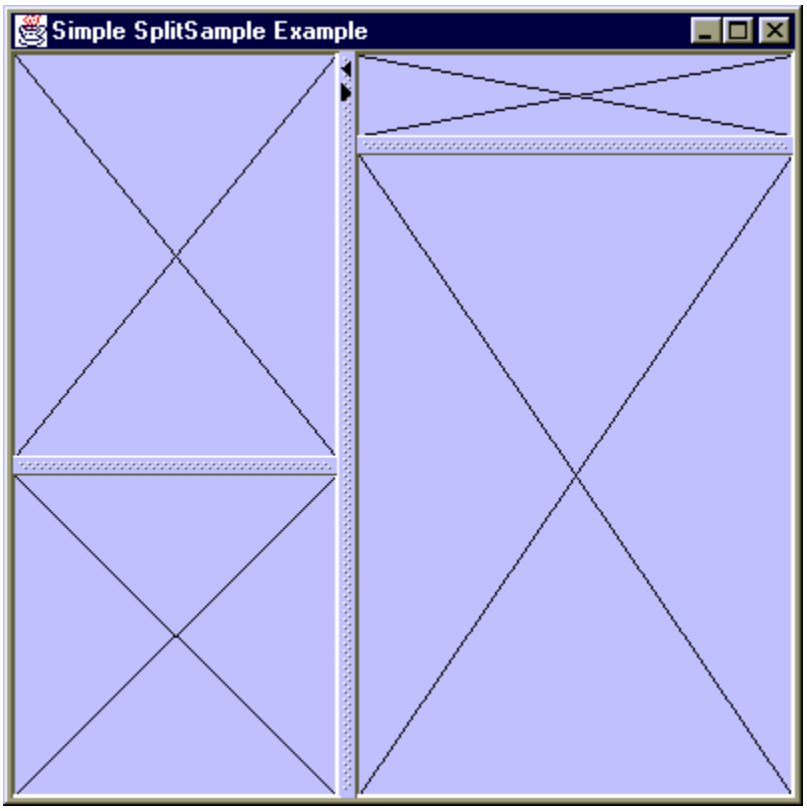


Figure 8.1 SplitPane example displaying simple custom panels.
<<file figure8-1.gif>>

The Code: SplitSample.java
see \Chapter8\1

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

import javax.swing.*;

public class SplitSample extends JFrame
{
  protected JSplitPane m_sp;

  public SplitSample() {
    super("Simple SplitSample Example");
    setSize(400, 400);

    Component c11 = new SimplePanel();
    Component c12 = new SimplePanel();
```

```
    JSplitPane spLeft = new JSplitPane(
      JSplitPane.VERTICAL_SPLIT, c11, c12);
    spLeft.setDividerSize(8);
    spLeft.setContinuousLayout(true);

    Component c21 = new SimplePanel();
    Component c22 = new SimplePanel();
    JSplitPane spRight = new JSplitPane(
      JSplitPane.VERTICAL_SPLIT, c21, c22);
    spRight.setDividerSize(8);
    spRight.setContinuousLayout(true);

    m_sp = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
      spLeft, spRight);
    m_sp.setContinuousLayout(false);
    m_sp.setOneTouchExpandable(true);

    getContentPane().add(m_sp, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public static void main(String argv[]) {
    new SplitSample();
  }
}

class SimplePanel extends JPanel
{
  public Dimension getPreferredSize() {
    return new Dimension(200, 200);
  }

  public Dimension getMinimumSize() {
    return new Dimension(40, 40);
  }

  public void paintComponent(Graphics g) {
    g.setColor(Color.black);
    Dimension sz = getSize();
    g.drawLine(0, 0, sz.width, sz.height);
    g.drawLine(sz.width, 0, 0, sz.height);
  }
}
```

Understanding the Code

Class SplitSample

Four instances of `SimplePanel` (see below) are used to fill a 2x2 structure. Two left components (`c11` and `c12`) are placed in the vertically split `spLeft` panel. The two right components (`c21` and `c22`) are placed in the vertically split `spRight` panel. The `spLeft` and `spRight` panels are placed in the horizontally split `m_sp` panel.

Several properties are assigned to demonstrate JSplitPane's behavior. The continuousLayout property is set to true for spLeft and spRight, and false for m_sp panel. So as the divider moves inside the left and right panels, child components are repainted continuously, producing immediate results. However, as the vertical divider is moved it is denoted by a black line until a new position is chosen (i.e. the mouse is released). Only then are its child components validated and repainted. The first kind of behavior is recommended for simple components that can be rendered quickly, while the second is recomended for components whose repainting can take a significant amount of time.

Also note that the oneTouchExpandable property is set to true for the vertical JSplitPane. This places small arrow widgets on the divider. By pressing these arrows with the mouse we can instantly move the divider to the leftmost or rightmost position. When in the left or right-most positions, pressing these arrows will then move the divider to its most recent location, maintained by the lastDividerLocation property.

Class SimplePanel

SimplePanel represents a simple Swing component to be used in this example. Method paintComponent() draws two diagonal lines across this component. Note that the overridden getMinimumSize() method defines the minimum space required for this component. JSplitPane will prohibit the user from moving the divider if the resulting child size will become less than its minimum size.

Note: The arrow widgets associated with the oneTouchExpandable property will move the divider to the extreme location without regard to minimum sizes of child components.

Running the Code

Note how child components can be resized with dividers. Also note the difference between resizing with continuous layout (side panes) and without it (center pane). Play with the "one touch expandable" widgets for quick expansion and collapse.

8.3 Gas model simulation using a split pane

In this section we'll use JSplitPane for an interesting scientific experiment: a simulation of the gas model. Left and right components represent containers holding a two-dimensional "ideal gas." The JSplitPane component provides a moveable divider between them. By moving the divider we observe how gas reacts when its volume is changed. Online educational software is ever-increasing as the internet flourishes, and here we show how Swing can be used to demonstrate one of the most basic laws of thermodynamics!

Note: As you may remember from a physics or chemistry course, an ideal gas is a physical model in which gas atoms or molecules move in random directions bouncing elastically from a container's bounds. Mutual collisions are negligible. The speed of the atoms depends on gas temperature. Several laws can be demonstrated with this model. One states that under the condition of constant temperature, multiplication of pressure P and volume V is constant: PV = const.

To model the motion of atoms we'll use threads. So this example also gives a good example of using several threads in Swing.

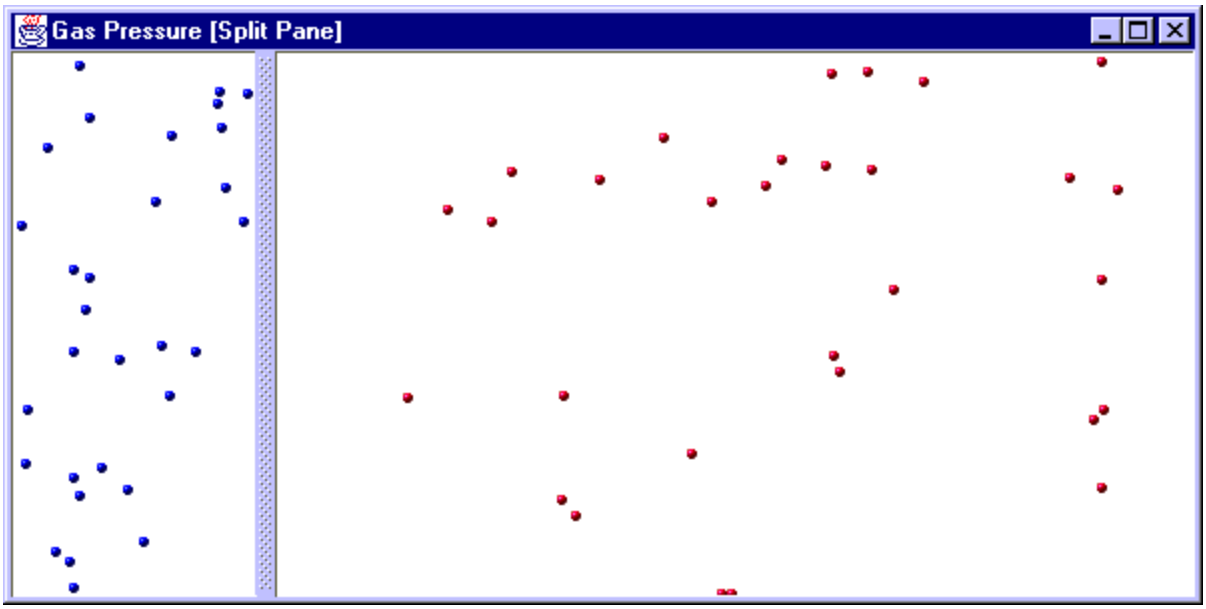


Figure 8.2 Gas model simulation showing moving atoms.

<<file figure8-2.gif>>

The Code: Split.java

see \Chapter8\2

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class Split extends JFrame implements Runnable
{
  protected GasPanel m_left;
  protected GasPanel m_right;

  public Split() {
    super("Gas Pressure [Split Pane]");
    setSize(600, 300);

    ImageIcon ball1 = new ImageIcon("ball1.gif");
    m_left = new GasPanel(30, ball1.getImage());
    ImageIcon ball2 = new ImageIcon("ball2.gif");
    m_right = new GasPanel(30, ball2.getImage());
    JSplitPane sp = new JSplitPane(
      JSplitPane.HORIZONTAL_SPLIT, m_left, m_right);
    sp.setDividerSize(10);
    sp.setContinuousLayout(true);
    getContentPane().add(sp, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);
```

```
    setVisible(true);

    new Thread(m_left).start();
    new Thread(m_right).start();
    new Thread(this).start();
  }

  public void run() {
    while (true) {
      int p1  = (int)m_left.m_px2;
      int pv1 = p1*m_left.getWidth();
      int p2 = (int)m_right.m_px1;
      int pv2 = p2*m_right.getWidth();
      System.out.println("Left: p="+p1+"\tpv="+pv1+
        "\tRight: p="+p2+"\tpv="+pv2);
      m_left.clearCounters();
      m_right.clearCounters();

      try {
        Thread.sleep(20000);
      }
      catch(InterruptedException e) {}
    }
  }

  public static void main(String argv[]) { new Split(); }
}

class GasPanel extends JPanel implements Runnable
{
  protected Atom[] m_atoms;
  protected Image  m_img;
  protected Rectangle m_rc;

  public double m_px1 = 0;
  public double m_px2 = 0;
  public double m_py1 = 0;
  public double m_py2 = 0;

  public GasPanel(int nAtoms, Image img) {
    setBackground(Color.white);
    enableEvents(ComponentEvent.COMPONENT_RESIZED);

    m_img = img;
    m_atoms = new Atom[nAtoms];
    m_rc = new Rectangle(getPreferredSize());
    for (int k=0; k<nAtoms; k++) {
      m_atoms[k] = new Atom(this);
    }
  }

  public Dimension getPreferredSize() {
    return new Dimension(300, 300);
  }

  public void run() {
    while (true) {
      for (int k=0; k<m_atoms.length; k++)
        m_atoms[k].move(m_rc);
      repaint();

      try {
```

```
        Thread.sleep(100);
      }
      catch(InterruptedException e) {}
    }
  }

  public void paintComponent(Graphics g) {
    g.setColor(getBackground());
    g.fillRect(m_rc.x, m_rc.y, m_rc.width, m_rc.height);

    for (int k=0; k<m_atoms.length; k++)
      g.drawImage(m_img, m_atoms[k].getX(),
        m_atoms[k].getY(), this);
  }

  protected void processComponentEvent(ComponentEvent e) {
    if (e.getID() == ComponentEvent.COMPONENT_RESIZED) {
      m_rc.setSize(getSize());
      for (int k=0; k<m_atoms.length; k++)
        m_atoms[k].ensureInRect(m_rc);
    }
  }

  public void clearCounters() {
    m_px1 = 0;
    m_px2 = 0;
    m_py1 = 0;
    m_py2 = 0;
  }
}

class Atom
{
  protected double m_x;
  protected double m_y;
  protected double m_vx;
  protected double m_vy;

  protected GasPanel m_parent;

  public Atom(GasPanel parent) {
    m_parent = parent;
    m_x = parent.m_rc.x + parent.m_rc.width*Math.random();
    m_y = parent.m_rc.y + parent.m_rc.height*Math.random();
    double angle = 2*Math.PI*Math.random();
    m_vx = 10*Math.cos(angle);
    m_vy = 10*Math.sin(angle);
  }

  public void move(Rectangle rc) {
    double x = m_x + m_vx;
    double y = m_y + m_vy;
    int x1 = rc.x;
    int x2 = rc.x + rc.width;
    int y1 = rc.y;
    int y2 = rc.y + rc.height;
    for (int bounce = 0; bounce<2; bounce++) {
      if (x < x1) {
        x += 2*(x1-x);
        m_vx = - m_vx;
        m_parent.m_px1 += 2*Math.abs(m_vx);
      }
```

```
      if (x > x2) {
        x -= 2*(x-x2);
        m_vx = - m_vx;
        m_parent.m_px2 += 2*Math.abs(m_vx);
      }
      if (y < y1) {
        y += 2*(y1-y);
        m_vy = - m_vy;
        m_parent.m_py1 += 2*Math.abs(m_vy);
      }
      if (y > y2) {
        y -= 2*(y-y2);
        m_vy = - m_vy;
        m_parent.m_py2 += 2*Math.abs(m_vy);
      }
    }
    m_x = x;
    m_y = y;
  }

  public void ensureInRect(Rectangle rc) {
    if (m_x < rc.x)
      m_x = rc.x;
    if (m_x > rc.x + rc.width)
      m_x = rc.x + rc.width;
    if (m_y < rc.y)
      m_y = rc.y;
    if (m_y > rc.y + rc.height)
      m_y = rc.y + rc.height;
  }

  public int getX() { return (int)m_x; }
  public int getY() { return (int)m_y; }
}
```

Understanding the Code

Class Split

The constructor of the `Split` frame creates two instances of the `GasPanel` class (which models a gas container, see below) and places them in a `JSplitPane`. All other code requires little discussion, but we must comment on one thing. Both the `Split` and `GasPanel` classes implement the `Runnable` interface, so threads are created and started to run all three instances.

Reminder: The `Runnable` interface should be implemented by classes which do not intend to use any `Thread` functionality other than the `run()` method. In such a case we don't have to sub-class the `Thread` class. Instead we can simply implement `Runnable` and define the `run()` method. In this method we can use the static `Thread.sleep()` method to yield control to other threads for a specified amount of time.

The `run()` method of our `Split` class periodically interrogates the pressure on the divider from the left and right containers, as well as each container's width, which is proportional to the container's volume. Then it prints out the results of our measurements, clears the counters of the containers (see below) and sleeps for another 20 seconds.

Note: Because of the random nature of the gas model all observations are statistical. Each time you run this simulation you're likely to observe a slightly different results. The more atoms that constitute the gas, the more accurate the results achieved will be. A real gas has about 10^{22} atoms/cm^3, and fluctuations in its parameters are very small. In our model only a few dozen "atoms" are participating, so fluctuations are considerable, and we

have to wait a bit before we can obtain meaningful measurements (through averaging several results).

Class GasPanel

Class `GasPanel` models a two-dimensional gas container. It implements the `Runnable` interface to model the random motion of its contained atoms. Seven instance variables have the following meaning:

`Atom[] m_atoms`: array of `Atom` instances hosted by this container.

`Image m_img`: image used to draw atoms.

`Rectangle m_rc`: container's rectangular bounds.

`double m_px1`: counter to measure pressure on the left wall.

`double m_px2`: counter to measure pressure on the right wall.

`double m_py1`: counter to measure pressure on the top wall.

`double m_py2`: counter to measure pressure on the bottom wall.

The `GasPanel` constructor takes a number of atoms to be created as a parameter as well as a reference to the image to represent them. Method `enableEvents()` is called to enable the processing of resize events on this component. Finally, an array of atoms is created (see the `Atom` class below).

The `getPreferredSize()` method returns the preferred size of this component (used by our split pane to determine the initial position of its divider).

The `run()` method activates the gas container. For each child atom it invokes the `Atom` `move()` method (see below) which changes an `Atom`'s coordinates. The component is then repainted and the calling thread sleeps for 100 ms to provide smooth continuous motion.

The `paintComponent()` method is overridden to draw each child atom. It clears the component's area and, for each atom, draws the specified image (typically a small ball) at the current atom location.

The `processComponentEvent()` method is overridden to process resizing events. It updates the `m_rc` rectangle (used to limit atom motion) and calls the `ensureInRect()` method for all child atoms to force them to stay inside this component's bounds. A check for the `COMPONENT_RESIZED` ID is done to skip the processing of `COMPONENT_MOVED` events which are also delivered to this component even though we've explicitly asked for only `COMPONENT_RESIZED` events (see `enableEvents()` call in constructor).

The `clearCounters()` method clears the counters used to measure the pressure on each of the walls.

Class Atom

An `Atom` represents a single object moving within a specific rectangular region and bouncing elastically from its walls. Instance variables:

`double m_x`: current x-coordinate.

`double m_y`: current y-coordinate.

`double m_vx`: current x-component of velocity.

`double m_vy`: current y-component of velocity.

`GasPanel m_parent`: reference to parent container.

The `Atom` constructor takes a reference to a parent `GasPanel` as parameter. It initializes its coordinates randomly within the parent's boudning rectangle using `Math.random()` as a random number generator. An

atom's velocity vector is assigned a fixed absolute magnitude (10) and a random orientation in the x-y plane.

Note: In a more realistic model, velocity would be a normally distributed random value. However, this is not very significant for our purposes.

The `move()` method moves an atom to a new position and is called during each time the parent GasPanel's `run()` loop is executed. When new coordinates `x, y` are calculated, this method checks for possible bounces from this container's walls:

```
  public void move(Rectangle rc) {
    double x = m_x + m_vx;
    double y = m_y + m_vy;
    int x1 = rc.x;
    int x2 = rc.x + rc.width;
    int y1 = rc.y;
    int y2 = rc.y + rc.height;
    for (int bounce = 0; bounce<2; bounce++) {
      // pseudo-code
      if (x < x1) { // bounce off of left wall... }
      if (x > x2) { // bounce off of right wall... }
      if (y < y1) { // bounce off of top wall... }
      if (y > y2) { // bounce off of bottom wall... }
    }
    m_x = x;
    m_y = y;
  }
```

If a new point lies behind one of four walls, a bounce occurs, which changes the coordinate and velocity vector. This contributes to the pressure on the wall the bounce occurred on (as an absolute change in the velocity's component), which is accumulated in the parent `GasPanel`. Note that bouncing is checked twice to take into account the rare case that two subsequent bounces occur in a single step. That can occur near the container's corners, when, after the first bounce, the moving particle is repositioned beyond the nearest perpendicular wall.

The final methods of our `Atom` class are fairly straightforward. The `ensureInRect()` method is called to ensure that an `Atom`'s coordinates lie within the given rectangle, and the `getX()` and `getY()` methods return the current coordinates as integers.

Running the Code

Note how the gas reacts to the change in the parent container's volume by adjusting the position of the split pane divider. Also try adjusting the size of the application frame.

The following are some P and PV measurements we obtained when experimenting with this example:

```
Left: p=749     pv=224700       Right: p=996    pv=276888
Left: p=701     pv=210300       Right: p=1006   pv=279668
Left: p=714     pv=214200       Right: p=1028   pv=285784
Left: p=770     pv=231000       Right: p=1018   pv=283004
Left: p=805     pv=241500       Right: p=1079   pv=299962
Left: p=1586    pv=190320       Right: p=680    pv=311440
Left: p=1757    pv=210840       Right: p=594    pv=272052
Left: p=1819    pv=218280       Right: p=590    pv=270220
Left: p=1863    pv=223560       Right: p=573    pv=262434
Left: p=1792    pv=215040       Right: p=621    pv=284418
```

We can see tell at a certain time the divider had been moved from right to left by the increase in pressure on

the left side, and a decrease in pressure on the right side. However, the PV value (in arbitrary units) remains practically unchanged.

Chapter 9. Combo Boxes

In this chapter:

- JComboBox
- Basic JComboBox example
- Custom model and renderer
- Combo box with memory
- Custom editing

9.1 JCombobox

class javax.swing.JComboBox

This class represents a basic GUI component which consists of two parts:

1. A popup menu (an implementation of `javax.swing.plaf.basic.ComboPopup`). By default this is a `JPopupMenu` sub-class (`javax.swing.plaf.basic.BasicComboPopup`) containing a `JList` in a `JScrollPane`.
2. A button acting as a container for an editor or renderer component, and an arrow button used to display the popup menu.

The `JList` uses a `ListSelectionModel` (see chapter 10) allowing `SINGLE_SELECTION` only. Apart from this, `JComboBox` directly uses only one model, a `ComboBoxModel`, which manages data in its `JList`.

A number of constructors are available to build a `JComboBox`. The default constructor can be used to create a combo box with an empty list, or we can pass data to a constructor as a one-dimensional array, a `Vector`, or as an implementation of the `ComboBoxModel` interface (see below). The last variant allows maximum control over the properties and appearance of a `JComboBox`, as we will see.

As other complex Swing components, `JComboBox` allows a customizable renderer for displaying each item in its drop-down list (by default a `JLabel` sub-class implementation of `ListCellRenderer`), and a customizable editor to be used as the combo box's data entry component (by default an instance of `ComboBoxEditor` which uses a `JTextField`). We can use the existing default implementations of `ListCellRenderer` and `ComboBoxEditor`, or we can create our own according to our particular needs (which we will see later in ths chapter). Note that unless we use a custom renderer, the default renderer will display each element as a `String` defined by that object's `toString()` method (the only exceptions to this are `Icon` implementations which will be rendered as they would be in any `JLabel`). Also note that a renderer returns a `Component`, but that component is not interactive and is only used for display purposes (i.e. it acts as a "rubber stamp"[API]). For instance, if a `JCheckBox` is used as a renderer we will not be able to check and uncheck it. Editors, however, are fully interactive.

Similar to `JList` (next chapter), this class uses `ListDataEvents` to deliver information about changes in the state of its drop-down list's model. `ItemEvents` and `ActionEvents` are fired when the current selection

changes (from any source--programmatic or direct user input). Correspondingly, we can attach ItemListeners and ActionListeners to receive these events.

The drop-down list of a JComboBox is a popup menu containing a JList (this is actually defined in the UI delegate, not the component itself) and can be programmatically displayed using the showPopup() and hidePopup() methods. As any other Swing popup menu (which we will discuss in chapter 12), it can be displayed as either heavyweight or lightweight. JComboBox provides the setLightWeightPopupEnabled() method allowing us to choose between these modes.

JComboBox also defines an inner interface called KeySelectionManager that declares one method, selectionForKey(char aKey, ComboBoxModel aModel), which should be overriden to return the index of the list element to select when the list is visible (popup is showing) and the given keyboard character is pressed.

The JComboBox UI delegate represents JComboBox graphically by a container with a button which encapsulates an arrow button and either a renderer displaying the currently selected item, or an editor allowing changes to be made to the currently selected item. The arrow button is displayed on the right of the renderer/editor and will show the popup menu containing the drop-down list when clicked.

Note: Because of the JComboBox UI delegate construction, setting the border of a JComboBox does not have the expected effect. Try this and you will see that the container containing the main JComboBox button gets the assigned border, when in fact we want that button to recieve the border. There is no easy way to set the border of this button without customizing the UI delegate, and we hope to see this limitation disappear in a future version.

When a JComboBox is editable (which it is not by default) the editor component will allow modification of the currenly selected item. The default editor will appear as a JTextField accepting input. This text field has an ActionListener attached that will accept an edit and change the selected item accoringly when/if the Enter key is pressed. If the focus changes while editing, all editing will be cancelled and a change will not be made to the selected item.

JComboBox can be made editable with its setEditable() method, and we can specify a custom ComboBoxEditor with JComboBox's setEditor() method.. Setting the editable property to true causes the UI delegate to replace the renderer component in the button to the specified editor component. Similarly, setting this property to false causes the editor in the button to be replaced by a renderer.

The cell renderer used for a JComboBox can be assigned/retrieved with the setRenderer()/getRenderer() methods. Calls to these methods actually get passed to the JList contained in the combo box's popup menu.

9.1.1 The ComboBoxModel interface

abstract interface javax.swing.ComboBoxModel

This interface extends the ListModel interface which handles the combo box drop-down list's data. This model separately handles its selected item with two methods, setSelectedItem() and getSelectedItem().

9.1.2 The MutableComboBoxModel interface

abstract interface javax.swing.MutableComboBoxModel

This interface extends ComboBoxModel and adds four methods to modify the model's contents dynamically: addElement(), insertElementAt(), removeElement(), removeElementAt().

9.1.3 DefaultComboBoxModel

class javax.swing.DefaultComboBoxModel

This class represents the default model used by `JComboBox`, and implements `MutableComboBoxModel`. To programmatically select an item we can call its `setSelectedItem()` method. Calling this method, as well as any of the `MutableComboBoxModel` methods mentioned above, will cause a `ListDataEvent` to be fired. To capture these events we can attatch `ListDataListeners` with `DefaultComboBoxModel`'s `addListDataListener()` method. We can also remove these listeners with its `removeListDataListener()` method.

9.1.4 The ListCellRenderer interface

abstract interface javax.swing.ListCellRenderer

This is a simple interface used to define the component to be used as a renderer for the `JComboBox` drop-down list. It declares one method, `getListCellRendererComponent(JList list, Object value, int Index, boolean isSelected, boolean cellHasFocus)`, which is called to return the component used to represent a given combo box element visually. The component returned by this method is not at all interactive and is used for display purposes only (referred to as a "rubber stamp" in the API docs).

When in noneditable mode, `-1` will be passed to this method to return the component used to represent the selected item in the main `JComboBox` button. Normally this component is the same as the component used to display that same element in the drop-down list.

9.1.5. DefaultListCellRenderer

class javax.swing.DefaultListCellRenderer

This is the concrete implementation of the `ListCellRenderer` interface used by `JList` by default (and this by `JComboBox`'s `JList`). This class extends `JLabel` and its `getListCellRenderer()` method returns a `this` reference, renders the given value by setting its text to the `String` returned by the value's `toString()` method (unless the value is an instance of `Icon`, in which case it will be rendered as it would be in any `JLabel`), and uses `JList` foreground and background colors depending on whether or not the given item is selected.

Note: Unfortunately there is no easy way to access `JComboBox`'s drop-down `JList`, which prevents us from assigning new foreground and background colors. Ideally `JComboBox` would provide this communication with its `JList`, and we hope to see this functionality in a future version.

A single static `EmptyBorder` instance is used for all cells that do not have the current focus. This border has top, bottom, left, and right spacing of 1, and unfortunately cannot be re-assigned.

9.1.6 The ComboBoxEditor interface

abstract interface javax.swing.ComboBoxEditor

This interface describes the `JComboBox` editor. The default editor is provided by the only implementing class, `javax.swing.plaf.basic.BasicComboBoxEditor`. But we are certainly not limited to this component. It is the purpose of this interface to allow us to implement our own custom editor. The `getEditorComponent()` method should be overridden to return the editor component to use. `BasicComboBoxEditor`'s `getEditorComponent()` method returns a `JTextField` that will be used for the currently selected combo box item. Unlike cell renderers, components returned by the `getEditorComponent()` method are fully interactive and do not act like rubber stamps.

The `setItem()` method is intended to tell the editor which element to edit (this is called when an item is selected from the drop-down list). The `getItem()` method is used to return the object being edited (a `String` using the default editor).

The `selectAll()` method is intended to select all items to be edited, and the default editor implements this by selecting all text in the text field (though this method is not used in the default implementation, we might consider calling it from our own the `setItem()` method to show all text selected when editing starts).

`ComboBoxEditor` also declares functionality for attaching and removing `ActionListeners` which are notified when an edit is accepted. In the default editor this occurs when Enter is pressed while the text field has the focus.

Note: Unfortunately Swing does not provide an easily reusable `ComboBoxEditor` implementation, forcing custom implementations to manage all `ActionListener` and item selection/modification functionality from scratch (we hope to see this limitation accounted for in a future Swing release).

UI Guideline: Advice on Usage and Design Usage

Comboboxes and List Boxes are very similar. In fact a Combobox is an Entry Field with a drop down List Box. Deciding when to use one or another can be difficult. Our advice is to think about reader output rather than data input. When the reader only needs to see a single item then a Combobox is the choice. Use a Combobox where a single selection is made from a collection and for reading purposes it is only necessary to see a single item, e.g. Currency USD.

Design

There are a number of things which affect the usability of a combobox. Beyond more than a few items, they become unusable unless the data is sorted in some logical fashion e.g. alphabetical, numerical. When a list gets longer, usability is affected again. Once a list gets beyond a couple of hundred items, even when sorted, it becomes very slow for the user to locate specific item in the list. Some implementations have solved this by offering an ability to type in partial text and the list "jumps" to the best match or partial match item e.g. type in "ch" and the combobox will jump to "Chevrolet" as in the example in this chapter. You may like to consider such an enhancement to a JCombobox to improve the usability in longer lists.

There are a number of graphical considerations too. Like all other data entry fields, comboboxes should be aligned to fit attractively into a panel. However, this can be problematic. You must avoid making a combobox which is simply too big for the list items contained e.g. a combobox for currency code (typically USD for U.S. Dollars) only needs to be 3 characters long. So don't make it big enough to take 50 characters. It will look unbalanced. Another problem, is the nature of the list items. If you have 50 items in a list where most items are around 20 characters but one item is 50 characters long then should you make the combobox big enough to display the longer one? Well maybe but for most occasions your display will be unbalanced again. It is probably best to optimise for the more common length, providing the the longer one still has meaning when read in its truncated form. One solution to displaying the whole length of a truncated item is to use the tooltip facility. When the User places the mouse over an item, a tooltip appears with the full length data.

One thing you must never do is dynamically resize the combobox to fit a varying length item selection. This will provide alignment problems and may also add a usability problem because the pull-down button may become a moving target which denies the user the option to learn its position with directional memory.

9.2 Basic JComboBox example

This example displays information about popular cars in two symmetrical panels to provide a natural means of comparison. To be more or less realistic, we need to take into account that any car model comes in several trim lines which actually determine the car's characteristics and price. Numerous characteristics of cars are available on the web. For this simple example we've selected the following two-level data structure:

CAR

Name	Type	Description
Name	String	Model's name
Manufacturer	String	Company manufacturer
Image	Icon	Model's photograph
Trims	Vector	A collection of model's trims

TRIM

Name	Type	Description
Name	String	Trim's name
MSRP	int	Manufacturer's suggested retail price
Invoice	int	Invoice price
Engine	String	Engine description

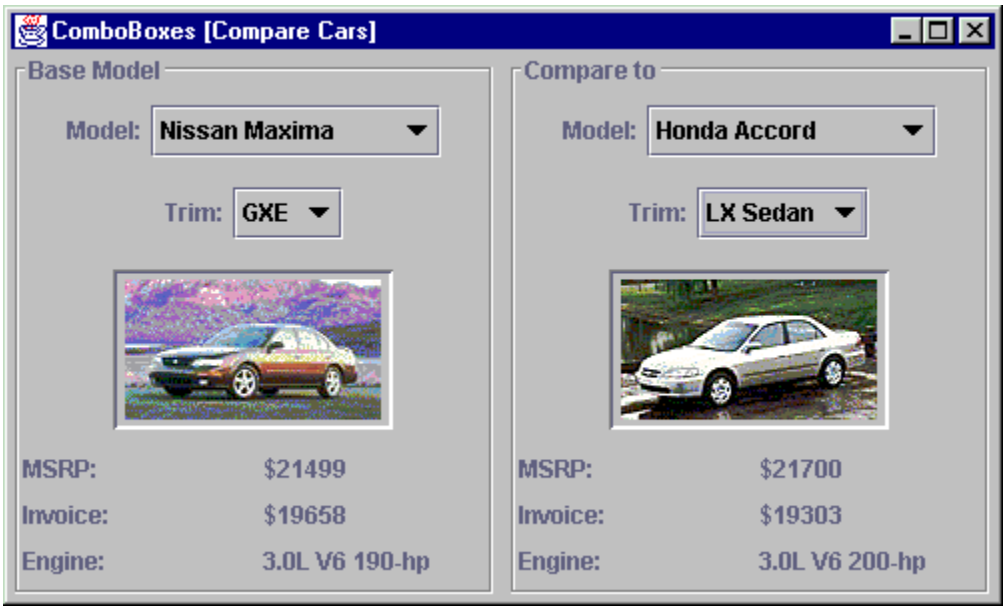


Figure 9.1 Dynamically changeable JComboBoxes allowing comparison of car model and trim information.
<<file figure9-1.gif>>

The Code: ComboBox1.java
see \Chapter9\1

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class ComboBox1 extends JFrame
{
  public ComboBox1() {
    super("ComboBoxes [Compare Cars]");
    getContentPane().setLayout(new BorderLayout());

    Vector cars = new Vector();
    Car maxima = new Car("Maxima", "Nissan", new ImageIcon(
      "maxima.gif"));
```

```
    maxima.addTrim("GXE", 21499, 19658, "3.0L V6 190-hp");
    maxima.addTrim("SE",  23499, 21118, "3.0L V6 190-hp");
    maxima.addTrim("GLE", 26899, 24174, "3.0L V6 190-hp");
    cars.addElement(maxima);

    Car accord = new Car("Accord", "Honda", new ImageIcon(
      "accord.gif"));
    accord.addTrim("LX Sedan", 21700, 19303, "3.0L V6 200-hp");
    accord.addTrim("EX Sedan", 24300, 21614, "3.0L V6 200-hp");
    cars.addElement(accord);

    Car camry = new Car("Camry", "Toyota", new ImageIcon(
      "camry.gif"));
    camry.addTrim("LE V6", 21888, 19163, "3.0L V6 194-hp");
    camry.addTrim("XLE V6", 24998, 21884, "3.0L V6 194-hp");
    cars.addElement(camry);

    Car lumina = new Car("Lumina", "Chevrolet", new ImageIcon(
      "lumina.gif"));
    lumina.addTrim("LS", 19920, 18227, "3.1L V6 160-hp");
    lumina.addTrim("LTZ", 20360, 18629, "3.8L V6 200-hp");
    cars.addElement(lumina);

    Car taurus = new Car("Taurus", "Ford", new ImageIcon(
      "taurus.gif"));
    taurus.addTrim("LS", 17445, 16110, "3.0L V6 145-hp");
    taurus.addTrim("SE", 18445, 16826, "3.0L V6 145-hp");
    taurus.addTrim("SHO", 29000, 26220, "3.4L V8 235-hp");
    cars.addElement(taurus);

    Car passat = new Car("Passat", "Volkswagen", new ImageIcon(
      "passat.gif"));
    passat.addTrim("GLS V6", 23190, 20855, "2.8L V6 190-hp");
    passat.addTrim("GLX", 26250, 23589, "2.8L V6 190-hp");
    cars.addElement(passat);

    getContentPane().setLayout(new GridLayout(1, 2, 5, 3));
    CarPanel pl = new CarPanel("Base Model", cars);
    getContentPane().add(pl);
    CarPanel pr = new CarPanel("Compare to", cars);
    getContentPane().add(pr);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    pl.selectCar(maxima);
    pr.selectCar(accord);
    setResizable(false);
    pack();
    setVisible(true);
  }

  public static void main(String argv[]) {
    new ComboBox1();
  }
}

class Car
```

```
{
  protected String m_name;
  protected String m_manufacturer;
  protected Icon   m_img;
  protected Vector m_trims;

  public Car(String name, String manufacturer, Icon img) {
    m_name = name;
    m_manufacturer = manufacturer;
    m_img = img;
    m_trims = new Vector();
  }

  public void addTrim(String name, int MSRP, int invoice,
   String engine) {
    Trim trim = new Trim(this, name, MSRP, invoice, engine);
      m_trims.addElement(trim);
  }

  public String getName() { return m_name; }

  public String getManufacturer() { return m_manufacturer; }

  public Icon getIcon() { return m_img; }

  public Vector getTrims() { return m_trims; }

  public String toString() { return m_manufacturer+" "+m_name; }
}

class Trim
{
  protected Car    m_parent;
  protected String m_name;
  protected int    m_MSRP;
  protected int    m_invoice;
  protected String m_engine;

  public Trim(Car parent, String name, int MSRP, int invoice,
   String engine) {
    m_parent = parent;
    m_name = name;
    m_MSRP = MSRP;
    m_invoice = invoice;
    m_engine = engine;
  }

  public Car getCar() { return m_parent; }

  public String getName() { return m_name; }

  public int getMSRP() { return m_MSRP; }

  public int getInvoice() { return m_invoice; }

  public String getEngine() { return m_engine; }

  public String toString() { return m_name; }
}

class CarPanel extends JPanel
{
```

```
protected JComboBox m_cbCars;
protected JComboBox m_cbTrims;
protected JLabel m_lblImg;
protected JLabel m_lblMSRP;
protected JLabel m_lblInvoice;
protected JLabel m_lblEngine;

public CarPanel(String title, Vector cars) {
  super();
  setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
  setBorder(new TitledBorder(new EtchedBorder(), title));

  JPanel p = new JPanel();
  p.add(new JLabel("Model:"));
  m_cbCars = new JComboBox(cars);
  ActionListener lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      Car car = (Car)m_cbCars.getSelectedItem();
      if (car != null)
        showCar(car);
    }
  };
  m_cbCars.addActionListener(lst);
  p.add(m_cbCars);
  add(p);

  p = new JPanel();
  p.add(new JLabel("Trim:"));
  m_cbTrims = new JComboBox();
  lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      Trim trim = (Trim)m_cbTrims.getSelectedItem();
      if (trim != null)
        showTrim(trim);
    }
  };
  m_cbTrims.addActionListener(lst);
  p.add(m_cbTrims);
  add(p);
  p = new JPanel();
  m_lblImg = new JLabel();
  m_lblImg.setHorizontalAlignment(JLabel.CENTER);
  m_lblImg.setPreferredSize(new Dimension(140, 80));
  m_lblImg.setBorder(new BevelBorder(BevelBorder.LOWERED));
  p.add(m_lblImg);
  add(p);

  p = new JPanel();
  p.setLayout(new GridLayout(3, 2, 10, 5));
  p.add(new JLabel("MSRP:"));
  m_lblMSRP = new JLabel();
  p.add(m_lblMSRP);

  p.add(new JLabel("Invoice:"));
  m_lblInvoice = new JLabel();
  p.add(m_lblInvoice);

  p.add(new JLabel("Engine:"));
  m_lblEngine = new JLabel();
  p.add(m_lblEngine);
  add(p);
}
```

```
  public void selectCar(Car car) { m_cbCars.setSelectedItem(car); }

  public void showCar(Car car) {
    m_lblImg.setIcon(car.getIcon());
    if (m_cbTrims.getItemCount() > 0)
      m_cbTrims.removeAllItems();
    Vector v = car.getTrims();
    for (int k=0; k<v.size(); k++)
      m_cbTrims.addItem(v.elementAt(k));
    m_cbTrims.grabFocus();
  }

  public void showTrim(Trim trim) {
    m_lblMSRP.setText("$"+trim.getMSRP());
    m_lblInvoice.setText("$"+trim.getInvoice());
    m_lblEngine.setText(trim.getEngine());
  }
}
```

Understanding the Code

Class ComboBox1

Class `ComboBox1` extends `JFrame` to implement the frame container for this example. It has no instance variables. The constructor of the `ComboBox1` class creates a data collection with car information as listed above. A collection of cars is stored in `Vector cars`, and each car, in turn, receives one or more `Trim` instances. Other than this, the `ComboBox1` constructor doesn't do much. It creates two instances of `CarPanel` (see below) and arranges them in a `GridLayout`. These panels are used to select and display car information. Finally two cars are initially selected in both panels.

Class Car

`Car` is a typical data object encapsulating three data fields listed at the beginning of this section: car name, manufacturer, and image. In addition, it holds the `m_trims` vector representing a collection of `Trim` instances.

Method `addTrim()` creates a new `Trim` instance and adds it to the `m_trims` vector. The rest of this class implements typical `getXX()` methods to allow access to the `protected` data fields.

Class Trim

`Trim` encapsulates four data fields listed at the beginning of this section: trim name, suggested retail price, invoice price, and engine type. In addition, it holds a reference to the parent `Car` instance. The rest of this class implements typical `getXX()` methods to allow access to the `protected` data fields.

Class CarPanel

This class extends `JPanel` to provide the GUI framework for displaying car information. Six components are declared as instance variables:

`JComboBox m_cbCars`: Combo box to select a car model.

`JComboBox m_cbTrims`: Combo box to select a car trim for the selected model.

`JLabel m_lblImg`: Label to display the model's image.

`JLabel m_lblMSRP`: Label to display the MSRP.

`JLabel m_lblInvoice`: Label to display the invoice price.

`JLabel m_lblEngine`: Label to display the engine description.

Two combo boxes are used to select cars and trims respectively. Note that `Car` and `Trim` data objects are used to populate these combo boxes, so the actual displayed text is determined by their `toString()` methods. Both combo boxes receive `ActionListeners` to handle item selection. Then a `Car` item is selected, which triggers a call to the `showCar()` method described below. Similarly, a selection of a `Trim` item triggers a call to the `showTrim()` method.

The rest of the `CarPanel` constructor builds `JLabels` to display a car's image and trim data. Note how layouts are used in this example. A y-oriented `BoxLayout` creates a vertical axis used to align and position all components. The combo boxes and supplementary labels are encapsulated in horizontal `JPanels`. `JLabel m_lblImg` receives a custom preferred size to reserve enough space for the photo image. This label is encapsulated in a panel (with its default `FlowLayout`) to ensure that this component will be centered over the container's space. The rest of `CarPanel` is occupied by the six labels, which are hosted by a 3x2 `GridLayout`.

Method `selectCar()` allows us to select a car programmatically from outside this class. It invokes the `setSelectedItem()` method on the `m_cbCars` combo box. Note that this call will trigger an `ActionEvent` which will be captured by the proper listener, resulting in a `showCar()` call.

Method `showCar()` updates the car image and updates the `m_cbTrims` combo box to display the corresponding trims of the selected model. The `(getItemCount() > 0)` condition is necessary because Swing throws an exception if `removeAllItems()` is invoked on an empty `JComboBox`. Finally, focus is transferred to the `m_cbTrims` component.

Method `showTrim()` updates the contents of the labels displaying trim information: MSRP, invoice price, and engine type.

Running the Code

Figure 9.1 shows the `ComboBox1` application displaying two cars simultaneously for comparison. Note that all initial information is displayed correctly. Try experimenting with various selections and note how the combo box contents change dynamically.

UI Guideline: Symmetrical Layout

In this example, the design avoids the problem of having to align the different length comboboxes by using a symmetrical layout. Overall the window has a good balance and good use of white space, as in turn do each of the bordered panes used for individual car selections.

9.3 Custom model and renderer

Ambitious Swing developers may want to provide custom rendering in combo boxes to display structured data in the drop-down list. Different levels of structure can be identified by differing left margins and icons, just as is done in trees (which we will study in chapter 17). Such complex combo boxes can enhance functionality and provide a more sophisticated appearance.

In this section we will show how to merge the model and trim combo boxes from the previous section into a single combo box. To differentiate between model and trim items in the drop-down list, we can use different left margins and different icons for each. Our list should looke something like this:

```
Nissan Maxima
  GXE
  SE
```

GLE

We also need to prevent the user from selecting models (e.g. "Nissan Maxima" above), since they do not provide complete information about a specific car, and only serve as separators between sets of trims.

Note: The hierarchical list organization shown here can easily be extended for use in a JList, and can handle an arbitrary number of levels. We only use two levels in this example, however, the design does not limit us to this.

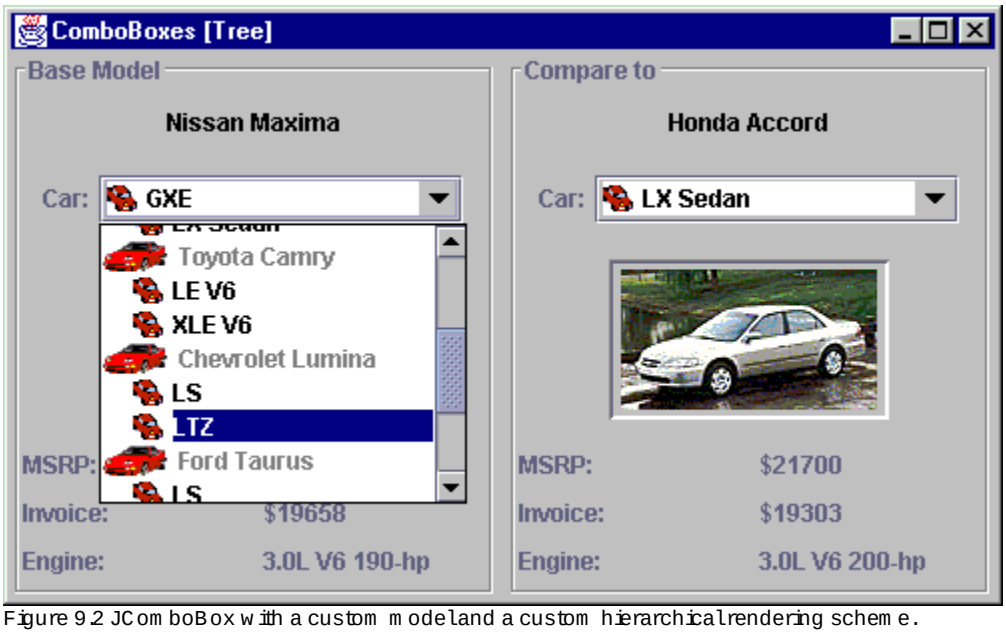


Figure 9.2 JComboBox with a custom model and a custom hierarchical rendering scheme.

<<file figure9-2.gif>>

The Code: ComboBox2.java

see \Chapter9\2

```
// Unchanged code from section 9.2

class CarPanel extends JPanel
{
  protected JComboBox m_cbCars;
  protected JLabel m_txtModel;
  protected JLabel m_lblImg;
  protected JLabel m_lblMSRP;
  protected JLabel m_lblInvoice;
  protected JLabel m_lblEngine;

  public CarPanel(String title, Vector cars) {
    super();
    setLayout(new BoxLayout(this, BoxLayout.Y_AXIS));
    setBorder(new TitledBorder(new EtchedBorder(), title));

    JPanel p = new JPanel();
    m_txtModel = new JLabel("");
    m_txtModel.setForeground(Color.black);
    p.add(m_txtModel);
    add(p);

    p = new JPanel();
```

```
    p.add(new JLabel("Car:"));
    CarComboBoxModel model = new CarComboBoxModel(cars);
    m_cbCars = new JComboBox(model);
    m_cbCars.setRenderer(new IconComboRenderer());
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        ListData data = (ListData)m_cbCars.getSelectedItem();
        Object obj = data.getObject();
        if (obj instanceof Trim)
          showTrim((Trim)obj);
      }
    };
    m_cbCars.addActionListener(lst);
    p.add(m_cbCars);
    add(p);

    //Unchanged code from section 9.2
  }

  public synchronized void selectCar(Car car) {
    for (int k=0; k < m_cbCars.getItemCount(); k++) {
      ListData obj = (ListData)m_cbCars.getItemAt(k);
      if (obj.getObject() == car) {
        m_cbCars.setSelectedItem(obj);
        break;
      }
    }
  }

  public synchronized void showTrim(Trim trim) {
    Car car = trim.getCar();
    m_txtModel.setText(car.toString());
    m_lblImg.setIcon(car.getIcon());
    m_lblMSRP.setText("$" + trim.getMSRP());
    m_lblInvoice.setText("$" + trim.getInvoice());
    m_lblEngine.setText(trim.getEngine());
  }
}

class ListData
{
  protected Icon    m_icon;
  protected int     m_index;
  protected boolean m_selectable;
  protected Object  m_data;

  public ListData(Icon icon, int index, boolean selectable,
   Object data) {
    m_icon = icon;
    m_index = index;
    m_selectable = selectable;
    m_data = data;
  }

  public Icon getIcon() { return m_icon; }

  public int getIndex() { return m_index; }

  public boolean isSelectable() { return m_selectable; }

  public Object getObject() { return m_data; }
```

```
  public String toString() { return m_data.toString(); }
}

class CarComboBoxModel extends DefaultComboBoxModel
{
  public static final ImageIcon ICON_CAR =
    new ImageIcon("car.gif");
  public static final ImageIcon ICON_TRIM =
    new ImageIcon("trim.gif");

  public CarComboBoxModel(Vector cars) {
    for (int k=0; k<cars.size(); k++) {
      Car car = (Car)cars.elementAt(k);
      addElement(new ListData(ICON_CAR, 0, false, car));

      Vector v = car.getTrims();
      for (int i=0; i < v.size(); i++) {
        Trim trim = (Trim)v.elementAt(i);
        addElement(new ListData(ICON_TRIM, 1, true, trim));
      }
    }
  }

  // This method only allows trims to be selected
  public void setSelectedItem(Object item) {
    if (item instanceof ListData) {
      ListData ldata = (ListData)item;
      if (!ldata.isSelectable()) {
        Object newItem = null;
        int index = getIndexOf(item);
        for (int k = index + 1; k < getSize(); k++) {
          Object item1 = getElementAt(k);
          if (item1 instanceof ListData) {
            ListData ldata1 = (ListData)item1;
            if (!ldata1.isSelectable())
              continue;
          }
          newItem = item1;
          break;
        }
        if (newItem==null)
          return;        // Selection failed
        item = newItem;
      }
    }
    super.setSelectedItem(item);
  }
}

class IconComboRenderer extends JLabel implements ListCellRenderer
{
  public static final int OFFSET = 16;

  protected Color m_textSelectionColor = Color.white;
  protected Color m_textNonSelectionColor = Color.black;
  protected Color m_textNonselectableColor = Color.gray;
  protected Color m_bkSelectionColor = new Color(0, 0, 128);
  protected Color m_bkNonSelectionColor = Color.white;
  protected Color m_borderSelectionColor = Color.yellow;

  protected Color  m_textColor;
  protected Color  m_bkColor;
```

```
protected boolean m_hasFocus;
protected Border[] m_borders;

public IconComboRenderer() {
  super();
  m_textColor = m_textNonSelectionColor;
  m_bkColor = m_bkNonSelectionColor;
  m_borders = new Border[20];
  for (int k=0; k < m_borders.length; k++)
    m_borders[k] = new EmptyBorder(0, OFFSET * k, 0, 0);
  setOpaque(false);
}

public Component getListCellRendererComponent(JList list,
 Object obj, int row, boolean sel, boolean hasFocus) {
  if (obj == null)
    return this;
  setText(obj.toString());
  boolean selectable = true;
  if (obj instanceof ListData) {
    ListData ldata = (ListData)obj;
    selectable = ldata.isSelectable();
    setIcon(ldata.getIcon());
    int index = 0;
    if (row >= 0)    // no offset for editor (row=-1)
      index = ldata.getIndex();
    Border b = (index < m_borders.length ? m_borders[index] :
      new EmptyBorder(0, OFFSET * index, 0, 0));
    setBorder(b);
  }
  else
    setIcon(null);

  setFont(list.getFont());
  m_textColor = (sel ? m_textSelectionColor :
    (selectable ? m_textNonSelectionColor :
    m_textNonselectableColor));
  m_bkColor = (sel ? m_bkSelectionColor :
    m_bkNonSelectionColor);
  m_hasFocus = hasFocus;
  return this;
}

public void paint (Graphics g) {
  Icon icon = getIcon();
  Border b = getBorder();

  g.setColor(m_bkNonSelectionColor);
  g.fillRect(0, 0, getWidth(), getHeight());

  g.setColor(m_bkColor);
  int offset = 0;
  if(icon != null && getText() != null) {
    Insets ins = getInsets();
    offset = ins.left + icon.getIconWidth() + getIconTextGap();
  }
  g.fillRect(offset, 0, getWidth() - 1 - offset,
    getHeight() - 1);

  if (m_hasFocus) {
    g.setColor(m_borderSelectionColor);
```

```
      g.drawRect(offset, 0, getWidth()-1-offset, getHeight()-1);
    }

    setForeground(m_textColor);
    setBackground(m_bkColor);
    super.paint(g);
  }
}
```

Understanding the Code

Class CarPanel

Classes `ComboBox2` (formerly `ComboBox1`), `Car`, and `Trim` remain unchanged in this example, so we'll start from the `CarPanel` class. Compared to the example in the previous section, we've removed combo box `m_cbTrims`, and added `JLabel m_txtModel`, which is used to display the current model's name (when the combo box popup is hidden, the user can see only the selected trim; so we need to display the corresponding model name separately). Curiously, the constructor of the `CarPanel` class places this label component in its own `JPanel` (using its default `FlowLayout`) to ensure it's location in the center of the base panel.

Note: The problem is that `JLabel m_txtModel` has a variable length, and the `BoxLayout` which manages `CarPanel` cannot dynamically center this component correctly. By placing this label in a `FlowLayout` panel it will always be centered.

The single combo box, `m_cbCars`, has a bit in common with the component of the same name in the previous example. First it receives a custom model, an instance of the `CarComboBoxModel` class, which will be described below. It also receives a custom renderer, an instance of the `IconComboRenderer` class, also described below.

The combo box is populated by both `Car` and `Trim` instances encapsulated in `ListData` objects (see below). This requires some changes in the `actionPerformed()` method which handles combo box selection. First we extract the data object from the selected `ListData` instance by calling the `getObject()` method. If this call returns a `Trim` object (as it should, since `Cars` cannot be selected), we call the `showTrim()` method to display the selected data.

Method `selectCar()` has been modified. As we mentioned above, our combo box now holds `ListData` objects, so we cannot pass a `Car` object as a parameter to the `setSelectedItem()` method. Instead we have to examine in turn all items in the combo box, cast them to `ListData` objects, and verify that the encapsulated data object is equal to the given `Car` instance. The `==` operator verifies that the address in memory of the object corresponding to the combo box is the same as the address of the given object. This assumes that the `Car` object passed to `selectCar()` is taken from the collection of objects used to populate this combo box. (To avoid this limitation we could alternatively implement an `equals()` method in the `Car` class.)

Method `showTrim()` now does the job of displaying the model data as well as the trim data. To do this we obtain a parent `Car` instance for a given `Trim` and display the model's name and icon. The rest of this method remains unchanged.

Class ListData

This class encapsulates the data object to be rendered in the combo box and adds new attributes for our rendering needs.

Instance variables:

`m_icon` `Icon`: icon associated with the data object.

`m_index int`: item's index which determines the left margin (i.e. the hierarchical level).

`m_selectable boolean`: flag indicating that this item can be selected.

`m_data Object`: encapsulated data object.

All variables are filled with parameters passed to the constructor. The rest of the `ListData` class represents four `getXX()` methods and a `toString()` method, which delegate calls to the `m_data` object.

Class CarComboBoxModel

This class extends `DefaultComboBoxModel` to serve as a data model for our combo box. First it creates two static `ImageIcons` to represent model and trim. The constructor takes a `Vector` of `Car` instances and converts them and their trims into a linear sequence of `ListData` objects. Each `Car` object is encapsulated in a `ListData` instance with an `ICON_CAR` icon, index set to `0`, and `m_selectable` flag set to `false`. Each `Trim` object is encapsulated in a `ListData` instance with `ICON_TRIM` icon, index set to `1`, and `m_selectable` flag set to `true`.

These manipulations could have been done without implementing a custom `ComboBoxModel`, of course. The real reason we do implement a custom model is to override the `setSelectedItem()` method to control item selection in the combo box. As we learned above, only `ListData` instances with the `m_selectable` flag set to `true` should be selectable. To achieve this goal, the overridden `setSelectedItem()` method casts the selected object to a `ListData` instance and examines its selection property using `isSelectable()`.

If `isSelectable()` returns `false`, a special action needs to be handled to move the selection to the first item following this item for which `isSelectable()` returns `true`. If no such item can be found our `setSelectedItem()` method returns and the selection in the combo box remains unchanged. Otherwise the `item` variable receives a new value which is finally passed to the `setSelectedItem()` implementation of the superclass `DefaultComboBoxModel`.

Note: You may notice that the `selectCar()` method discussed above selects a `Car` instance which cannot be selected. This internally triggers a call to the `setSelectedItem()` of the combo box model, which shifts the selection to the first available `Trim` item. You can verify this when running the example.

Class IconComboRenderer

This class extends `JLabel` and implements the `ListCellRenderer` interface to serve as a custom combo box renderer.

Instance variables:

`int OFFSET`: offset in pixels of image and text (different for cars and trims).

`Color m_textColor`: current text color.

`Color m_bkColor`: current background color.

`boolean m_hasFocus`: flag indicating whether this item has focus.

`Border[] m_borders`: an array of borders used for this component.

The constructor of the `IconComboRenderer` class initializes these variables. `EmptyBorders` are used to provide left margins while rendering components of the drop-down list. To avoid generation of numerous temporary objects, an array of 20 `Borders` is prepared with increasing left offsets corresponding to array index (incremented by `OFFSET`). This provides us with a set of different borders to use for white space in representing data at 20 distinct hierarchical levels.

Note: Even though we only use two levels in this example, `IconComboRenderer` has been designed for maximum reusability. 20 levels should be enough for most hierarchies, but if more levels are necessary we've designed `getListCellRendererComponent()` (see below) to create a new `EmptyBorder` in the event that more than 20 levels are used.

The `opaque` property is set to `false` because we intend to draw the background ourselves.

Method `getListCellRendererComponent()` is called prior to the painting of each cell in the drop-down list. We first set this component's text to that of the given object (passed as parameter). Then, if the object is an instance of `ListData`, we set the icon and left margin by using the appropriate `EmptyBorder` from the previously prepared array (based on the given `ListData`'s `m_index` property--if the index is greater than the). Note that a call to this method with `row=-1` will be invoked prior to the rendering of the combo box editor, which is the part of the combo box that is always visible (see 9.1). In this case we don't need to use any border offset. Offset only makes sense when there are hierarchical differences between items in the list, not when an item is rendered alone.

The rest of the `getListCellRendererComponent()` method determines the background and foreground colors to use, based on whether is selected and selectable, and stores them in instance variables for use within the `paint()` method. Non-selectable items receive their own foreground to distinguish them from selectable items.

The `paint()` method performs a bit of rendering before invoking the super-class implementation. It fills the background with the stored `m_bkColor` (from above) excluding the icon's area (note that the left margin is already taken into account by the component's `Border`). It also draws a border-like rectangle if the component currently has the focus. This method then ends with a call to its super-class's `paint()` method which takes responsibility for painting the label text and icon

Running the Code

Figure 9.2 shows our hierarchical drop-down list in action. Note that models and trim lines can be easily differentiated because of the varying icons and offsets. In addition, models have a gray foreground to imply that they cannot be selected.

This implementation is more user-friendly than the previous example because it displays all available data in a single drop-down list. Try selecting different trims and note how this changes data for both the model and trim information labels. Try selecting a model and note that it will result in the selection of the first trim of that model.

UI Guideline: Improved Usability

From a usability perspective the solution in fig 9.2 is an improvement over the one presented in fig 9.1. By using a combobox with a hierarchical data model, the designer has reduced the data entry to a single selection and has presented the information in an accessible and logical manner which also produces a visually cleaner result.

Further improvements could be made here by sorting the hierarchical data. In this example it would seem appropriate to sort in a two tiered fashion: alphabetically by manufacturer; and alphabetically by model. Thus Toyota would come after Ford and Toyota Corolla would come after Toyota Camry.

This is an excellent example of how the programmer can improve UI Design and Usability by doing additional work to make the User's Goal easier to achieve.

9.4 Comboboxes with memory

In some situations it is desirable to use editable combo boxes which keep a historical list of choices for future reuse. This conveniently allows the user to select a previous choice rather than typing identical text. A typical example of an editable combo box with memory can be found in find/replace dialogs in many modern applications. Another example, familiar to almost every modern computer user, is provided in many Internet browsers which use an editable URL combo box with history mechanism. These combo boxes accumulate typed addresses so the user can easily return to any previously visited site by selecting it from the drop-down list instead of manually typing it in again.

The following example shows how to create a simple browser application using an editable combo box with memory. It uses the serialization mechanism to save data between program sessions, and the JEditorPane component (described in more detail in chapters 11 and 19) to display non-editable HTML files.

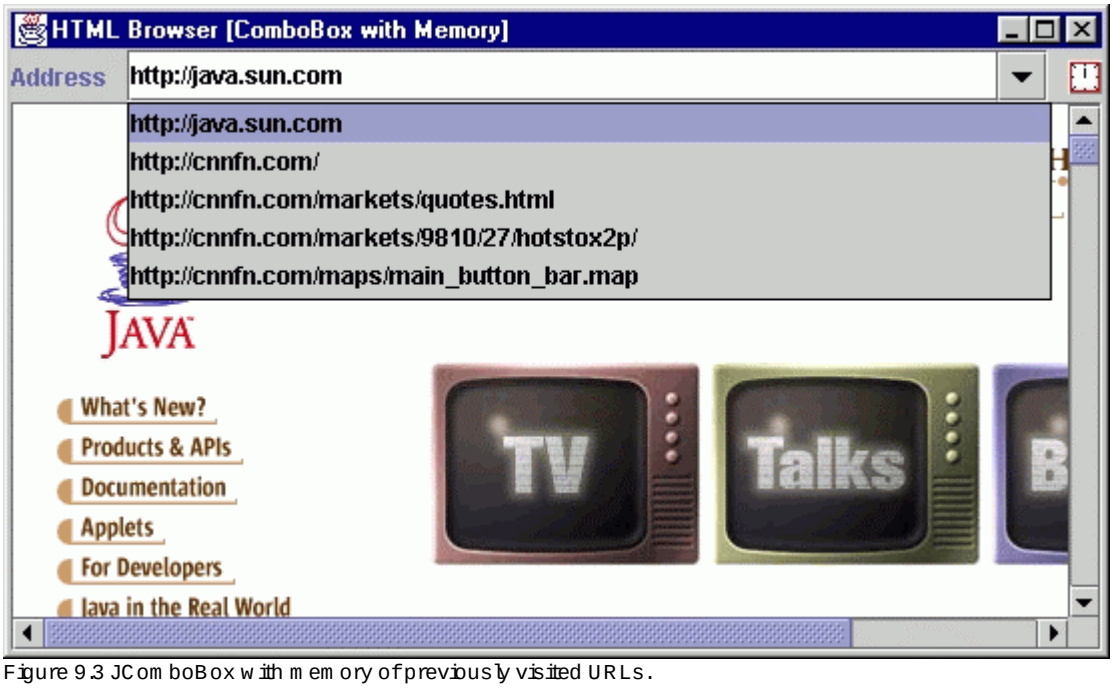


Figure 9.3 JComboBox with memory of previously visited URLs.
<<file figure9-3.gif>>

The Code: Browser.java
see \Chapter9\3

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

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import javax.swing.text.html.*;

public class Browser extends JFrame
{
  protected JEditorPane m_browser;
  protected MemComboBox m_locator;
  protected AnimatedLabel m_runner;
```

```
  public Browser() {
    super("HTML Browser [ComboBox with Memory]");
    setSize(500, 300);

    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
    p.add(new JLabel("Address"));
    p.add(Box.createRigidArea(new Dimension(10, 1)));

    m_locator = new MemComboBox();
    m_locator.load("addresses.dat");
    BrowserListener lst = new BrowserListener();
    m_locator.addActionListener(lst);

    p.add(m_locator);
    p.add(Box.createRigidArea(new Dimension(10, 1)));

    m_runner = new AnimatedLabel("clock", 8);
    p.add(m_runner);
    getContentPane().add(p, BorderLayout.NORTH);

    m_browser = new JEditorPane();
    m_browser.setEditable(false);
    m_browser.addHyperlinkListener(lst);

    JScrollPane sp = new JScrollPane();
    sp.getViewport().add(m_browser);
    getContentPane().add(sp, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        m_locator.save("addresses.dat");
          System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
    m_locator.grabFocus();
  }

  class BrowserListener implements ActionListener, HyperlinkListener
  {
    public void actionPerformed(ActionEvent evt) {
      String sUrl = (String)m_locator.getSelectedItem();
      if (sUrl == null || sUrl.length() == 0 ||
       m_runner.getRunning())
        return;
      BrowserLoader loader = new BrowserLoader(sUrl);
      loader.start();
    }

    public void hyperlinkUpdate(HyperlinkEvent e) {
      URL url = e.getURL();
      if (url == null || m_runner.getRunning())
        return;
      BrowserLoader loader = new BrowserLoader(url.toString());
      loader.start();
    }
  }

  class BrowserLoader extends Thread
```

```
  {
    protected String m_sUrl;

    public BrowserLoader(String sUrl) { m_sUrl = sUrl; }

    public void run() {
      setCursor( Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
      m_runner.setRunning(true);

      try {
        URL source = new URL(m_sUrl);
        m_browser.setPage(source);
        m_locator.add(m_sUrl);
      }
      catch (Exception e) {
        JOptionPane.showMessageDialog(Browser.this,
          "Error: "+e.toString(),
          "Warning", JOptionPane.WARNING_MESSAGE);
      }
      m_runner.setRunning(false);
      setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
    }
  }

  public static void main(String argv[]) { new Browser(); }
}

class MemComboBox extends JComboBox
{
  public static final int MAX_MEM_LEN = 30;

  public MemComboBox() {
    super();
    setEditable(true);
  }

  public void add(String item) {
    removeItem(item);
    insertItemAt(item, 0);
    setSelectedItem(item);
    if (getItemCount() > MAX_MEM_LEN)
      removeItemAt(getItemCount()-1);
  }

  public void load(String fName) {
    try {
      if (getItemCount() > 0)
        removeAllItems();
      File f = new File(fName);
      if (!f.exists())
        return;
      FileInputStream fStream =
        new FileInputStream(f);
      ObjectInput  stream  =
        new  ObjectInputStream(fStream);
      Object obj = stream.readObject();
      if (obj instanceof ComboBoxModel)
        setModel((ComboBoxModel)obj);
      stream.close();
      fStream.close();
    }
    catch (Exception e) {
```

```
      e.printStackTrace();
      System.err.println("Serialization error: "+e.toString());
    }
  }

  public void save(String fName) {
    try {
      FileOutputStream fStream =
        new FileOutputStream(fName);
      ObjectOutput  stream  =
        new  ObjectOutputStream(fStream);
      stream.writeObject(getModel());
      stream.flush();
      stream.close();
      fStream.close();
    }
    catch (Exception e) {
      e.printStackTrace();
      System.err.println("Serialization error: "+e.toString());
    }
  }
}

class AnimatedLabel extends JLabel implements Runnable
{
  protected Icon[] m_icons;
  protected int m_index = 0;
  protected boolean m_isRunning;

  public AnimatedLabel(String gifName, int numGifs) {
    m_icons = new Icon[numGifs];
    for (int k=0; k<numGifs; k++)
      m_icons[k] = new ImageIcon(gifName+k+".gif");
    setIcon(m_icons[0]);

    Thread tr = new Thread(this);
    tr.setPriority(Thread.MAX_PRIORITY);
    tr.start();
  }

  public void setRunning(boolean isRunning) {
    m_isRunning = isRunning;
  }

  public boolean getRunning() { return m_isRunning; }

  public void run() {
    while(true) {
      if (m_isRunning) {
        m_index++;
        if (m_index >= m_icons.length)
          m_index = 0;
        setIcon(m_icons[m_index]);
        Graphics g = getGraphics();
        m_icons[m_index].paintIcon(this, g, 0, 0);
      }
      else {
        if (m_index > 0) {
          m_index = 0;
          setIcon(m_icons[0]);
        }
      }
```

```
      try { Thread.sleep(500); } catch(Exception ex) {}
    }
  }
}
```

Understanding the Code

Class Browser

This class extends `JFrame` to implement the frame container for our browser. Instance variables:

`JEditorPane m_browser`: text component to parse and render HTML files.

`MemComboBox m_locator`: combo box to enter/select URL address.

`AnimatedLabel m_runner`: traditional animated icon alive while the browser is requesting a URL.

The constructor creates the custom combo box, `m_locator`, and an associated explanatory label. Then it creates the `m_runner` icon and places all three components in the northern region of our frame's content pane. `JEditorPane m_browser` is created and placed in a `JScrollPane` to provide scrolling capabilities. This is then added to the center of the content pane.

Note that the `WindowListener`, as used in many previous examples to close the frame and terminate execution, receives an additional function: it invokes our custom `save()` method (see below) on our custom combo box component before destroying the frame. This saves the list of visited URLs entered as a file called "addresses.dat" in the current running directory.

Class Browser.BrowserListener

This inner class implements both the `ActionListener` and `HyperlinkListener` interfaces to manage navigation to HTML pages. The `actionPerformed()` method is invoked when the user selects a new item in the combo box. It verifies that the selection is valid and the browser is not currently running (i.e. requesting a URL). If these checks are passed it then creates and starts a new `BrowserLoader` instance (see below) for the specified address.

Method `hyperlinkUpdate()` is invoked when the user clicks a hyperlink in the currently loaded web page. This method also determines the selected URL address and starts a new `BrowserLoader` to load it.

Class Browser.BrowserLoader

This inner class extends `Thread` to load web pages into the `JEditorPane` component. It takes a URL address parameter in the constructor and stores it in a instance variable. The `run()` method sets the mouse cursor to hourglass (`Cursor.WAIT_CURSOR`) and starts the animated icon to indicate that the browser is busy.

The core functionality of this thread is enclosed in its `try/catch` block. If an exception occurs during processing of the requested URL, it is displayed in simple dialog message box (we will learn discuss `JOptionPane` in chapter 14).

The actual job of retrieving, parsing, and rendering the web page is hidden in a single call to the `setPage()` method. So why do we need to create this separate thread instead of making that simple call, say, in `BrowserListener`? The reason is, as we discussed in chapter 2, by creating separate threads to do potentially time-consuming operations we avoid clogging up the event-dispatching thread.

Class MemComboBox

This class extends `JComboBox` to add a historical mechanism for this component. The constructor creates an underlying `JComboBox` component and sets its `editable` property to `true`.

The `add()` method adds a new text string to the beginning of the list. If this item is already present in the list, it is removed from the old position. If the resulting list is longer than the pre-defined `maximum` length then the last item in the list is truncated.

Method `load()` loads a previously stored `ComboBoxModel` from file "addresses.dat" using the serialization mechanism. The significant portion of this method reads an object from an `ObjectInputStream` and sets it as the `ComboBoxModel`. Note that any possible exceptions are only printed to the standard output and purposefully do not distract the user (since this serialization mechanism should be considered an optional feature).

Similarly, the `save()` method serializes our combo box's `ComboBoxModel`. Any possible exceptions are, again, printed to standard output and do not distract the user.

Class AnimatedLabel

Surprisingly, Swing does not provide any special support for animated components, so we have to create our own component for this purpose. This provides us with an interesting example of using threads in Java.

Note: Animated GIFs are fully supported by `ImageIcon` (see chapter 5) but we want complete control over each animated frame here.

`AnimatedLabel` extends `JLabel` and implements the `Runnable` interface. Instance variables:

`Icon[] m_icons`: an array of images to be used for animation.

`int m_index`: index of the current image.

`boolean m_isRunning`: flag indicating whether the animation is running.

The constructor takes a common name of a series of GIF files containing images for animation, and the number of those files. These images are loaded and stored into an array. When all images are loaded a thread with maximum priority is created and started to run this `Runnable` instance.

The `setRunning()` and `getRunning()` methods simply manage the `m_isRunning` flag.

In the `run()` method we cyclically increment the `m_index` variable and draw an image from the `m_icons` array with the corresponding index, exactly as you would expect from an animated image. This is done only when the `m_isRunning` flag is set to `true`. Otherwise, the image with index 0 is displayed. After an image is painted, `AnimatedLabel` yields control to other threads and sleeps for 500 ms.

The interesting thing about this component is that it runs in parallel with other threads which do not necessary yield control explicitly. In our case the concurrent `BrowserLoader` thread spends the main part of its time inside the `setPage()` method, and our animated icon runs in a separate thread signaling to the user that something is going on. This is made possible because this animated component is running in the thread with the maximum priority. Of course, we should use such thread priority with caution. In our case it is appropriate since our thread consumes only a small amount of the processor's time and does yield control to the lesser-priority threads (when it sleeps).

Note: As a good exercise try using threads with normal priority or Swing's `Timer` component in this example. You will find that this doesn't work as expected: the animated icon does not show any animation while the browser is running.

Running the Code

Figure 9.3 shows the `Browser` application displaying a web page. Note that the animated icon comes to life

when the browser requests a URL. Also note how the combo box is populated with URL addresses as we navigate to different web pages. Now quit the application and re-start it. Note that our addresses have been saved and restored (by serializing the combo box model, as discussed above).

Note: HTML rendering functionality is not yet matured. Do not be surprised if your favorite web page looks signigicantly different in our Swing-based browser. As a matter of fact even the JavaSoft home page throws several exceptions while being displayed in this Swing component. (These exceptions occur outside our code, during the JEditorPane rendering--this is why they are not caught and handled by our code.)

UI Guideline: Usage of a Memory Combobox

The example given here is a good usage for such a device. However, a memory combobox will not always be appropriate. Remember the advice that usability of an unsorted comboboxes tends to degrade rapidly as the number of items grows. Therefore, it is sensible to deploy this technique where the likelihood of more than say 20 entries is very small. The browser example is good because it is unlikely that a user would type more than 20 URLs in a single web surfing session.

Where you have a domain problem which is likely to need a larger number of memory items but you still want to use a memory combobox, consider adding a sorting algorithm, so that rather than most recent first, you sort into a meaningful index such as alphabetical order. This will improve usability and mean that you could easily populate the list up to 2 or 3 hundred items.

9.5 Custom editing

In this section we will discuss a custom editing feature to make the example from the last section even more convenient and similar to modern browser applications. We will attach a key event listener to our combo box's editor and search for previously visited URLs with matching beginning strings. If a match occurs the remainder of that URL is displayed in the editor, and by pressing Enter we can accept the suggestion. Most modern browsers also provide this functionality.

Note that the caret position will remain unchanged as well as the text on the left side of the caret (i.e. the text most likely typed by the user). The text on the right side of the caret represents the browser's suggestion which may or may not correspond to the user's intentions. To avoid distracting the user, this portion of the text is highlighted, so any newly typed character will replace that suggested text.

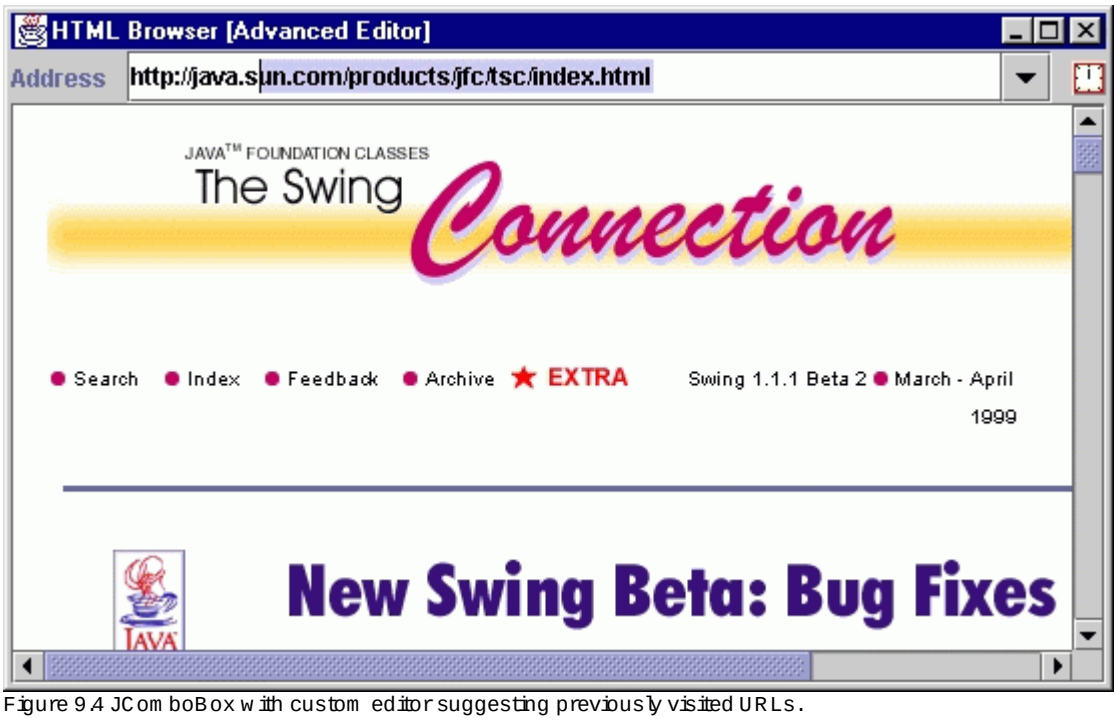


Figure 9.4 JComboBox with custom editor suggesting previously visited URLs.

<<file figure9-4.gif>>

The Code: Browser.java

see \Chapter9\4

```
public class Browser extends JFrame
{
  // Unchanged code from section 9.4

  public Browser() {
    super("HTML Browser [Advanced Editor]");

    // Unchanged code from section 9.4

    MemComboAgent agent = new MemComboAgent(m_locator);

    // Unchanged code from section 9.4
  }
  // Unchanged code from section 9.4
}

class MemComboAgent extends KeyAdapter
{
  protected JComboBox   m_comboBox;
  protected JTextField  m_editor;

  public MemComboAgent(JComboBox comboBox) {
    m_comboBox = comboBox;
    m_editor = (JTextField)comboBox.getEditor().
      getEditorComponent();
    m_editor.addKeyListener(this);
  }

  public void keyReleased(KeyEvent e) {
    char ch = e.getKeyChar();
    if (ch == KeyEvent.CHAR_UNDEFINED || Character.isISOControl(ch))
```

```
      return;
    int pos = m_editor.getCaretPosition();
    String str = m_editor.getText();
    if (str.length() == 0)
      return;

    for (int k=0; k<m_comboBox.getItemCount(); k++) {
      String item = m_comboBox.getItemAt(k).toString();
      if (item.startsWith(str)) {
        m_editor.setText(item);
        m_editor.setCaretPosition(item.length());
        m_editor.moveCaretPosition(pos);
        break;
      }
    }
  }
}
```

Understanding the Code

Class Browser

This class has only one change in comparison with the previous example: it creates an instance of our custom `MemComboAgent` class and passes it a reference to our `m_locator` combo box.

Class MemComboAgent

This class extends `KeyAdapter` to listen for keyboard activity. It takes a reference to a `JComboBox` component and stores it in an instance variable along with the `JTextField` component used as that combo box's editor. Finally, a `MemComboAgent` object adds itself to that editor as a `KeyListener` to be notified of all keyboard input that is passed to the editor component.

Method `keyReleased()` is the only method we implement. First this method retrieves the pressed characters and verifies that they are not control characters. We also retrieve the contents of the text field and check that it is not empty (to avoid annoying the user with suggestions in an empty field). Note that when this method is invoked the pressed key will already have been included in this text.

This method then walks through the list of combo box items and searches for an item starting with the combo box editor text. If such an item is found it is set as the combo box editor's text. Then we place the caret at the end of that string using `setCaretPosition()`, and move it back to its initial position in the backward direction using the `moveCaretPosition()` method. This final `JTextComponent` method places the caret in its original position and highlights all text to its right (see chapters 11 and 19).

Note: A more sophisticated realization of this idea may include separate processing of URL protocol and host, as well as using threads for smooth execution.

Running the Code

Figure 9.4 shows our custom combo box's editor displaying a portion of a URL address taken from its list. Try entering some new addresses and browsing to them. After some experimentation, try typing in an address that you have already visited with this application. Notice that the enhanced combo box suggests the remainder of this address from its pull-down list. Press "Enter" as soon as an address matches your intended selection to avoid typing the complete URL.

Chapter 10. List Boxes

In this chapter:

- JList
- Basic JList example
- Custom renderer
- Processing keyboard input and searching
- List of check boxes

10.1 List API overview

class javax.swing.JList

This class represents a basic GUI component allowing the selection of one or more items from a list of choices. JList has two models: ListModel which handles data in the list, and ListSelectionModel which handles item selection (three different selection modes are supported which we will discuss below). JList also supports custom rendering, as we learned in the last chapter, through the implementation of the ListCellRenderer interface. We can use existing default implementation of ListCellRenderer (DefaultListCellRenderer) or create our own according to our particular needs (which we will see later in ths chapter). Note that unless we use a custom renderer, the default renderer will display each element as a String defined by that object's toString() method (the only exceptions to this are Icon implementations which will be renderered as they would be in any JLabel). Also note that a ListCellRenderer returns a Component, but that component is not interactive and is only used for display purposes (i.e. it acts as a "rubber stamp"[API]). For instance, if a JCheckBox is used as a renderer we will not be able to check and uncheck it. Unlike JComboBox, however, JList does not support editing of any sort.

A number of constructors are available to create a JList component. We can use the default constructor or pass list data to a constructor as a one-dimensional array, a Vector, or as an implementation of the ListModel interface. The last variant provides maximum control over a list's properties and appearance. We can also assign data to a JList using the setModel() method, or one of the overloaded setListData() methods.

JList does not provide direct access to its elements, and we must access its ListModel to gain access to this data. JList does, however, provide direct access to its selection data by implementing all ListSelectionModel methods, and delegating their traffic to the actual ListSelectionModel instance. To avoid repetition we will discuss selection funcionality below in our overview of ListSelectionModel.

JList maintains selection foreground and background colors (assigned by its UI delegate when installed), and the default cell renderer, DefaultListCellRenderer, will use these colors to render selected cells. These colors can be assigned with setSelectedForeground() and setSelectedBackground(). Nonselected cells will be rendered with the component foreground and background colors assigned to JList with setForeground() and setBackground().

JList implements the Scrollable interface (see chapter 7) to provide vertical unit incremental scrolling corresponding the list cell height, and vertical block incremental scrolling corresponding to the number of visible cells. Horizontal unit increment scrolling corresponds to the size of the list's font (1 if the font is null), and horizontal block unit increment scrolling corresponds to the current width of the list. Thus JList does not directly support scrolling and is intended to be placed in a JScrollPane.

The `visibleRowCount` property specifies how many cells should be visible when a `JList` is placed in a scroll pane. This defaults to 8 and can be set with the `setVisibleRowCount()` method. Another interesting method provided by `JList` is `ensureIndexIsVisible()`, which forces the list to scroll itself so that the element corresponding to the given index becomes visible. Note that `JList` also supports autoscrolling (i.e. it will scroll element by element every 100ms if the mouse is dragged below or above its bounds and if an item has been selected).

By default the width of each cell is the width of the widest item, and the hieght of each cell corresponds to the hieght of the tallest item. We can overpower this behavior and specify our own fixed cell width and height of each list cell using the `setFixedCellWidth()` and `setFixedCellHeight()` methods.

Another way to control the width and height of each cell is through use of the `setPrototypeCellValue()` method. This method takes an `Object` parameter and uses it to automatically determine the `fixedCellWidth` and `fixedCellHeight`. A typical use of this method would be to give it a `String`. This forces the list to use a fixed cell width and hieght equal to the width and hieght of that string when rendered in the `Font` currently assigned to the `JList`.

`JList` also provides a method called `locationToIndex()` which will return the index of a cell at the given `Point` (in list coordinates). `-1` will be returned if none is found. Unfortunately `JList` does not provide support for double-clicking, but this method comes in very handy in implementing our own. The following psuedocode shows how we can use a `MouseAdapter`, `MouseEvent`, and the `locationToIndex()` method to determine the `JList` cell a double-click occurs on:

```
myJist.addMouseListener( new MouseAdapter() {
  public void mouseClicked(MouseEvent e) {
    if (e.getClickCount() == 2) {
      int cellIndex = myJList.locationToIndex(e.getPoint());
      // We now have the index of the double-clicked cell..
    }
  }
});
```

10.1.1 The ListModel interface

abstract interface javax.swing.ListModel

This interface describes a data model that holds a list of items. Method `getElementAt()` retrieves the item at the given position as an `Object` instance. Method `getSize()` returns the number of items in the list. It also contains two methods allowing the registration of `ListDataListeners` (see below) to be notified of additions, removals, and any changes that occur to this model. Note that this interface leaves the job of specifying how we store and structure the data, as well as how we add, remove, or change an item, completely up to its implementations.

10.1.2 AbstractListModel

abstract class javax.swing.AbstractListModel

This class represents a partial implementation of the `ListModel` interface. It defines the default event handling functionality, and implements the add/remove `ListDataListener` methods, as well as methods to fire `ListDataEvents` (see below) when additions, removals, and changes occur. The remainder of `ListModel` methods `getElementAt()` and `getSize()`, must be implemented in any concrete sub-class.

10.1.3 DefaultListModel

class javax.swing.DefaultListModel

This class represents the concrete default implementation of the `ListModel` interface. It extends `AbstractListModel` and uses a `java.util.Vector` to store its data. Almost all of the methods of this class correspond directly to `Vector` methods and we will not discuss them here. Familiarity with `Vectors` implies familiarity with how `DefaultListModel` works (see API docs).

10.1.4 The ListSelectionModel interface

abstract interface javax.swing.ListSelectionModel

This interface describes the model used for selecting list items. It defines three modes of selection: single selection, single contiguous interval selection, and multiple contiguous interval selection. A selection is defined as an indexed range, or set of ranges, of list elements. The beginning of a selected range (where it originated) is referred to as the anchor, while the last item is referred to as the lead (the anchor can be greater than, less than, or equal to the lead). The lowest selected index is referred to as the minimum, and the highest selected index is referred to as the maximum (regardless of the order in which selection takes place). Each of these indices represents a `ListSelectionModel` property. The minimum and maximum properties should be `-1` when no selection exists, and the anchor and lead maintain their most recent value until a new selection occurs.

To change selection mode we use the `setSelectionMode()` method, passing it one of the following constants: `MULTIPLE_INTERVAL_SELECTION`, `SINGLE_INTERVAL_SELECTION`, and `SINGLE_SELECTION`. In `SINGLE_SELECTION` mode only one item can be selected. In `SINGLE_INTERVAL_SELECTION` mode a contiguous group of items can be selected by selecting an anchor item, holding down the SHIFT key, and choosing a lead item (which can be at a higher or lower index than the anchor). In `MULTIPLE_INTERVAL_SELECTION` mode any number of items can be selected regardless of their location by holding down the CTRL key and clicking. Multiple selection mode also allows use of SHIFT to select a contiguous interval, however, this has the effect of clearing the current selection.

`ListSelectionModel` provides several methods for adding, removing, and manipulating ranges of selections. Methods for registering/removing `ListSelectionListeners` (see below) are provided as well. Each of these methods is self-explanitory in the API docs and we will not describe them in detail here.

`JList` defines all the methods declared in this interface and simply delegates all traffic to its `ListSelectionModel` instance. This allows access to selection data without the need to directly communicate with the selection model.

10.1.5 DefaultListSelectionModel

class javax.swing.DefaultListSelectionModel

This class represents the concrete default implementation of the `ListSelectionModel` interface. It defines methods to fire `ListSelectionEvents` (see below) when a selection range changes.

10.1.6 The ListCellRenderer interface

abstract interface javax.swing.ListCellRenderer

This interface describes a component used for rendering a list item. We discussed this interface, as well as its default concrete implementation, `DefaultListCellRenderer`, in the last chapter (see 9.1.4 and 9.1.4). We will show how to construct several custom renderers in the examples that follow.

10.1.7 The ListDataListener interface

abstract interface javax.swing.event.ListDataListener

Defines three methods for dispatching `ListDataEvents` when list elements are added, removed or changed in the `ListModel`: `intervalAdded()`, `intervalRemoved()`, and `contentsChanged()`.

10.1.8 ListDataEvent

class javax.swing.event.ListDataEvent

This class represents the event delivered when changes occur in a list's `ListModel`. It includes the source of the event as well as the index of the lowest and highest indexed elements affected by the change. It also includes the type of event that occurred. Three `ListDataEvent` types are defined as static `ints`: `CONTENTS_CHANGED`, `INTERVAL_ADDED`, and `INTERVAL_REMOVED`. We can use the `getType()` method to discover the type of any `ListDataEvent`.

10.1.9 The ListSelectionListener interface

abstract interface javax.swing.event.ListSelectionListener

This interface describes a listener which listens for changes in a list's `ListSelectionModel`. It declares the `valueChanged()` method which accepts a `ListSelectionEvent`.

10.1.10 ListSelectionEvent

class javax.swing.event.ListSelectionEvent

This class represents an event delivered by `ListSelectionModel` when changes occur in its selection. It is almost identical to `ListDataEvent`, except that the indices specified signify where there has been a change in the selection model, rather than the data model.

UI Guideline: Advice on Usage and Design Usage

Much of the UI Guideline advice for List Boxes is similar to that of Comboboxes. Clearly the two things are different and are intended for different purposes. Deciding when to use one or another can be difficult. Our advice is to think about reader output rather than data input. When the reader needs to see a collection of items then a List Box is the correct choice. Use a List Box where there is a collection of data, which may grow dynamically, and for reading purposes it is useful to see the whole collection or as much of the collection as can reasonably be fitted in the available space, e.g. DepartmentStaff<ListofStaffName>.

Design

Like Comboboxes, there are a number of things which affect the usability of a List Box. Beyond more than a few items, they become unusable unless the data is sorted in some logical fashion e.g. alphabetical, numerical. List Boxes are designed to be used with ScrollPane. It is assumed that the list will most often be too long to display in the available screen space. Using a sensible sorted order for the list, allows the user to predict how much they need to scroll to find what they are looking for.

When a list gets longer, usability is affected again. Once a list gets beyond a couple of hundred items, even when sorted, it becomes very slow for the user to locate specific item in the list. When a list becomes so long, it may be better to consider providing a Search Facility, or grouping the data inside the list using a Tree.

Graphical considerations for List Boxes are much like those for Comboboxes. List Boxes should be aligned to fit attractively into a panel. However, this can be problematic. You must avoid making a List Box which is simply too big for the list items contained e.g. a List Box showing supported file formats such as .gif need only be a few characters long, don't make it big enough to take 50 characters, as it will look unbalanced.

The nature of the list items must also be considered. If you have 50 items in a list where most items are around 20 characters but one item is 50 characters long then should you make the ListBox big enough to display the longer one? Well maybe, but for most occasions your display will be unbalanced again. It is probably best to optimise for the more common length, providing the the longer one still has meaning when read in its truncated form. One solution to displaying the whole length of a truncated item is to use the tooltip facility. When the User places the mouse over an item, a tooltip appears with the full length data.

10.2 Basic JList example

This example displays a list of the united states using an array of Strings in the following format:

2-character abbreviation<tab character>full name<tab character>capital

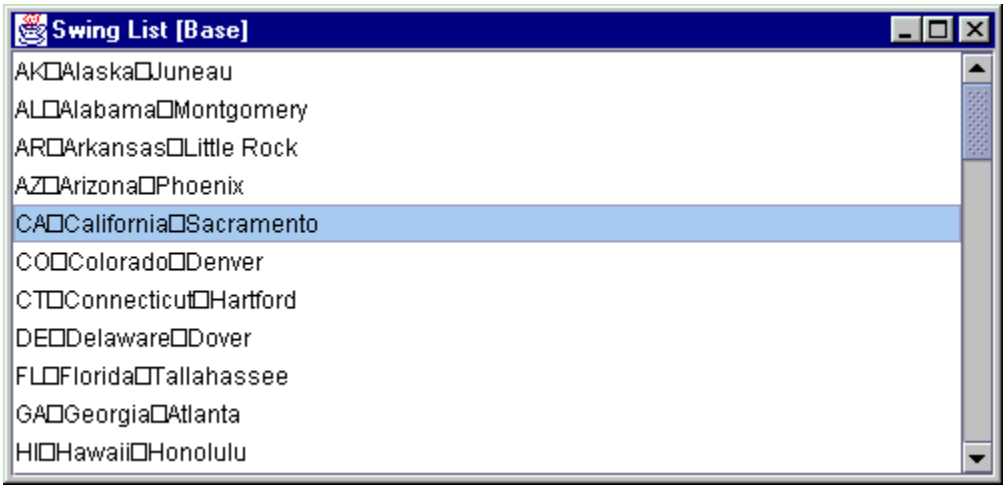


Figure 10.1 A JList displaying a list of Strings containing tab characters.
<<file figure10-1.gif>>

The Code: StatesList.java
see \Chapter10\1

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class StatesList extends JFrame
{
  protected JList m_statesList;

  public StatesList() {
    super("Swing List [Base]");
    setSize(500, 240);

    String [] states = {
      "AK\tAlaska\tJuneau",
      "AL\tAlabama\tMontgomery",
      "AR\tArkansas\tLittle Rock",
      "AZ\tArizona\tPhoenix",
      "CA\tCalifornia\tSacramento",
```

```
      "CO\tColorado\tDenver",
      "CT\tConnecticut\tHartford",
      "DE\tDelaware\tDover",
      "FL\tFlorida\tTallahassee",
      "GA\tGeorgia\tAtlanta",
      "HI\tHawaii\tHonolulu",
      "IA\tIowa\tDes Moines",
      "ID\tIdaho\tBoise",
      "IL\tIllinois\tSpringfield",
      "IN\tIndiana\tIndianapolis",
      "KS\tKansas\tTopeka",
      "KY\tKentucky\tFrankfort",
      "LA\tLouisiana\tBaton Rouge",
      "MA\tMassachusetts\tBoston",
      "MD\tMaryland\tAnnapolis",
      "ME\tMaine\tAugusta",
      "MI\tMichigan\tLansing",
      "MN\tMinnesota\tSt.Paul",
      "MO\tMissouri\tJefferson City",
      "MS\tMississippi\tJackson",
      "MT\tMontana\tHelena",
      "NC\tNorth Carolina\tRaleigh",
      "ND\tNorth Dakota\tBismarck",
      "NE\tNebraska\tLincoln",
      "NH\tNew Hampshire\tConcord",
      "NJ\tNew Jersey\tTrenton",
      "NM\tNew Mexico\tSantaFe",
      "NV\tNevada\tCarson City",
      "NY\tNew York\tAlbany",
      "OH\tOhio\tColumbus",
      "OK\tOklahoma\tOklahoma City",
      "OR\tOregon\tSalem",
      "PA\tPennsylvania\tHarrisburg",
      "RI\tRhode Island\tProvidence",
      "SC\tSouth Carolina\tColumbia",
      "SD\tSouth Dakota\tPierre",
      "TN\tTennessee\tNashville",
      "TX\tTexas\tAustin",
      "UT\tUtah\tSalt Lake City",
      "VA\tVirginia\tRichmond",
      "VT\tVermont\tMontpelier",
      "WA\tWashington\tOlympia",
      "WV\tWest Virginia\tCharleston",
      "WI\tWisconsin\tMadison",
      "WY\tWyoming\tCheyenne"
    };

    m_statesList = new JList(states);

    JScrollPane ps = new JScrollPane();
    ps.getViewport().add(m_statesList);
    getContentPane().add(ps, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }
```

```
  public static void main(String argv[]) {
    new StatesList();
  }
}
```

Understanding the Code

Class StatesList

Class `StatesList` extends `JFrame` to implement the frame container for this example. One instance variable, `JList m_statesList`, is used to store an array of state `Strings` (as described above). This list is created by passing the states `String` array to the `JList` constructor. It is then added to a `JScrollPane` instance to provide scrolling capabilities.

Running the Code

Figure 10.1 shows `StatesList` in action displaying the list of states and their capitals. Note that the separating tab character is displayed as an unpleasant square symbol (we'll fix this in the next example).

UI Guideline: Unbalanced Layout

In this example, the design is unbalanced due to the tab character not being displayed correctly. The box is ugly but the spacing is also wrong. The large whitespace area to the right ought to be avoided. The next example corrects this.

10.3 Custom rendering

In this section we'll add the ability to align `Strings` containing tab separators into a table-like arrangement. We want each tab character to shift all text to its right, to a specified location instead of being rendered as the square symbol we saw above. These locations should be determined uniformly for all elements of the list to form columns that line up correctly.

Note that this example works well with proportional fonts as well as with fixed width fonts (i.e. it doesn't matter what font we use because alignment is not designed to be font-dependent). This makes `JList` a powerful but simple component, which can be used in place of `JTable` in simple cases such as the example presented here (where the involvement of `JTable` would create unnecessary overhead).

To accomplish the desired rendering we construct a custom renderer, `TabListCellRenderer`, which exposes accessor methods to specify and retreive tab positions based on the index of a tab character in a `String` being rendered:

- `getDefaultTab()/setDefaultTab(int)`: manages the default tab size (defaults to 50). In case a position is not specified for a given tab index, we use a default size to determine how far to offset a portion of text.
- `getTabs()/setTabs(int[])`: manages an array of positions based on the index of a tab character in a `String` being rendered. These positions used in rendering each element in the list to provide consisitent alignment.

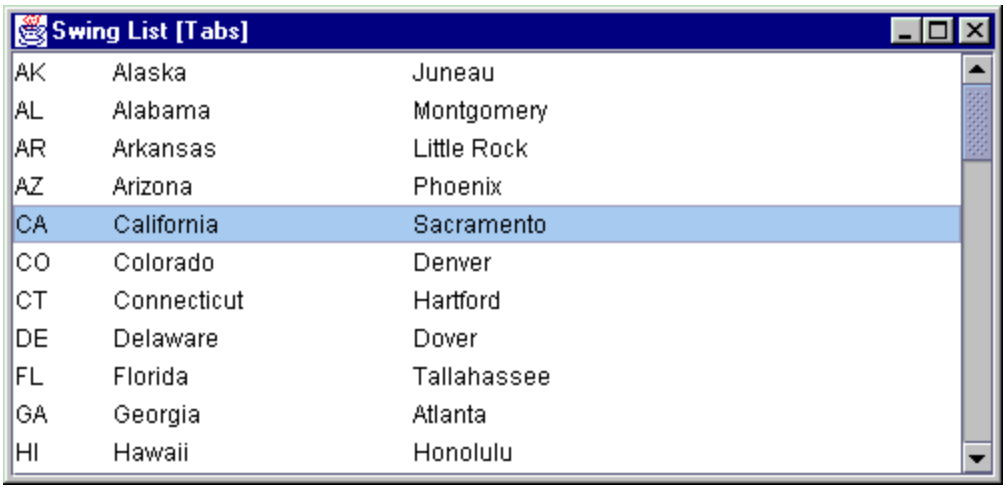


Figure 10.2 Custom ListCellRenderer to display tab-separated Strings in a table-like fashion.

<<file figure10-2.gif>>

The Code: StatesList.java

see \Chapter10\2

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class StatesList extends JFrame
{
  protected JList m_statesList;

  public StatesList() {
    // Unchanged code from section 10.2

    m_statesList = new JList(states);
    TabListCellRenderer renderer = new TabListCellRenderer();
    renderer.setTabs(new int[] {50, 200, 300});
    m_statesList.setCellRenderer(renderer);

    // Unchanged code from section 10.2
  }
}

class TabListCellRenderer extends JLabel
 implements ListCellRenderer
{
  protected static Border m_noFocusBorder;
  protected FontMetrics m_fm = null;
  protected Insets m_insets = new Insets(0, 0, 0, 0);

  protected int m_defaultTab = 50;
  protected int[] m_tabs = null;

  public TabListCellRenderer() {
    super();
    m_noFocusBorder = new EmptyBorder(1, 1, 1, 1);
    setOpaque(true);
```

```
    setBorder(m_noFocusBorder);
  }

  public Component getListCellRendererComponent(JList list,
   Object value, int index, boolean isSelected, boolean cellHasFocus)
  {
    setText(value.toString());

    setBackground(isSelected ? list.getSelectionBackground() :
      list.getBackground());
    setForeground(isSelected ? list.getSelectionForeground() :
      list.getForeground());

    setFont(list.getFont());
    setBorder((cellHasFocus) ? UIManager.getBorder(
      "List.focusCellHighlightBorder") : m_noFocusBorder);

    return this;
  }

  public void setDefaultTab(int defaultTab) {
    m_defaultTab = defaultTab;
  }

  public int getDefaultTab() { return m_defaultTab; }

  public void setTabs(int[] tabs) { m_tabs = tabs; }

  public int[] getTabs() { return m_tabs; }

  public int getTab(int index) {
    if (m_tabs == null)
      return m_defaultTab*index;
    int len = m_tabs.length;
    if (index>=0 && index<len)
      return m_tabs[index];
    return m_tabs[len-1] + m_defaultTab*(index-len+1);
  }


  public void paint(Graphics g) {
    m_fm = g.getFontMetrics();

    g.setColor(getBackground());
    g.fillRect(0, 0, getWidth(), getHeight());
    getBorder().paintBorder(this, g, 0, 0, getWidth(), getHeight());

    g.setColor(getForeground());
    g.setFont(getFont());
    m_insets = getInsets();
    int x = m_insets.left;
    int y = m_insets.top + m_fm.getAscent();

    StringTokenizer    st = new StringTokenizer(getText(), "\t");
    while (st.hasMoreTokens()) {
      String sNext = st.nextToken();
      g.drawString(sNext, x, y);
      x += m_fm.StringWidth(sNext);
      if (!st.hasMoreTokens())
        break;
      int index = 0;
      while (x >= getTab(index))
```

```
        index++;
      x = getTab(index);
    }
  }
}
```

Understanding the Code

Class StatesList

Minor changes have been made to this class (compared to `StatesList` from the previous section). We create an instance of our custom `TabListCellRenderer`, pass it an array of positions and set it as the renderer for our `JList` component.

Class TabListCellRenderer

Class `TabListCellRenderer` extends `JLabel` and implements the `ListCellRenderer` interface to be used as our custom renderer.

Class variable:

`Border m_noFocusBorder`: border to be used when a list item has no focus.

Instance variables:

`FontMetrics m_fm`: used in calculating text positioning when drawing.

`Insets m_insets`: insets of the cell being rendered.

`int m_defaultTab`: default tab size.

`int[] m_tabs`: an array of positions based on tab index in a `String` being rendered.

The constructor creates assigns text, sets its opaque property to `true` (to render the component's area with the specified background), and sets the border to `m_noFocusBorder`.

The `getListCellRendererComponent()` method is required when implementing `ListCellRenderer`, and is called each time a cell is about to be rendered. It takes five parameters:

`JList list`: reference to the list instance.

`Object value`: data object to be painted by the renderer.

`int index`: index of the item in the list.

`boolean isSelected`: `true` if the cell is currently selected.

`boolean cellHasFocus`: `true` if the cell currently has the focus.

Our implementation of this method assigns new text, sets the background and foreground (depending on whether or not the cell is selected), sets the font to that taken from the parent list component, and sets the border according to whether or not the cell has input focus.

Four additional methods provide set/get support for the `m_defaultTab` and `m_tabs` variables, and do not require detailed explanation beyond the code listing. Now let's take a close look at the `getTab()` method which calculates and returns the position for a given tab index. If no tab array, `m_tabs`, is set, this method returns the `m_defaultTab` distance (defaults to 50) multiplied by the given tab index. If the `m_tabs` array is not `null` and the tab index is less than it's length, the proper value from that array is returned. Otherwise, if the tab index is greater than the array's length, we have no choice but to use the default tab size again.

Since the `JLabel` component does not render tab characters properly, we do not benefit a lot from its

inheritance and implement the `paint()` method to draw tabbed `Strings` ourselves.

Note: Because this is a very simple component that we do not plan to enhance with custom UI functionality, overriding `paint()` is acceptable.

First, our `paint()` method requests a reference to the `FontMetrics` instance for the given `Graphics`. Then we fill the component's rectangle with the background color (which is set in the `getListCellRendererComponent()` method depending on whether or not the cell is selected, see above), and paint the component's border.

Note: Alternatively we could use the `drawTabbedText()` method from the `javax.swing.text.Utilities` class to draw tabbed text. However, this requires us to implement the `TabExpander` interface. In our case it's easier to draw text directly without using that utility. As an interesting exercise you can modify the code from this example to use `drawTabbedText()` method. We will discuss working with tabs more in chapter 19.

In the next step we prepare to draw the tabbed `String`. We set the foreground color, font, and determine the initial `x` and `y` positions for drawing the text, taking into account the component's insets.

Reminder: To draw text in Java you need to use a baseline y-coordinate. This is why the `getAscent()` value is added to the y position. The `getAscent()` method returns the distance from the font's baseline to the top of most alphanumeric characters. See chapter 2 for more information on drawing text and Java 2 `FontMetrics` caveats.

We then use a `StringTokenizer` to parse the `String` and extract the portions separated by tabs. Each portion is drawn with the `drawString()` method, and the x-coordinate is adjusted to the length of the text. We cycle through this process, positioning each portion of text by calling the `getTab()` method, until no more tabs are found.

Running the Code

Figure 10.2 shows `StatesList` displaying an array of tab-separated `Strings`. Note that the tab symbols are not drawn directly, but form consistently aligned columns inside the list.

UI Guideline: Improved Balance

With the tab character now being displayed correctly, the list box now has much better balance. The available area for Capital City is still very large and as designer you may wish to consider reduing this, thus reducing the excessive white space to the right hand side. Such a decision would normally be made after the List Box is seen in situation and necessary alignment and overall panel balance is taken into consideration.

10.4 Processing keyboard input and searching

In this section we will continue to enhance our `JList` states example by adding the ability to select an element whose text starts with a character corresponding to a key press. We will also show how to extend this functionality to search for an element whose text starts with a sequence of typed key characters.

To do this, we must use a `KeyListener` to listen for keyboard input, and accumulate this input in a `String`. Each time a key is pressed, the listener must search through the list and select the first element whose text matches the `String` that we have accumulated. If the time interval between two key presses exceeds a certain pre-defined value, the accumulated `String` must be cleared before appending a new character to avoid

overflow.

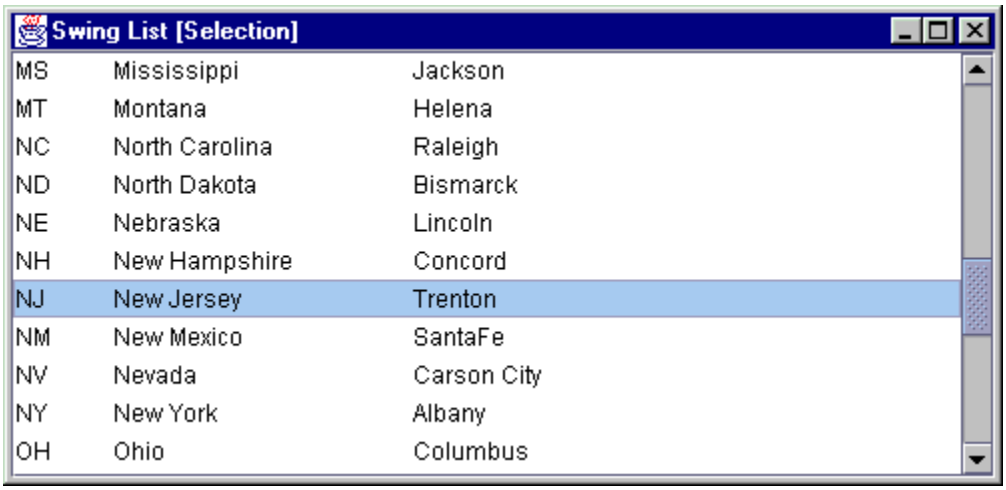


Figure 10.3 JList allowing accumulated keyboard input to search for a matching item.

<<file figure10-3.gif>>

The Code: StatesList.java

see \Chapter10\3

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class StatesList extends JFrame
{
  protected JList m_statesList;

  public StatesList() {
    // Unchanged code from section 10.3

    m_statesList = new JList(states);
    TabListCellRenderer renderer = new TabListCellRenderer();
    renderer.setTabs(new int[] {50, 200, 300});
    m_statesList.setCellRenderer(renderer);
    m_statesList.addKeyListener(new ListSearcher(m_statesList));

    // Unchanged code from section 10.3
  }
}

class ListSearcher extends KeyAdapter
{
  protected JList m_list;
  protected ListModel m_model;
  protected String m_key = "";
  protected long m_time = 0;

  public static int CHAR_DELTA = 1000;

  public ListSearcher(JList list) {
    m_list = list;
```

```
    m_model = m_list.getModel();
  }

  public void keyTyped(KeyEvent e) {
    char ch = e.getKeyChar();
    if (!Character.isLetterOrDigit(ch))
      return;

    if (m_time+CHAR_DELTA < System.currentTimeMillis())
      m_key = "";
    m_time = System.currentTimeMillis();

    m_key += Character.toLowerCase(ch);
    for (int k=0; k<m_model.getSize(); k++) {
      String str = ((String)m_model.getElementAt(k)).toLowerCase();
      if (str.startsWith(m_key)){
        m_list.setSelectedIndex(k);
        m_list.ensureIndexIsVisible(k);
        break;
      }
    }
  }
}
```

Understanding the Code

Class StatesList

An instance of `ListSearcher` is added to the `m_statesList` component as a `KeyListener`. This is the only difference made to this class with respect to the previous example.

Class ListSearcher

Class `ListSearcher` extends the `KeyAdapter` class and defines on class variable:

`int CHAR_DELTA`: static variable to hold the maximum time interval in ms between two subsequent key presses before clearing the search key character `String`.

Instance variables:

`JList m_list`: list component to search and change selection based on keyboard input.

`ListModel m_model`: list model of `m_list`.

`String m_key`: key character `String` used to search for a match.

`long m_time`: time in ms of the last key press.

The `ListSearcher` constructor simply takes a reference to the parent `JList` component and stores it in instance variable `m_list`, and its model in `m_model`.

The `keyTyped()` method is called each time a new character is typed (i.e. a key is pressed and released). Our implementation first obtains a typed character and returns if that character is not letter or digit. `keyTyped()` then checks the time interval between now and the time when the previous key type event occurred. If this interval exceeds `CHAR_DELTA`, the `m_key String` is cleared. Finally, this method walks through the list and performs a case-insensitive comparison of the list `Strings` and searching `String (m_key)`. If an element's text starts with `m_key`, this element is selected and it is forced to appear within our current `JList` view using the `ensureIndexIsVisible()` method.

Running the Code

Try out the search functionality. Figure 10.3 shows our list's selection after pressing "n" immediately followed by "j". As expected, New Jersey is selected.

UIGuideline: Extending Usability and List Size

This technique of allowing accumulated keyboard input to sift and select a List item, improves usability by making the task of search and locating an item in the list easier. This extends the number of items you can put in a list and still have a usable design. A technique like this can easily improve the usefulness of the list up to several thousand entries.

This is another good example of improved usability when the developer takes extra time to provide additional code to make the User's task easier.

10.5 List of check boxes

Lists can certainly be used for more than just Strings. We can easily imagine a list of Swing components. A list of check boxes is actually common in software packages when prompting for selection of optional constituents during installation. In Swing such a list can be constructed by implementing a custom renderer that uses the JCheckBox component. The catch is that mouse and keyboard events must be handled manually to check/uncheck these boxes.

The following example shows how to create a list of check boxes representing imaginary optional program constituents. Associated with each component is an instance of our custom InstallData class with the following fields:

Field	Type	Description
m_name	String	Component's name
m_size	int	Component's size in KB
m_selected	boolean	true if component is selected

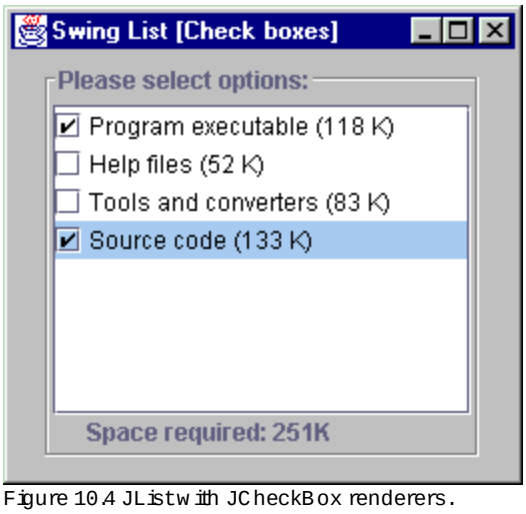


Figure 10.4 JList with JCheckBox renderers.
<<file figure10-4.gif>>

The Code: CheckBoxList.java
see \Chapter10\4

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

```
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class CheckBoxList extends JFrame
{
  protected JList  m_list;
  protected JLabel m_total;

  public CheckBoxList() {
    super("Swing List [Check boxes]");
    setSize(280, 250);
    getContentPane().setLayout(new FlowLayout());

    InstallData[] options = {
      new InstallData("Program executable", 118),
      new InstallData("Help files", 52),
      new InstallData("Tools and converters", 83),
      new InstallData("Source code", 133)
    };

    m_list = new JList(options);
    CheckListCellRenderer renderer = new CheckListCellRenderer();
    m_list.setCellRenderer(renderer);
    m_list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    CheckListener lst = new CheckListener(this);
    m_list.addMouseListener(lst);
    m_list.addKeyListener(lst);

    JScrollPane ps = new JScrollPane();
    ps.getViewport().add(m_list);

    m_total = new JLabel("Space required: 0K");

    JPanel p = new JPanel();
    p.setLayout(new BorderLayout());
    p.add(ps, BorderLayout.CENTER);
    p.add(m_total, BorderLayout.SOUTH);
    p.setBorder(new TitledBorder(new EtchedBorder(),
      "Please select options:") );
    getContentPane().add(p);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);

    recalcTotal();
  }

  public void recalcTotal() {
    ListModel model = m_list.getModel();
    int total = 0;
    for (int k=0; k<model.getSize(); k++) {
      InstallData data = (InstallData)model.getElementAt(k);
      if (data.isSelected())
        total += data.getSize();
```

```
    }
    m_total.setText("Space required: "+total+"K");
  }

  public static void main(String argv[]) {
    new CheckBoxList();
  }
}

class CheckListCellRenderer extends JCheckBox
 implements ListCellRenderer
{
  protected static Border m_noFocusBorder =
    new EmptyBorder(1, 1, 1, 1);

  public CheckListCellRenderer() {
    super();
    setOpaque(true);
    setBorder(m_noFocusBorder);
  }

  public Component getListCellRendererComponent(JList list,
   Object value, int index, boolean isSelected, boolean cellHasFocus)
  {
    setText(value.toString());

    setBackground(isSelected ? list.getSelectionBackground() :
      list.getBackground());
    setForeground(isSelected ? list.getSelectionForeground() :
      list.getForeground());

    InstallData data = (InstallData)value;
      setSelected(data.isSelected());

    setFont(list.getFont());
    setBorder((cellHasFocus) ?
    UIManager.getBorder("List.focusCellHighlightBorder")
      : m_noFocusBorder);

    return this;
  }
}

class CheckListener implements MouseListener, KeyListener
{
  protected CheckBoxList m_parent;
  protected JList m_list;

  public CheckListener(CheckBoxList parent) {
    m_parent = parent;
    m_list = parent.m_list;
  }

  public void mouseClicked(MouseEvent e) {
    if (e.getX() < 20)
    doCheck();
  }

  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
```

```
  public void keyPressed(KeyEvent e) {
    if (e.getKeyChar() == ' ')
      doCheck();
  }

  public void keyTyped(KeyEvent e) {}
  public void keyReleased(KeyEvent e) {}

  protected void doCheck() {
    int index = m_list.getSelectedIndex();
    if (index < 0)
      return;
    InstallData data = (InstallData)m_list.getModel().
      getElementAt(index);
    data.invertSelected();
    m_list.repaint();
    m_parent.recalcTotal();
  }
}

class InstallData
{
  protected String m_name;
  protected int m_size;
  protected boolean m_selected;

  public InstallData(String name, int size) {
    m_name = name;
    m_size = size;
    m_selected = false;
  }

  public String getName() { return m_name; }

  public int getSize() { return m_size; }

  public void setSelected(boolean selected) {
    m_selected = selected;
  }

  public void invertSelected() { m_selected = !m_selected; }

  public boolean isSelected() { return m_selected; }

  public String toString() { return m_name+" ("+m_size+" K)"; }
}
```

Understanding the Code

Class CheckBoxList

`CheckBoxList` extends `JFrame` to provide the basic frame for this example. Instance variables:

- `JList m_list`: list to display program constituents.
- `JLabel m_total`: label to display total space required for installation based on selected constituents.

An array of four `InstallData` objects is passed to the constructor of our `JList` component (note that we use the `DefaultListModel`, which is sufficient for our purposes here). `SINGLE_SELECTION` is used as our list's selection mode. An instance of our custom `CheckListCellRenderer` is created and set as the cell

renderer for our list. An instance of our custom `CheckListener` is then registered as both a mouse and key listener to handle item checking/unchecking for each check box (see below).

The list component is added to a `JScrollPane` to provide scrolling capabilities. Then `JLabel m_total` is created to display the total amount of space required for installation based on the currently selected check boxes.

In previous examples the `JList` component occupied all of our frame's available space. In this example, however, we are required to consider a different layout. `JPanel p` is now used to hold both the list and label (`m_total`). To ensure that the label will always be placed below the list we use a `BorderLayout`. We also use a `TitledBorder` for this panel's border to provide visual grouping.

Method `recalcTotal()` steps through the sequence of `InstallData` instances contained in the list, and calculates the sum of sizes of the selected items. The result is then displayed in the `m_total` label.

Class CheckListCellRenderer

This class implements the `ListCellRenderer` interface and is similar to our `TabListCellRenderer` class from section 10.3. An important difference is that `CheckListCellRenderer` extends `JCheckBox` (not `JLabel`) and uses that component to render each item in our list. Method `getListCellRendererComponent()` sets the check box text, determines whether or not the current list item is selected, and sets the check box's selection state accordingly (using its inherited `JCheckBox.setSelected()` method).

Note: Alternatively we could use `JLabels` with custom icons to imitate checked and unchecked boxes. However, the use of `JCheckBox` is preferred for graphical consistency with other parts of a GUI.

Class CheckListener

This class implements both `MouseListener` and `KeyListener` to process all user input which can change the state of check boxes in the list. Its constructor takes a `CheckBoxList` instance as parameter in order to gain access to the `CheckBoxList.recalcTotal()` method.

We've assumed in this example, that an item's checked state should be changed if:

1. The user clicks the mouse close enough to the item's check box (say, up to 20 pixels from the left edge).
2. The user transfers focus to the item (with the mouse or keyboard) and then presses the space bar.

Bearing this in mind, two methods need to be implemented: `mouseClicked()` and `keyPressed()`. They both call `protected` method `doCheck()` if either of the conditions described above are satisfied. All other methods from the `MouseListener` and `KeyListener` interfaces have empty implementations.

Method `doCheck()` determines the first selected index (the only selected index--recall that our list uses single selection mode) in the list component and retrieves the corresponding `InstallData` object. This method then calls `invertSelected()` to change the checked state of that object. It then repaints the list component, and displays the new total by calling the `recalcTotal()` method.

Class InstallData

Class `InstallData` handles a data unit for this example (it functions as a custom model). `InstallData` encapsulates three variables described at the beginning of this section: `m_name`, `m_size`, and `m_selected`. Its only constructor takes three parameters to fill these variables. Besides the obvious set/get methods, the `invertSelected()` method is defined to negate the value of `m_selected`. Method `toString()` determines the `String` representation of this object to be used by the list renderer.

Running the Code

Figure 10.4 shows our list composed of check boxes in action. Select any item and click over the check box, or press space bar to change its checked state. Note that the total kilobytes required for these imaginary implementations is dynamically displayed in the label at the bottom.

UI Guideline: When to use Check Boxes in a List Check boxes tend to be used inside bordered panes to show groupings of mutually related binary attributes. Such a technique is good for a fixed number of attributes, however, it becomes problematic when the number of items can vary.

The technique shown here is a good way to solve the problem when the collection of attributes or data is of an undetermined size. Use a CheckBox list for binary (True/False) selection of items from a collection of a size which cannot be determined at design time.

For example, imagine the team selection for a football team. The coach has a pool of players and needs to indicate who has been picked for the Saturday game. With such a problem, you could show the whole pool of players (sorted alphabetically or by number) in the list and allow the coach to check off each selected player.

Chapter 11. Text Components and Undo

In this chapter:

- TextComponents overview
- Undo/Redo

11.1 TextComponents overview

This chapter summarizes the most basic and commonly used text component features, and introduces the `undo` package. In the next chapter we develop a basic `JTextArea` application in the process of demonstrating the use of menus and toolbars. In chapter 19 we discuss the inner workings of text components in much more detail. In chapter 20 we develop an extensive `JTextPane` word processor application with powerful font, style, paragraph, find & replace, and spell-checking dialogs.

11.1.1 JTextComponent

abstract class javax.swing.text.JTextComponent

The `JTextComponent` class serves as the super-class of each Swing text component. All text component functionality is defined by this class, along with the plethora of supporting classes and interfaces provided in the `text` package. The text components themselves are members of the `javax.swing` package: `JTextField`, `JPasswordField`, `JTextArea`, `JEditorPane`, and `JTextPane`.

Note: We have purposely left out most of the details behind text components in this chapter to provide only the information that you will most likely need on a regular basis. If, after reading this chapter, you would like a more thorough understanding of how text components work, and how to customize them or take advantage of some of the more advanced features, see chapters 19 and 20.

JTextComponent is an abstract sub-class of JComponent, and implements the Scrollable interface (see chapter 7). Each multi-line text component is designed to be placed in a JScrollPane.

Textual content is maintained in instances of the javax.swing.text.Document interface, which acts as the text component model. The text package includes two concrete Document implementations: PlainDocument and StyledDocument. PlainDocument allows one font, one color, and is limited to character content. StyledDocument is much more complex, allowing multiple fonts, colors, embedded images and components, and various sets of hierarchically resolving textual attributes. JTextField, JPasswordField, and JTextArea each use a PlainDocument model. JEditorPane and JTextPane use a StyledDocument model. We can retrieve a text component's Document with getDocument(), and assign one with setDocument(). We can also attach DocumentListeners to a document to listen for changes in that document's textual content (this is much different than a key listener because all document events are dispatched after a change has been made).

We can assign and retrieve the color of a text component's Caret with setCaretColor() and getCaretColor(). We can also assign and retrieve the current Caret position in a text component with setCaretPosition() and getCaretPosition().

The disabledColor property allows assignment of a font color to use when in the disabled state. The foreground and background properties inherited from JComponent also apply, where the foreground color is used as the font color when a text component is enabled, and the background color is used as the background for the whole text component. The font property specifies the font to render the text in. Note that the font property, and the foreground and background color properties, do not overpower any attributes assigned to styled text components such as JEditorPane and JTextPane.

All text components maintain information about their current selection. We can retrieve the currently selected text as a String with getSelectedText(), and we can assign and retrieve specific background and foreground colors to use for selected text with setSelectionBackground()/getSelectionBackground() and setSelectionForeground()/getSelectionForeground() respectively.

JTextComponent also maintains a bound focusAccelerator property, which is a char that is used to transfer focus to a text component when the corresponding key is pressed simultaneously with the ALT key. This works internally by calling requestFocus() on the text component, and will occur as long as the top-level window containing the given text component is currently active. We can assign/retrieve this character with setFocusAccelerator()/getFocusAccelerator(), and we can turn this functionality off by assigning '\0'.

The read() and write() methods provide convenient ways to read and write text documents respectively. The read() method takes a java.io.Reader and an Object describing the Reader stream, and creates a new document model appropriate to the given text component containing the obtained character data. The write() method stores the content of the document model into a given java.io.Writer stream.

Warning: We can customize any text component's document model. However, it is important to realize that whenever the read() method is invoked a new document will be created. Unless this method is overriden, a custom document that had been assigned with setDocument() will be lost whenever read() is invoked, because the current document will be replaced by a default instance.

11.1.2 JTextField

class javax.swing.JTextField

`JTextField` is a single-line text component using a `PlainDocument` model. The `horizonatlAlignment` property specifies text justification within the textfield. We can assign/retrive this property with `setHorizontalAlignment()`/`getHorizontalAlignment`, and acceptable values are `JTextField.LEFT`, `JTextField.CENTER`, and `JTextField.RIGHT`.

There are several `JTextField` constructors, two of which allow us to specify a number of columns. We can also assign/retrieve this number, the `columns` property, with `setColumns()`/`getColumns()`. Specifying a specific number of columns rarely corresponds to the actual number of characters that will fit in that textfield. Firstly because a textfield might not receive its preferred size due to the current layout manager. Second, the width of a column is the width of the character 'm' in the current font. Unless a monospaced font is being used, this width will be greater than most other characters.

The following example creates 14 `JTextFields` with a varying number of columns. Each field contains a number of 'm's equal to its number of columns.

The Code: JTextFieldTest.java
see \Chapter11\1

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

public class JTextFieldTest extends JFrame
{
  public JTextFieldTest() {
    super("JTextField Test");

    getContentPane().setLayout(new FlowLayout());

    JTextField textField1 = new JTextField("m",1);
    JTextField textField2 = new JTextField("mm",2);
    JTextField textField3 = new JTextField("mmm",3);
    JTextField textField4 = new JTextField("mmmm",4);
    JTextField textField5 = new JTextField("mmmmm",5);
    JTextField textField6 = new JTextField("mmmmmm",6);
    JTextField textField7 = new JTextField("mmmmmmm",7);
    JTextField textField8 = new JTextField("mmmmmmmm",8);
    JTextField textField9 = new JTextField("mmmmmmmmm",9);
    JTextField textField10 = new JTextField("mmmmmmmmmm",10);
    JTextField textField11 = new JTextField("mmmmmmmmmmm",11);
    JTextField textField12 = new JTextField("mmmmmmmmmmmm",12);
    JTextField textField13 = new JTextField("mmmmmmmmmmmmm",13);
    JTextField textField14 = new JTextField("mmmmmmmmmmmmmm",14);

    getContentPane().add(textField1);
    getContentPane().add(textField2);
    getContentPane().add(textField3);
    getContentPane().add(textField4);
    getContentPane().add(textField5);
    getContentPane().add(textField6);
    getContentPane().add(textField7);
    getContentPane().add(textField8);
    getContentPane().add(textField9);
    getContentPane().add(textField10);
    getContentPane().add(textField11);
```

```
    getContentPane().add(textField12);
    getContentPane().add(textField13);
    getContentPane().add(textField14);

    setSize(300,170);
    setVisible(true);
  }

  public static void main(String argv[]) {
    new JTextFieldTest();
  }
}
```

Figure 11.1 illustrates. Note that none of the text completely fits in its field. This is because `JTextField` does not factor in the size of its border when calculating its preferred size, as we might expect. To work around this problem, although not an ideal solution, we can add one more column to each text field. The result is shown in figure 11.2. This solution is more appropriate when a fixed width font (monospaced) is being used. Figure 11.3 illustrates.

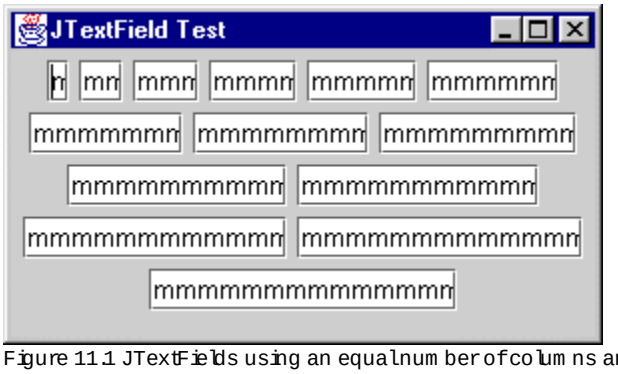


Figure 11.1 JTextFields using an equal number of columns and 'm' characters.

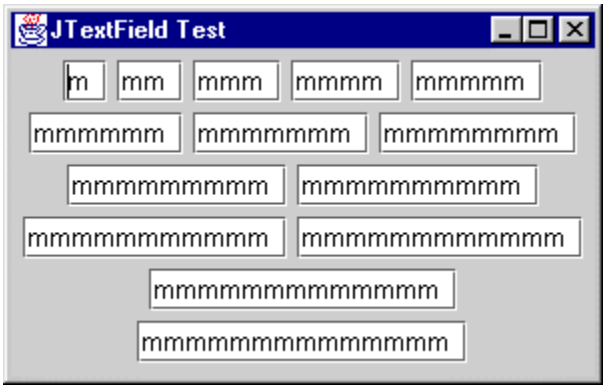


Figure 11.2 JTextFields using one more column than number of 'm' characters.

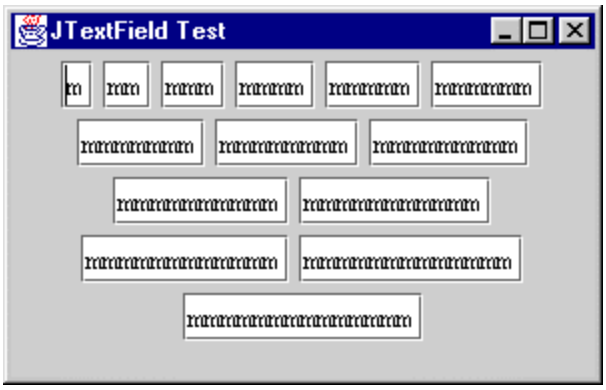


Figure 11.3 JTextFields using a monospaced font, and one more column than number of 'm characters.

Note: Using a monospaced font is always more appropriate when a fixed character limit is desired.

`JTextField` also maintains a `BoundedRangeModel` (see chapter 13) as its `horizontalVisibility` property. This model is used to keep track of the amount of currently visible text. The `minimum` is zero (the beginning of the document), and the `maximum` is equal to the width of the text field or the total length of the text in pixels (whichever is greater). The `value` is the current offset of the text displayed at the left edge of the field, and the `extent` is the width of the text field in pixels.

By default a `KeyStroke` (see 2.13.2) is established with `ENTER` that causes an `ActionEvent` to be fired. By simply adding an `ActionListener` to a `JTextField` we will receive events whenever `ENTER` is pressed while that field has the current focus. This is very convenient functionality, but it may also get in the way of things. To remove this registered keystroke we can do the following:

```
KeyStroke enter = KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0);
Keymap map = myJTextField.getKeymap();
map.removeKeyStrokeBinding(enter);
```

`JTextField`'s document model can be customized to allow only certain forms of input by extending `PlainDocument` and overriding the `insertString()` method. The following code shows a class that will only allow six or less digits to be entered. We can assign this document to a `JTextField` with the `setDocument()` method (see chapter 19 for more about working with `Documents`).

```
class SixDigitDocument extends PlainDocument
{
  public void insertString(int offset,
   String str, AttributeSet a)
    throws BadLocationException {
      char[] insertChars = str.toCharArray();

      boolean valid = true;
      boolean fit = true;
      if (insertChars.length + getLength() <= 6) {
        for (int i = 0; i < insertChars.length; i++) {
          if (!Character.isDigit(insertChars[i])) {
            valid = false;
            break;
          }
        }
      }
      else
        fit = false;
```

```
        if (fit && valid)
          super.insertString(offset, str, a);
        else if (!fit)
          getToolkit().beep();
      }
    }
```

UIGuideline: Don't overly restrict input

Filtering text fields during data entry is a powerful aid to usability. It helps prevent the user from making a mistake and can speed operation by removing the need for validation and correction procedures. However, it is important not to overly restrict the allowable input. Ensure that all reasonable input is expected and accepted.

For example, with a phone number, allow "00 1 44 654 7777" and allow "00+1 44 654 7777" and allow "00-1-1-654-7777" as well as "00144654777". Phone numbers can contain more than just numbers! Another example might be dates. You should allow "04-06-99", "04/06/99" and "04:06:99" as well as "040699".

11.1.3 JPasswordField

class javax.swing.JPasswordField

`JPasswordField` is a fairly simple extension of `JTextField` that displays an echo character instead of the actual content that is placed in its model. This echo character defaults to '*', and we can assign a different one with `setEchoChar()`.

Unlike other text components, we cannot retrieve the actual content of a `JPasswordField` with `getText()` (this method, along with `setText()` have been deprecated in `JPasswordField`). Instead we must use the `getPassword()` method, which returns an array of `chars`. Also note that `JPasswordField` overrides the `JTextComponent` `copy()` and `cut()` methods to do nothing but emit a beep (for security reasons).

Figure 11.4 shows the `JTextFieldDemo` example of section 11.1.2, using `JPasswordFields` instead, each using a monospaced font.

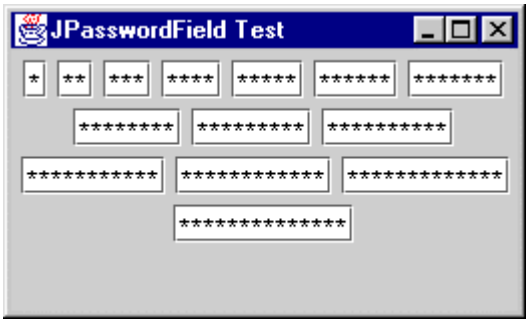


Figure 11.4 JPasswordFields using a monospaced font, and one more column than number of characters.

11.1.4 JTextArea

class javax.swing.JTextArea

`JTextArea` allows multiple lines of text and, like `JTextField`, it uses a `PlainDocument` model. As we discussed above, `JTextArea` cannot display multiple fonts or font colors. `JTextArea` can perform line wrapping, and when line wrapping is enabled, we can specify whether lines break on word boundaries or not. To enable/disable line wrapping we set the `lineWrap` property with `setLineWrap()`. To enable/disable

wrapping on boundaries (which will only have an effect when `lineWrap` is set to `true`) we set the `wrapStyleWord` property using `setWrapStyleWord()`. Note that both `lineWrap` and `wrapStyleWord` are bound properties.

`JTextArea` overrides `isManagingFocus()` (see 2.12) to return `true`, indicating that the `FocusManager` will not transfer focus out of a `JTextArea` when `TAB` is pressed. Instead a `TAB` is inserted into the document, which is a number of spaces equal to `tabSize`. We can assign/retrieve the tab size with `setTabSize()`/`getTabSize()` respectively. `tabSize` is also a bound property.

There are several ways to add text to a `JTextArea`'s document. We can pass this text in to one of the constructors, append it to the end of the document using the `append()` method, insert a string at a given character offset using the `insert()` method, or replace a given range of text with the `replaceRange()` method. As with any text component, we can also set the text with the `JTextComponent` `setText()` method, and we can add and remove text directly from its `Document` (see chapter 19 for more details about the `Document` interface).

`JTextArea` maintains `lineCount` and `rows` properties which can easily be confused. The `rows` property specifies how many rows of text `JTextArea` is actually displaying. This may change whenever a text area is resized. The `lineCount` property specifies how many lines of text the document contains. This usually means a set of characters ending with a line break ('\n'). We can retrieve the character offset of the end of a given line with `getLineEndOffset()`, the character offset of the beginning of a given line with `getLineStartOffset()`, and the line number that contains a given offset with `getLineOfOffset()`.

The `rowHeight` and `columnWidth` properties are determined by the height and with of the current font respectively. The width of one column is equal to the width of the 'm' character in the current font. We cannot assign new values to the properties, but we can override the `getColumnWidth()` and `getRowHeight()` methods in a sub-class to return any value we like. We can explicitly set the number of rows and columns a text area contains with `setRows()` and `setColumns()`, and the `getRows()` and `getColumns()` methods will only return these explicitly assigned values (not the current row and column count as we might assume at first glance).

Unless placed in a `JScrollPane` or a container using a layout manager which enforces a certain size (or a layout manager using the `preferredSize` of its children), `JTextArea` will resize itself dynamically depending on the amount of text entered. This behavior is rarely desired.

11.1.5 JEditorPane

class javax.swing.JEditorPane

`JEditorPane` is a multi-line text component capable of displaying and editing various different types of content. Swing provides support for HTML and RTF, but there is nothing stopping us from defining our own, or implementing support for an alternate format.

Note: Swing's support for HTML and RTF is located in the `javax.swing.text.html` and `javax.swing.text.rtf` packages respectively. At the time of this writing the `html` package was still going through major changes. The `rtf` package was also still under development, but was closer to a finalized state than HTML support. For this reason we devoted chapter 20 to the step-wise construction of an RTF word processor application. We expect to see stronger support for displaying and editing HTML in a future Java 2 release.

Support for different content is accomplished in part through the use of custom `EditorKit` objects. `JEditorPane`'s `contentType` property is a `String` representing the type of document the editor pane is currently set up to display. The `EditorKit` maintains this value which, for `DefaultEditorKit`, defaults

to "text/plain." HTMLEditorKit and RTFEditorKit have contentType values of "text/html" and "text/rtf" respectively (see chapter 19 for more about EditorKits).

In chapter 9 we built a simple web browser using a non-editable JEditorPane by passing a URL to its constructor. When in non-editable mode JEditorPane displays HTML pretty much as we might expect (although it has a long way to go to match Netscape). By allowing editing, JEditorPane will display an HTML document with many of its tags specially rendered as shown in figure 11.1 (compare this to figure 9.4).

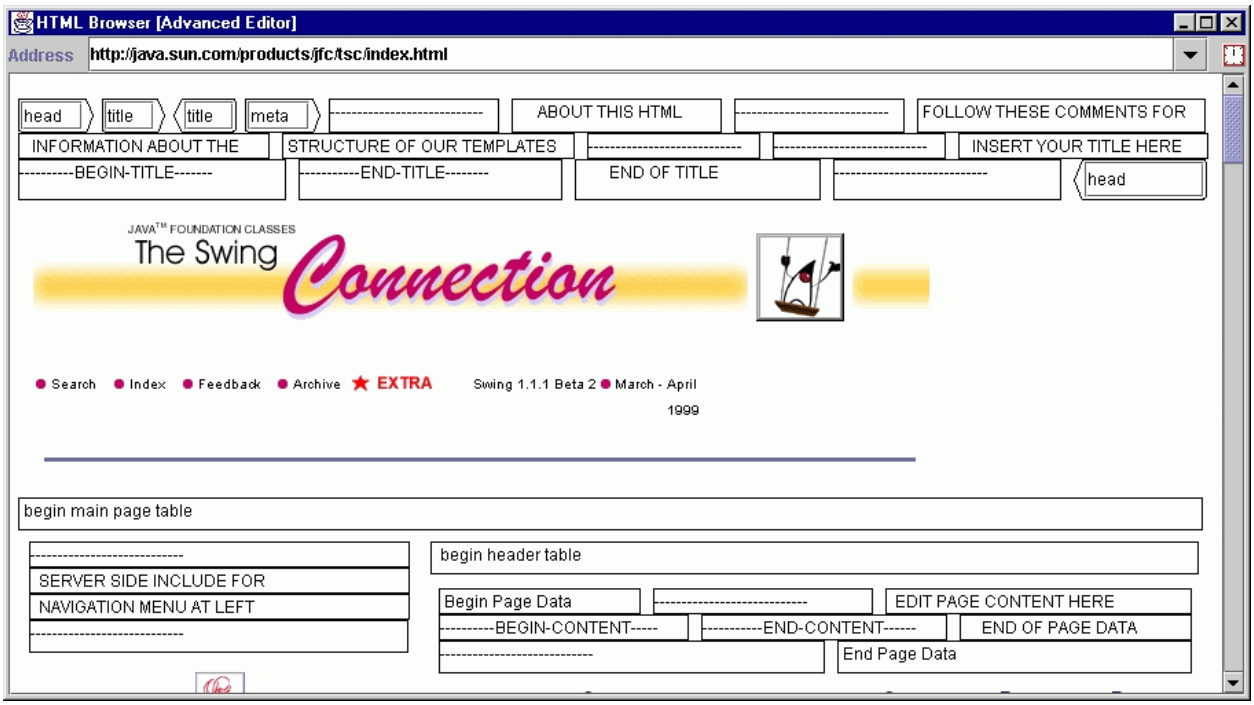


Figure 11.5 JEditorPane displaying HTML in editable mode.

JEditorPane is smart enough to use an appropriate EditorKit, if one is available, to display a document passed to it. When displaying an HTML document, JEditorPane fires HyperlinkEvents (defined in the javax.swing.event package). We can attach HyperlinkListeners to JEditorPane to listen for hyperlink invocations, as demonstrated by the examples at the end of chapter 9. The following code shows how simple it is to construct an HTML browser using an active HyperlinkListener.

```
m_browser = new JEditorPane(
  new URL("http://java.sun.com/products/jfc/tsc/index.html"));
m_browser.setEditable(false);
m_browser.addHyperlinkListener( new HyperlinkListener() {
  public void hyperlinkUpdate(HyperlinkEvent e) {
    if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
      URL url = e.getURL();
      if (url == null)
        return;
      try { m_browser.setPage(e.getURL); }
      catch (IOException e) { e.printStackTrace(); }
    }
  }
}
```

JEditorPane uses a Hashtable to store its editor kit/content type pairs. We can query this table and retrieve the editor kit associated with a particular content type, if there is one, using the getEditorKitForContentType() method. We can get the current editor kit with getEditorKit(),

and the current content type with `getContentType()`. We can set the current content type with `setContentType()`, and if there is an appropriate editor kit in `JEditorPane`'s hash table the appropriate editor kit will replace the current one. We can also assign an editor kit for a given content type using the `setEditorKitForContentType()` method (we will discuss `EditorKits`, and the ability to construct our own, in chapter 19).

`JEditorPane` uses a `DefaultStyledDocument` as its model. In HTML mode an `HTMLDocument`, which extends `DefaultStyledDocument`, is used. `DefaultStyledDocument` is quite powerful, allowing us to associate attributes with characters, paragraphs, and apply logical styles (see chapter 19).

11.1.6 JTextPane

class javax.swing.JTextPane

`JTextPane` extends `JEditorPane` and thus inherits its abilities to display various types of content. The most significant functionality `JTextPane` offers is the ability to programmatically assign attributes to regions of its content, and embed components and images within its document.

To assign attributes to a region of document content we use an `AttributeSet` implementation. We will describe `AttributeSets` in detail in chapter 19, but it suffices to say here that they contain a group of attributes such as font type, font style, font color, paragraph and character properties, etc. These attributes are assigned through the use of various static methods defined in the `StyleConstants` class, which we will also discuss further in chapter 19.

The following example demonstrates embedded icons, components, and stylized text. Figure 11.6 illustrates.

Figure 11.6 JTextPane with inserted ImageIcons, text with attributes, and an active JButton.

The Code: JTextPaneDemo.java
see \Chapter11\2

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

public class JTextPaneDemo extends JFrame
{
  static SimpleAttributeSet ITALIC_GRAY = new SimpleAttributeSet();
  static SimpleAttributeSet BOLD_BLACK = new SimpleAttributeSet();
  static SimpleAttributeSet BLACK = new SimpleAttributeSet();

  JTextPane m_editor = new JTextPane();

  // Best to reuse attribute sets as much as possible.
  static {
    StyleConstants.setForeground(ITALIC_GRAY, Color.gray);
    StyleConstants.setItalic(ITALIC_GRAY, true);
    StyleConstants.setFontFamily(ITALIC_GRAY, "Helvetica");
    StyleConstants.setFontSize(ITALIC_GRAY, 14);

    StyleConstants.setForeground(BOLD_BLACK, Color.black);
    StyleConstants.setBold(BOLD_BLACK, true);
    StyleConstants.setFontFamily(BOLD_BLACK, "Helvetica");
    StyleConstants.setFontSize(BOLD_BLACK, 14);

    StyleConstants.setForeground(BLACK, Color.black);
    StyleConstants.setFontFamily(BLACK, "Helvetica");
    StyleConstants.setFontSize(BLACK, 14);
  }

  public JTextPaneDemo() {
    super("JTextPane Demo");

    JScrollPane scrollPane = new JScrollPane(m_editor);
    getContentPane().add(scrollPane, BorderLayout.CENTER);

    setEndSelection();
    m_editor.insertIcon(new ImageIcon("manning.gif"));
    insertText("\nHistory: Distant\n\n", BOLD_BLACK);

    setEndSelection();
    m_editor.insertIcon(new ImageIcon("Lee_fade.jpg"));
    insertText("                                        ", BLACK);
    setEndSelection();
    m_editor.insertIcon(new ImageIcon("Bace_fade.jpg"));

    insertText("\n      Lee Fitzpatrick            "
      + "                          "
      + "Marjan Bace\n\n", ITALIC_GRAY);

    insertText("When we started doing business under " +
      "the Manning name, about 10 years ago, we were a very " +
      "different company. What we are now is the end result of " +
      "an evolutionary process in which accidental " +
      "events played as big a role, or bigger, as planning and " +
```

```
      "foresight.\n", BLACK);

    setEndSelection();
    JButton manningButton = new JButton("Visit Manning");
    manningButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_editor.setEditable(false);
        try { m_editor.setPage("http://www.manning.com"); }
        catch (IOException ioe) { ioe.printStackTrace(); }
      }
    });
    m_editor.insertComponent(manningButton);

    setSize(500,450);
    setVisible(true);
  }

  protected void insertText(String text, AttributeSet set) {
    try {
      m_editor.getDocument().insertString(
        m_editor.getDocument().getLength(), text, set);
    }
    catch (BadLocationException e) {
      e.printStackTrace();
    }
  }

  protected void setEndSelection() {
    m_editor.setSelectionStart(m_editor.getDocument().getLength());
    m_editor.setSelectionEnd(m_editor.getDocument().getLength());
  }

  public static void main(String argv[]) {
    new JTextPaneDemo();
  }
}
```

As demonstrated above, we can insert images and components with `JTextPane`'s `insertIcon()` and `insertComponent()` methods. Note that these methods insert the given object by replacing the current selection. If there is no current selection they will be placed at the beginning of the document. This is why we defined the `setEndSelection()` method in our example above to point the selection to the end of the document where we want to do insertions.

When inserting text, we cannot simply append it to the textpane itself. Instead we retrieve its document and call `insertString()`. To give attributes to inserted text we can construct `AttributeSet` implementations, and assign attributes to that set using the `StyleConstants` class. In the example above we do this by constructing three `SimpleAttributeSets` as static instances (so that they may be reused as much as possible).

As an extension of `JEditorPane`, `JTextPane` uses a `DefaultStyledDocument` for its model. Text panes use a special editor kit, `DefaultStyledEditorKit`, to manage their `Actions` and `Views`. `JTextPane` also supports the use of `Styles`, which are named collections of attributes. We will discuss styles as well as many other advanced features of `JTextPane` in chapters 19 and 20.

11.2 Undo/Redo

Undo/redo options are commonplace in applications such as paint programs and word processors, and have been used extensively throughout the writing of this book! It is interesting that this functionality is provided as part of the Swing library, as it is completely Swing independent. In this section we will briefly introduce the

javax.swing.undo constituents, and in the process of doing so, we will present an example showing how undo/redo functionality can be integrated into any app. The text components come with built-in undo/redo functionality, and we will also discuss how to take advantage of this.

11.2.1 The UndoableEdit interface

abstract interface javax.swing.undo.UndoableEdit

This interface acts as a template definition for anything that can be undone/redone. Implementations should normally be very lightweight, as undo/redo operations commonly occur quickly in succession.

UndoableEdits are designed to have three states: undoable, redoable, and dead. When an UndoableEdit is in the undoable state, calling undo() will perform an undo operation. Similarly, when an UndoableEdit is in the redoable state, calling redo() will perform a redo operation. Methods canUndo() and canRedo() provide ways to check whether or not an UndoableEdit is in the undoable or redoable state respectively. We can use the die() method to explicitly send an UndoableEdit to the dead state. When in the dead state, an UndoableEdit cannot be undone or redone, and any attempt to do so will generate an exception.

UndoableEdits maintains three String properties (normally used as menu item text): presentationName, undoPresentationName, and redoPresentationName. Methods addEdit() and replaceEdit() are meant to be used to merge two edits and replace an edit, respectively. UndoableEdit also defines the concept of significant and insignificant edits. An insignificant edit is one that UndoManager (see 11.2.6) ignores when an undo/redo request is made. CompoundEdit (see 11.2.3), however, will pay attention to both significant and insignificant edits. The significant property of an UndoableEdit can be queried with isSignificant().

11.2.2 AbstractUndoableEdit

class javax.swing.undo.AbstractUndoableEdit

AbstractUndoableEdit implements UndoableEdit and defines two boolean properties representing the three UndoableEdit states. The alive property is true when an edit is not dead. The done property is true when an undo can be performed, and false when a redo can be performed.

The default behavior provided by this class is good enough for most sub-classes. All AbstractUndoableEdits are significant, and the undoPresentationName and redoPresentationName properties are formed by simply appending "Undo" and "Redo" to presentationName.

The following example demonstrates a basic square painting program with undo/redo functionality. This app simply draws a square outline wherever a mouse press occurs. A Vector of Points is maintained which represent the upper left-hand corner of each square that is drawn on the canvas. We create an AbstractUndoableEdit sub-class to maintain a reference to a Point, with undo() and redo() methods that remove and add that Point from the Vector, respectively. Figure 11.7 illustrates.

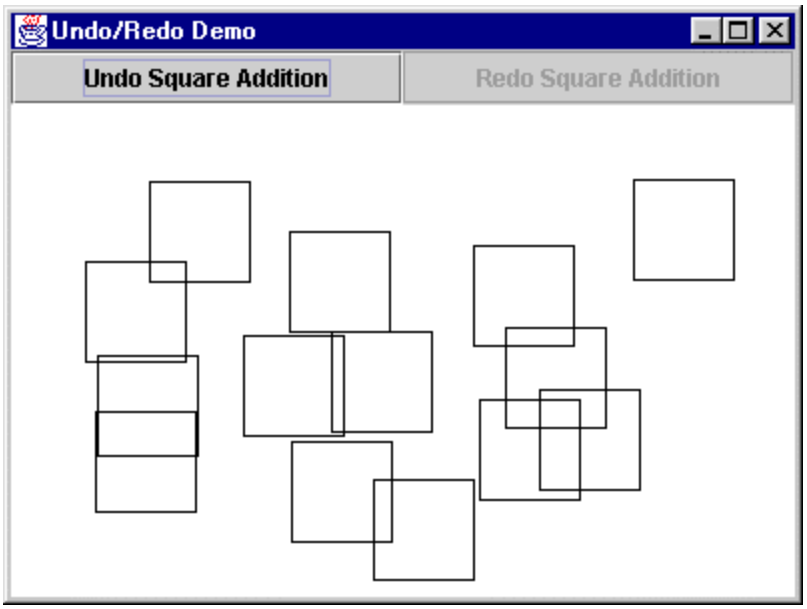


Figure 11.7 A square painting app with one level of undo/redo.

The Code: UndoRedoPaintApp.java
see \Chapter11\3

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

public class UndoRedoPaintApp extends JFrame
{
  protected Vector m_points = new Vector();
  protected PaintCanvas m_canvas = new PaintCanvas(m_points);
  protected UndoablePaintSquare m_edit;
  protected JButton m_undoButton = new JButton("Undo");
  protected JButton m_redoButton = new JButton("Redo");

  public UndoRedoPaintApp() {
    super("Undo/Redo Demo");

    m_undoButton.setEnabled(false);
    m_redoButton.setEnabled(false);

    JPanel buttonPanel = new JPanel(new GridLayout());
    buttonPanel.add(m_undoButton);
    buttonPanel.add(m_redoButton);

    getContentPane().add(buttonPanel, BorderLayout.NORTH);
    getContentPane().add(m_canvas, BorderLayout.CENTER);

    m_canvas.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        Point point = new Point(e.getX(), e.getY());
        m_points.addElement(point);
        m_edit = new UndoablePaintSquare(point, m_points);
        m_undoButton.setText(m_edit.getUndoPresentationName());
        m_redoButton.setText(m_edit.getRedoPresentationName());
        m_undoButton.setEnabled(m_edit.canUndo());
```

```
        m_redoButton.setEnabled(m_edit.canRedo());
        m_canvas.repaint();
      }
    });

    m_undoButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try { m_edit.undo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        m_canvas.repaint();
        m_undoButton.setEnabled(m_edit.canUndo());
        m_redoButton.setEnabled(m_edit.canRedo());
      }
    });

    m_redoButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try { m_edit.redo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        m_canvas.repaint();
        m_undoButton.setEnabled(m_edit.canUndo());
        m_redoButton.setEnabled(m_edit.canRedo());
      }
    });

    setSize(400,300);
    setVisible(true);
  }

  public static void main(String argv[]) {
    new UndoRedoPaintApp();
  }
}

class PaintCanvas extends JPanel
{
  Vector m_points;
  protected int width = 50;
  protected int height = 50;

  public PaintCanvas(Vector vect) {
    super();
    m_points = vect;
    setOpaque(true);
    setBackground(Color.white);
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.black);
    Enumeration enum = m_points.elements();
    while(enum.hasMoreElements()) {
      Point point = (Point) enum.nextElement();
      g.drawRect(point.x, point.y, width, height);
    }
  }
}

class UndoablePaintSquare extends AbstractUndoableEdit
{
  protected Vector m_points;
  protected Point m_point;
```

```
  public UndoablePaintSquare(Point point, Vector vect) {
    m_points = vect;
    m_point = point;
  }

  public String getPresentationName() {
    return "Square Addition";
  }

  public void undo() {
    super.undo();
    m_points.remove(m_point);
  }

  public void redo() {
    super.redo();
    m_points.add(m_point);
  }
}
```

One thing to note about this example is that it is extremely limited. Because we are not maintaining an ordered collection of `UndoableEdits` we can only perform one undo/redo. `CompoundEdit` and `UndoManager` directly address this limitation.

11.2.3 CompoundEdit

class javax.swing.undo.CompoundEdit

This class extends `AbstractUndoableEdit` to support an ordered collection of `UndoableEdits`, maintained as a protected `Vector` called `edits`. `UndoableEdits` can be added to this vector with `addEdit()`, but they cannot so easily be removed (for this, a sub-class would be necessary).

Even though `CompoundEdit` is more powerful than `AbstractUndoableEdit`, it is far from the ideal solution. Edits cannot be undone until all edits have been added. Once all `UndoableEdits` are added we are expected to call `end()`, at which point `CompoundEdit` will no longer accept any additional edits. Once `end()` is called, a call to `undo()` will undo all edits, whether significant or not. A `redo()` will then redo them all, and we can continue to cycle back and forth like this as long as the `CompoundEdit` itself remains alive. For this reason, `CompoundEdit` is useful for a pre-defined or intentionally limited set of states.

`CompoundEdit` introduces an additional state property called `inProgress`, which is `true` if `end()` has not been called. We can retrieve the value of `inProgess` with `isInProgress()`. The `significant` property, inherited from `UndoableEdit`, will be `true` if one or more of the contained `UndoableEdits` is significant, and `false` otherwise.

11.2.4 UndoableEditEvent

class javax.swing.event.UndoableEditEvent

This event encapsulates a source `Object` and an `UndoableEdit`, and is meant to be passed to implementations of the `UndoableEditListener` interface.

11.2.5 The UndoableEditListener interface

class javax.swing.event.UndoableEditListener

This listener is intended for use by any class wishing to listen for operations that can be undone/redone. When such an operation occurs an `UndoableEditEvent` can be sent to an `UndoableEditListener` for processing. `UndoManager` implements this interface so we can simply add it to any class defining undoable/redoable operations. It is important to emphasize that `UndoableEditEvents` are not fired when an undo or redo actually occurs, but when an operation occurs which has an `UndoableEdit` associated with it. This interface declares one method, `undoableEditHappened()`, which accepts an `UndoableEditEvent`. We are generally responsible for passing `UndoableEditEvents` to this method. The example in the next section demonstrates.

11.2.6 UndoManager

class javax.swing.undo.UndoManager

`UndoManager` extends `CompoundEdit` and relieves us of the limitation where undos and redos cannot be performed until `edit()` is called. It also relieves us of the limitation where all edits are undone or redone at once. Another major difference from `CompoundEdit` is that `UndoManager` simply skips over all insignificant edits when `undo()` or `redo()` is called, effectively not paying them any attention. Interestingly, `UndoManager` allows us to add edits while `inProgress` is `true`, but if `end()` is ever called, `UndoManager` immediately starts acting like a `CompoundEdit`.

`UndoManager` introduces a new state called `undoOrRedo` which, when `true`, signifies that calling `undo()` or `redo()` is valid. This property can only be `true` if there is more than one edit stored, and only if there is at least one edit in the undoable state and one in the redoable state. The value of this property can be retrieved with `canUndoOrRedo()`, and the `getUndoOrRedoPresentationName()` method will return an appropriate name for use in a menu item or elsewhere.

We can retrieve the next significant `UndoableEdit` that is scheduled to be undone or redone with `editToBeUndone()` or `editToBeRedone()` respectively. We can kill all stored edits with `discardAllEdits()`. Methods `redoTo()` and `undoTo()` can be used to programmatically invoke `undo()` or `redo()` on all edits from the current edit to the edit provided as parameter.

We can set the maximum number of edits that can be stored with `setLimit()`. The value of the `limit` property (100 by default) can be retrieved with `getLimit()`, and if it is set to a value smaller than the current number of edits, they edits will be reduced using the protected `trimForLimit()` method. Based on the index of the current edit within the `edits` vector, this method will attempt to kill the most balanced number of edits in undoable and redoable states as it can in order to achieve the given limit. The further an edit is (based on its vector index in the edits vector) the more of a candidate it is for death when a trim occurs, as edits sentenced to death are taken from the extreme ends of the `edits` vector.

It is very important to note that when an edit is added to the `edits` vector, all edits in the redoable state (i.e. those appearing after the index of the current edit) do not simply get moved up one index. Rather, they are killed off. So, for example, suppose in a word processor application you enter some text, change the style of 10 different regions of that text, and then undo the most recent 5 style additions. Then a new style change is made. The first 5 style changes that were made remain in the undoable state, and the new edit is added, also in the undoable state. However, the 5 style changes that were undone (i.e. moved to the redoable state) are now completely lost.

Note: All public `UndoManager` methods are synchronized to enable thread-safety, and make `UndoManager` a good candidate for use as a central undo/redo manager for any number of functionalities.

The following code shows how we can modify our UndoRedoPaintApp example to allow multiple undos and redos using an UndoManager. Because UndoManager implements UndoableEditListener, we should normally add UndoableEdits to it using the undoableEditHappened() method rather than addEdit(). This is because undoableEditHappened() calls addEdit() for us, and at the same time allows us to keep track of the source of the operation. This enables UndoManager to act as a central location for all undo/redo edits in an app.

The Code: UndoRedoPaintApp.java
see \Chapter11\4

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

public class UndoRedoPaintApp extends JFrame
{
  protected Vector m_points = new Vector();
  protected PaintCanvas m_canvas = new PaintCanvas(m_points);
  protected UndoManager m_undoManager = new UndoManager();
  protected JButton m_undoButton = new JButton("Undo");
  protected JButton m_redoButton = new JButton("Redo");

  public UndoRedoPaintApp() {
    super("Undo/Redo Demo");

    m_undoButton.setEnabled(false);
    m_redoButton.setEnabled(false);

    JPanel buttonPanel = new JPanel(new GridLayout());
    buttonPanel.add(m_undoButton);
    buttonPanel.add(m_redoButton);

    getContentPane().add(buttonPanel, BorderLayout.NORTH);
    getContentPane().add(m_canvas, BorderLayout.CENTER);

    m_canvas.addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        Point point = new Point(e.getX(), e.getY());
        m_points.addElement(point);

        m_undoManager.undoableEditHappened(new UndoableEditEvent(m_canvas,
          new UndoablePaintSquare(point, m_points)));

        m_undoButton.setText(m_undoManager.getUndoPresentationName());
        m_redoButton.setText(m_undoManager.getRedoPresentationName());
        m_undoButton.setEnabled(m_undoManager.canUndo());
        m_redoButton.setEnabled(m_undoManager.canRedo());
        m_canvas.repaint();
      }
    });

    m_undoButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try { m_undoManager.undo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        m_canvas.repaint();
```

```
        m_undoButton.setEnabled(m_undoManager.canUndo());
        m_redoButton.setEnabled(m_undoManager.canRedo());
      }
    });

    m_redoButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try { m_undoManager.redo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        m_canvas.repaint();
        m_undoButton.setEnabled(m_undoManager.canUndo());
        m_redoButton.setEnabled(m_undoManager.canRedo());
      }
    });

    setSize(400,300);
    setVisible(true);
  }

  public static void main(String argv[]) {
    new UndoRedoPaintApp();
  }
}

// Classes PaintCanvas and UndoablePaintSquare unchanged
// from 11.2.2
```

Run this example and note that we can have up to 100 squares in the undoable or redoable state at any given time. Also note that when several squares are in the redoable state, adding a new square will kill them, and the redo button will become disabled (indicating that no redos can be performed).

11.2.7 The StateEditable interface

abstract interface javax.swing.undo.StateEditable

The `StateEditable` interface is intended for use by objects that wish to maintain specific before (pre) and after (post) states. This provides an alternative to managing undos and redos in `UndoableEdits`. Once a before and after state is defined we can use a `StateEdit` object to switch between the two states. Two methods must be implemented by `StateEditable` implementations. `storeState()` is intended to be used by an object to store its state as a set of key/value pairs in a given `Hashtable`. Normally this entails storing the name of an object and a copy of that object (unless a primitive is stored). `restoreState()` is intended to be used by an object to restore its state according to the key/vaue pairs stored in a given `Hashtable`.

11.2.8 StateEdit

class javax.swing.undo.StateEdit

`StateEdit` extends `AbstractUndoableEdit`, and is meant to store the before and after `Hashtables` of a `StateEditable` instance. When a `StateEdit` is instantiated it is passed a `StateEditable` object, and a protected `Hashtable` called `preState` is passed to that `StateEditable`'s `storeState()` method. Similarly, when `end()` is called on a `StateEdit`, a protected `Hashtable` called `postState` is passed to the corresponding `StateEditable`'s `storeState()` method. After `end()` is called, undos and redos toggle the state of the `StateEditable` between `postState` and `preState` by passing the appropriate `Hashtable` to that `StateEditable`'s `restoreState()` method.

11.2.9 UndoableEditSupport

class javax.swing.undo.UndoableEditSupport

This convenience class is used for managing `UndoableEditListeners`. We can add and remove an `UndoableEditListener` with `addUndoableEditListener()` and `removeUndoableEditListener()` respectively. `UndoableEditSupport` maintains an `updateLevel` property which basically specifies how many times the `beginUpdate()` method has been called. As long as this value is above `0`, `UndoableEdits` added with the `postEdit()` method will be stored in a temporary `CompoundEdit` object without being fired. The `endEdit()` method decrements the `updateLevel` property. When `updateLevel` is `0`, any calls to `postEdit()` will fire the edit passed in, or the `CompoundEdit` that has been accumulating edits up to that point.

Warning: The `endUpdate()` and `beginUpdate()` methods may call `undoableEditHappened()` in each `UndoableEditListener`, possibly resulting in deadlock if these methods are actually invoked from one of the listeners themselves.

11.2.10 CannotUndoException

class javax.swing.undo.CannotUndoException

This exception is thrown when `undo()` is invoked on an `UndoableEdit` that cannot be undone.

11.2.11 CannotRedoException

class javax.swing.undo.CannotRedoException

This exception is thrown when `redo()` is invoked on an `UndoableEdit` that cannot be redone.

11.2.12 Using built-in text component undo/redo functionality

All default text component `Document` models fire `UndoableEdits`. For `PlainDocuments` this involves keeping track of text insertions and removals, as well as any structural changes. For `StyledDocuments`, however, this involves keeping track of a much larger group of changes. Fortunately this work has been built into these document models for us. The following example shows how easy it is to add undo/redo support to text components. Figure 11.8 illustrates.

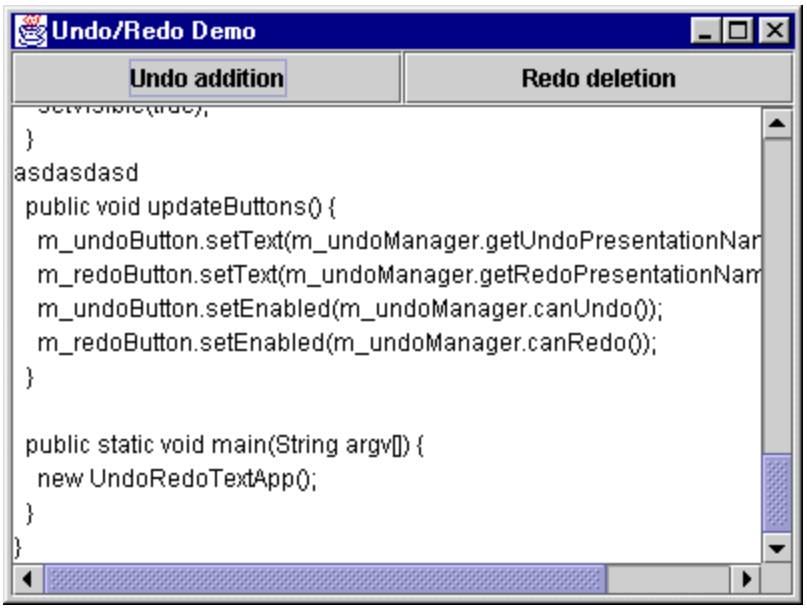


Figure 11.8 Undo/Redo functionality added to a JTextArea.

The Code: UndoRedoTextApp.java
see \Chapter11\5

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

public class UndoRedoTextApp extends JFrame
{
  protected JTextArea m_editor = new JTextArea();
  protected UndoManager m_undoManager = new UndoManager();
  protected JButton m_undoButton = new JButton("Undo");
  protected JButton m_redoButton = new JButton("Redo");

  public UndoRedoTextApp() {
    super("Undo/Redo Demo");

    m_undoButton.setEnabled(false);
    m_redoButton.setEnabled(false);

    JPanel buttonPanel = new JPanel(new GridLayout());
    buttonPanel.add(m_undoButton);
    buttonPanel.add(m_redoButton);

    JScrollPane scroller = new JScrollPane(m_editor);

    getContentPane().add(buttonPanel, BorderLayout.NORTH);
    getContentPane().add(scroller, BorderLayout.CENTER);

    m_editor.getDocument().addUndoableEditListener(
     new UndoableEditListener() {
      public void undoableEditHappened(UndoableEditEvent e) {
        m_undoManager.addEdit(e.getEdit());
        updateButtons();
      }
```

```
    });

    m_undoButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try { m_undoManager.undo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        updateButtons();
      }
    });

    m_redoButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try { m_undoManager.redo(); }
        catch (CannotRedoException cre) { cre.printStackTrace(); }
        updateButtons();
      }
    });

    setSize(400,300);
    setVisible(true);
  }

  public void updateButtons() {
    m_undoButton.setText(m_undoManager.getUndoPresentationName());
    m_redoButton.setText(m_undoManager.getRedoPresentationName());
    m_undoButton.setEnabled(m_undoManager.canUndo());
    m_redoButton.setEnabled(m_undoManager.canRedo());
  }

  public static void main(String argv[]) {
    new UndoRedoTextApp();
  }
}
```

Chapter 12. Menus, Toolbars, and Actions

In this chapter:

- Menus, Toolbars, and Actions overview
- Basic texteditor: part I - Menus
- Basic texteditor: part II - Toolbars and Actions
- Basic texteditor: part III - Custom toolbar components
- Basic texteditor: part IV - Custom menu components

12.1 Menus, toolbars, and actions overview

Drop-down menu bars, context-sensitive popup menus, and draggable toolbars have become commonplace in many modern applications. It is no surprise that Swing offers these features, and in this section we will discuss the classes and interfaces that underly them. The remainder of this chapter is then devoted to the stepwise

construction of a basic text editor application to demonstrate each feature discussed here.

12.1.1 The SingleSelectionModel interface

abstract interface javax.swing.SingleSelectionModel

This simple interface describes a model which maintains a single selected element from a given collection. Methods to assign, retrieve, and clear a selected index are declared, as well as methods for attaching and removing `ChangeListeners`. Implementations are responsible for the storage and manipulation of the collection to be selected from, maintaining an `int` property representing the selected element, maintaining a `boolean` property specifying whether or not an element is selected, and are expected to fire `ChangeEvents` whenever the selected index changes.

12.1.2 DefaultSingleSelectionModel

class javax.swing.DefaultSelectionModel

This is the default implementation of `SingleSelectionModel` used by `JMenuBar` and `JMenuItem`. The `selectedIndex` property represents the selected index at any given time and is -1 when nothing is selected. As expected we can add and remove `ChangeListeners`, and the protected `fireStateChanged()` method is responsible for dispatching `ChangeEvents` whenever the `selectedIndex` property changes.

12.1.3 JMenuBar

class javax.swing.JMenuBar

`JMenuBar` is a container for `JMenus` layed out horizontally in a row, typically residing at the top of a frame or applet. We use the `add(JMenu menu)` method to add a new `JMenu` to a `JMenuBar`. We use the `setJMenuBar()` method in `JFrame`, `JDialog`, `JApplet`, `JRootPane`, and `JInternalFrame` to set the menu bar for these containers (recall that each of these containers implements `RootPaneContainer`, which enforces the definition of `setJMenuBar()`--see chapter 3). `JMenuBar` uses a `DefaultSingleSelectionModel` to enforce the selection of only one child at any given time.

A `JMenuBar` is a `JComponent` subclass and, as such, can be placed anywhere in a container just as any other Swing component (this functionality is not available with AWT menu bars).

Warning: `JMenuBar` defines the method `setHelpMenu(JMenu menu)` which is intened to mark a single menu contained in a `JMenuBar` as the designated 'help' menu. The `JMenuBar` UI delegate may be responsible for positioning and somehow treating this menu differently than others. However, this is not implemented as of Java 2 FCS, and generates an exception if used.

Note: One lacking feature in the current `JMenuBar` implementation, or its UI delegate, is the ability to easily control the spacing between its `JMenu` children. As of Java 2 FCS, the easiest way to control this is by overriding `JMenuBar` and manually taking control of its layout. By default `JMenubar` uses an x-oriented `BoxLayout`.

`JMenuBar` provides several methods to retrieve its child components, set/get the currently selected item, register/unregister with the current `KeyBoardManager` (see chapter 2, section 2.13), and the `isManagingFocus()` method which simply returns `true` to indicate that `JMenuBar` handles focus management internally. Public methods `processKeyEvent()` and `processMouseEvent()` are implemented only to satisfy the `MenuElement` interface (see below) requirements, and do nothing by default.

12.1.4 JMenuItem

class javax.swing.JMenuItem

This class extends `AbstractButton` (see chapter 4, section 4.1) and represents a single menu item. We can assign icons and keyboard mnemonics just as we can with buttons. A mnemonic is represented graphically by underlining the first instance of the corresponding character, just as it is in buttons. Icon and text placement can be dealt with identically to how we deal with this functionality in buttons.

We can also attach keyboard accelerators to a `JMenuItem` (i.e. we can register keyboard actions with a `JMenuItem` -- see chapter 2 section 2.13). Every `JComponent` decendent inherits similar functionality. When assigned to a `JMenuItem`, the accelerator will appear as small text to the right of the menu item text. An accelerator is a key or combination of keys that can be used to activate a menu item. Contrary to a mnemonic, an accelerator will invoke a menu item even when the popup containing it is not visible. The only necessary condition for accelerator activation is that the window containing the target menu item is currently active. To add an accelerator corresponding to CTRL+A we can do the following:

```
myJMenuItem.setAccelerator(KeyStroke.getKeyStroke(
  KeyEvent.VK_A, KeyEvent.CTRL_MASK, false);
```

Note: `JMenuItem` is the only Swing component that graphically displays an assigned keyboard accelerator.

We normally attach an `ActionListener` to a menu item. As any button, whenever the menu item is clicked the `ActionListener` is notified. Alternatively we can use `Actions` (discussed below and briefly in 2.13) which provide a convenient means of creating a menu item as well as definining the corresponding action handling code. A single `Action` instance can be used to create an arbitrary number of `JMenuItems` and `JButtons` with identical action handling code. We will see how this is done soon enough. It suffices to say here that when an `Action` is disabled, all `JMenuItems` associated with that `Action` are disabled and, as buttons always do in the disabled state, appear grayed out.

As any `AbstractButton` decendent, `JMenuItem` fires `ActionEvents` and `ChangeEvents` and allows attachment of `ActionListeners` and `ChangeListeners` accordingly. `JMenuItem` will also fire `MenuDragMouseEvents` (see below) when the mouse enters, exits, is dragged, or a mouse button is released inside its bounds, and `MenuKeyEvents` when a key is pressed, typed, or released. Both of these Swing-specific events will only be fired when the popup containing the corresponding menu item is visible. As expected, we can add `MenuDragMouseListeners` and `MenuKeyEventListeners` for notification of these events. Several public `processXXEvent()` methods are also provided to receive and respond to events dispatched to a `JMenuItem`, some of which are forewarded from the current `MenuSelectionManager` (see below).

12.1.5 JMenu

class javax.swing.JMenu

This class extends `JMenuItem` and is usually added to a `JMenuBar` or to another `JMenu`. In the former case it will act as a menu item which pops up a `JPopupMenu` containing child menu items. If a `JMenu` is added to another `JMenu` it will appear in that menu's corresponding popup as a menu item with an arrow on its right side. When that menu item is activated by mouse movement or keyboard selection a popup will appear displaying its corresponding child menu items. Each `JMenu` maintains a `topLevelMenu` property which is `false` for sub-menus and `true` otherwise.

`JMenu` uses a `DefaultButtonModel` to manage its state, and it holds a private instance of `JPopupMenu` (see below) used to display its associated menu items when it is activated with the mouse or a keyboard

mnemonic.

Note: Unlike its JMenuItem parent, JMenu specifically overrides setAccelerator() with an empty implementation to disallow keyboard accelerators. This is because it assumes that we will only want to activate a menu (i.e. display its popup) when it is already visible; and for this we can use a mnemonic.

We can display/hide the associated popup programmatically by setting the popupMenuVisible property, and we can access the popup using getPopupMenu(). We can set the coordinate location where the popup is displayed with setMenuLocation(). We can assign a specific delay time in milliseconds using setDelay() to specify how long a JMenu should wait before displaying its popup when activated.

We use the overloaded add() method to add JMenuItems, Components, Actions (see below) or Strings to a JMenu. (Adding a String simply creates a JMenuItem child with the given text.) Similarly we can use several variations of overloaded insert() and remove() methods to insert and remove existing children. JMenu also directly supports creation and insertion of separator components in its popup, using addSeparator(), which provides a convenient means of visually organizing child components into groups.

The protected createActionChangeListener() method is used when an Action is added to a JMenu to create a PropertyChangeListener for internal use in responding to bound property changes that occur in that Action (see below). The createWinListener() method is used to create an instance of the protected inner class JMenu.WinListener which is used to deselect a menu when its corresponding popup closes. We are rarely concerned with these methods, and only subclasses desiring a more complete customization will override them.

Along with event dispatching/handling inherited from JMenuItem, JMenu adds functionality for firing and capturing MenuEvents (see below) used to notify attached MenuListeners when its current selection changes.

UI Guideline: Flat and wide design
Recent research in usability has shown that menus with too many levels of hierachy don't work well. Features get buried too many layers deep. Some operating systems restrict menus to 3 levels i.e. the main menu bar, a pull-down menu and a single walking pop-up menu.
A maximum 3 Levels would be appear to be a good rule of thumb. Don't be tempted to use popup menus to create a complex series of hierarchical choices. Keep menus flatter.
For each menu, another good rule of thumb is to provide 7 +/- 2 options. However, if you have too many choices, it is better to break this rule and go to 10 or more than to introduce additional hierarchy.

12.1.6 JPopupMenu

class javax.swing.JPopupMenu

This class represents a small window which pops up and contains a collection of components layed out in a single column by default using, suprisingly, a GridBagLayout (note that there is nothing stopping us from changing JPopupMenu's layout manager). JPopupMenu uses a DefaultSingleSelectionModel to enforce the selection of only one child at any given time.

JMenu simply delegates all its add(), remove(), insert(), addSeparator(), etc., calls to its internal JPopupMenu. As expected, JPopupMenu provides similar methods. The addSeparator() method inserts an instance of the inner class JPopupMenu.Separator (a subclass of JSeparator -- discussed below). The show() method displays a JPopupMenu at a given position within the corrdinate system of a given component. This component is referred to as the invoker component, and JPopupMenu can be assigned an invoker by setting its invoker property. JComponent's setVisible() method is overriden to display a

JPopupMenu with respect to its current invoker, and we can change the location it will appear using setLocation(). We can also control a JPopupMenu's size with the overloaded setPopupSize() methods, and we can use the pack() method (similar to the java.awt.Window method of the same name) to request that a popup change size to the minimum required for correct display of its child components.

Note: JComboBox's UI delegate uses a JPopupMenu subclass to display its popup list.

When the need arises to display our own JPopupMenu, it is customary, but certainly not necessary, to do so in response to a platform-dependent mouse gesture (e.g. a right-click on Windows platforms). Thus, the java.awt.event.MouseEvent class provides a simple method we can use in a platform-independent manner to check whether a platform-dependent popup gesture has occurred. This method, isPopupTrigger(), will return true if the MouseEvent it is called on represents the current operating system's popup trigger gesture.

JPopupMenu has the unique ability to act as either a heavyweight or lighweight component. It is smart enough to detect when it will be displayed completely within a Swing container or not and adjust itself accordingly. However, there may be cases in which the default behavior may not be acceptable. Recall from chapter 2 that we must set JPopupMenu's lightWeightPopupEnabled property to false to force it to be heavyweight and allow overlapping of other heavyweight components that might reside in the same container. Setting this property to true will force a JPopupMenu to remain lightweight. The static setDefaultLightWeightPopupEnabled() method serves the same purpose, but affects all JPopupMenu's created from that point on (in the current implementation all popups existing before this method is called will retain their previous lightweight/heavyweight settings).

Bug Alert! Due to an AWT bug, all popups are forced into lightweight mode when displayed in modal dialogs (regardless of the state of the lightWeightPopupEnabled property).

The protected createActionChangeListener() method is used when an Action (see below) is added to a JPopupMenu to create a PropertyChangeListener for internal use in responding to bound property changes that occur in that Action.

A JPopupMenu fires PopupMenuEvents (discussed below) whenever it is made visible, hidden, and cancelled. As expected we can attatch PopupMenuListeners to capture these events.

12.1.7 JSeparator

class javax.swing.JSeparator

This class represents a simple separator component with a UI delegate responsible for displaying a horizontal or vertical line. We can specify which orientation a JSeparator should use by changing its orientation property. This class is most often used in menus and toolbars, however, it is a standard Swing component and there is nothing stopping us from using JSeparators anywhere we like.

We normally do not use JSeparator explicitly. Rather, we use the addSeparator() method of JMenu, JPopupMenu, and JToolBar. JMenu delegates this call to its JPopupMenu which, as we know, uses an instance of its own custom JSeparator subclass which is rendered as a horizontal line. JToolBar also uses its own custom JSeparator sub-class which has no graphical representation, and appears as just an empty region. Unlike menu separators, however, JToolBar's separator allows explicit instantiation and provides a method for assigning a new size in the form of a Dimension.

UI Guideline: Use of a separator

Use a separator to group related menu choices and separate them from others. This gives better visual communication and better usability by providing a space between the target areas for groups of choices. This reduces the chance of an error when making a selection with the mouse.

12.1.8 JCheckBoxMenuItem

class javax.swing.JCheckBoxMenuItem

This class extends `JMenuItem` and can be selected, deselected, and rendered identical to `JCheckBox` (see chapter 4). We use the `isSelected()/setSelected()` or `getState()/setState()` methods to determine/set the selection state respectively. `ActionListeners` and `ChangeListeners` can be attached to a `JCheckBoxMenuItem` for notification about changes in its state (see `JMenuItem` discussion for inherited functionality). We often use `JCheckBoxMenuItems` in `ButtonGroups` to enforce the selection of only one item in a group at any given time.

12.1.9 JRadioButtonMenuItem

class javax.swing.JRadioButtonMenuItem

This class extends `JMenuItem` and can be selected, deselected, and rendered identical to `JRadioButton` (see chapter 4). We use the `isSelected()/setSelected()` or `getState()/setState()` methods to determine/set the selection state respectively. `ActionListeners` and `ChangeListeners` can be attached to a `JRadioButtonMenuItem` for notification about changes in its state (see `JMenuItem` discussion for inherited functionality). We often use `JRadioButtonMenuItems` in `ButtonGroups` to enforce the selection of only one item in a group at any given time.

UI Guideline: Widget overloading

As a general rule in UI Design, it is not desirable to overload components and use them for two purposes. By adding Checkboxes or Radio Buttons to a menu, you are changing the purpose of a menu from one of navigation to one of selection. This is an important point to understand.

Making this change is an acceptable design technique when it will speed operation and enhance usability by removing the need for a cumbersome dialog or option pane. However, it is important to assess that it does not otherwise adversely affect usability.

Groups of Radio Button or Checkbox menu items are probably best isolated by using a JSeparator.

12.1.10 The MenuElement interface

abstract interface javax.swing.MenuElement

This interface must be implemented by all components that wish to act as menu items. By implementing the methods of this interface any components can act as a menu item, making it quite easy to build our own.

The `getSubElements()` method returns an array of `MenuElements` containing the given item's sub-elements. The `processKeyEvent()` and `processMouseEvent()` methods are called to process keyboard and mouse events respectively when the implementing component has the focus. Unlike methods with the same name in the `java.awt.Component` class, these two methods receive three parameters: the `KeyEvent` or `MouseEvent`, respectively, which should be processed, an array of `MenuElements` which forms the menu path to the implementing component, and the current `MenuSelectionManager` (see below). The `menuSelectionChanged()` method is called by the `MenuSelectionManager` when the implementing component is added or removed from its current selection state. The `getComponent()` method returns a

reference to a component that is responsible for the rendering of the implementing component.

Note: The `getComponent()` method is interesting, as it allows classes that are not `Components` themselves to implement the `MenuElement` interface and act as menu elements when necessary. Such a class must provide a `Component` used for display in a menu, and this `Component` would be returned by `getComponent()`. This design has curious implications, allowing us to design robust JavaBeans that encapsulate an optional GUI representation. We can imagine a complex spell-checker or dictionary class implementing the `MenuElement` interface and providing a custom component for display in a menu (a powerful and highly object-oriented bean indeed).

`JMenuItem`, `JMenuBar`, `JPopupMenu`, and `JMenu` all implement this interface. Note that each of their `getComponent()` methods simply return a `this` reference. Also note that by extending any of these implementing classes, we inherit `MenuElement` functionality and are therefore not required to implement it. (We won't explicitly use this interface in any examples, as the custom component we will build at the end of this chapter is an extension of `JMenu`.)

12.1.11 MenuSelectionManager

class javax.swing.MenuSelectionManager

`MenuSelectionManager` is a service class responsible for managing menu selection throughout a single Java session. (Note that unlike most other service classes in Swing, `MenuSelectionManager` does not register its shared instance with `AppContext`--see chapter 2.) When `MenuElement` implementations receive `MouseEvents` or `KeyEvents`, these events should not be processed directly. Rather, they should be handed off to the `MenuSelectionManager` so that it may forward them to sub-components automatically. For instance, whenever a `JMenuItem` is activated by keyboard or mouse, or whenever a `JMenuItem` selection occurs, the menu item UI delegate is responsible for forwarding the corresponding event to the `MenuSelectionManager` if necessary. The following code shows how `BasicMenuItemUI` deals with mouse releases:

```
public void mouseReleased(MouseEvent e) {
  MenuSelectionManager manager =
    MenuSelectionManager.defaultManager();
  Point p = e.getPoint();
  if(p.x >= 0 && p.x < menuItem.getWidth() &&
   p.y >= 0 && p.y < menuItem.getHeight()) {
    manager.clearSelectedPath();
    menuItem.doClick(0);
  }
  else {
    manager.processMouseEvent(e);
  }
}
```

The static `defaultManager()` method returns the `MenuSelectionManager` shared instance, and the `clearSelectedPath()` method tells the currently active menu hierarchy to close and unselect all menu components. In the code shown above, `clearSelectedPath()` will only be called if the mouse release occurs within the corresponding `JMenuItem` (in which case there is no need for the event to propogate any further). If this is not the case, the event is sent to `MenuSelectionManager`'s `processMouseEvent()` method which forwards it to other sub-components. `JMenuItem` doesn't have any sub-components by default so not much interesting happens in this case. However, in the case of `JMenu`, which considers its popup menu a sub-component, sending a mouse released event to the `MenuSelectionManager` is expected no matter what (from `BasicMenuUI`):

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

```
      MenuSelectionManager manager =
        MenuSelectionManager.defaultManager();
      manager.processMouseEvent(e);
      if (!e.isConsumed())
        manager.clearSelectedPath();
    }
```

`MenuSelectionManager` will fire `ChangeEvents` whenever its `setSelectedPath()` method is called (i.e. each time a menu selection changes). As expected, we can attach `ChangeListeners` to listen for these events.

12.1.12 The MenuDragMouseListener interface

abstract interface javax.swing.event.MenuDragMouseListener

This listener receives notification when the mouse cursor enters, exits, is released, or is moved over a menu item.

12.1.13 MenuDragMouseEvent

class javax.swing.event.MenuDragMouseEvent

This event class is used to deliver information to `MenuDragMouseListeners`. It encapsulates the component source, event id, time of the event, bitwise or-masked `int` specifying which mouse button and/or keys (`CTRL`, `SHIFT`, `ALT`, or `META`) were pressed at the time of the event, x and y mouse coordinates, number of clicks immediately preceding the event, whether or not the event represents the platform-dependent popup trigger, an array of `MenuElements` leading to the source of the event, and the current `MenuSelectionManager`. This event inherits all `MouseEvent` functionality (see API docs) and adds two methods for retrieving the array of `MenuElements` and the `MenuSelectionManager`.

12.1.14 The MenuKeyListener interface

abstract interface javax.swing.event.MenuKeyListener

This listener is notified when a menu item receives a key event corresponding to a key press, release, or type. These events don't necessarily correspond to mnemonics or accelerators, and are received whenever a menu item is simply visible on the screen.

12.1.15 MenuKeyEvent

class javax.swing.event.MenuKeyEvent

This event class is used to deliver information to `MenuKeyListeners`. It encapsulates the component source, event id, time of the event, bitwise or-masked `int` specifying which mouse button and/or keys (`CTRL`, `SHIFT`, or `ALT`) were pressed at the time of the event, an `int` and `char` identifying the source key that caused the event, an array of `MenuElements` leading to the source of the event, and the current `MenuSelectionManager`. This event inherits all `KeyEvent` functionality (see API docs) and adds two methods for retrieving the array of `MenuElements` and the `MenuSelectionManager`.

12.1.16 The MenuListener interface

abstract interface javax.swing.event.MenuListener

This listener receives notification when a menu is selected, deselected, or cancelled. Three methods must be implemented by `MenuListeners`, and each takes a `MouseEvent` parameter: `menuCanceled()`,

menuDeselected(), and menuSelected().

12.1.17 MenuEvent

class javax.swing.event.MenuEvent

This event class is used to deliver information to MenuListeners. It simply encapsulates a reference to its source Object.

12.1.18 The PopupMenuListener interface

abstract interface javax.swing.event.PopupMenuListener

This listener receives notification when a JPopupMenu is about to become visible, hidden, or when it is cancelled. Canceling a JPopupMenu also causes it to be hidden, so two PopupMenuEvents are fired in this case. A cancel occurs when the invoker component is resized or the window containing the invoker changes size or location. Three methods must be implemented by PopupMenuListeners, and each takes a PopupMenuEvent parameter: popupMenuCanceled(), popupMenuWillBecomeVisible(), and popupMenuWillBecomeInvisible().

12.1.19 PopupMenuEvent

class javax.swing.event.PopupMenuEvent

This event class is used to deliver information to PopupMenuListeners. It simply encapsulates a reference to its source Object.

12.1.20 JToolBar

class javax.swing.JToolBar

This class represents the Swing implementation of a toolbar. Toolbars are often placed directly below menu bars at the top of a frame or applet, and act as a container for any component (buttons and combo boxes are most common). The most convenient way to add buttons to a JToolBar is to use Actions (discussed below).

Note: Components often need their alignment setting tweaked to provide uniform positioning within JToolBar. This can be accomplished through use of the setAlignmentY() and setAlignmentX() methods. We will see that this is necessary in the final example of this chapter.

JToolBar also allows convenient addition of an inner JSeparator sub-class, JToolBar.Separator, to provide an empty space for visually grouping components. These separators can be added with either of the overloaded addSeparator() methods, one of which takes a Dimension parameter specifying the size of the separator.

Two orientations are supported, VERTICAL and HORIZONTAL, and managed by JToolBar's orientation property. It uses a BoxLayout layout manager which is dynamically changed between Y_AXIS and X_AXIS when the orientation property changes.

JToolBar can be dragged in and out of its parent container if its floatable property is set to true. When dragged out of its parent, a JToolBar appears as a floating window and its border changes color depending on whether it can re-dock in its parent at a given location. If a JToolBar is dragged outside of its parent and released, it will be placed in its own JFrame which is fully maximizable, minimizable, and closable. When this frame is closed JToolBar will jump back into its most recent dock position in its original parent, and the

floating `JFrame` will disappear. It is recommended that `JToolBar` is placed in one of the four sides of a container using a `BorderLayout`, leaving the other sides unused. This allows the `JToolBar` to be docked in any of that container's side regions.

The protected `createActionChangeListener()` method is used when an `Action` (see below) is added to a `JToolBar` to create a `PropertyChangeListener` for internal use in responding to bound property changes that occur in that `Action`.

UI Guideline: 3 Uses for a Toolbar

Toolbars have become ubiquitious in modern software. However, they are often overused or misused and fail to achieve their objective of increased usability. There are three key uses which have subtle differences and implications.

Tool Selection or Mode Selection

Perhaps the most effective use of a toolbar is, as the name suggests, for the selection of a tool or operational mode. This is most common in drawing or image manipulation packages. The user selects the toolbar button to change the mode from "paintbrush" to "filler" to "draw box" to "cut" etc.. This is a highly effective use of toolbar, as the small icons are usually sufficient to render a suitable tool image. Many images for this purpose have been adopted as a defacto standard. If you are developing such a tool selection toolbar, it would be advisable to stick closely to icons which have been used by similar existing products.

Functional Selection

The earliest use of toolbar was to replace the selection of a specific function from the menu. This led to them being called "speedbars" or "menubars". The idea was that the small icon button was faster and easier to acquire than the menu selection and that usability was enhanced as a result. This worked well for many common functions in file oriented applications, such as "open file", "new file", "save", "cut", "copy" and "paste". In fact, most of us would recognise the small icons for all of these functions. However, with other more application specific functions, it has been much harder for icon designers to come up with appropriate designs. This often leads to applications which have a confusing and intimidating array of icons across the top of the screen. This can detract from usability. As a general rule of thumb, stick to common cross application functions when overloading menu selections with toolbar buttons. If you do need to break the rule then consider selecting annotated buttons for the toolbar.

Navigational Selection

The 3rd use for toolbars has been for navigational selection. This often means replacing or overloading menu options from a "Window", "Go" or "Form" menu. These menu options are used to select a specific screen to come to the front. The toolbar buttons replace or overload the menu option and allow the navigational selection to be made supposedly by faster means. However, again this suffers from the problem of appropriate icon design. It is usually too difficult to devise a suitable set of icons which have clear and unambiguous meaning. Therefore, as a rule of thumb, consider the use of annotated buttons on the toolbar.

12.1.2.1 Custom JToolBar separators

Unfortunately Swing does not include a toolbar-specific separator component that will display a vertical or horizontal line depending on current toolbar orientation. The following psuedo-code shows how we can build such a component under the assumption that it will always have a `JToolBar` as direct parent:

```
public class MyToolBarSeparator extends JComponent
{
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (getParent() instanceof JToolBar) {
      if (((JToolBar) getParent()).getOrientation()
       == JToolBar.HORIZONTAL) {
```

```
        // paint a vertical line
      }
      else {
        // paint a horizontal line
      }
    }
  }

  public Dimension getPreferredSize() {
    if (getParent() instanceof JToolBar) {
      if (((JToolBar) getParent()).getOrientation()
       == JToolBar.HORIZONTAL) {
        // return size of vertical bar
      }
      else {
        // return size of horizontal bar
      }
    }
  }
}
```

UI Guideline: Use of a separator

The failure to include a separator for toolbars really was an oversight by the Swing designers. Again, use the separator to group related functions or tools. For example, if the function all belong on the same menu then group them together, or if the tools (or modes) are related such as "cut", "copy", "paste" then group them together and separate them from others with a separator.

Visual grouping like this improves visual separation by introducing a visual layer. The viewer can first acquire a group of buttons and then a specific button. They will also learn with directional memory the approximate position of each group. By separating them you will improve the usability by helping them to acquire the target better when using the mouse.

12.1.22 Changing JToolBar's floating frame behavior

The behavior of `JToolBar`'s floating `JFrame` is certainly useful, but it is arguable whether the maximization and resizability should be allowed. Though we cannot control whether or not a `JFrame` can be maximized, we can control whether or not it can be resized. To enforce non-resizability in `JToolBar`'s floating `JFrame` (and set its dispayed title while we're at it) we need to override its UI delegate and customize the `createFloatingFrame()` method as follows:

```
public class MyToolBarUI
 extends javax.swing.plaf.metal.MetalToolBarUI {
  protected JFrame createFloatingFrame(JToolBar toolbar) {
    JFrame frame = new JFrame(toolbar.getName());
    frame.setTitle("My toolbar");
    frame.setResizable(false);
    WindowListener wl = createFrameListener();
    frame.addWindowListener(wl);
    return frame;
  }
}
```

To assign `MyToolBarUI` as a `JToolBar`'s UI delegate we can do the following:

```
mytoolbar.setUI(new MyToolBarUI());
```

To force use of this delegate on a global basis we can do the following before any `JToolBars` are

instantiated:

```
UIManager.getDefaults().put(
  "ToolBarUI","com.mycompany.MyToolBarUI");
```

Note that we may also have to add an associated `Class` instance to the `UIDefaults` table for this to work (see chapter 21).

UIGuideline: Use of a Floating Frame

It is probably best to restrict the use of a floating toolbar frame to toolbars being used for tool or mode selection (see guideline in 12.1.20).

12.1.23 The Action interface

abstract interface javax.swing.Action

This interface describes a helper object which extends `ActionListener` and which supports a set of bound properties. We use appropriate `add()` methods in the `JMenu`, `JPopupMenu`, and `JToolBar` classes to add an `Action` which will use information from the given instance to create and return a component that is appropriate for that container (a `JMenuItem` in the case of the first two, a `JButton` in the case of the latter). The same `Action` instance can be used to create an arbitrary number of menu items or toolbar buttons.

Because `Action` extends `ActionListener`, the `actionPerformed()` method is inherited and can be used to encapsulte appropriate `ActionEvent` handling code. When a menu item or toolbar button is created using an `Action`, the resulting component is registered as a `PropertyChangeListener` with the `Action`, and the `Action` is registered as an `ActionListener` with the component. Thus, whenever a change occurs to one of that `Action`'s bound properties, all components with registered `PropertyChangeListeners` will receive notification. This provides a convenient means for allowing identical functionality in menus, toolbars, and popup menus with minimum code repetition and object creation.

The `putValue()` and `getValue()` methods are intended to work with a `Hashtable`-like structure to maintain an `Action`'s bound properties. Whenever the value of a property changes, we are expected to fire `PropertyChangeEvents` to all registered listeners. As expected, methods to add and remove `PropertyChangeListeners` are provided.

The `Action` interface defines five static property keys intended for use by `JMenuItems` and `JButtons` created with an `Action` instance:

`String DEFAULT`: [not used]

`String LONG_DESCRIPTION`: Used for a lengthy description of an `Action`.

`String NAME`: Used as the text in associated and displayed in `JMenuItems` and `JButtons`.

`String SHORT_DESCRIPTION`: Used for the tooltip text of associated `JMenuItems` and `JButtons`.

`String SMALL_ICON`: Used as the icon in associated `JMenuItems` and `JButtons`.

12.1.24 AbstractAction

class javax.swing.AbstractAction

This class is an abstract implementation of the `Action` interface. Along with the properties inherited from `Action`, `AbstractAction` defines the `enabled` property which provides a means of enabling/disabling all

associated components registered as `PropertyChangeListeners`. A `SwingPropertyChangeSupport` instance is used to manage the firing of `PropertyChangeEvents` to all registered `PropertyChangeListeners` (see chapter 2 for more about `SwingPropertyChangeSupport`).

12.2 Basic texteditor: partI - menus

In this example we begin the construction of a basic texteditor application using a menu bar and several menu items. The menu bar contains two `JMenus` labeled "File" and "Font." The "File" menu contains `JMenuItems` for creating a new (empty) document, opening a text file, saving the current document as a text file, and exiting the application. The "Font" menu contains `JCheckBoxMenuItems` for making the document bold and/or italic, as well as `JRadioButtonMenuItems` organized into a `ButtonGroup` allowing selection of a single font.

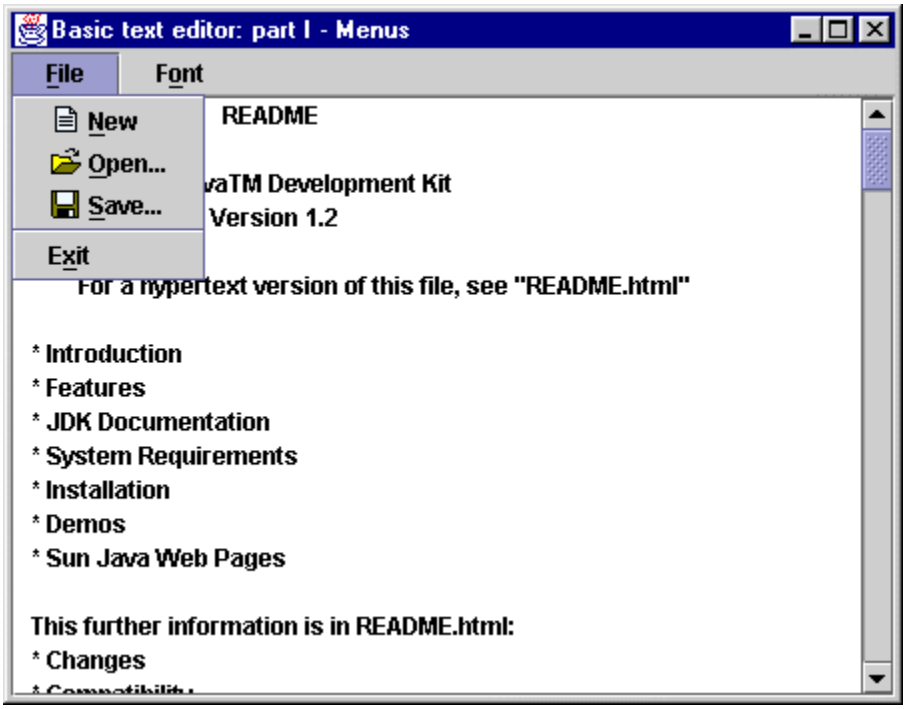


Figure 12.1 JMenu containing JMenuItems with mnemonics and icons.

<<file figure12-1.gif>>

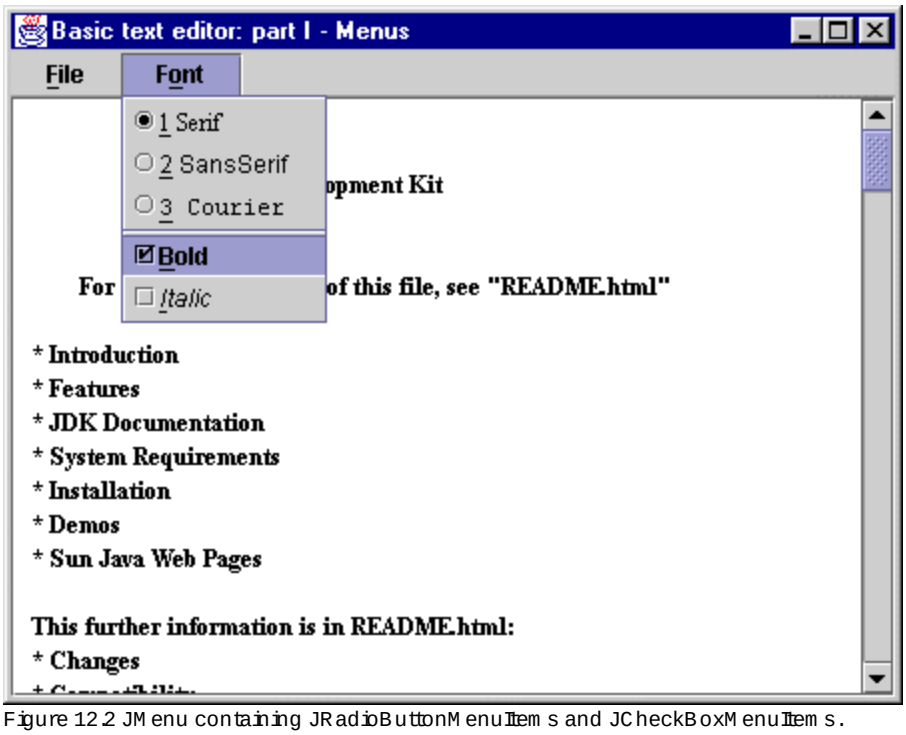


Figure 12.2 JMenu containing JRadioButtonMenuItems and JCheckBoxMenuItems.

<<file figure12-2.gif>>

The Code: BasicTextEditor.java

see \Chapter12\1

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

import javax.swing.*;
import javax.swing.event.*;

public class BasicTextEditor extends JFrame
{
  public static final String FONTS[] =
    { "Serif", "SansSerif", "Courier" };

  protected Font m_fonts[];

  protected JTextArea m_monitor;
  protected JMenuItem[] m_fontMenus;
  protected JCheckBoxMenuItem m_bold;
  protected JCheckBoxMenuItem m_italic;
  protected JFileChooser m_chooser;

  public BasicTextEditor(){
    super("Basic text editor: part I - Menus");
    setSize(450, 350);

    m_fonts = new Font[FONTS.length];
    for (int k=0; k<FONTS.length; k++)
      m_fonts[k] = new Font(FONTS[k], Font.PLAIN, 12);

    m_monitor = new JTextArea();
    JScrollPane ps = new JScrollPane(m_monitor);
```

```
    getContentPane().add(ps, BorderLayout.CENTER);

    m_monitor.append("Basic text editor");

    JMenuBar menuBar = createMenuBar();
    setJMenuBar(menuBar);

    m_chooser = new JFileChooser();
    m_chooser.setCurrentDirectory(new File("."));

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    updateMonitor();
    setVisible(true);
  }

  protected JMenuBar createMenuBar() {
    final JMenuBar menuBar = new JMenuBar();

    JMenu mFile = new JMenu("File");
    mFile.setMnemonic('f');

    JMenuItem item = new JMenuItem("New");
    item.setIcon(new ImageIcon("file_new.gif"));
    item.setMnemonic('n');
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_monitor.setText("");
      }
    };
    item.addActionListener(lst);
    mFile.add(item);

    item = new JMenuItem("Open...");
    item.setIcon(new ImageIcon("file_open.gif"));
    item.setMnemonic('o');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        BasicTextEditor.this.repaint();
        if (m_chooser.showOpenDialog(BasicTextEditor.this) !=
         JFileChooser.APPROVE_OPTION)
          return;
        Thread runner = new Thread() {
          public void run() {
            File fChoosen = m_chooser.getSelectedFile();
            try {
              FileReader in = new FileReader(fChoosen);
              m_monitor.read(in, null);
              in.close();
            }
            catch (IOException ex) { ex.printStackTrace(); }
          }
        };
        runner.start();
      }
    };
    item.addActionListener(lst);
```

```
mFile.add(item);

item = new JMenuItem("Save...");
item.setIcon(new ImageIcon("file_save.gif"));
item.setMnemonic('s');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e){
    BasicTextEditor.this.repaint();
    if (m_chooser.showSaveDialog(BasicTextEditor.this) !=
     JFileChooser.APPROVE_OPTION)
      return;
    Thread runner = new Thread() {
      public void run() {
        File fChoosen = m_chooser.getSelectedFile();
        try {
          FileWriter out = new FileWriter(fChoosen);
          m_monitor.write(out);
          out.close();
        }
        catch (IOException ex) { ex.printStackTrace(); }
      }
    };
    runner.start();
  }
};
item.addActionListener(lst);
mFile.add(item);

mFile.addSeparator();

item = new JMenuItem("Exit");
item.setMnemonic('x');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
};
item.addActionListener(lst);
mFile.add(item);
menuBar.add(mFile);

ActionListener fontListener = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    updateMonitor();
  }
};

JMenu mFont = new JMenu("Font");
mFont.setMnemonic('o');

ButtonGroup group = new ButtonGroup();
m_fontMenus = new JMenuItem[FONTS.length];
for (int k = 0; k < FONTS.length; k++) {
  int m = k+1;
  m_fontMenus[k] = new JRadioButtonMenuItem(
    m + " " + FONTS[k]);
  boolean selected = (k == 0);
  m_fontMenus[k].setSelected(selected);
  m_fontMenus[k].setMnemonic('1' + k);
  m_fontMenus[k].setFont(m_fonts[k]);
  m_fontMenus[k].addActionListener(fontListener);
  group.add(m_fontMenus[k]);
```

```
      mFont.add(m_fontMenus[k]);
    }

    mFont.addSeparator();

    m_bold = new JCheckBoxMenuItem("Bold");
    m_bold.setMnemonic('b');
    Font fn = m_fonts[1].deriveFont(Font.BOLD);
    m_bold.setFont(fn);
    m_bold.setSelected(false);
    m_bold.addActionListener(fontListener);
    mFont.add(m_bold);

    m_italic = new JCheckBoxMenuItem("Italic");
    m_italic.setMnemonic('i');
    fn = m_fonts[1].deriveFont(Font.ITALIC);
    m_italic.setFont(fn);
    m_italic.setSelected(false);
    m_italic.addActionListener(fontListener);
    mFont.add(m_italic);

    menuBar.add(mFont);

    return menuBar;
  }

  protected void updateMonitor() {
    int index = -1;
    for (int k = 0; k < m_fontMenus.length; k++) {
      if (m_fontMenus[k].isSelected()) {
        index = k;
        break;
      }
    }
    if (index == -1)
      return;

    if (index == 2) {  // Courier
      m_bold.setSelected(false);
      m_bold.setEnabled(false);
      m_italic.setSelected(false);
      m_italic.setEnabled(false);
    }
    else {
      m_bold.setEnabled(true);
      m_italic.setEnabled(true);
    }

    int style = Font.PLAIN;
    if (m_bold.isSelected())
      style |= Font.BOLD;
    if (m_italic.isSelected())
      style |= Font.ITALIC;
    Font fn = m_fonts[index].deriveFont(style);
    m_monitor.setFont(fn);
    m_monitor.repaint();
  }

  public static void main(String argv[]) {
    new BasicTextEditor();
  }
}
```

Class BasicTextEditor

This class extends `JFrame` and provides the parent frame for our example. One class variable is declared:

`String FONTS[]`: an array of font family names.

Instance variables:

`Font[] m_fonts`: an array of `Font` instances which can be used to render our `JTextArea` editor.

`JTextArea m_monitor`: used as our text editor.

`JMenuItem[] m_fontMenus`: an array of menu items representing available fonts.

`JCheckBoxMenuItem m_bold`: menu item which sets/unsets the bold property of the current font.

`JCheckBoxMenuItem m_italic`: menu item which sets/unsets the italic property of the current font.

`JFileChooser m_chooser`: used to load and save simple text files.

The `Menu1` constructor populates our `m_fonts` array with `Font` instances corresponding to the names provided in `FONTS[]`. The `m_monitor JTextArea` is then created and placed in a `JScrollPane`. This scrollpane is added to the center of our frame's content pane and we append some simple text to `m_monitor` for display at startup. Our `createMenuBar()` method is called to create the menu bar to manage this application, and this menu bar is then added to our frame using the `setJMenuBar()` method.

The `createMenuBar()` method creates and returns a `JMenuBar`. Each menu item receives an `ActionListener` to handle it's selection. Two menus are added titled "File" and "Font." The "New" menu item in the "File" menu is responsible for creating a new (empty) document. It doesn't really replace `JTextArea`'s `Document`. Instead it simply clears the contents of our editor component. Note that an icon is used in this menu item. Also note that this menu item can be selected with the keyboard by pressing 'n' when the "File" menu's popup is visible, because we assigned it 'n' as a mnemonic. The File menu was also assigned a mnemonic character, 'f,' and pressing ALT-F while the application frame is active, the "File" popup will be displayed allowing navigation with either the mouse or keyboard (all other menus and menu items in this example also receive appropriate mnemonics).

Bug Alert! Even though ALT-F will get processed by the corresponding menu in the menu bar, the key event that is generated is still received by `JTextArea`. Thus, you will often see an 'f' added to the text area even though ALT was held down when 'f' was pressed. It is possible to work around this problem by extending `JTextArea` (or any other `JTextComponent`) as follows:

```
public class MyTextArea extends JTextArea
{
  protected synchronized void processComponentKeyEvent(
   KeyEvent anEvent) {
    super.processComponentKeyEvent(anEvent);
    ivLastKeyEventWasAlt = anEvent.isAltDown();
  }

  protected synchronized void processInputMethodEvent(
```

```
    InputMethodEvent e) {
     if (ivLastKeyEventWasAlt) {
       e.consume();
     }
     super.processInputMethodEvent(e);
   }
   private transient boolean ivLastKeyEventWasAlt = false;
}
```

Note that this workaround was orginally posted to the JDC Bug Parade and it should only be considered a temporary solution until the problem is actually fixed.

The "Open" menu item brings up our `m_chooser` `JFileChooser` component (discussed in chapter 14) to allow selection of a text file to open. Once a text file is selected, we open a `FileReader` on it and invoke `read()` on our `JTextArea` component to read the file's content (which creates a new `PlainDocument` containing the selected file's content to replace the current `JTextArea` document--see chapter 11). The "Save" menu item brings up `m_chooser` to select a destination and file name to save the current text to. Once a text file is selected, we open a `FileWriter` on it and invoke `write()` on our `JTextArea` component to write its content to the destination file. Both of these I/O operations are wrapped in separate threads to avoid clogging up the event dispatching thread.

Bug Alert! We have added code to repaint the whole application before the I/O operations occur. This is because when a Swing menu is hidden by a dialog (in this case a `JFileChooser` dialog) that pops up immediately in response to a menu selection, the menu will often not perform all necessary repainting of itself and components that lie below its popup. To make sure that no remnants of menu rendering are left around we force a repaint of the whole frame before moving on.

The "Exit" menu item terminates program execution. It is separated from the first three menu items with a menu separator to create a more logical display.

The "Font" menu consists of several menu items used to select the font and font style used in our editor. All of these items receive the same `ActionListener` which invokes our `updateMonitor()` method (see below). To give the user an idea of how each font looks, each font is used to render the corresponding menu item text. Since only one font can be selected at any given time, we use `JRadioButtonMenuItems` for these menu items, and add them all to a `ButtonGroup` instance which manages a single selection.

To create each menu item we iterate through our `FONTS` array and create a `JRadioButtonMenuItem` corresponding to each entry. Each item is set to unselected (except for the first one), assigned a numerical mnemonic corresponding to the current `FONTS` array index, assigned the appropriate `Font` instance for rendering its text, assigned our multi-purpose `ActionListener`, and added to our `ButtonGroup` along with the others.

The two other menu items in the "Font" menu manage the bold and italic font properties. They are implemented as `JCheckBoxMenuItems` since these properties can be selected or unselected independently. These items also are assigned the same `ActionListener` as the radio button items to process changes in their selected state.

The `updateMonitor()` method updates the current font used to render the editing component by checking the state of each check box item and determining which radio button item is currently selected. The `m_bold`

and m_italic components are disabled and unselected if the Courier font is selected, and enabled otherwise. The appropriate m_fonts array element is selected and a Font instance is derived from it corresponding to the current state of the check box items using Font's deriveFont() method (see chapter 2).

Note: Surprisingly the ButtonGroup class does not provide a direct way to determine which component is currently selected. So we have to examine m_fontMenus array elements in turn to determine the selected font index. Alternatively we could save the font index in an enhanced version of our ActionListener.

Running the Code

Open a text file, make some changes, and save it as a new file. Change the font options and note how the text area is updated. Select the Courier font and note how it disables the bold and italic check box items (it also unchecks them if they were previously checked). Select another font and note how this re-enables check box items. Figure 12.1 shows BasicTextEditor's "File" menu, and figure 12.2 shows the "Font" menu. Note how the mnemonics are underlined and the images appear to the right of the text by default (just like buttons).

UI Guideline: File oriented applications

This is an example of a menu being used on a file oriented application. Menus were first developed to be used in this fashion. The inclusion of a menu on such an application is essential, as users have come to expect one. There are clearly defined platform standards for the layout of such a menu and it is best that you ahere to these. Almost always the "File" menu comes first (to the LHS).

Note also the use of the elipsis "..." on the "Open..." and "Save..." options. This is a standard technique which gives a visual affordance to the behaviour that a dialog will open when the menu item is selected.

UI Guideline: Correct use of separator and widget overloading This example shows clearly how a menu in a simple application can have selection controls added to speed operation and ease usability. The separator is used to group and separate the selection of the font type from the font style.

12.3 Basic text editor: part II - toolbars and actions

Swing provides the Action interface to simplify the creation of menu items. As we know, implementations of this interface encapsulate both knowledge of what to do when a menu item or toolbar button is selected (by extending the ActionListener interface) and knowledge of how to render the component itself (by holding a collection of bound properties such as NAME, SMALL_ICON, etc.). We can create both a menu item and a toolbar button from a single Action instance, conserving code and providing a reliable means of ensuring consistency between menus and toolbars.

The following example uses the AbstractAction class to add a toolbar to our BasicTextEditor application. By converting the ActionListeners used in the example above to AbstractActions, we can use these actions to create both toolbar buttons and menu items with very little additional work.

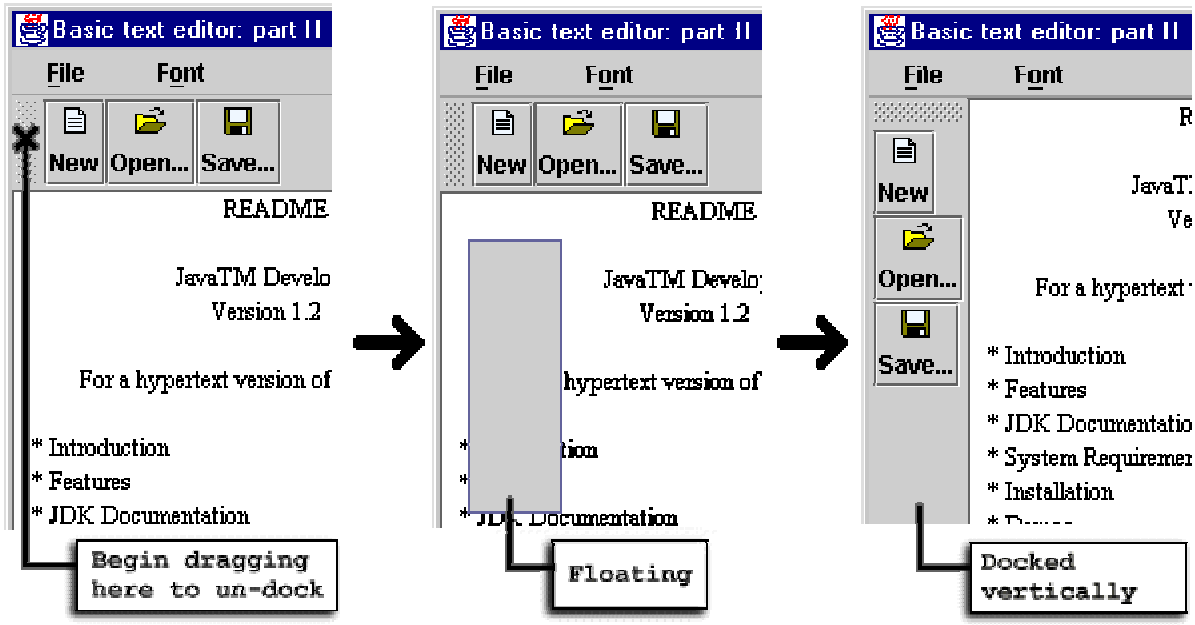


Figure 12.3 Process of un-docking, dragging, and docking a floating JToolBar.
<<file figure12-3.gif>>

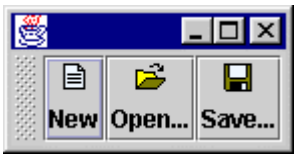


Figure 12.4 A floating JToolBar placed in a non-dockable region.
<<file figure12-4.gif>>

The Code: BasicTextEditor.java
see \Chapter12\2

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

import javax.swing.*;
import javax.swing.event.*;

public class BasicTextEditor extends JFrame
{
  // Unchanged code from section 12.2

  protected JToolBar m_toolBar;

  protected JMenuBar createMenuBar() {
    final JMenuBar menuBar = new JMenuBar();
    JMenu mFile = new JMenu("File");
    mFile.setMnemonic('f');

    ImageIcon iconNew = new ImageIcon("file_new.gif");
    Action actionNew = new AbstractAction("New", iconNew) {
      public void actionPerformed(ActionEvent e) {
        m_monitor.setText("");
      }
    };
```

```
    JMenuItem item =  mFile.add(actionNew);
    item.setMnemonic('n');

    ImageIcon iconOpen = new ImageIcon("file_open.gif");
    Action actionOpen = new AbstractAction("Open...", iconOpen) {
      public void actionPerformed(ActionEvent e) {
        // Unchanged code from section 12.2
      }
    };

    item =  mFile.add(actionOpen);
    item.setMnemonic('o');

    ImageIcon iconSave = new ImageIcon("file_save.gif");
    Action actionSave = new AbstractAction("Save...", iconSave) {
      public void actionPerformed(ActionEvent e) {
        // Unchanged code from section 12.2
      }
    };

    item =  mFile.add(actionSave);
    item.setMnemonic('s');

    mFile.addSeparator();

    Action actionExit = new AbstractAction("Exit") {
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    };

    item =  mFile.add(actionExit);
    item.setMnemonic('x');
    menuBar.add(mFile);

    m_toolBar = new JToolBar();
    JButton btn1 = m_toolBar.add(actionNew);
    btn1.setToolTipText("New text");
    JButton btn2 = m_toolBar.add(actionOpen);
    btn2.setToolTipText("Open text file");
    JButton btn3 = m_toolBar.add(actionSave);
    btn3.setToolTipText("Save text file");

    // Unchanged code from section 12.2

    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    return menuBar;
  }

  // Unchanged code from section 12.2
}
```

Understanding the Code

Class BasicTextEditor

This class now declares one more instance variable, `JToolBar m_toolBar`. The constructor remains unchanged and is not listed here. The `createMenuBar()` method now creates `AbstractAction` instances instead of `ActionListeners`. These objects encapsulate the same action handling code we defined in the

last example, as well as the text and icon to display in associated menu items and toolbar buttons. This allows us to create `JMenuItems` using the `JMenu.add(Action a)` method, and `JButtons` using the `JToolBar.add(Action a)` method. These methods return instances that we can treat as any other button component and do things such as set the background color assign a different text alignment.

Our `JToolBar` component is placed in the `NORTH` region of our content pane, and we make sure to leave the `EAST`, `WEST`, and `SOUTH` regions empty allowing it to dock on all sides.

Running the Code

Verify that the toolbar buttons work as expected by opening and saving a text file. Try dragging the toolbar from its 'handle' and note how it is represented by an empty gray window as it is dragged. The border will change to a dark color when the window is in a location where it will dock if the mouse is released. If the border does not appear dark, releasing the mouse will result in the toolbar being placed in its own `JFrame`. Figure 12.3 illustrates the simple process of undocking, dragging, and docking our toolbar in a new location. Figure 12.4 shows our toolbar in its own `JFrame` when undocked and released outside of a dockable region (also referred to as a hotspot).

Note: The current `JToolBar` implementation does not easily allow the use of multiple floating toolbars as is common in many modern applications. We hope to see more of this functionality built into future versions of Swing.

UI Guideline: Vertical or Horizontal?

In some applications, you may prefer to leave the selection of a vertical or horizontal toolbar to the user. More often than not, you as designer can make that choice for them. Consider whether vertical or horizontal space is more valuable for what you need to display. If, for example, you are displaying Letter text then you probably need vertical space more than horizontal space. Usually in PC applications, vertical space is at a premium.

When vertical space is at a premium, place the toolbar vertically thus freeing up valuable vertical space. When horizontal space is at a premium, place the toolbar horizontally thus freeing up valuable horizontal space.

Almost never allow a floating toolbar, as it has a tendency to get lost under other windows. Floating toolbars are for advanced users who understand the full operation of the computer system. Consider the technical level of your user group before making the design choice for a floating toolbar.

12.4 Basic text editor: part III - custom toolbar components

Using `Actions` to create toolbar buttons is easy, but not always desirable if we wish to have complete control over our toolbar components. In this section we build off of `BasicTextEditor` and place a `JComboBox` in the toolbar allowing `Font` selection. We also use instances of our own custom buttons, `SmallButton` and `SmallToggleButton`, in the toolbar. Both of these button classes use different borders to signify different states. `SmallButton` uses a raised border when the mouse passes over it, no border when the mouse is not within its bounds, and a lowered border when a mouse press occurs. `SmallToggleButton` uses a raised border when unselected and a lowered border when selected.

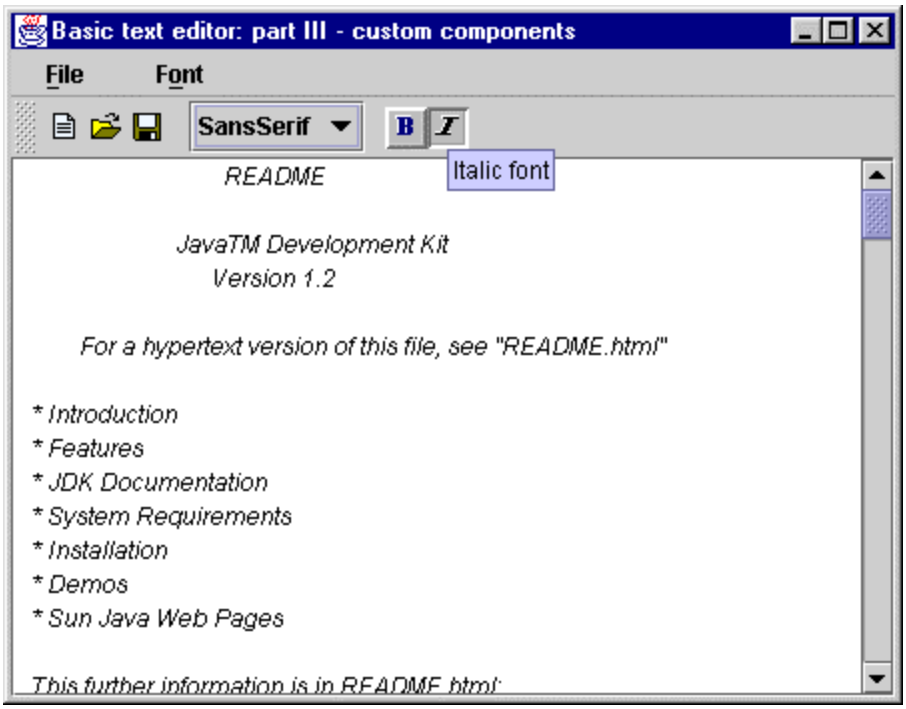


Figure 12.5 JToolBar with custom buttons and a JComboBox.

<<file figure12-5.gif>>

The Code: BasicTextEditor.java

see \Chapter12\3

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

import javax.swing.*;
import javax.swing.event.*;

public class BasicTextEditor extends JFrame
{
  // Unchanged code from section 12.3

  protected JComboBox m_cbFonts;
  protected SmallToggleButton m_bBold;
  protected SmallToggleButton m_bItalic;

  // Unchanged code from section 12.3

  protected JMenuBar createMenuBar()
  {
    // Unchanged code from section 12.3

    m_toolBar = new JToolBar();
    JButton bNew = new SmallButton(actionNew,
      "New text");
    m_toolBar.add(bNew);

    JButton bOpen = new SmallButton(actionOpen,
      "Open text file");
    m_toolBar.add(bOpen);
```

```
    JButton bSave = new SmallButton(actionSave,
      "Save text file");
    m_toolBar.add(bSave);

    JMenu mFont = new JMenu("Font");
    mFont.setMnemonic('o');

    // Unchanged code from section 12.3

    mFont.addSeparator();

    m_toolBar.addSeparator();
    m_cbFonts = new JComboBox(FONTS);
    m_cbFonts.setMaximumSize(m_cbFonts.getPreferredSize());
    m_cbFonts.setToolTipText("Available fonts");
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int index = m_cbFonts.getSelectedIndex();
        if (index < 0)
          return;
        m_fontMenus[index].setSelected(true);
        updateMonitor();
      }
    };
    m_cbFonts.addActionListener(lst);
    m_toolBar.add(m_cbFonts);

    m_bold = new JCheckBoxMenuItem("Bold");
    m_bold.setMnemonic('b');
    Font fn = m_fonts[1].deriveFont(Font.BOLD);
    m_bold.setFont(fn);
    m_bold.setSelected(false);
    m_bold.addActionListener(fontListener);
    mFont.add(m_bold);

    m_italic = new JCheckBoxMenuItem("Italic");
    m_italic.setMnemonic('i');
    fn = m_fonts[1].deriveFont(Font.ITALIC);
    m_italic.setFont(fn);
    m_italic.setSelected(false);
    m_italic.addActionListener(fontListener);
    mFont.add(m_italic);

    menuBar.add(mFont);

    m_toolBar.addSeparator();

    ImageIcon img1 = new ImageIcon("font_bold1.gif");
    ImageIcon img2 = new ImageIcon("font_bold2.gif");
    m_bBold = new SmallToggleButton(false, img1, img2,
      "Bold font");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_bold.setSelected(m_bBold.isSelected());
        updateMonitor();
      }
    };
    m_bBold.addActionListener(lst);
    m_toolBar.add(m_bBold);

    img1 = new ImageIcon("font_italic1.gif");
    img2 = new ImageIcon("font_italic2.gif");
```

```
    m_bItalic = new SmallToggleButton(false, img1, img2,
      "Italic font");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_italic.setSelected(m_bItalic.isSelected());
        updateMonitor();
      }
    };
    m_bItalic.addActionListener(lst);
    m_toolBar.add(m_bItalic);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);
    return menuBar;
  }

  protected void updateMonitor() {
    int index = -1;
    for (int k=0; k<m_fontMenus.length; k++) {
      if (m_fontMenus[k].isSelected()) {
        index = k;
        break;
      }
    }
    if (index == -1)
      return;
    boolean isBold = m_bold.isSelected();
    boolean isItalic = m_italic.isSelected();

    m_cbFonts.setSelectedIndex(index);

    if (index==2) {   //Courier
      m_bold.setSelected(false);
      m_bold.setEnabled(false);
      m_italic.setSelected(false);
      m_italic.setEnabled(false);
      m_bBold.setSelected(false);
      m_bBold.setEnabled(false);
      m_bItalic.setSelected(false);
      m_bItalic.setEnabled(false);
    }
    else {
      m_bold.setEnabled(true);
      m_italic.setEnabled(true);
      m_bBold.setEnabled(true);
      m_bItalic.setEnabled(true);
    }

    if (m_bBold.isSelected() != isBold)
      m_bBold.setSelected(isBold);
    if (m_bItalic.isSelected() != isItalic)
      m_bItalic.setSelected(isItalic);

    int style = Font.PLAIN;
    if (isBold)
      style |= Font.BOLD;
    if (isItalic)
      style |= Font.ITALIC;
    Font fn = m_fonts[index].deriveFont(style);
    m_monitor.setFont(fn);
    m_monitor.repaint();
  }
```

```
  public static void main(String argv[]) {
    new BasicTextEditor();
  }
}

class SmallButton extends JButton implements MouseListener
{
  protected Border m_raised;
  protected Border m_lowered;
  protected Border m_inactive;

  public SmallButton(Action act, String tip) {
    super((Icon)act.getValue(Action.SMALL_ICON));
    m_raised = new BevelBorder(BevelBorder.RAISED);
    m_lowered = new BevelBorder(BevelBorder.LOWERED);
    m_inactive = new EmptyBorder(2, 2, 2, 2);
    setBorder(m_inactive);
    setMargin(new Insets(1,1,1,1));
    setToolTipText(tip);
    addActionListener(act);
    addMouseListener(this);
    setRequestFocusEnabled(false);
  }

  public float getAlignmentY() { return 0.5f; }

  public void mousePressed(MouseEvent e) {
    setBorder(m_lowered);
  }
  public void mouseReleased(MouseEvent e) {
    setBorder(m_inactive);
  }
  public void mouseClicked(MouseEvent e) {}
  public void mouseEntered(MouseEvent e) {
    setBorder(m_raised);
  }
  public void mouseExited(MouseEvent e) {
    setBorder(m_inactive);
  }
}

class SmallToggleButton extends JToggleButton
  implements ItemListener
{
  protected Border m_raised;
  protected Border m_lowered;

  public SmallToggleButton(boolean selected,
   ImageIcon imgUnselected, ImageIcon imgSelected, String tip) {
    super(imgUnselected, selected);
    setHorizontalAlignment(CENTER);
    setBorderPainted(true);
    m_raised = new BevelBorder(BevelBorder.RAISED);
    m_lowered = new BevelBorder(BevelBorder.LOWERED);
    setBorder(selected ? m_lowered : m_raised);
    setMargin(new Insets(1,1,1,1));
    setToolTipText(tip);
    setRequestFocusEnabled(false);
    setSelectedIcon(imgSelected);
    addItemListener(this);
  }
```

```
  public float getAlignmentY() { return 0.5f; }

  public void itemStateChanged(ItemEvent e) {
    setBorder(isSelected() ? m_lowered : m_raised);
  }
}
```

Understanding the Code

Class BasicTextEditor

`BasicTextEditor` now declares three new instance variables:

- `JComboBox m_cbFonts`: combo box containing available font names.
- `SmallToggleButton m_bBold`: custom toggle button representing the bold font style.
- `SmallToggleButton m_bItalic`: custom toggle button representing the italic font style.

The `createMenuBar()` method now creates three instances of the `SmallButton` class (see below) corresponding to our pre-existing "New," "Open," and "Save" toolbar buttons. These are constructed by passing the appropriate `Action` (which we built in part II) as well as a tooltip `String` to the `SmallButton` constructor. Then we create a combo box with all available font names and add it to the toolbar. The `setMaximumSize()` method is called on the combo box to reduce its size to a necessary maximum (otherwise it will fill all unoccupied space in our toolbar). An `ActionListener` is then added to monitor combo box selection. This listener selects the corresponding font menu item because the combo box and font radio button menu items must always be in synch. It then calls our `updateMonitor()` method.

Two `SmallToggleButtons` are created and added to our toolbar to manage the bold and italic font properties. Each button receives an `ActionListener` which selects/deselects the corresponding menu item (because both the menu items and toolbar buttons must remain in synch) and calls our `updateMonitor()` method.

Our `updateMonitor()` method receives some additional code to provide consistency between our menu items and toolbar controls. This method relies on the state of the menu items, which is why the toolbar components first set the corresponding menu items when selected. The code added here is self-explanatory and just involves enabling/disabling and selecting/deselecting components to preserve consistency.

Class SmallButton

`SmallButton` represents a small push button to be used in a toolbar. It implements the `MouseListener` interface to process mouse input. Three instance variables are declared:

- `Border m_raised`: border to be used when the mouse cursor is located over the button.
- `Border m_lowered`: border to be used when the button is pressed.
- `Border m_inactive`: border to be used when the mouse cursor is located outside the button.

The `SmallButton` constructor takes an `Action` parameter, which is added as an `ActionListener` and performs an appropriate action when the button is pressed, and a `String` representing the tooltip text. Several familiar properties are assigned and the icon encapsulated within the `Action` is used for this buttons icon. `SmallButton` also adds itself as a `MouseListener` and sets its tooltip text to the given `String` passed to the constructor. Note that the `requestFocusEnabled` property is set to `false` so that when this button is clicked focus will not be transferred out of our `JTextArea` editor component.

The `getAlignmentY()` method is overriden to return a constant value of `0.5f`, indicating that this button should always be placed in the middle of the toolbar in the vertical direction. The remainder of `SmallButton` represents an implementation of the `MouseListener` interface which sets the border based

on mouse events. The border is set to `m_inactive` when the is mouse located outside its bounds, `m_active` when the mouse is located inside its bounds, and `m_lowered` when the button is pressed.

Class SmallToggleButton

`SmallToggleButton` extends `JToggleButton` and implements the `ItemListener` interface to process changes in the button's selection state. Two instance variables are declared:

`Border m_raised`: border to be used when the button is unselected (unchecked).

`Border m_lowered`: border to be used when the button is selected (checked).

The `SmallToggleButton` constructor takes four arguments:

`boolean selected`: initial selection state.

`ImageIcon imgUnselected`: icon for use when unselected.

`ImageIcon imgSelected`: icon for use when selected.

`String tip`: tooltip message.

In the constructor, several familiar button properties are set, and a raised or lowered border is assigned depending on the initial selection state. Each instance is added to itself as an `ItemListener` to receive notification about changes in its selection. Thus the `itemStateChanged()` method is implemented which simply sets the button's border according to the corresponding to the new selected state.

Running the Code

Verify that the toolbar components (combobox and toggle buttons) change the editor's font as expected. Check which menu and toolbar components work consistently (menu item selections result in changes in the toolbar controls, and vice versa).

UI Guideline: Tooltip Help Tooltip Help on mouse-over is a "must have" technical addition for small toolbar buttons. The relatively recent innovation of tooltips has greatly improved the usability of toolbars. Don't get caught delivering a toolbar without one, but make sure that your tooltip text is meaningful to the user!

12.5 Basic text editor: part IV - custom menu components

In this section we will show how to build a custom menu component, `ColorMenu`, which allows selection of a color from a grid of small colored panes (which are instances of the inner class `ColorMenu.ColorPane`). By extending `JMenu` we inherit all `MenuElement` functionality (see 12.1.10), making custom menu creation quite easy.

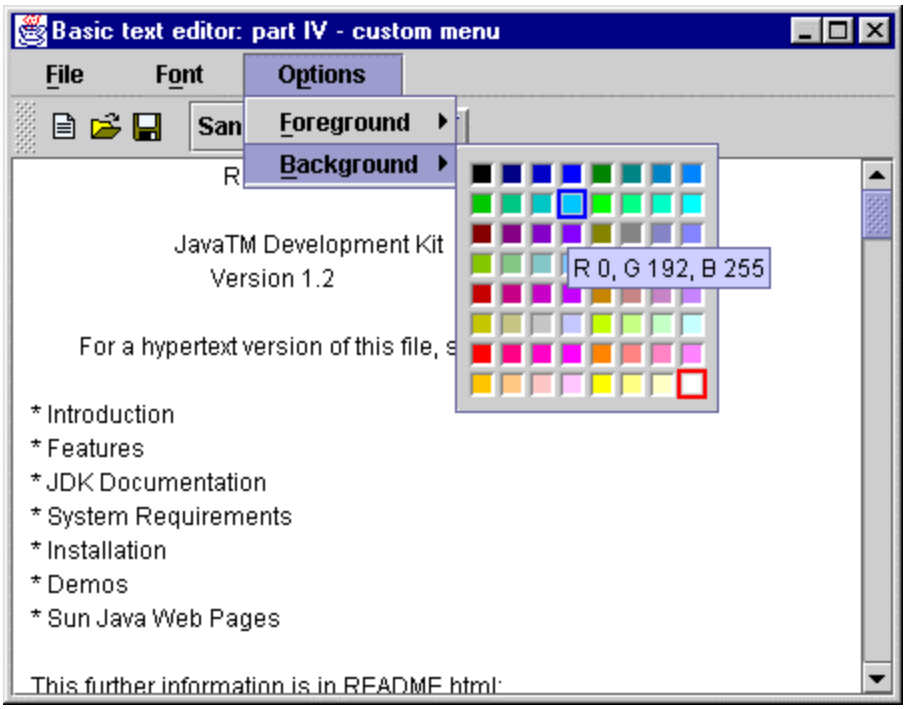


Figure 12.6 Custom menu component used for quick color selection.

<<file figure12-6.gif>>

The Code: BasicTextEditor.java

see \Chapter12\4

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

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class BasicTextEditor extends JFrame
{
  // Unchanged code from section 12.4

  protected JMenuBar createMenuBar()
  {
    // Unchanged code

    JMenu mOpt = new JMenu("Options");
    mOpt.setMnemonic('p');

    ColorMenu cm = new ColorMenu("Foreground");
    cm.setColor(m_monitor.getForeground());
    cm.setMnemonic('f');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        ColorMenu m = (ColorMenu)e.getSource();
        m_monitor.setForeground(m.getColor());
      }
    };
    cm.addActionListener(lst);
    mOpt.add(cm);
```

```
    cm = new ColorMenu("Background");
    cm.setColor(m_monitor.getBackground());
    cm.setMnemonic('b');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        ColorMenu m = (ColorMenu)e.getSource();
        m_monitor.setBackground(m.getColor());
      }
    };
    cm.addActionListener(lst);
    mOpt.add(cm);
    menuBar.add(mOpt);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);
    return menuBar;
  }

  // Unchanged code
}

class ColorMenu extends JMenu
{
  protected Border m_unselectedBorder;
  protected Border m_selectedBorder;
  protected Border m_activeBorder;

  protected Hashtable m_panes;
  protected ColorPane m_selected;

  public ColorMenu(String name) {
    super(name);
    m_unselectedBorder = new CompoundBorder(
      new MatteBorder(1, 1, 1, 1, getBackground()),
      new BevelBorder(BevelBorder.LOWERED,
      Color.white, Color.gray));
    m_selectedBorder = new CompoundBorder(
      new MatteBorder(2, 2, 2, 2, Color.red),
      new MatteBorder(1, 1, 1, 1, getBackground()));
    m_activeBorder = new CompoundBorder(
      new MatteBorder(2, 2, 2, 2, Color.blue),
      new MatteBorder(1, 1, 1, 1, getBackground()));

    JPanel p = new JPanel();
    p.setBorder(new EmptyBorder(5, 5, 5, 5));
    p.setLayout(new GridLayout(8, 8));
    m_panes = new Hashtable();

    int[] values = new int[] { 0, 128, 192, 255 };
    for (int r=0; r<values.length; r++) {
      for (int g=0; g<values.length; g++) {
        for (int b=0; b<values.length; b++) {
          Color c = new Color(values[r], values[g], values[b]);
          ColorPane pn = new ColorPane(c);
          p.add(pn);
          m_panes.put(c, pn);
        }
      }
    }
    add(p);
  }
```

```
public void setColor(Color c) {
  Object obj = m_panes.get(c);
  if (obj == null)
    return;
  if (m_selected != null)
    m_selected.setSelected(false);
  m_selected = (ColorPane)obj;
  m_selected.setSelected(true);
}

public Color getColor() {
  if (m_selected == null)
    return null;
  return m_selected.getColor();
}

public void doSelection() {
  fireActionPerformed(new ActionEvent(this,
    ActionEvent.ACTION_PERFORMED, getActionCommand()));
}

class ColorPane extends JPanel implements MouseListener
{
  protected Color m_c;
  protected boolean m_selected;

  public ColorPane(Color c) {
    m_c = c;
    setBackground(c);
    setBorder(m_unselectedBorder);
    String msg = "R "+c.getRed()+", G "+c.getGreen()+
      ", B "+c.getBlue();
    setToolTipText(msg);
    addMouseListener(this);
  }

  public Color getColor() { return m_c; }

  public Dimension getPreferredSize() {
    return new Dimension(15, 15);
  }
  public Dimension getMaximumSize() { return getPreferredSize(); }
  public Dimension getMinimumSize() { return getPreferredSize(); }

  public void setSelected(boolean selected) {
    m_selected = selected;
    if (m_selected)
      setBorder(m_selectedBorder);
    else
      setBorder(m_unselectedBorder);
  }

  public boolean isSelected() { return m_selected; }

  public void mousePressed(MouseEvent e) {}

  public void mouseClicked(MouseEvent e) {}

  public void mouseReleased(MouseEvent e) {
    setColor(m_c);
    MenuSelectionManager.defaultManager().clearSelectedPath();
    doSelection();
```

```
    }

    public void mouseEntered(MouseEvent e) {
      setBorder(m_activeBorder);
    }

    public void mouseExited(MouseEvent e) {
      setBorder(m_selected ? m_selectedBorder :
        m_unselectedBorder);
    }
  }
}
```

Understanding the Code

Class BasicTextEditor

The `createMenuBar()` method now creates a new `JMenu` titled "Options" and populates it with two `ColorMenus`. The first of these menus receives an `ActionListener` which requests the selected color, using `ColorMenu`'s `getColor()` method, and assigns it as the foreground color of our editor component. Similarly, the second `ColorMenu` receives an `ActionListener` which manages our editor's background color.

Class ColorMenu

This class extends `JMenu` and represents a custom menu component which serves as a quick color chooser. Instance variables:

- `Border m_unselectedBorder`: border to be used for a `ColorPane` (see below) when it is not selected and the mouse cursor is located outside of its bounds.
- `Border m_selectedBorder`: border to be used for a `ColorPane` when it is selected and the mouse cursor is located outside of its bounds.
- `Border m_activeBorder`: border to be used for a `ColorPane` when the mouse cursor is located inside its bounds.
- `Hashtable m_panes`: a collection of `ColorPanes`.
- `ColorPane m_selected`: a reference to the currently selected `ColorPane`.

The `ColorMenu` constructor takes a menu name as parameter and creates the underlying `JMenu` component using that name. This creates a root menu item which can be added to another menu or to a menu bar. Selecting this menu item will display its `JPopupMenu` component, which normally contains several simple menu items. In our case, however, we add a `JPanel` to it using `JMenu`'s `add(Component c)` method. This `JPanel` serves as a container for 64 `ColorPanes` (see below) which are used to display the available selectable colors, as well as the current selection. A triple `for` cycle is used to generate the constituent `ColorPanes` in 3-dimensional color space. Each `ColorPane` takes a `Color` instance as constructor parameter, and each `ColorPane` is placed in our `Hashtable` collection, `m_panes`, using its associated `Color` as the key.

The `setColor()` method finds a `ColorPane` which holds a given `Color`. If such a component is found this method clears the previously selected `ColorPane` and selects a new one by calling its `setSelected()` method. The `getColor()` method simply returns the currently selected color.

The `doSelection()` method sends an `ActionEvent` to registered listeners notifying them that an action has been performed on this `ColorPane`, which means a new color may have been selected.

Class ColorMenu.ColorPane

This inner class is used to display a single color available for selection in a `ColorMenu`. It extends `JPanel` and implements `MouseListener` to process its own mouse events. This class uses the three `Border` variables from the parent `ColorMenu` class to represent its state, whether selected, unselected, or active. Instance variables:

`Color m_c`: color instance represented by this pane.

`boolean m_selected`: a flag indicating whether or not this pane is currently selected.

The `ColorPane` constructor takes a `Color` instance as parameter and stores it in our `m_c` instance variable. The only thing we need to do to display that color is set it as the pane's background. We also add a tooltip indicating the red, green, and blue components of this color.

All MouseListener related methods should be familiar by now. However, take note of the `mouseReleased()` method which plays the key role in color selection: If the mouse is released over a `ColorPane` we first assign the associated `Color` to the parenting `ColorMenu` component using the `setColor()` method (so it later can be retrieved by any attached listeners). We then hide all opened menu components by calling the `MenuSelectionManager.clearSelectedPath()` method since menu selection is completed at this point. Finally we invoke the `doSelection()` method on the parenting `ColorMenu` component to notify all attached listeners.

Running the Code

Experiment with changing the editor's background and foreground colors using our custom menu component available in the "Options" menu.. Note that a color selection will not affect anything until the mouse is released, and a mouse release also triggers the collapse of all menu popups in the current path. Figure 12.6 shows `ColorMenu` in action.

UI Guideline: Usability and Design alternatives

A more traditional approach to this exampe would be to have an elipsis on the menu option and open a `Color Chooser Dialog`. Consider what an improvement the presented design makes to usability. Within a limited range of colours, this design allows faster selection with the possible minor con that there is more chance of a mistake being made in the selection. However, such a mistake is low cost as it can easily be corrected. As you will see in the next chapter, knowing that you have a bounded range of input selections can be put to good use when improving a design and usability.

UI references:
Human Factors International at http://www.humanfactors.com
A Test to give you Fitt's at http://www.asktog.com

Chapter 13. Progress Bars, Sliders, and ScrollBars

In this chapter:

- Bounded-range components overview
- Basic JScrollBar example
- JSlider date chooser
- JSliders in a JPEG image editor
- JProgressBar in an FTP client application

13.1 Bounded-range components overview

`JScrollBar`, `JSlider`, and `JProgressBar` provide visualization and selection within a bounded interval, allowing the user to conveniently select a value from that interval, or simply observe its current state. In this section we'll give a brief overview of these components and the significant classes and interfaces that support them.

13.1.1 The BoundedRangeModel interface

abstract interface javax.swing.BoundedRangeModel

The `BoundedRangeModel` interface describes a data model used to define an integer value between `minimum` and `maximum` values. This value can have a subrange called an extent, which can be used to define the size of, for instance, a scrollbar "thumb." Often the extent changes dynamically corresponding to how much of the entire range of possible values is visible. Note that the value can never be set larger than the `maximum` or `minimum` values, and the extent always starts at the current value and never extends past the `maximum`. Another property called `valueIsAdjusting` declared and expected to be true when the value is in the state of being adjusted.

Implementations are expected to fire `ChangeEvents` when any of the `minimim`, `maximum`, `value`, `extent`, or `valueIsAdjusting` properties change state. Thus, `BoundedRangeModel` includes method decarations for adding and removing `ChangeListeners`: `addChangeListener()` and `removeChangeListener()`. This model is used by `JProgressBar`, `JSlider`, and `JScrollBar`.

`UIGuideline:` Why choose a bounded range component

The bounded range components are essentially analog devices in nature. They are good at providing relative, positional, approximate or changing (in time) data. They are also excellent at visually communicating the bounds or limits of a data selection and at communicating percentage of the whole through approximate visual means. Where you have several values which share the same bounds e.g. RGB values for a ColorChooser, then you can easily communicate relative values of the three choices through use of a bounded range component. The position of each component shows the relative value of one against the other.

So use bounded range components when there is advantage in communicating the range of values and an approximate, relative, position or changing value needs to be communicated to the User.

13.1.2 DefaultBoundedRangeModel

class javax.swing.DefaultBoundedRangeModel

`DefaultBoundedRangeModel` is the default concrete implementation of the `BoundedRangeModel` interface. The default constructor initializes a model with `0` for `minimum`, `100` for `maximum`, and `0` for the value and extent properties. Another constructor allows specification of each of these initial values as `int` parameters. As expected, this implementation does fire `ChangeEvents` whenever one of its properties changes.

13.1.3 JScrollBar

class javax.swing.JScrollBar

Scrollbars can be used to choose a new value from a specified interval by sliding a knob (often referred to as the thumb) between given `maximum` and `minimum` bounds, or by using small buttons at the ends of the component. The area not occupied by the thumb and buttons is known as the paging area, and this can also be used to change the current scrollbar value. The thumb represents the extent of this bounded-range component, and its value is stored in the `visibleAmount` property.

`JScrollBar` can be oriented horizontally or vertically, and its value increases to the right or upward respectively. To specify orientation, stored in the `orientation` property, we call the `setOrientation()` method and pass it one of the `JScrollBar.HORIZONTAL` or `JScrollBar.VERTICAL` constants.

Clicking on a button moves the thumb (and thus the value--recall that a bounded-range component's value lies at the beginning of the extent) by the value of `JScrollBar's` `unitIncrement` property. Similarly, clicking the paging area moves the thumb by the value of `JScrollBar's` `blockIncrement` property.

Note: It is common to match the `visibleAmount` property with the `blockIncrement` property. This is a simple way to visually signify to the user how much of the available range of data is currently visible.

UI Guideline: Usage of a Scrollbar Background

The Scrollbar is really a computer enhanced development from an original analog mechanical idea. Scrollbars are in some respects more advanced than Sliders (see section 13.1.4). The Thumb of the Scrollbar can very cleverly be used to show the current data as a percentage of a whole (as described in the Note above). If the scrollbar is placed onto an image and the thumb is approximately 50% of the total size, then the User is given a clear indication that the viewing area is roughly half of the total. The ability for the Thumb in a scrollbar to change size to accurately reflect this, is something which could not have been achieved with a mechanical device. Scrollbar is in this respect a very good example of taking a metaphor based on a mechanical device and enhancing it to improve usability.

Choosing Position

By far the best use of a scrollbar is position selection. They are in nature analog and to the viewer are giving only an approximate position. Position selection is how they are used throughout Swing inside a JScrollPane. Users have become used to this usage and its very natural. For most other occasions where you wish to use a sliding control for selection, a Slider is probably best.

As expected, `JScrollBar` uses a `DefaultBoundedRangeModel` by default. In addition to the `ChangeEvents` fired by this model, `JScrollPane` fires `PropertyChangeEvents` when its `orientation`, `unitIncrement`, or `blockIncrement` properties change state. `JScrollPane` also fires

AdjustmentEvents whenever any of its bound properties change, or when any of it's model's properties change (this is done solely for backward compatibility with the AWT scrollbar class). Accordingly, JScrollPane provides methods to add and remove AdjustmentListeners (we don't need to provide methods for adding and removing PropertyChangeListeners because this functionality is inherited from JComponent).

Note: AdjustmentListeners receive AdjustmentEvents. Both are defined in java.awt.event--see APIdocs.

13.1.4 JSlider

class javax.swing.JSlider

Sliders can be used to choose a desirable numerical value from a specified interval by sliding a knob between given borders (using the mouse, arrow keys, or PageDown and PageUp). Sliders are very useful when we know in advance the range of input the user should be able to choose from.

JSlider supports horizontal and vertical orientations, and its orientation property can be set to one of JSlider.HORIZONTAL or JSlider.VERTICAL. The extent property specifies the number of values to skip forward/up or back/down when PageUp or PageDown is pressed, respectively. Tick marks can be used to denote value locations. Minor and major tick marks are supported, where major ticks are usually longer and spread farther apart than minor ticks. In the case where a major and minor tick fall on the same location, the major tick takes precedence and the minor tick will not be displayed. Spacing between minor tick marks is specified by the minorTickSpacing property, and spacing between major tick marks is specified by the majorTickSpacing property.

Note: The tick spacing properties specify the number of values to be skipped between each successive tick. Their names are somewhat misleading because they actually have nothing to do with the physical space (in pixels) between each tick. They would be more appropriately named "minorTickDisplayInterval" and "majorTickDisplayInterval."

Setting either spacing property to 0 has a disabling effect, and the paintTicks property also provides a way of turning ticks on and off.

Note: The snapToTicks property is intended to only allow the slider knob to lie on a tick-marked value, however, this feature does not work as expected as of Java 2 FCS.

Major ticks can be annotated by components, and by default each of JSlider's major ticks are adorned with JLabel's denoting the integer tick value. We can turn on/off this functionality by changing the paintLabels property, and we can customize which components are used to annotate, and at what values they are placed, by passing a Dictionary of Integer/Component pairs to the setLabelTable() method. The createStandardLabels() method is used by default to setup JSlider with its JLabel's at each major tick value. This method returns a Hashtable (sub-class of Dictionary) which can then be assigned to JSlider using setLabelTable().

By default JSlider's values increment from left-to-right or bottom-to-top depending on whether horizontal or vertical orientation is used respectively. To reverse the direction of incrementation we can set the inverted property to true.

UIGuideline: Usage of a Slider

By origin a slider is an analog device. Sliders are really a close graphical and behavioural representation of a real world analog slider from, for example, a hi-fi system or an older TV volume control. As such sliders are analog devices and are designed for use in determining an approximate or positional setting for something. Usually they rely on direct user feedback to help with the choice of position. With the TV volume control example, the volume would go up and down as the slider is moved and the User would stop moving it when the volume was at a comfortable level.

The Swing version of a slider is actually a digital device disguised as an analog one. Each tick of the slider is a digital increment. The slider can therefore be used to determine an accurate value providing some additional digital feedback is given for the User such as a numeric display of the absolute value or a scale along the side of the slider. Where accurate values are important, such as with a colour chooser, be sure to provide an absolute value as output along side the slider.

Feedback
: Immediate feedback is important with sliders due to the analog nature of the device. Provide actual feedback such as the brightness of a picture which increases or decreases as the slider is moved or provide an absolute numeric value readout which can be observed to change as the slider is moved. Therefore, judicious use of the Change Event with a ChangeListener is important so that the feedback mechanism can be updated e.g. brightness or contrast in an image.

Movement
: The two default orientations of Slider are conventions which date back to original analog electronic devices. When vertical, the down position is lower and you move it up to increase in value. When horizontal, left is lower and you move it right to increase in value. Users are likely to be very familiar with this convention. If you wish to switch it, then you should have a very very good reason for doing. We wouldn't recommend it!

Slider vs. Scrollbar

On the whole, use a Slider for choosing a value when the value needed is approximate and subjective such as color, volume, brightness and requires User feedback to make the subjective judgement. Use a scrollbar for positional choice, where the desired position is again approximate and judged relative to the whole.

The `paintTrack` property specifies whether the whole slider track is filled in or not. The MetalL&F UI delegate for `JSlider` pays attention to client property with key "`JSlider.isFilled`" and `Boolean` value. Adding this property to a `JSlider`'s client properties hashtable (using `putClientProperty()`--see chapter 2) with a value of `Boolean.TRUE`, will fill in only the lower half of the slider track from the position of the knob. Note that this client property will have no effect if the `paintTrack` property is set to `true`, and will only work if the slider is using the MetalL&F UI delegate.

As expected, `JSlider` uses a `DefaultBoundedRangeModel` by default. In addition to the `ChangeEvents` fired by this model, `JSlider` fires `PropertyChangeEvents` when any of its properties described above change state. Unlike `JScrollBar`, this class provides the ability to add and remove `ChangeListeners` directly, vs. `AdjustmentListeners`.

13.1.5 JProgressBar

class javax.swing.JProgressBar

Progress bars can be used to display how far or close a given numerical value is from the bounds of a specified interval. They are typically used to indicate progress during a certain lengthy job to provide simple feedback to the user showing that the job being monitored is alive and active. As with `JScrollBar` and `JSlider`, `JProgressBar` can be oriented horizontally or vertically. Note also that `JProgressBar` acts identically to `JSlider` with respect to incrementing: left-to-right in horizontal orientation, bottom-to-top in vertical orientation.

A JProgressBar is painted filled from the minimum value to its current value (with the exception of the Windows L&F, which paints a series of small rectangles). A percentage representing how much of a job has been completed can optionally by displayed in the center of JProgressBar. The string property represents the String to be painted (usually of the form XX%, where X is a digit), stringPainted specifies whether or not string should be painted, and percentComplete is a double between 0 and 1 specifying how much of the job has been completed so far.

Note: We normally do not need to take control of this rendering functionality, because by setting the string property to null, and the stringPainted property to true, the percentComplete property is convereted to the XX% form for us, and displayed in the progress bar.

JProgressBar's foreground and background can be assigned as any JComponent, however, the color used to render its status text is not directly modifiable. Instead this is handled by the UI delegate, and the easiest way to assign specific colors is to replace the appropriate UI resources in UIManager's defaults table (see chapter 21 for more about L&F customization).

The borderPainted property (defaults to true) specifies whether or not a border is rendered around JProgressBar. As expected, JProgressBar uses a DefaultBoundedRangeModel by default, and ChangeListeners can be added to receive ChangeEvents when any of JProgressBar's properties change state.

During a monitored operation we simply call setValue() on a JProgressBar and all updating is taken care of for us. Note that we must be careful to make this call in the event dispatching thread. Consider the following basic example, figure 13.1 illustrates:

Figure 13.1 A basic JProgressBar example showing custom colors and proper updating.
<<file figure13-1.gif>>

The Code: JProgressBarDemo.java
see \Chapter13\1

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

import javax.swing.*;
import javax.swing.event.*;

public class JProgressBarDemo
  extends JFrame
{
  protected int m_min = 0;
  protected int m_max = 100;
  protected int m_counter = 0;
  protected JProgressBar jpb;

  public JProgressBarDemo()
  {
    super("JProgressBar Demo");
    setSize(300,50);

    UIManager.put("ProgressBar.selectionBackground", Color.black);
    UIManager.put("ProgressBar.selectionForeground", Color.white);
    UIManager.put("ProgressBar.foreground", new Color(8,32,128));
```

```
    jpb = new JProgressBar();
    jpb.setMinimum(m_min);
    jpb.setMaximum(m_max);
    jpb.setStringPainted(true);

    JButton start = new JButton("Start");
    start.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Thread runner = new Thread() {
          public void run() {
            m_counter = m_min;
            while (m_counter <= m_max) {
              Runnable runme = new Runnable() {
                public void run() {
                  jpb.setValue(m_counter);
                }
              };
              SwingUtilities.invokeLater(runme);
              m_counter++;
              try {
                Thread.sleep(100);
              }
              catch (Exception ex) {}
            }
          }
        };
        runner.start();
      }
    });

    getContentPane().add(jpb, BorderLayout.CENTER);
    getContentPane().add(start, BorderLayout.WEST);

    WindowListener wndCloser = new WindowAdapter()
    {
      public void windowClosing(WindowEvent e)
      {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);
    setVisible(true);
  }

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

Note: The `JProgressBar` UI delegate centers the progress text horizontally and vertically. However, its centering scheme enforces a certain amount of white space around the text and has undesirable effects when using thin progress bars. In order to fix this we can override `BasicProgressBarUI`'s `getStringPlacement()` method (see API docs and BasicProgressBarUI.java source) to return the desired `Point` location where the text should be rendered.

UI Guideline: Usage of Progress Bar Long Operations

Progress Bars are commonly used as a fill in for operations which will take a long time. A long time in human

interaction is often defined as around 1 second or longer. The progress bar is usually rendered inside a JOptionPane.

Special attention will need to be paid to the business logic code so that it is capable of notifying a Progress Bar of the progress of an operation.

Progress Bars are inherintly analog in nature. Analog data is particularly good for displaying change and for relative comparison. It is not good for exact measurement. The analog nature of a Progress Bar means that it is good for showing that something is happening and that progress is taking place. However, it is not good for giving an exact measure of completeness. If your problem domain requires the ability to measure exactly how complete a task is then you may to to supplement the progress bar with a digital reading of progress. This is common with Internet download dialogs and option panes.

A digital readout is particularly useful when the task to be completed will take a very long time. The progress bar may only be giving you a granularity of around 3% for each graphic. If it takes significantly long to progress by such a jump, say greater than 5 seconds, then the digital readout will give you a finer grained reading at 1% and will change approximately 3 times faster than your progress bar. The combination of the two helps to pass the time for the viewer and gives them the reassurance that something is happening whilst also giving them a very accurate view of progress. That is why the dual combination of digital and analog progress is popular with Internet download dialogs, as the task can be very long and cannot be determined by the application developer.

13.1.6 ProgressMonitor

class javax.swing.ProgressMonitor

The `ProgressMonitor` class is a convenient and intelligent means of deploying a dynamic progress bar in an application that performs time-consuming operations. This class is a direct sub-class of `Object` (thus it does not exist in the component hierarchy).

`ProgressMonitor` displays a `JDialog` containing a `JOptionPane`-style component. The `note` property represents a `String` that can change during the course of an operation and is displayed in a `JLabel` above the `JProgressBar` (if `null` is used this label is not displayed).

Two buttons, "OK" and "Cancel," are placed at the bottom and serve to dismiss the dialog and abort the operation respectively. The "OK " button simply hides the dialog. The "Cancel" button also hides the dialog, and sets the `canceled` property `true`, providing us with a way to test whether the user has canceled the operation or not. Since most time-consuming operations occur in loops, we can test this property during each iteration and abort if necessary.

The `millisToDecideToPopup` property is an `int` value specifying the number of milliseconds to wait before `ProgressMonitor` should determine whether to pop up a dialog (defaults to 500). This is used to allow a certain amount of time to pass before questioning whether the job is long enough to warrant popping up a dialog. The `millisToPopup` property is an `int` value specifying the minimum time a job must take in order to warrant popping up a dialog (defaults to 2000). If `ProgressMonitor` determines that the job will take less than `millisToPopup` milliseconds the dialog will not be shown.

The `progress` property is an `int` value specifying the current value of the `JProgressBar`. During an operation we are expected to update the `note` and `progress` in the event-dispatching thread (as demonstrated in the example above).

Warning: In light of theses properties we should only use a `ProgressMonitor` for simple, predictable jobs. `ProgressMonitor` bases the estimated time to completion on the value of its `JProgressBar` from start time to current evaluation time and assumes that a constant rate of progression will exist throughout the whole job. For transfering a single file this may be a fairly valid assumption. However, the rate of progress is highly

dependant on how the job is constructed. If

Note: ProgressMonitor does not currently give us access to its JProgressBar component. We hope that in future implementations this will be accounted for, as this makes customization more difficult.

13.1.6 ProgressMonitorInputStream

class javax.swing.ProgressMonitorInputStream

This class extends java.io.FilterInputStream and contains a ProgressMonitor. When used in place of an InputStream, this class provides a very simple means of displaying job progress. This InputStream's overloaded read() methods read data and update the ProgressMonitor at the same time. We can access ProgressMonitorInputStream's ProgressMonitor with getProgressMonitor() but we cannot assign it a new one. (See the API docs for more about InputStreams.)

13.2 Basic JScrollBar example

The JScrollBar component is most often seen as part of a JScrollPane. We rarely use this component alone, unless customized scrolling is desired. In this section we'll show how to use JScrollBar to create a simple custom scrolling pane from scratch.

Figure 13.2 Running ScrollDemo example showing an image in the custom scrollpane.
<<file figure13-2.gif>>

The Code: ScrollDemo.java
see \Chapter13\2

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

import javax.swing.*;
import javax.swing.event.*;

public class ScrollDemo extends JFrame
{
  public ScrollDemo() {
    super("JScrollBar Demo");
    setSize(300,250);
```

```
    ImageIcon ii = new ImageIcon("earth.jpg");
    CustomScrollPane sp = new CustomScrollPane(new JLabel(ii));
    getContentPane().add(sp);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);
    setVisible(true);
  }

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

class CustomScrollPane extends JPanel
{
  protected JScrollBar m_vertSB;
  protected JScrollBar m_horzSB;
  protected CustomViewport m_viewport;
  protected JComponent m_comp;

  public CustomScrollPane(JComponent comp) {
    setLayout(null);
    m_viewport = new CustomViewport();
    m_viewport.setLayout(null);
    add(m_viewport);
    m_comp = comp;
    m_viewport.add(m_comp);

    m_vertSB = new JScrollBar(
      JScrollBar.VERTICAL, 0, 0, 0, 0);
    m_vertSB.setUnitIncrement(5);
    add(m_vertSB);

    m_horzSB = new JScrollBar(
      JScrollBar.HORIZONTAL, 0, 0, 0, 0);
    m_horzSB.setUnitIncrement(5);
    add(m_horzSB);

    AdjustmentListener lst = new AdjustmentListener() {
      public void adjustmentValueChanged(AdjustmentEvent e) {
        m_viewport.doLayout();
      }
    };
    m_vertSB.addAdjustmentListener(lst);
    m_horzSB.addAdjustmentListener(lst);
  }

  public void doLayout() {
    Dimension d = getSize();
    Dimension d0 = m_comp.getPreferredSize();
    Dimension d1 = m_vertSB.getPreferredSize();
    Dimension d2 = m_horzSB.getPreferredSize();

    int w = Math.max(d.width - d1.width-1, 0);
    int h = Math.max(d.height - d2.height-1, 0);
    m_viewport.setBounds(0, 0, w, h);
    m_vertSB.setBounds(w+1, 0, d1.width, h);
    m_horzSB.setBounds(0, h+1, w, d2.height);
```

```
    int xs = Math.max(d0.width - w, 0);
    m_horzSB.setMaximum(xs);
    m_horzSB.setBlockIncrement(xs/5);
    m_horzSB.setEnabled(xs > 0);

    int ys = Math.max(d0.height - h, 0);
    m_vertSB.setMaximum(ys);
    m_vertSB.setBlockIncrement(ys/5);
    m_vertSB.setEnabled(ys > 0);

    m_horzSB.setVisibleAmount(m_horzSB.getBlockIncrement());
    m_vertSB.setVisibleAmount(m_vertSB.getBlockIncrement());
  }

  public Dimension getPreferredSize() {
    Dimension d0 = m_comp.getPreferredSize();
    Dimension d1 = m_vertSB.getPreferredSize();
    Dimension d2 = m_horzSB.getPreferredSize();
    Dimension d = new Dimension(d0.width+d1.width,
      d0.height+d2.height);
    return d;
  }

  class CustomViewport extends JPanel
  {
    public void doLayout() {
      Dimension d0 = m_comp.getPreferredSize();
      int x = m_horzSB.getValue();
      int y = m_vertSB.getValue();
      m_comp.setBounds(-x, -y, d0.width, d0.height);
    }
  }
}
```

Understanding the Code

Class ScrollDemo

This simple frame-based class creates a `CustomScrollPane` instance to scroll a large image. This class is very similar to the first example in the chapter 7 and does not require additional explanation.

Class CustomScrollPane

This class extends `JPanel` to represent a simple custom scrollpane. Four instance variables are declared:

- `JScrollBar m_vertSB`: vertical scrollbar.
- `JScrollBar m_horzSB`: horizontal scrollbar.
- `CustomViewport m_viewport`: custom viewport component.
- `JComponent m_comp`: component to be placed in our custom viewport.

The `CustomScrollPane` constructor takes a component to be scrolled as parameter. It instantiates the instance variables described above and adds them to itself using a `null` layout (because this component acts as its own layout manager). Note that the `JScrollBars` are created with proper orientation and zero values accross the board (because these are meaningless if not based on the size of the component being scrolled).

An `AdjustmentListener` is created and added to both scrollbars. The `adjustmentValueChanged()` method calls the `doLayout()` method on the `m_viewport` component to perform the actual component

scrolling according to the new scrollbars values.

The `doLayout()` method sets the bounds for the viewport (in the center), vertical scrollbar (on the right), and horizontal scrollbar (on the bottom). New maximum values and block increment values are set for the scrollbars based on the sizes of the scrolling pane and component to be scrolled. Note that if the maximum value reaches zero, the corresponding scrollbar is disabled. The `visibleAmount` property of each is set to the corresponding `blockIncrement` value to provide proportional thumb sizes.

The `getPreferredSize()` method simply calculates the preferred size of this component based on the preferred sizes of it's children.

Class CustomViewport

This class extends `JPanel` and represents a simple realization of a viewport for our custom scrolling pane. The only implemented method, `doLayout()`, reads the current scrollbar values and assigns bounds to the scrolling component accordingly.

Running the Code

Figure 13.2 shows an image in the custom scrollpane. Use the horizontal and vertical scrollbars to verify that scrolling works as expected. Resize the frame component to verify that the scrollbar values and thumbs are adjusted correctly as the container's size is changed.

13.3 JSlider date chooser

In this example we'll show how three `JSliders` can be combined to allow date selection. We will also address some resizing issues and show how to dynamically change `JSlider`'s annotation components and tick spacing based on size constraints.

Note: While month and day are limited values, year is not. We can use a `JSlider` to select year only if we define a limited range of years to choose from. In this example we bound the year slider value between 1990 and 2010.

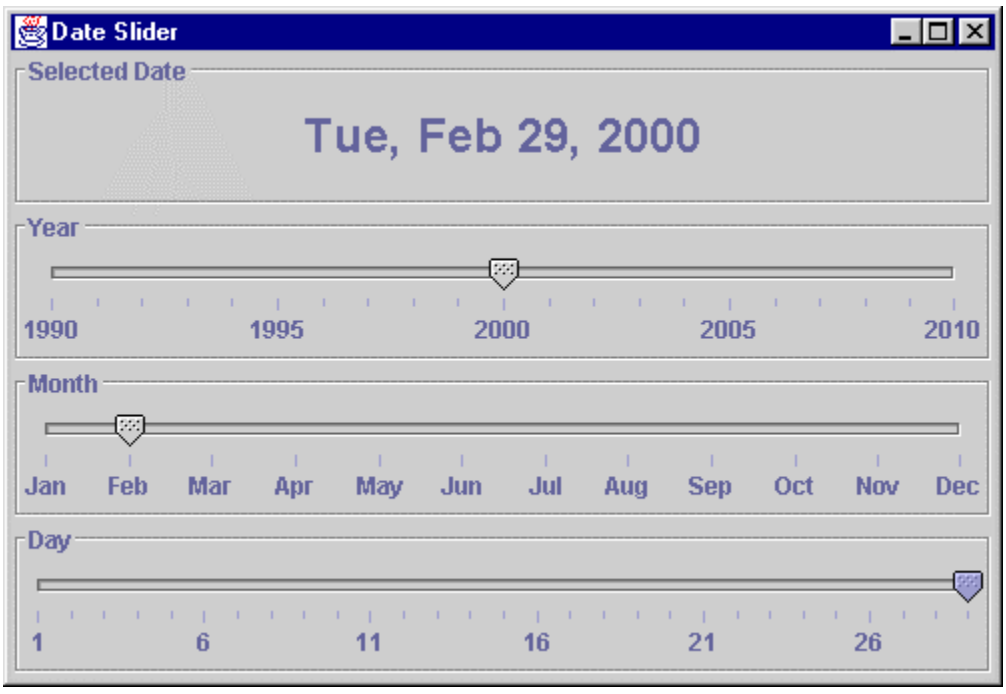


Figure 13.3 JSliders with dynamically changable bound values, tick spacing, and annotation components.
<<file figure13-3.gif>>

UIGuideline:

Feedback in Readable form

Using Sliders to pick the values for a date may be an interesting method for data input, however, it does not lend itself to reading and clear output communication. This is fixed by the use of the clearly human readable form at the top of the dialog. This directly follows the advice that Sliders should be used with immediate visual feedback.

VisualNoise

Visualnoise or clutter is avoided through the spacing of annotations and the avoiding the temptation to annotate each day and each year. The change in rendering as the device is resized to smaller is also a clear example of how extra coding and the adoption of an advanced technique can aid visual communication and usability.

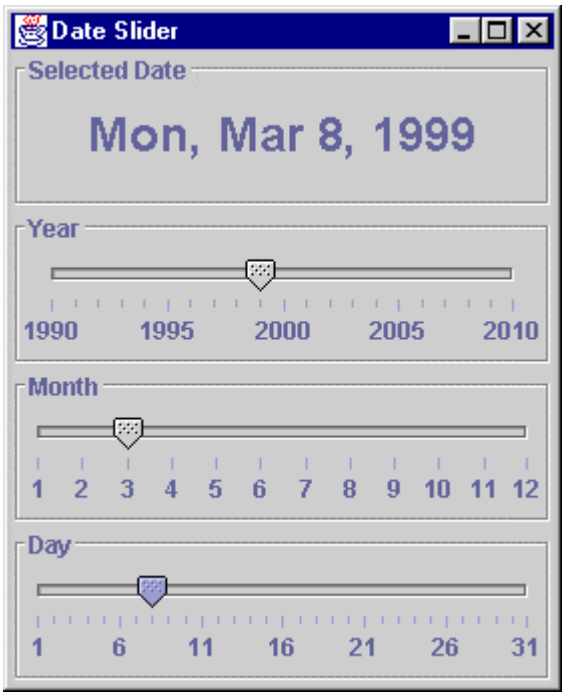


Figure 13.4 JSliders showing altered maximum bound and annotation labels.
<<file figure13-4.gif>>

The Code: DateSlider.java
see \Chapter13\3

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class DateSlider extends JFrame
{
  public final static Dimension RIGID_DIMENSION =
    new Dimension(1,3);
```

```
protected JLabel  m_lbDate;
protected JSlider m_slYear;
protected JSlider m_slMonth;
protected JSlider m_slDay;
protected Hashtable m_labels;
protected GregorianCalendar m_calendar;
protected SimpleDateFormat m_dateFormat;

public DateSlider() {
  super("Date Slider");
  setSize(500, 340);

  m_calendar = new GregorianCalendar();
  Date currDate = new Date();
  m_calendar.setTime(currDate);
  m_dateFormat = new SimpleDateFormat("EEE, MMM d, yyyyy");

  JPanel p1 = new JPanel();
  p1.setLayout(new GridLayout(4, 1));

  JPanel p = new JPanel();
  p.setBorder(new TitledBorder(new EtchedBorder(),
    "Selected Date"));
  m_lbDate = new JLabel(
    m_dateFormat.format(currDate) + "     ");
  m_lbDate.setFont(new Font("Arial",Font.BOLD,24));
  p.add(m_lbDate);
  p1.add(p);

  m_slYear = new JSlider(JSlider.HORIZONTAL, 1990, 2010,
    m_calendar.get(Calendar.YEAR));
  m_slYear.setPaintLabels(true);
  m_slYear.setMajorTickSpacing(5);
  m_slYear.setMinorTickSpacing(1);
  m_slYear.setPaintTicks(true);
  DateListener lst = new DateListener();
  m_slYear.addChangeListener(lst);

  p = new JPanel();
  p.setBorder(new TitledBorder(new EtchedBorder(), "Year"));
  p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
  p.add(Box.createRigidArea(RIGID_DIMENSION));
  p.add(m_slYear);
  p.add(Box.createRigidArea(RIGID_DIMENSION));
  p1.add(p);

  m_slMonth = new JSlider(JSlider.HORIZONTAL, 1, 12,
    m_calendar.get(Calendar.MONTH)+1);
  String[] months =
    (new DateFormatSymbols()).getShortMonths();
  m_labels = new Hashtable(12);
  for (int k=0; k<12; k++)
    m_labels.put(new Integer(k+1), new JLabel(
      months[k], JLabel.CENTER ));
  m_slMonth.setLabelTable(m_labels);
  m_slMonth.setPaintLabels(true);
  m_slMonth.setMajorTickSpacing(1);
  m_slMonth.setPaintTicks(true);
  m_slMonth.addChangeListener(lst);

  p = new JPanel();
  p.setBorder(new TitledBorder(new EtchedBorder(), "Month"));
```

```
    p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
    p.add(Box.createRigidArea(RIGID_DIMENSION));
    p.add(m_slMonth);
    p.add(Box.createRigidArea(RIGID_DIMENSION));
    p1.add(p);

    int maxDays = m_calendar.getActualMaximum(
      Calendar.DAY_OF_MONTH);
    m_slDay = new JSlider(JSlider.HORIZONTAL, 1, maxDays,
      m_calendar.get(Calendar.DAY_OF_MONTH));
    m_slDay.setPaintLabels(true);
    m_slDay.setMajorTickSpacing(5);
    m_slDay.setMinorTickSpacing(1);
    m_slDay.setPaintTicks(true);
    m_slDay.addChangeListener(lst);

    p = new JPanel();
    p.setBorder(new TitledBorder(new EtchedBorder(), "Day"));
    p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
    p.add(Box.createRigidArea(RIGID_DIMENSION));
    p.add(m_slDay);
    p.add(Box.createRigidArea(RIGID_DIMENSION));
    p1.add(p);

    getContentPane().add(p1, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    enableEvents(ComponentEvent.COMPONENT_RESIZED);
    setVisible(true);
  }

  protected void processComponentEvent(ComponentEvent e) {
    if (e.getID() == ComponentEvent.COMPONENT_RESIZED) {
      int w = getSize().width;

      m_slYear.setLabelTable(null);
      if (w > 200)
        m_slYear.setMajorTickSpacing(5);
      else
        m_slYear.setMajorTickSpacing(10);
      m_slYear.setPaintLabels(w > 100);

      m_slMonth.setLabelTable(w > 300 ? m_labels : null);
      if (w <= 300 && w >=200)
        m_slMonth.setMajorTickSpacing(1);
      else
        m_slMonth.setMajorTickSpacing(2);
      m_slMonth.setPaintLabels(w > 100);

      m_slDay.setLabelTable(null);
      if (w > 200)
        m_slDay.setMajorTickSpacing(5);
      else
        m_slDay.setMajorTickSpacing(10);
      m_slDay.setPaintLabels(w > 100);
    }
  }
```

```
  public void showDate() {
    m_calendar.set(m_slYear.getValue(),
    m_slMonth.getValue()-1, 1);
    int maxDays = m_calendar.getActualMaximum(
      Calendar.DAY_OF_MONTH);

    if (m_slDay.getMaximum() != maxDays) {
      m_slDay.setValue(
        Math.min(m_slDay.getValue(), maxDays));
      m_slDay.setMaximum(maxDays);
      m_slDay.repaint();
    }

    m_calendar.set(
      m_slYear.getValue(), m_slMonth.getValue()-1,
      m_slDay.getValue());
    Date date = m_calendar.getTime();
    m_lbDate.setText(m_dateFormat.format(date));
  }

  class DateListener implements ChangeListener
  {
    public void stateChanged(ChangeEvent e) {
      showDate();
    }
  }

  public static void main(String argv[]) {
    new DateSlider();
  }
}
```

Understanding the Code

Class DateSlider

`DateSlider` extends `JFrame` and declares seven instance variables and one class constant. Class constant:

`Dimension RIGID_DIMENSION`: used to create rigid areas above and below each slider.

Instance variables:

`JLabel m_lbDate`: label to display the selected date.

`JSlider m_slYear`: slider to select year.

`JSlider m_slMonth`: slider to select month.

`JSlider m_slDay`: slider to select day.

`Hashtable m_labels`: collection of labels to denote months by short names rather than numbers.

`GregorianCalendar m_calendar`: calendar to perform date manipulations.

`SimpleDateFormat m_dateFormat`: object to format the date as a string.

The `DateSlider` constructor initializes the `m_calendar` instance defined above, and date format `m_dateFormat`. A `JPanel` with a `GridLayout` of one column and four rows is used as a base panel, `p1`. `JLabel m_lbDate` using a large font is created, embedded in a `JPanel` with a simple `TitledBorder`, and placed in the first row.

The `m_slYear` slider is created and placed in the second row. This is used to select the year from the interval 1990 to 2010. Note that it takes its initial value from the current date. A number of settings are applied to `m_slYear`. The `paintLabels` and `paintTicks` properties are set to `true` to allow drawing ticks and labels, `majorTickSpacing` is set to 5 to draw majors ticks for every fifth value, and `minorTickSpacing` is set to 1 to draw minor ticks for every value. Finally a new `DateListener` instance (see below) is added as a `ChangeListener` to monitor changes to this slider's properties. Note that `m_slYear` is placed in a `JPanel` surrounded by a `TitledBorder`. Two rigid areas are added to ensure vertical spacing between our slider and this parent panel (see chapter 4 for more about `Box` and its invisible `Filler` components).

The `m_slMonth` slider is created and placed in the third row. This slider is used to select the month from the interval 1 to 12. This component is constructed similar to `m_slYear`, but receives a `Hashtable` of `JLabels` to denote months by short names rather than numbers. These names are taken from an instance of the `DateFormatSymbols` class (see API docs) and used to create pairs in a local `m_labels` `Hashtable` in the form: `Integer` representing slider value (from 1 to 12) as key, and `JLabel` with the proper text as value. Finally, the `setLabelTable()` method is invoked to assign these custom labels to the slider.

The `m_slDay` slider is created and placed in the fourth row. It is used to select the day of the month from an interval which dynamically changes depending on the current month and, for February, the year. Aside from this difference, `m_slDay` is constructed very similar to `m_slYear`.

Because a slider's tick annotation components may overlap each other and become unreadable if not enough space provided, it is up to us to account for this possibility. This becomes a more significant problem when (as in this example) slider components can resized by simply resizing the parent frame. To work around this problem we can simply enforce a certain frame size, however, this may not be desirable in all situations. If we are ever in such a situation we need to change our slider's properties dynamically depending on its size. For this reason the `processComponentEvent()` method is overridden to process resizing events that occur on the parent frame. Processing of these events is enabled in the `DateSlider` constructor with the `enableEvents()` method.

The `processComponentEvent()` method only responds to `ComponentEvents` with ID `COMPONENT_RESIZED`. For each of our three sliders this method changes the `majorTickSpacing` property based on the container's width. `m_slDay` and `m_slYear` receive a spacing of 5 if the width if greater than 200, and 10 otherwise. `m_slMonth` receives a `majorTickSpacing` of 1 if the conatiner's width is anywhere from 200 to 300, and 2 otherwise. If this width is greater than 300 our custom set of labels is used to annotate `m_slMonth`'s major ticks. The default numerical labels are used otherwise. For each slider, if the width is less than 100 the `paintLabels` property is set to `false`, which disables all annotations. Otherwise `paintLabels` is set to `true`.

Our custom `showDate()` method is used to retrieve values from our sliders and display them in `m_lbDate` as the new selected date. First we determine the maximum number of days for the selected month by passing `m_calendar` a year, a month, and 1 as the day. Then, if necessary, we reset `m_slDay`'s current and maximum values. Finally, we pass `m_calendar` a year, month, and the selected (possibly adjusted) day, retrieve a `Date` instance corresponding to these values, and invoke `format()` to retrieve a textual representation of the date.

Note: Java 2 does not really provide a direct way to convert a year, month, and day triplet into a `Date` instance (this functionality has been deprecated). We need to use `Calendar.set()` and `Calendar.getTime()` for this. Be aware that the day parameter is not checked against the maximum value for the selected month. Setting the day to 30 when the month is set to February will be silently treated as March, 2.

Class DateSlider.DateListener

The `DateListener` inner class implements the `ChangeListener` interface and is used to listen for changes in each of our sliders' properties. Its `stateChanged()` method simply calls the `showDate()` method described above.

Running the Code

Note how the date is selected and displayed, and the range of the "Day" slider is adjusted when a new month is selected. Figure 13.3 shows selection of February 29th 2000, demonstrating that this is a leap year.

Note: A leap year is a year evenly divisible by four, but not evenly divisible by 100. The first rule takes precedence, so years evenly divisible by 400 are leap years (2000 is a leap year, while 1900 is not).

Now try resizing the application frame to see how the slider annotations and ticks change to their more compact variants as the available space shrinks. Figure 13.4 illustrates.

UI Guideline:

Exact value selection

Although Sliders are best used for selection when an exact value is not needed, this example gets around it by providing an adequate gap between ticks, making an exact choice easy to achieve.

The use of a Slider for Year is an unusual choice, as Year is not normally a bounded input. However, in certain domains it may be a more suitable choice. You may for example know the limits of available years e.g. years on which an Olympic games was held. The tick value would be 4 and the bound would be from the first games in 1896 to the next in 2000. Once Year and Month have been displayed using Sliders it is visually attractive and consistent to use a Slider for Day. There may be some debate about doing so as the bound will change depending on the month selected. However, it is fair to argue that the changing bound on Day, as Month is selected gives a clear, instant, visual feedback of how many days are in the month, which meets with the criteria of providing instant feedback when using a Slider.

13.4 JSliders in a JPEG image editor

Java 2 ships with a special package, `com.sun.image.codec.jpeg`, providing a set of classes and interfaces for working with JPEG images (created at least in part by Eastman Kodak Company). Although this package is not a part of Swing, it can be very useful in Swing-based applications. By reducing image quality (which is actually a result of compression), required storage space can be decreased. Using reduced quality JPEGs in web pages increases response time (by decreasing download time), and our editor application developed here allows us to load an existing JPEG, modify its quality, and then save the result. `JSliders` are used for the main editing components.

Note: JPEG stands for Joint Photographic Experts Group. It is a popular graphical format allowing compression of images up to 10 or 20 times.

Before deciding to use functionality in this package, you should know that, even though this package is shipped with Java 2, "... the classes in the `com.sun.image.codec.jpeg` package are not part of the core Java APIs. They are a part of Sun's JDK and JRE distributions. Although other licensees may choose to distribute these classes, developers cannot depend on their availability in non-Sun implementations. We expect that equivalent functionality will eventually be available in a core API or standard extension."

13.4.1 The JPEGDecodeParam interface com.sun.image.codec.jpeg

abstract interface com.sun.image.codec.jpeg.JPEGDecodeParam

This interface encapsulates parameters used to control the decoding of a JPEG image. It provides a rich set of `getXX()` and `isXX()` accessor methods. Instances contain information about how to decode a JPEG inpout stream, and are created automatically by `JPEGImageDecoder` (see below) if none is specified when an image is decoded. A `JPEGImageDecoder`'s associated `JPEGDecoderParam` can be obtained with its `getJPEGDecodeParam()` method.

13.4.2 The JPEGEncodeParam interface

abstract interface com.sun.image.codec.jpeg.JPEGEncodeParam

This interface encapsulates parameters used to control the encoding of a JPEG image stream. It provides a rich set of `getXX()` and `setXX()` accessor methods. Instances contain information about how to encode a JPEG to an output stream, and a default instance will be created atomatically by `JPEGImageEncoder` (see below) if none is specified when an image is encoded. A `JPEGImageEncoder`'s associated `JPEGEncodeParam` can be obtained with its `getJPEGEncodeParam()` method, or one of its overriden `getDefaultJPEGEncodeParam()` methods.

Particularly relevant to this example are `JPEGEncodeParam`'s `xDensity`, `yDensity`, and `quality` properties, which all can be assigned using typical `setXX()` methods. `xDensity` and `yDensity` represent horizontal and vertical pixel density, which depends on `JPEGEncoderParam`'s current pixel density setting. The pixel density setting is controlled with `JPEGEncodeParam`'s `setDensityUnit()` method and can be, for instance, `DENSITY_UNIT_DOTS_INCH`, which means pixel density will be interpreted as pixels per inch. The quality property is specified as a float within the range `0.0` to `1.0`, where `1.0` means perfect quality. In general: `0.75` means high quality, `0.5` means medium quality, and `0.25` means low quality.

13.4.3 The JPEGImageDecoder interface

abstract interface com.sun.image.codec.jpeg.JPEGImageDecoder

This interface describes an object used to decode a JPEG data stream into an image. We invoke method `decodeAsBufferedImage()` to perform the actual decoding into a `BufferedImage` instance, or `decodeAsRaster()` to perform decoding into a `Raster` instance. An instance of this interface can be obtained with one of the `JPEGCodec.createJPEGDecoder()` methods, which takes the delivering data `InputStream` as parameter. `JPEGImageDecoder` performs decoding according to its associated `JPEGDecodeParam`, and a default instance will br provided for is we do not specify one.

13.4.4 The JPEGImageEncoder interface

abstract interface com.sun.image.codec.jpeg.JPEGImageEncoder

This interface describes an object used to encode an image into a JPEG data stream. We invoke the overloaded `encode()` method to perform the actual encoding. Instances of this interface can be obtained with one of the `JPEGCodec.createJPEGEncoder()` methods, which takes an `OutputStream` to output data to as parameter. `JPEGImageEncoder` performs encoding according to its associated `JPEGImageEncoder`, and a default instance will br provided for is we do not specify one.

13.4.5 JPEG Codec

class com.sun.image.codec.jpeg.JPEGCodec

This class contains a collection of static methods used to create JPEG encoders and decoders. Particularly useful are the overloaded `createJPEGDecoder()` and `createJPEGEncoder()` methods which take an `InputStream` and `OutputStream`, respectively, as parameter (along with an optional `JPEGDecodeParam` or `JPEGEncodeParam` instance).

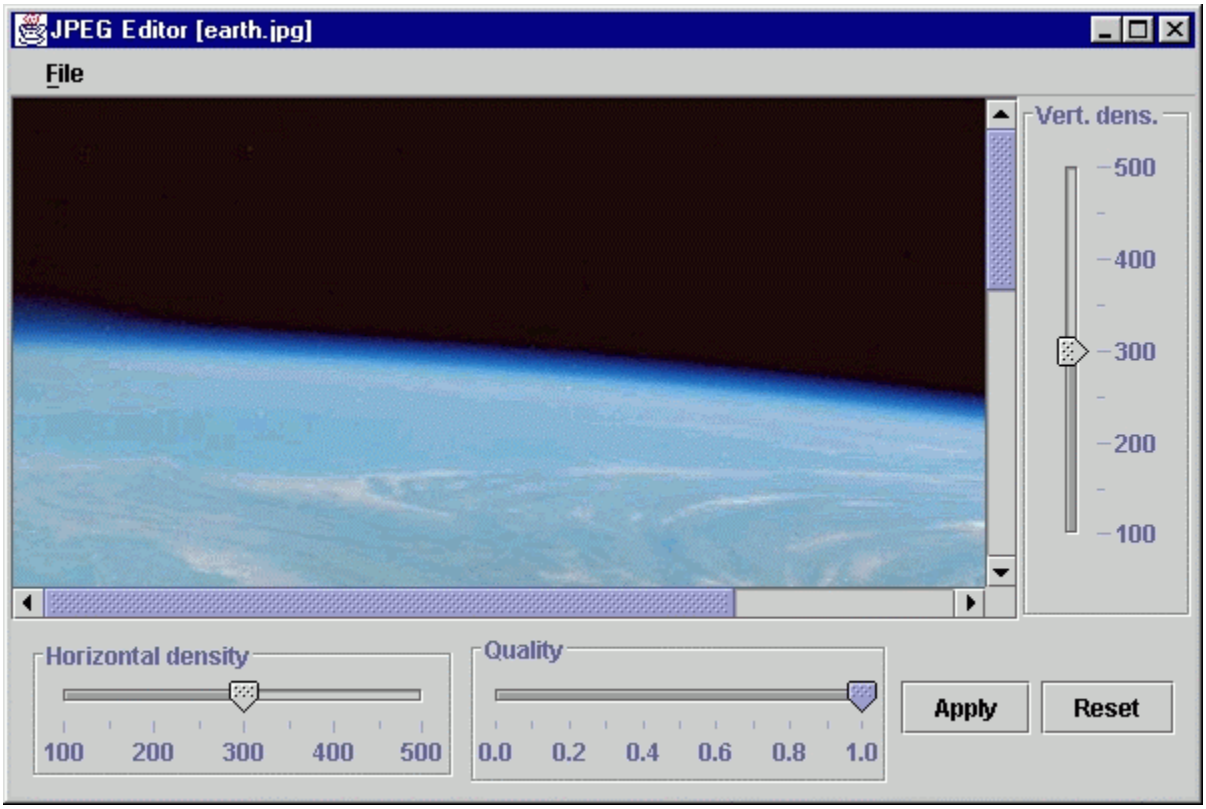


Figure 13.6 JPEG Editor showing a high-quality image of Earth (using JSliders with "isFilled" client property).
<<file figure13-6.gif>>

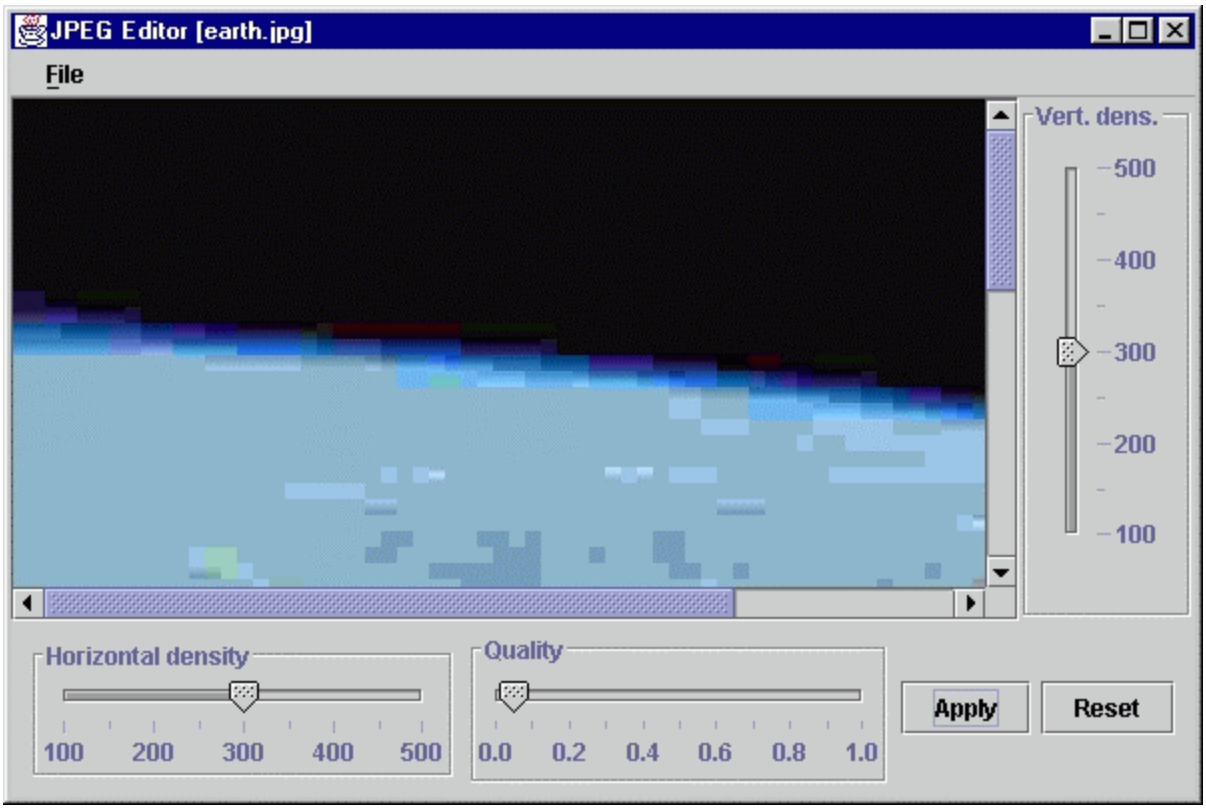


Figure 13.7 JPEG Editor showing a reduced quality image of Earth.

<<file figure13-7.gif>>

The Code: JPEGEditor.java

see \Chapter13\4

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.util.*;
import java.io.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.filechooser.*;

import com.sun.image.codec.jpeg.*;

public class JPEGEditor extends JFrame
{
  public final static Dimension VERTICAL_RIGID_SIZE
    = new Dimension(1,3);
  public final static Dimension HORIZONTAL_RIGID_SIZE
    = new Dimension(3,1);

  protected File m_currentDir = new File(".");
  protected File m_currentFile = null;

  protected JFileChooser m_chooser;
  protected JPEGPanel m_panel;
  protected JSlider m_slHorzDensity;
  protected JSlider m_slVertDensity;
  protected JSlider m_slQuality;
```

```
protected BufferedImage m_bi1, m_bi2;

public JPEGEditor() {
  super("JPEG Editor");
  setSize(600, 400);

  m_chooser = new JFileChooser();
  SimpleFilter filter = new SimpleFilter("jpg",
    "JPEG Image Files");
  m_chooser.setFileFilter(filter);
  m_chooser.setCurrentDirectory(m_currentDir);

  m_panel = new JPEGPanel();
  JScrollPane ps = new JScrollPane(m_panel,
    JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,
    JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
  getContentPane().add(ps, BorderLayout.CENTER);

  JPanel p, p1;

  m_slVertDensity = new JSlider(JSlider.VERTICAL,
    100, 500, 300);
  m_slVertDensity.setExtent(50);
  m_slVertDensity.setPaintLabels(true);
  m_slVertDensity.setMajorTickSpacing(100);
  m_slVertDensity.setMinorTickSpacing(50);
  m_slVertDensity.setPaintTicks(true);
  m_slVertDensity.putClientProperty(
    "JSlider.isFilled", Boolean.TRUE);

  p = new JPanel();
  p.setBorder(new TitledBorder(new EtchedBorder(),
    "Vert. dens."));
  p.add(Box.createRigidArea(HORIZONTAL_RIGID_SIZE));
  p.add(m_slVertDensity);
  p.add(Box.createRigidArea(HORIZONTAL_RIGID_SIZE));
  getContentPane().add(p, BorderLayout.EAST);

  m_slHorzDensity = new JSlider(JSlider.HORIZONTAL,
    100, 500, 300);
  m_slHorzDensity.setExtent(50);
  m_slHorzDensity.setPaintLabels(true);
  m_slHorzDensity.setMajorTickSpacing(100);
  m_slHorzDensity.setMinorTickSpacing(50);
  m_slHorzDensity.setPaintTicks(true);
  m_slHorzDensity.putClientProperty(
    "JSlider.isFilled", Boolean.TRUE);

  p = new JPanel();
  p.setBorder(new TitledBorder(new EtchedBorder(),
    "Horizontal density"));
  p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
  p.add(Box.createRigidArea(VERTICAL_RIGID_SIZE));
  p.add(m_slHorzDensity);
  p.add(Box.createRigidArea(VERTICAL_RIGID_SIZE));
  p1 = new JPanel();
  p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
  p1.add(p);

  m_slQuality = new JSlider(JSlider.HORIZONTAL,
    0, 100, 100);
  Hashtable labels = new Hashtable(6);
```

```
    for (float q = 0; q <= 1.0; q += 0.2)
      labels.put(new Integer((int)(q*100)),
      new JLabel("" + q, JLabel.CENTER ));
    m_slQuality.setLabelTable(labels);
    m_slQuality.setExtent(10);
    m_slQuality.setPaintLabels(true);
    m_slQuality.setMinorTickSpacing(10);
    m_slQuality.setPaintTicks(true);
    m_slQuality.putClientProperty(
      "JSlider.isFilled", Boolean.TRUE);

    p = new JPanel();
    p.setBorder(new TitledBorder(new EtchedBorder(),
      "Quality"));
    p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
    p.add(Box.createRigidArea(VERTICAL_RIGID_SIZE));
    p.add(m_slQuality);
    p.add(Box.createRigidArea(VERTICAL_RIGID_SIZE));
    p1.add(p);

    JButton btApply = new JButton("Apply");
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        apply();
      }
    };
    btApply.addActionListener(lst);
    p1.add(btApply);

    JButton btReset = new JButton("Reset");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        reset();
      }
    };
    btReset.addActionListener(lst);
    p1.add(btReset);
    getContentPane().add(p1, BorderLayout.SOUTH);

    setJMenuBar(createMenuBar());

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  protected JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();

    JMenu mFile = new JMenu("File");
    mFile.setMnemonic('f');

    JMenuItem mItem = new JMenuItem("Open...");
    mItem.setMnemonic('o');
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if (m_chooser.showOpenDialog(JPEGEditor.this) !=
          JFileChooser.APPROVE_OPTION)
```

```
            return;
          m_currentDir = m_chooser.getCurrentDirectory();
          File fChoosen = m_chooser.getSelectedFile();
          openFile(fChoosen);
        }
      };
      mItem.addActionListener(lst);
      mFile.add(mItem);

      mItem = new JMenuItem("Save");
      mItem.setMnemonic('s');
      lst = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          saveFile(m_currentFile);
        }
      };
      mItem.addActionListener(lst);
      mFile.add(mItem);

      mItem = new JMenuItem("Save As...");
      mItem.setMnemonic('a');
      lst = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          m_chooser.setSelectedFile(m_currentFile);
          if (m_chooser.showSaveDialog(JPEGEditor.this) !=
            JFileChooser.APPROVE_OPTION)
          return;
          m_currentDir = m_chooser.getCurrentDirectory();
          File fChoosen = m_chooser.getSelectedFile();
          if (fChoosen!=null && fChoosen.exists()) {
            String message = "File " + fChoosen.getName()+
              " already exists. Override?";
            int result = JOptionPane.showConfirmDialog(
              JPEGEditor.this, message, getTitle(),
              JOptionPane.YES_NO_OPTION);
            if (result != JOptionPane.YES_OPTION)
              return;
          }
          setCurrentFile(fChoosen);
          saveFile(fChoosen);
        }
      };
      mItem.addActionListener(lst);
      mFile.add(mItem);

      mFile.addSeparator();

      mItem = new JMenuItem("Exit");
      mItem.setMnemonic('x');
      lst = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          System.exit(0);
        }
      };
      mItem.addActionListener(lst);
      mFile.add(mItem);
      menuBar.add(mFile);
      return menuBar;
    }

    protected void setCurrentFile(File file) {
      if (file != null) {
        m_currentFile = file;
```

```
      setTitle("JPEG Editor ["+file.getName()+"]");
    }
  }

  protected void openFile(final File file) {
    if (file == null || !file.exists())
      return;
    setCurrentFile(file);

    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    Thread runner = new Thread() {
      public void run() {
        try {
          FileInputStream in = new FileInputStream(file);
          JPEGImageDecoder decoder =
            JPEGCodec.createJPEGDecoder(in);
          m_bi1 = decoder.decodeAsBufferedImage();
          m_bi2 = null;
          in.close();
          SwingUtilities.invokeLater( new Runnable() {
            public void run() { reset(); }
          });
        }
        catch (Exception ex) {
          ex.printStackTrace();
          System.err.println("openFile: "+ex.toString());
        }
        setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
      }
    };
    runner.start();
  }

  protected void saveFile(final File file) {
    if (file == null || m_panel.getBufferedImage() == null)
      return;

    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    Thread runner = new Thread() {
      public void run() {
        try {
          FileOutputStream out = new FileOutputStream(file);
          JPEGImageEncoder encoder =
            JPEGCodec.createJPEGEncoder(out);
          encoder.encode(m_panel.getBufferedImage());
          out.close();
        }
        catch (Exception ex) {
          ex.printStackTrace();
          System.err.println("apply: "+ex.toString());
        }
        setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
      }
    };
    runner.start();
  }

  protected void apply() {
    if (m_bi1 == null)
      return;
```

```
    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    Thread runner = new Thread() {
      public void run() {
        try {
          ByteArrayOutputStream out = new ByteArrayOutputStream();
          JPEGImageEncoder encoder =
            JPEGCodec.createJPEGEncoder(out);
          JPEGEncodeParam param =
            encoder.getDefaultJPEGEncodeParam(m_bi1);

          float quality = m_slQuality.getValue()/100.0f;
          param.setQuality(quality, false);

          param.setDensityUnit(
            JPEGEncodeParam.DENSITY_UNIT_DOTS_INCH);
          int xDensity = m_slHorzDensity.getValue();
          param.setXDensity(xDensity);
          int yDensity = m_slVertDensity.getValue();
          param.setYDensity(yDensity);

          encoder.setJPEGEncodeParam(param);
          encoder.encode(m_bi1);

          ByteArrayInputStream in = new ByteArrayInputStream(
            out.toByteArray());
          JPEGImageDecoder decoder =
            JPEGCodec.createJPEGDecoder(in);
          final BufferedImage bi2 = decoder.decodeAsBufferedImage();
          SwingUtilities.invokeLater( new Runnable() {
            public void run() {
              m_panel.setBufferedImage(bi2);
            }
          });
        }
        catch (Exception ex) {
          ex.printStackTrace();
          System.err.println("apply: "+ex.toString());
        }
        setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
      }
    };
    runner.start();
  }

  protected void reset() {
    if (m_bi1 != null) {
      m_panel.setBufferedImage(m_bi1);
      m_slQuality.setValue(100);
      m_slHorzDensity.setValue(300);
      m_slVertDensity.setValue(300);
    }
  }

  public static void main(String argv[]) {
    new JPEGEditor();
  }
}

class JPEGPanel extends JPanel
{
  protected BufferedImage m_bi = null;
```

```
  public JPEGPanel() {}

  public void setBufferedImage(BufferedImage bi) {
    if (bi == null)
      return;
    m_bi = bi;
    Dimension d = new Dimension(m_bi.getWidth(this),
      m_bi.getHeight(this));
    setPreferredSize(d);
    revalidate();
    repaint();
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Dimension d = getSize();
    g.setColor(getBackground());
    g.fillRect(0, 0, d.width, d.height);
    if (m_bi != null)
      g.drawImage(m_bi, 0, 0, this);
  }

  public BufferedImage getBufferedImage() {
    return m_bi;
  }
}

//class SimpleFilter taken from chapter 14
```

Understanding the Code

Class JPEGEditor

Class variables:

`Dimension VERTICAL_RIGID_SIZE`: size of rigid area used for vertical spacing.

`Dimension HORIZONTAL_RIGID_SIZE`: size of rigid area used for horizontal spacing.

Instance variables:

`File m_currentDir`: current directory navigated to by our `JFileChooser`.

`File m_currentFile`: JPEG image file currently in our editing environment.

`JFileChooser m_chooser`: file chooser used for loading and saving JPEGs.

`JPEGPanel m_panel`: custom component used to display JPEGs.

`JSlider m_slHorzDensity`: slider to choose horizontal pixel density.

`JSlider m_slVertDensity`: slider to choose vertical pixel density.

`JSlider m_slQuality`: slider to choose image quality.

`BufferedImage m_bi1`: original image.

`BufferedImage m_bi2`: modified image.

`JPEGEditor`'s constructor starts by instantiating our `JFileChooser` and applying a `SimpleFilter` (see chapter 14) file filter to it, in order to restrict file selection to JPEG images (files with a ".jpg" extension). creates and initializes the GUI components for this example. Custom panel `m_panel` is used to display a JPEG image (see the `JPEGPanel` class below). It is added to a `JScrollPane` to provide scrolling

capabilities. Three sliders are used to select JPEGEncodeParam properties as described above: xDensity, yDensity, and quality. Each is surrounded by a TitledBorder with an appropriate title. Similar to the previous example, RigidAreas are used to ensure proper spacing between the slider and the border. Note that each slider makes use of the MetalL&F client property "JSlider.isFilled" with value Boolean.TRUE to force the lower portion of each slider track to be filled.

Note that the m_slQuality slider must represent values from 0 to 1.0. We scale this interval to [0,100], but display annotation labels 0.0, 0.2, 0.4, .., 1.0 stored in Hashtable labels. The selected image quality value is the slider's value divided by 100. Also note the usage of setExtent() for each slider in this example. Recall that the value of the extent property is used when the slider has focus and the user presses the PgUp or PgDn key to increment or decrement the slider's value respectively.

An "Apply" button is created and assigned an ActionListener to retrieve current slider settings and apply them to current JPEG image by calling our custom apply() method (because of the large amount of work the apply method performs, it does not make sense to do this on-the-fly by listening for slider change events). A "Reset" button reverts any changes and returns the image to its original state by calling our custom reset() method. Finally a JMenuBar is created with our createMenuBar() method.

The createMenuBar() method creates and returns a JMenuBar containing one menu titled "File," which, in turn, contains four menu items: "Open..," "Save," "Save As..," and "Exit." Each item receives its own ActionListener.

The "Open..." menu item invokes our JFileChooser for selecting a JPEG image file. After a successful selection the current directory is stored in our m_currentDir variable for future use, and our custom openFile() method is invoked to load the image into our environment. The "Save" menu item invokes our custom saveFile() method to save the image currently in our environment. The "Save As.." menu item instructs JFileChooser to prompt the user for a new name, and possibly location, to save the current image file to. This code is fairly similar to the code for the "Open..." menu, except that showSaveDialog() is used instead of showOpenDialog(). If the selected file already exists, a request for confirmation is invoked uwing JOptionPane.showConfirmDialog() (interestingly, this is not a standard feature of JFileChooser--see chapter 14 for more about JOptionPane and JFileChooser). Finally our saveFile() method is invoked to save the current image as the selected file. The "Exit" menu item calls System.exit(0) to quit this application.

The setCurrentFile() method stores a reference to the newly opened file in m_currentFile. This method also modifies the frame's title to display file name. It is called whenever the "Open..." and "Save As.." menu items are invoked.

The openFile() method opens a given File corresponding to a stored JPEG image. First it checks whether or not the selected file exists. If so, a new thread is created to execute all remaining code in this method to avoid clogging up the event-dispatching thread. A FileInputStream is opened and a JPEGImageDecoder is created for the given file. Then a call to decodeAsBufferedImage() retrieves a BufferedImage from the JPEGImageDecoder and stores it in our m_bi1 variable. The file stream is closed and our image is passed to JPEGPanel by calling the reset() method (see below). Note that because our reset method directly modifies the state of Swing components, we place this call in a Runnable and send it to the event-dispatching queue with SwingUtilities.invokeLater() (see chapter 2 for more about invokeLater()).

The saveFile() method saves the current image into the given File. In a separate thread, a FileOutputStream is opened and a JPEGImageEncoder is created corresponding to this File. Then a call to the JPEGImageEncoder's encode() method saves the current image (retrieved by our JPEGPanel'sgetBufferedImage() method) to the opened stream.

The `apply()` method applies the current slider settings to the current image. In a separate thread, this method creates a `ByteArrayOutputStream` to stream the operations in memory. Then a `JPEGImageEncoder` is created for this stream, and a `JPEGEncodeParam` is retrieved corresponding to the original image, `m_bi1` (which is assigned in `openFile()`). Three property values are retrieved from our sliders and sent to a `JPEGEncodeParam` object via `setXX()` methods: `quality`, `xDensity` and `yDensity` (note that quality is converted to a `float` through division by `100.0f`). Then this `JPEGEncodeParam` object is assigned to our `JPEGImageEncoder`, and the `encode()` method is used to perform the actual encoding of the `m_bi1` image. Next a new image is retrieved from this encoder by first retrieving a `ByteArrayInputStream` from our `ByteArrayOutputStream` using its `toByteArray()` method. A `JPEGImageDecoder` is created for this stream, and the `decodeAsBufferedImage()` method retrieves a `BufferedImage` instance. Finally, in a `Runnable` sent to `SwingUtilities.invokeLater()`, this image is assigned to our image panel for display with `JPEGPanel`'s `setBufferedImage()` method.

The `reset()` method, as you might guess from its name, resets the current image to its original state (the state it was in when opened) and resets the slider values.

Class JPEGPanel

`JPEGPanel` extends `JPanel` and provides a placeholder for JPEG images. It declares a single instance variable:

`BufferedImage m_bi`: holds the current JPEG.

Method `setBufferedImage()` assigns the given image to `m_bi`, and changes this panel's preferred size to the size of that image. The panel is then revalidated and repainted to display the new image properly.

Note: We learned in chapter 2 that when a `revalidate()` request is invoked on a component, all ancestors below the first ancestor whose `validateRoot` property is `true` get validated. `JRootPane`, `JScrollPane`, and `JTextField` are the only Swing components with a `true validateRoot` property by default. Thus, calling `revalidate()` on our `JPEGPanel` will result in validation of the `JScrollPane` it is contained in within our `JPEGEditor` application. This results in proper layout and display of `JPEGPanel` which would not occur by simply calling `repaint()`.

The `paintComponent()` method clears the background and draws the current image (if any). The `getBufferedImage()` method simply returns the most recent image associated with this panel.

Running the Code

Figure 13.5 shows `JPEGEditor` displaying a high-quality image of Earth. Applying our sliders to reduce the quality, and clicking the "Apply" button, we produce the image shown in Figure 13.6. By saving this image as a new file, we obtain a representation occupying much less disk space than the original. A balance between quality and size is a decision that often needs to be made when space or latency issues are important.

UI Guideline: Component Selection

This example provides some tricky problems for the designer. The nature of the calculation means that instant feedback is not possible. However, the User needs to see what the result of a choice would mean. This has been solved by the introduction of the "Apply" button. This is justifiable in a case such as this due to the complex and time consuming nature of the effect of the selection. It is not otherwise recommended.

The introduction of the shaded area on the Sliders gives a clear indication that an amount or quantity rather than an exact, discrete value is being selected and the amount is a percentage of the bounded whole. This adds to the visual affordance of the component and aids the viewer in understanding what is happening.

13.5 JProgressBar in an FTP client application

The following example uses `JProgressBar` to display progress in downloading and uploading files using File Transfer Protocol (FTP). The support for this protocol is provided in the `sun.net` and `sun.net.ftp` packages.

13.5.1 FtpClient

class sun.net.ftp.FtpClient

This class provides functionality for an FTP client. Methods particularly relevant to this example include:

- `FTPClient(String host)`: constructor to create a new instance and connect to the given host address.
- `login(String user, String password)`: login to an FTP host with given username and password.
- `cd(String directory)`: change directory.
- `binary()`: set mode to binary for proper file transfering.
- `closeSever()`: disconnect from host.
- `list()`: returns an `InputStream` supplying the printout of the `ls -l` command (list contents of directories, one per line).
- `get(String filename)`: returns an `InputStream` for retrieving the specified file from the host.
- `put(String filename)`: returns an `OutputStream` for writing the specified file to the host.

Note: This application's GUI is layed out using our custom `DialogLayout2` layout manager, developed in chapter 4. Refer back to this chapter for more information about how this manager works.

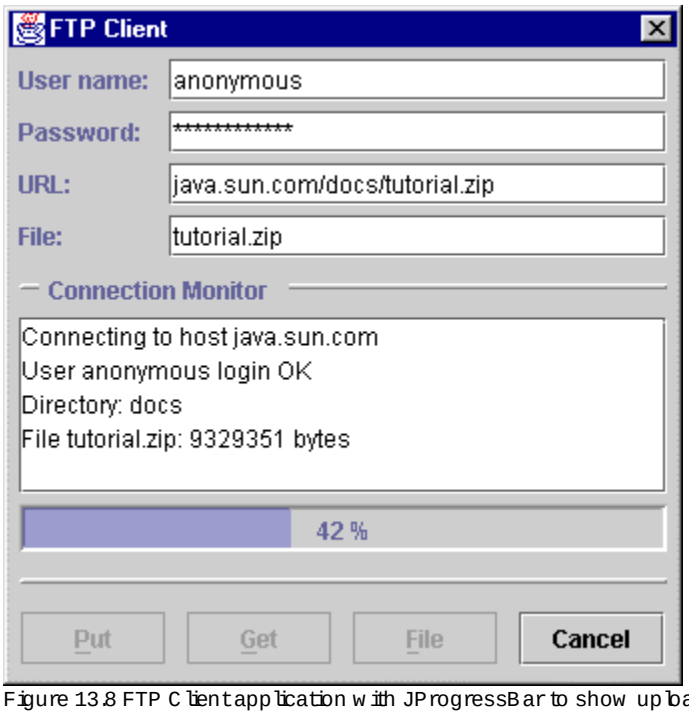


Figure 13.8 FTP Client application with JProgressBar to show upload/download status.

<<file figure13-8.gif>>

The Code: FTPApp.java
see \Chapter13\5

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.net.*;
import java.lang.reflect.*;

import sun.net.ftp.*;
import sun.net.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

import dl.*;

public class FTPApp extends JFrame
{
  public static int BUFFER_SIZE = 10240;

  protected JTextField m_txtUser;
  protected JPasswordField m_txtPassword;
  protected JTextField m_txtURL;
  protected JTextField m_txtFile;
  protected JTextArea  m_monitor;
  protected JProgressBar m_progress;
  protected JButton m_btPut;
  protected JButton m_btGet;
  protected JButton m_btFile;
  protected JButton m_btClose;
  protected JFileChooser m_chooser;

  protected FtpClient m_client;
  protected String m_sLocalFile;
  protected String m_sHostFile;

  public FTPApp() {
    super("FTP Client");

    JPanel p = new JPanel();
    p.setLayout(new DialogLayout2(10, 5));
    p.setBorder(new EmptyBorder(5, 5, 5, 5));

    p.add(new JLabel("User name:"));
    m_txtUser = new JTextField("anonymous");
    p.add(m_txtUser);

    p.add(new JLabel("Password:"));
    m_txtPassword = new JPasswordField();
    p.add(m_txtPassword);

    p.add(new JLabel("URL:"));
    m_txtURL = new JTextField();
    p.add(m_txtURL);

    p.add(new JLabel("File:"));
    m_txtFile = new JTextField();
    p.add(m_txtFile);
```

```
p.add(new DialogSeparator("Connection Monitor"));

m_monitor = new JTextArea(5, 20);
m_monitor.setEditable(false);
JScrollPane ps = new JScrollPane(m_monitor);
p.add(ps);

m_progress = new JProgressBar();
m_progress.setStringPainted(true);
m_progress.setBorder(new BevelBorder(BevelBorder.LOWERED,
  Color.white, Color.gray));
m_progress.setMinimum(0);
JPanel p1 = new JPanel(new BorderLayout());
p1.add(m_progress, BorderLayout.CENTER);
p.add(p1);

p.add(new DialogSeparator());
m_btPut = new JButton("Put");
ActionListener lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    if (connect()) {
      Thread uploader = new Thread() {
        public void run() {
          putFile();
          disconnect();
        }
      };
      uploader.start();
    }
  }
};
m_btPut.addActionListener(lst);
m_btPut.setMnemonic('p');
p.add(m_btPut);

m_btGet = new JButton("Get");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    if (connect()) {
      Thread downloader = new Thread() {
        public void run() {
          getFile();
          disconnect();
        }
      };
      downloader.start();
    }
  }
};
m_btGet.addActionListener(lst);
m_btGet.setMnemonic('g');
p.add(m_btGet);

m_btFile = new JButton("File");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    if (m_chooser.showSaveDialog(FTPApp.this) !=
      JFileChooser.APPROVE_OPTION)
        return;
    File f = m_chooser.getSelectedFile();
    m_txtFile.setText(f.getPath());
  }
};
```

```
    m_btFile.addActionListener(lst);
    m_btFile.setMnemonic('f');
    p.add(m_btFile);

    m_btClose = new JButton("Close");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if (m_client != null)
          disconnect();
        else
          System.exit(0);
      }
    };
    m_btClose.addActionListener(lst);
    m_btClose.setDefaultCapable(true);
    m_btClose.setMnemonic('g');
    p.add(m_btClose);

    getContentPane().add(p, BorderLayout.CENTER);

    m_chooser = new JFileChooser();
    m_chooser.setCurrentDirectory(new File("."));
    m_chooser.setApproveButtonToolTipText(
      "Select file for upload/download");

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        disconnect();
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setSize(340,340);
    setResizable(false);
    setVisible(true);
  }

  public void setButtonStates(boolean state) {
    m_btPut.setEnabled(state);
    m_btGet.setEnabled(state);
    m_btFile.setEnabled(state);
  }

  protected boolean connect() {
    m_monitor.setText("");
    setButtonStates(false);
    m_btClose.setText("Cancel");
    setCursor(Cursor.getPredefinedCursor(
      Cursor.WAIT_CURSOR));

    String user = m_txtUser.getText();
    if (user.length()==0) {
      message("Please enter user name");
      setButtonStates(true);
      return false;
    }
    String password = new String(m_txtPassword.getPassword());
    String sUrl = m_txtURL.getText();
    if (sUrl.length()==0) {
      message("Please enter URL");
      setButtonStates(true);
      return false;
```

```
    }
    m_sLocalFile = m_txtFile.getText();

    // Parse URL
    int index = sUrl.indexOf("//");
    if (index >= 0)
    sUrl = sUrl.substring(index+2);

    index = sUrl.indexOf("/");
    String host = sUrl.substring(0, index);
    sUrl = sUrl.substring(index+1);

    String sDir = "";
    index = sUrl.lastIndexOf("/");
    if (index >= 0) {
      sDir = sUrl.substring(0, index);
      sUrl = sUrl.substring(index+1);
    }
    m_sHostFile = sUrl;

    try {
      message("Connecting to host "+host);
      m_client = new FtpClient(host);
      m_client.login(user, password);
      message("User "+user+" login OK");
      message(m_client.welcomeMsg);
      m_client.cd(sDir);
      message("Directory: "+sDir);
      m_client.binary();
      return true;
    }
    catch (Exception ex) {
      message("Error: "+ex.toString());
      setButtonStates(true);
      return false;
    }
  }

  protected void disconnect() {
    if (m_client != null) {
      try { m_client.closeServer(); }
      catch (IOException ex) {}
      m_client = null;
    }
    Runnable runner = new Runnable() {
      public void run() {
        m_progress.setValue(0);
        setButtonStates(true);
        m_btClose.setText("Close");
        FTPApp.this.setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
      }
    };
    SwingUtilities.invokeLater(runner);
  }

  protected void getFile() {
    if (m_sLocalFile.length()==0) {
      m_sLocalFile = m_sHostFile;
      SwingUtilities.invokeLater( new Runnable() {
        public void run() {
          m_txtFile.setText(m_sLocalFile);
        }
```

```
      });
    }
    byte[] buffer = new byte[BUFFER_SIZE];
    try {
      int size = getFileSize(m_client, m_sHostFile);
      if (size > 0) {
        message("File " + m_sHostFile + ": " + size + " bytes");
        setProgressMaximum(size);
      }
      else
        message("File " + m_sHostFile + ": size unknown");
      FileOutputStream out = new
        FileOutputStream(m_sLocalFile);
      InputStream in = m_client.get(m_sHostFile);
      int counter = 0;
      while(true) {
        int bytes = in.read(buffer);
        if (bytes < 0)
          break;

        out.write(buffer, 0, bytes);
        counter += bytes;
        if (size > 0) {
          setProgressValue(counter);
          int proc = (int) Math.round(m_progress.
            getPercentComplete() * 100);
          setProgressString(proc + " %");
        }
        else {
          int kb = counter/1024;
          setProgressString(kb + " KB");
        }
      }
      out.close();
      in.close();
    }
    catch (Exception ex) {
      message("Error: "+ex.toString());
    }
  }

  protected void putFile() {
    if (m_sLocalFile.length()==0) {
      message("Please enter file name");
    }
    byte[] buffer = new byte[BUFFER_SIZE];
    try {
      File f = new File(m_sLocalFile);
      int size = (int)f.length();
      message("File " + m_sLocalFile + ": " + size + " bytes");
      setProgressMaximum (size);
      FileInputStream in = new
        FileInputStream(m_sLocalFile);
      OutputStream out = m_client.put(m_sHostFile);

      int counter = 0;
      while(true) {
        int bytes = in.read(buffer);
        if (bytes < 0)
          break;
        out.write(buffer, 0, bytes);
        counter += bytes;
        setProgressValue(counter);
```

```
      int proc = (int) Math.round(m_progress.
        getPercentComplete() * 100);
      setProgressString(proc + " %");
    }

    out.close();
    in.close();
  }
  catch (Exception ex) {
    message("Error: " + ex.toString());
  }
}

protected void message(final String str) {
  if (str != null) {
    Runnable runner = new Runnable() {
      public void run() {
        m_monitor.append(str + '\n');
        m_monitor.repaint();
      }
    };
    SwingUtilities.invokeLater(runner);
  }
}

protected void setProgressValue(final int value) {
  Runnable runner = new Runnable() {
    public void run() {
      m_progress.setValue(value);
    }
  };
  SwingUtilities.invokeLater(runner);
}

protected void setProgressMaximum(final int value) {
  Runnable runner = new Runnable() {
    public void run() {
      m_progress.setMaximum(value);
    }
  };
  SwingUtilities.invokeLater(runner);
}

protected void setProgressString(final String string) {
  Runnable runner = new Runnable() {
    public void run() {
      m_progress.setString(string);
    }
  };
  SwingUtilities.invokeLater(runner);
}

public static void main(String argv[]) {
  new FTPApp();
}

public static int getFileSize(FtpClient client, String fileName)
 throws IOException {
  TelnetInputStream lst = client.list();
  String str = "";
  fileName = fileName.toLowerCase();
  while(true) {
    int c = lst.read();
```

```
      char ch = (char) c;
      if (c < 0 || ch == '\n') {
        str = str.toLowerCase();
        if (str.indexOf(fileName) >= 0) {
          StringTokenizer tk = new StringTokenizer(str);
          int index = 0;
          while(tk.hasMoreTokens()) {
            String token = tk.nextToken();
            if (index == 4)
              try {
                return Integer.parseInt(token);
              }
              catch (NumberFormatException ex) {
                return -1;
              }
            index++;
          }
        }
        str = "";
      }
      if (c <= 0)
        break;
      str += ch;
    }
    return -1;
  }
}
```

Understanding the Code

Class FTPApp

Class variable:

`int BUFFER_SIZE`: the size of the buffer used for input/ouput operations.

Instance variables:

`JTextField m_txtUser`: login user name textfield.

`JPasswordField m_txtPassword`: login password field.

`JTextField m_txtURL`: field for URL of file to be downloaded/uploaded on the remote site.

`JTextField m_txtFile`: field for file name of the file to be uploaded/downloaded on the local machine.

`JTextArea m_monitor`: used as a log to display various status messages.

`JProgressBar m_progress`: used to indicate the progress of an upload/download operation.

`JButton m_btPut`: used to initiate uploading.

`JButton m_btGet`: used to initiate downloading.

`JButton m_btFile`: used to bring up a file chooser dialog to choose a local file or specify a file name and location.

`JButton m_btClose`: used to close the application.

`JFileChooser m_chooser`: used to choose a local file or specify a file name and location.

`FtpClient m_client`: client connection to host which manages I/O operations.

`String m_sLocalFile`: name of the most recent local file involved in a data transfer.

`String m_sHostFile`: name of the most recent host file involved in a data transfer.

The FTPApp constructor first creates a panel using a DialogLayout2 layout manager, and instantiates and adds our four text fields with corresponding labels (recall that our DialogLayout2 manager requires that label/input field pairs are added in the specific label1, field1, label2, field2, etc., order). The m_monitor text area is created and placed in a JScrollPane, and separated from the label/field panel (by an instance of our custom DialogSeparator class titled "Connection Monitor"). The m_progress JProgressBar is created and placed in a JPanel with a BorderLayout to ensure that DialogLayout2 allows it to occupy the maximum width across the frame, as well as its preferred height).

A plain DialogSeparator is added below the progress bar, and our four buttons are created with attached ActionListeners, and added to the DialogLayout2 panel resulting in a horizontal row at the bottom of the frame. The button titled "Put" attempts to connect to a host using our connect() method. If it is successful a new thread is started which calls putFile() to upload a selected file, and then calls disconnect() to terminate the connection to the host. Similarly, the button titled "Get" attempts to connect to a host, and, if successful, starts a thread which calls our getFile() method to download a file, and then disconnect(). The button titled "File" brings up our JFileChooser dialog to select a local file or specify a new file name and location. The button titled "Close" invokes disconnect() to terminate a connection to the host if FTP transfer is in progress (i.e. if m_client instance is not null). If a transfer is not in progress the application is terminated.

The setButtonStates() method takes a boolean parameter and enables/disables the "Put," "Get," and "File" buttons accordingly.

The connect() method establishes a connection to the remote host and returns true in the case of success, or false otherwise. First this method disables our "Put," "Get," and "File" push buttons, and sets the text of the last button to "Cancel." Then this method reads the content of the text fields to obtain login name, password, URL, and local file name. The URL is parsed and split into the host name, remote directory, and the host file name. A new FtpClient instance is created to connect to the remote host and stored in our m_client instance variable. Then the login() method is invoked on m_client to login to the server using the specified user name and password. If the login is successful we change the remote directory and set the connection type to binary (which is almost always required for file transfers). If no exceptions have been thrown during this process, connect() returns true. Otherwise it shows an exception in our m_monitor text area, re-enables our buttons, and returns false.

Note: The connect() method code involving connecting to a host and changing directory would be better off in a separate thread, and we suggest this enhancement for more professional implementations. All other time-intensive code in this example is executed in separate threads.

The disconnect() method invokes closeServer() on the current m_client FTPClient instance if it is in use. It then sets the m_client reference to null, allowing garbage collection of the FTPClient object. This method also clears the progress bar component, enables all push buttons which may have been disabled by the connect() method, and restores the text of the "Close" button. Note that all component updates are wrapped in a Runnable and sent to the event-dispatching queue with SwingUtilities.invokeLater().

The getFile() method downloads a prespecified file from the current host. If the name of the destination local file is not specified, the name of the remote file is used. This method tries to determine the size of the remote file by calling our getFileSize() helper method (see below). If that succeeds, the file size is set as the maximum value of the progress bar (the minimum value is always 0) with our custom setProgressMaximum() method. Then a FileOutputStream is openned to write to the local file, and an InputStream is retrieved from the FTPClient to read from the remote file. A while loop is set up to perform typical read/write operations until all content of the remote file is written to the local file. During this process the number of bytes read is accumulated in the counter local variable. If the size of the file is known,

this number is assigned to the progress bar using our custom `setProgressValue()` method. We also calculate the percentage of downloading complete with our custom `getPercentComplete()` method, and display it in the progress bar using our custom `setProgressString()` method. If the size of file is unknown (i.e. it is less than or equal to `0`), we can only display the number of kilobytes currently downloaded at any given time. To obtain this value we simply divide the current byte count, stored in the local `counter` variable, by `1024`.

The `putFile()` method uploads the content of a local file to a remote pre-specified `URL`. If the name of the local file is not specified, a message is printed, using our custom `message()` method, and we simply return. Otherwise, the size of the local file is determined and used as the maximum value of our progress bar using our custom `setMaximum()` method (the minimum value is always `0`). A `FileInputStream` is openned to read from the local file, and an `OutputStream` is retrieved from the `FTPClient` to write to the remote file. A while loop is set up to perform typical read/write operations until all content of the local file is written to the remote host. During this process the number of bytes written is accumulated in the `counter` local variable. This number is assigned to the progress bar using our custom `setProgressValue()` method. As in the `getFile()` method, we also calculate the percentage of downloading complete with our custom `getPercentComplete()` method, and display it in the progress bar using our custom `setProgressString()` method. Since we can always determine the size of a `localFile` object, there is no need to display the progress in terms of kilobytes (as we did in `getFile()` above).

The `message()` method takes a `String` parameter to display in our `m_monitor` text area. The `setProgressValue()` and `setProgressMaximum()` methods assign selected and maximum values to our progress bar respectively. Since each of these methods modifes the state of our progress bar component, and each is called from a custom thread, we wrap their bodies in `Runnables` and send them to the event-dispatching queue using `SwingUtilities.invokeLater()`.

Unfortunately the `FtpClient` class does not provide a direct way to determine the size of a remote file, as well as any other available file specifics. The only way we can get any information about files on the remote host using this class is to call the its `list()` method which returns a `TelnetInputStream` supplying the printout of the results of an `ls -l` command. Our `getFileSize()` method uses this method in an attempt to obtain the length of a remote file specified by a given file name and `FTPClient` instance. This method captures the printout from the remote server, splits it into lines separated by '\n' characters, and uses a `StringTokenizer` to parse them into tokens. According to the syntax of the `ls -l` command output, the the length of the file in bytes appears as the 5th token, and the last token should contain the file name. So we go character by character through each line until a line containing a matching file name is found, and the length is returned to the caller. If this does not succeed we return -1 to indicate that the server either does not allow browsing of its content or that an error has occured.

Running the Code

Figure 13.8 shows `FTPApp` in action. Try running this application and transfering a few files. Start by entering your user name, password, `URL` containing the host FTP sever, and (optionally) a local file name and path to act as the source or destination of a transfer. Press the "Get" button to download a specified remote file, or press the "Put" button to upload a specified local file to the host. If the required connection is established successfully, you will see the transfer progress updated incrementally in the progress bar.

In figure 13.8 we specify "anonymous" as username and use an email address as password. In our `URL` text field we specify the remote "tutorial.zip" file (the most recent Java Tutorial) on the "java.sun.com" FTP server in its "docs" directory. In our `File` text field we specify "tutorial.zip" as the destination file in the current running directory. Clicking on "Get" establishes a connection, changes remote directory to "docs," determines the size of the remote "tutorial.zip" file, and starts retrieving and storing it as a local file in the current running directory. Try performing this transfer and note how smoothly the progress bar updates itself (it can't hurt to

keep a local copy of the Java Tutorial, but be aware that this archive is close to 10 megabytes).

Note: In the next chapter we will customize `JFileChooser` to build a ZIP/JAR archive tool. This can be used to unpackage tutorial.zip if you do not have access to an equivalent tool.

Chapter 14. Dialogs

In this chapter:

- Dialogs and choosers overview
- Adding an "About" dialog
- JOptionPane message dialogs
- Customizing JColorChooser
- Customizing JFileChooser

14.1 Dialogs overview

Swing's `JDialog` class allows implementation of both modal and non-modal dialogs. In simple cases, when we need to post a short message or ask for a single value input, we can use standardized pre-built dialog boxes provided by the `JOptionPane` convenience class. Additionally, two special dialog classes provide powerful selection and navigation capabilities for choosing colors and files: `JColorChooser` and `JFileChooser`.

UI Guideline: Usage Advice When to Use a Dialog

Dialogs are intended for the acquisition of sets of related data. This may be the set of attributes for a particular object or group of objects. A Dialog is particularly useful when validation across those attributes must be performed before the data can be accepted. The validation code can be executed when an "accept" button is pressed and the dialog will only dismiss when the data is validated as good.

Dialogs are also useful for complex manipulations or selections. For example a dialog with two lists, "Available Players" and "Team for Saturday's Game," might allow the selection, addition, and deletion of items to and/or form each list. When the team for the Saturday game is selected, the User can accept the selection by pressing "OK".

Data entry and complex data manipulation which requires a clear boundary or definition of acceptance, are good uses for a dialog.

When to Use an Option Pane

Option Panes are designed for use when the system needs to hold a conversation with the User, either for simple directed data entry such as "Enter your name and password" or for navigation choices such as "View", "Edit", "Print".

When to Use a Chooser

Choosers have been introduced to facilitate consistency for common selections across a whole operating environment. If you have a need to select files or colors then you ought to be using the appropriate chooser. The User gets the benefit of only learning one component which appears again and again across applications. Using a Chooser when appropriate should improve customer acceptance for your application.

14.1.1 JDialog

class javax.swing.JDialog

This class extends `java.awt.Dialog` and is used to create a new dialog box in a separate native platform window. We typically extend this class to create our own custom dialog, as it is a container almost identical to `JFrame`.

Note: `JDialog` is a `JRootPane` container just as `JFrame`, and familiarity with chapter 3 is assumed here. All `WindowEvents`, default close operations, sizing and positioning, etc., can be controlled identically to `JFrame` and we will not repeat this material.

We can create a `JDialog` by specifying a dialog owner (`Frame` or `Dialog` instances), a dialog title, and a modal/non-modal state. The following code shows a typical custom dialog class constructor:

```
class MyDialog extends JDialog
{
  public MyDialog(Frame owner) {
    super(owner, "Sample Dialog", true);
    // Perform GUI initialization here
  }
}
```

We are not required to pass a valid parent and are free to use `null` as the parent reference. As we discussed in chapter 2, the `SwingUtilities` class maintains a non-visible `Frame` instance registered with the `AppContext` service mapping, which is used as the parent of all null-parent dialogs. If a valid parent is used the dialog's icon will be that of the parent frame set with the `setIconImage()` method.

A modal dialog will not allow other windows to become active (i.e. respond to user input) at the same time it is active. Modal dialogs also block the invoking thread of execution and do not allow it to continue until they are dismissed. Non-modal dialogs do allow other windows to be active and do not affect the invoking thread.

To populate a dialog we use the same layout techniques discussed for `JFrame`, and we are prohibited from changing the layout or adding components directly. Instead we are expected to deal with the dialog's content pane:

```
JButton btn = new JButton("OK");
myDialog.getContentPane().add(btn);
```

From the design prospective, it is very common to add push buttons to a dialog. Typical buttons are "OK" or "Save" to continue with an action or save data, and "Cancel" or "Close" to close the dialog and cancel an action or avoid saving data.

Bug Alert! `JDialog` is not garbage collected even when it is disposed, made non-visible, and retains no direct references in scope. This 'feature' is not documented and is most likely an AWT or Swing bug.

Bug Alert! Because of certain AWT limitations, `JDialog` will only allow lightweight popup menus. `JPopupMenu`, and the components that use it, such as `JComboBox` and `JMenu`, can only be lightweight in modal dialogs. For this reason we need to be especially careful when implementing a modal dialog containing heavyweight components.

As with `JFrame`, `JDialog` will appear in the upper left-hand corner of the screen unless another location is

specified. It is usually more natural to center a dialog with respect to its owner, as follows:

```
Window owner = myDialog.getParent();
myDialog.setResizable(false);
Dimension d1 = myDialog.getSize();
Dimension d2 = owner.getSize();
int x = Math.max((d2.width-d1.width)/2, 0);
int y = Math.max((d2.height-d1.height)/2, 0);
myDialog.setBounds(x, y, d1.width, d1.height);
```

Note: It is common practice to show dialogs in response to menu selections. In such cases a menu's popup may remain visible and the parent frame needs to be manually repainted. For this reason we suggest calling `repaint()` on the parent before displaying dialogs invoked by menus.

To display a `JDialog` window we can use either the `show()` method inherited from `java.awt.Dialog`, or we can use the `setVisible()` method inherited from `java.awt.Component`. The following snippet code illustrates a typical scheme of creation and usage of the custom dialog box:

Reference: In chapter 20 we will show how to construct quite complex dialogs and assume knowledge of the `JDialog` information presented here. We suggest supplementing this chapter's material with a brief look at the chapter 20 examples to get a feel for what lies ahead.

Note: When building complex dialogs it is normally preferable that one instance of that dialog be used throughout a given Java session. We suggest instiating such dialogs when the application/applet is started, and storing them as variables for repetitive use. This avoids the often significantly long delay time required to instantiate a dialog each time it is needed. We also suggest wrapping dialog instantiation in a separate thread to avoid clogging up the AWT event dispatching thread.

14.1.2 JOptionPane

class javax.swing.JOptionPane

This class provides an easy and convenient way to display standard dialogs used for posting a message, asking a question, or prompting for simple user input. Note that each `JOptionPane` dialog is modal and will block the invoking thread of execution, as described above (this does not apply to internal dialogs; we will discuss these soon enough).

It is important to understand that `JOptionPane` is not itself a dialog (note that it directly extends `JComponent`). Rather, it acts as a container that is normally placed in a `JDialog` or a `JInternalFrame`, and provides several convenient methods for doing so. There is nothing stopping us from creating a `JOptionPane` and placing it in any container we choose, but this will rarely be useful. Figure 14.1 illustrates the general `JOptionPane` component arrangement:

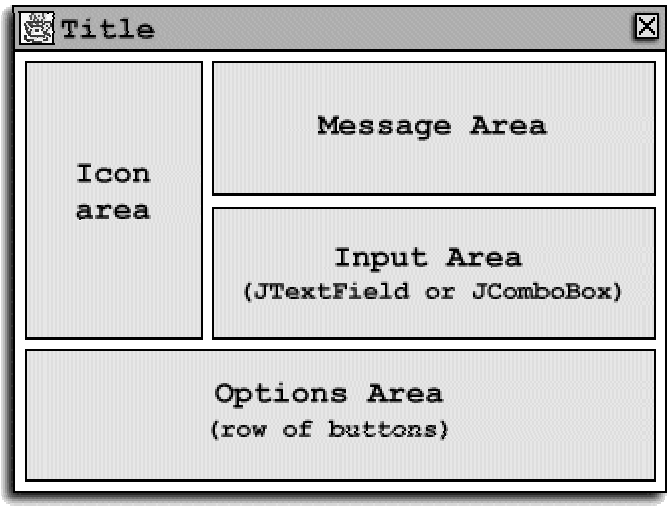


Figure 14-1 The components of a JOptionPane dialog.
<<file figure14-1.gif>>

JOptionPane class supports four pre-defined types: Message, Confirm, Input, and Option. We will discuss how to create and work with each type, but first we need to understand the constituents. To create a JOptionPane that is automatically placed in either a JDialog or JInternalFrame, we need to supply some or all of the following parameters to one if its static showXXDialog() methods (discussed below):

- A parentComponent. If the parent is Frame, the option pane will be placed in a JDialog and be centered with respect to the parent. If this parameter is null, it will instead be centered with respect to the screen. If the parent is a JDesktopPane, or is contained in one, the opyion pane will be contained in a JInternalFrame and placed in the parent desktop's MODAL_LAYER (see chapter 15). For other types of parent components a JDialog will be used and placed below that component on the screen.
- A message Object: a message to be displayed in the center of the pane (in the Message area). Typically this is a String, which may be broken into separate lines using '\n' characters. However this parameter has a generic Object type and JOptionPane deals with non-String objects in the following way:

  Icon: will be displayed in a JLabel.

  Component: will simply be placed in the message area.

  Object[]: dealt with as described here, these will be placed vertically in a column (this is done recursively).

  Object: the toString() method will be called to convert this to a String for display in a JLabel
- An int message type: can be one of the following static constants defined in JOptionPane: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE. This is used by the current L&F to customize the look of an option pane by displaying an appropriate icon (in the Icon area) corresponding to the message type's meaning (see below).
- An int option type: can be one of the following static constants defined in JOptionPane: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION. This parameter specifies a set of corresponding buttons to be displayed at the bottom of the pane (in the Options area). Another set of similar parameters returned from JOptionPane showXXDialog() methods (see below) specify which button was pressed: CANCEL_OPTION, CLOSED_OPTION, NO_OPTION, OK_OPTION, YES_OPTION. Note that CLOSED_OPTION is only returned when the pane is contained in a JDialog or JInternalFrame, and that container's close button (located in the title bar) is pressed.

- An `Icon`: displayed in the left side of the pane (in the Icon area). If not explicitly specified, the icon is determined by the current L&F based on the message type (this does not apply to panes using the `PLAIN_MESSAGE` message type).
- An array of option `Objects`: we can directly specify an array of option `Objects` to be displayed at the bottom of the pane (in the Options area). This array can also be specified with the `setOptions()` method. It typically contains an array of `Strings` to be displayed on a `JButtons`, but `JOptionPane` also honors `Icons` (also displayed in `JButtons`) and `Components` (placed directly in a row). Similar to message `Objects`, the `toString()` method will be called to convert all objects that are not `Icon`'s or `Components` to a `String` for display in a `JButton`. If `null` is used, the option buttons are determined by the specified option type.
- An initial value `Object`: specifies which button or component in the Options area has the focus when the pane is initially displayed.
- An array of selection value `Objects`: specifies an array of allowed choices the user can make. If this array contains more than twenty items a `JList` is used to display them using the default rendering behavior (see chapter 10). If the array contains less than twenty items a `JComboBox` is used to display them (also using the default `JList` rendering behavior). If `null` is used, an empty `JTextField` is displayed. In any case, the component used for selection is placed in the Input area.
- A `String` title: used for display as the titlebar title in the containing `JDialog` or `JInternalFrame`.

The following static methods are provided for convenient creation of `JOptionPanes` placed in `JDialogs`:

- `showConfirmDialog()`: this method displays a dialog with several buttons and returns an `int` option type corresponding to the button pressed (see above). Four overloaded methods are provided allowing specification of, at most, a parent component, message, title, option type, message type, and icon.
- `showInputDialog()`: this method displays a dialog which is intended to receive user input, and returns a `String` (if the input component is a text field) or an `Object` if the input component is a list or a combo box (see above). Four overloaded methods are provided allowing specification of, at most, a parent component, message, title, option type, message type, icon, array of possible selections, and an initially selected item. Two buttons are always displayed in the Options area: "OK" and "Cancel."
- `showMessageDialog()`: this method displays a dialog with an "OK" button, and doesn't return anything. Three overloaded methods are provided allowing specification of, at most, a parent component, message, title, message type, and icon.
- `showOptionDialog()`: this method displays a dialog which can be customized a bit more than the above dialogs, and returns either the index into the array of option `Objects` specified, or an option type if no option `Objects` are specified. Only one method is provided allowing specification of a parent component, message, title, option type, message type, icon, array of option `Objects`, and an option `Object` with the initial focus. The option `Objects` are layed out as a row in the Options area.

To create `JOptionPanes` contained in `JInternalFrames` rather than `JDialogs`, we can use the `showInternalConfirmDialog()`, `showInternalInputDialog()`, `showInternalMessageDialog()`, and `showInternalOptionDialog()` overloaded methods. These work the same as the methods described above, only they expect that a given parent is a `JDesktopPane` (or has a `JDesktopPane` ancestor).

Note: Internal dialogs are not modal and, as such, do not block execution of the invoking thread.

Alternatively, we can directly create a `JOptionPane` as illustrated by the following psuedo-code:

```
JOptionPane pane = new JOptionPane(...); // Specify parameters
```

```
pane.setXX(...); // Set additional parameters

JDialog dialog = pane.createDialog(parent, title);
dialog.show();

Object result = pane.getValue();
// Process result (may be null)
```

This code creates an instance of `JOptionPane` and specifies several parameters (see API docs). Additional settings are then provided with `setXX` accessor methods. The `createDialog()` method creates a `JDialog` instance containing our `JOptionPane` which is then displayed (we could also have used the `createInternalFrame()` method to wrap our pane in a `JInternalFrame`). Finally, the `getValue()` method retrieves the option selected by user, so the program may react accordingly. This value may be `null` (if the user closes dialog window). Note that because program execution blocks until the dialog is dismissed, `getValue()` will not be called until a selection is made.

Note: The advantage of `JOptionPane` is its simplicity and convenience. In general, there should be no need to customize it to any large extent. If you find yourself desiring a different layout we suggest writing your own container instead.

Figure 14.2 illustrates the use of `JOptionPane` where a custom dialog may be more suitable. Note the extremely long button, text field, and combo box. Such extreme sizes have detrimental affects on the overall usability and appearance of an app.

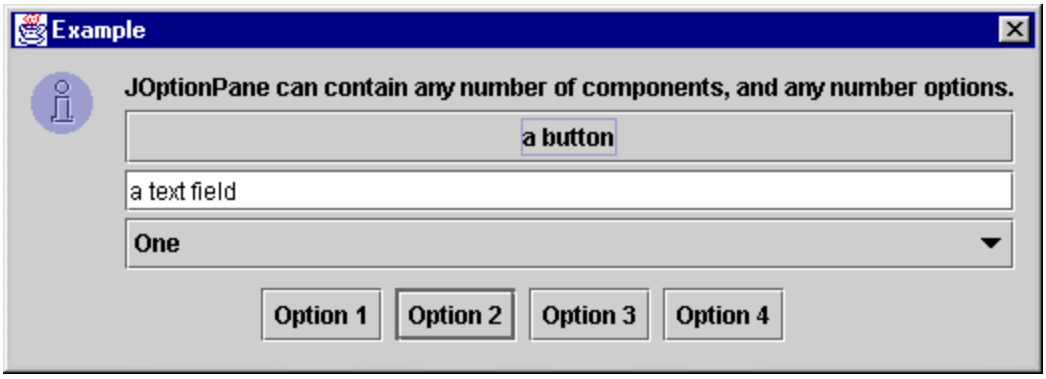


Figure 14.2 Awkward use of components in a JOptionPane.
<<file figure14-2.gif>>

UI Guideline: Usage of JOptionPane

JOptionPane is not designed as a general purpose input dialog. The primary restriction is the defined layout. JOptionPane is designed for use in conversations between the System and the User where the desired result is a navigation choice or a data selection, or where the User must be notified of an event.

Therefore, JOptionPane is best used with a single entry field or combobox selection, possibly with a set of buttons for selection or navigational choice.

For example an Answer Phone application might require an option dialog displaying "You have 1 message," with options "Play," "Save," "Record outgoing message," and "Delete messages." Such a requirement can be met with a JOptionPane which provides a single label for the message and 4 buttons for each of the choices available.

14.1.3 JColorChooser

class javax.swing.JColorChooser

This class represents a powerful, pre-built component used for color selection. `JColorChooser` is normally used in a modal dialog. It consists of tabbed pane containing three panels each offering a different method of choosing a color: Swatches, HSB, and RGB. A color preview pane is displayed below this tabbed pane and always displays the currently selected color. Figure 14.3 illustrates.

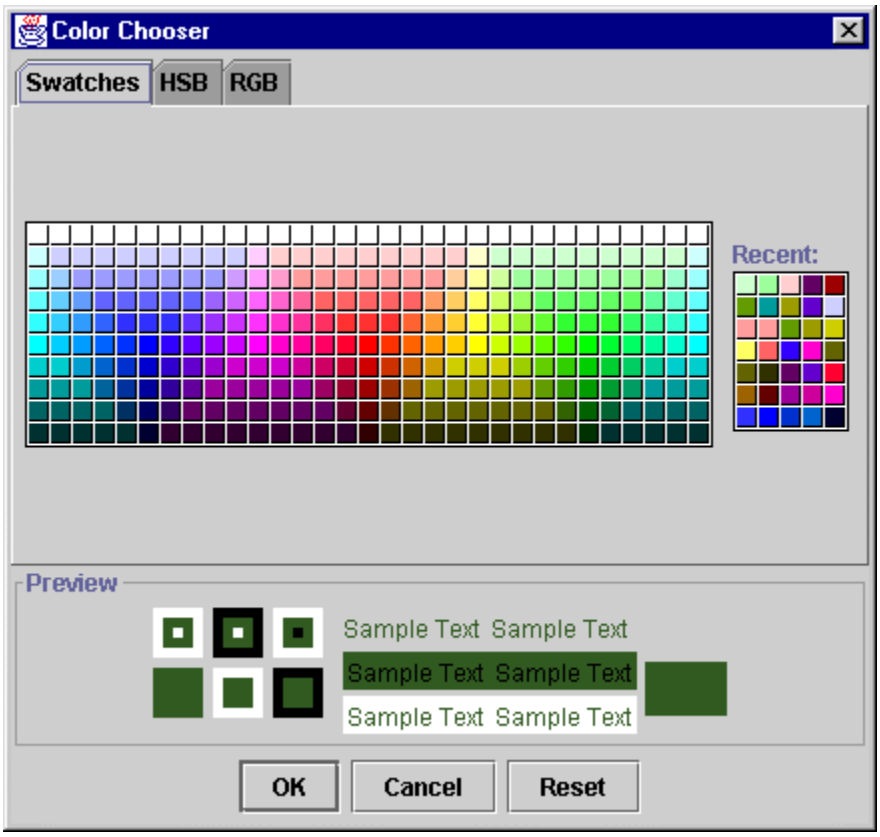


Figure 14.3 `JColorChooser` in a `JDialog`.
<<file figure14-3.gif>>

The static `showDialog()` method instatiates and displays a `JColorChooser` in a modal dialog, and returns the selected `Color` (or `null` if no selection is made):

```
Color color =  JColorChooser.showDialog(myComponent,
  "Color Chooser", Color.red);
if (color != null)
  myComponent.setBackground(c);
```

A more complex variant is the static `createDialog()` method which allows specification of two `ActionListeners` to be invoked when a selection is made or cancelled respectively. We can also do the following:

Retrieve color selection panels with the `getChooserPanels()` method and use them outside the dialog.

Add custom color selection panels using the `addChooserPanel()` method.

Assign a new custom color preview pane using the `setPreviewPanel()` method.

Several classes and interfaces supporting `JColorChooser` are grouped into the `javax.swing.colorchooser` package.

14.1.4 The ColorSelectionModel interface

abstract interface javax.swing.colorchooser.ColorSelectionModel

This is a simple interface describing the color selection model for `JColorChooser`. It declares methods for adding and removing `ChangeListeners` which are intended to be notified when the selected `Color` changes, and `getSelectedColor()`/`setSelectedColor()` accessors to retrieve and assign the currently selected `Color` respectively.

14.1.5 DefaultColorSelectionModel

class javax.swing.colorchooser.DefaultColorSelectionModel

This is the default concrete implementation of the `ColorSelectionModel` interface. It simply implements the necessary methods as expected, stores registered `ChangeListeners` in an `EventListenerList`, and implements an additional method to perform the actual firing of `ChangeEvents` to all registered listeners.

14.1.6 AbstractColorChooserPanel

abstract class javax.swing.colorchooser.AbstractColorChooserPanel

This abstract class describes a color chooser panel which can be added to `JColorChooser` as a new tab. We can sub-class `AbstractColorChooserPanel` to implement a custom color chooser panel of our own. The two most important methods that must be implemented are `buildChooser()` and `updateChooser()`. The former is normally called only once at instantiation time and is intended to perform all GUI initialization tasks. The latter is intended to update the panel to reflect a change in the associated `JColorChooser`'s `ColorSelectionModel`. Other required methods include those allowing access to a display name and icon used to identify the panel when it is displayed in `JColorChooser`'s tabbed pane.

14.1.7 ColorChooserComponentFactory

class javax.swing.colorchooser.ColorChooserComponentFactory

This is a very simple class that is responsible for creating and returning instances of the default color chooser panels and preview pane used by `JColorChooser`. The three color chooser panels are instances of private classes: `DefaultSwatchChooserPanel`, `DefaultRGBChooserPanel`, `DefaultHSBChooserPanel`. The preview pane is an instance of `DefaultPreviewPane`. Other private classes used in this package include two custom layout managers, `CenterLayout` and `SmartGridLayout`, a class for convenient generation of synthetic images, `SyntheticImage`, and a custom text field that only allows integer input, `JIntegerTextField`. These undocumented classes are very interesting and we urge curious readers to spend some time with the source code. Because they are only used within the `colorchooser` package and are defined as package private, we will not discuss them further here.

14.1.8 JFileChooser

class javax.swing.JFileChooser

This class represents the standard Swing directory navigation and file selection component which is normally used in a modal dialog. It consists of a `JList` and several button and input components all linked together offering functionality similar to the file dialogs we are used to on our native platforms. The `JList` is used to

display a list of files and sub-directories residing in the current directory being navigated. Figure 14.4 illustrates.

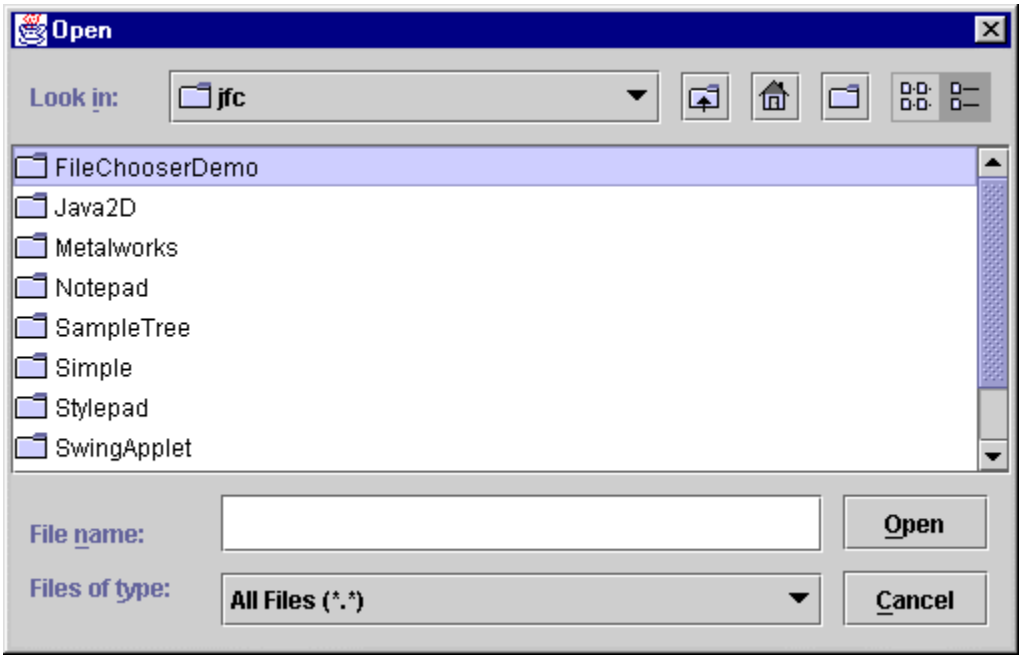


Figure 14.4 JFileChooser in a JDialog.
<<file figure14-4.gif>>

UI Guideline: Cross Application Consistency

The key reason for promoting the use of a standard file chooser dialog is to promote the consistency of such an operation across the whole operating system or machine environment. The User experience is improved because file selection is always the same no matter which application they are running. This is an important goal and worthy of recognition. Thus, if you have a requirement to manipulate files, you ought to be using the JFileChooser component.

The fact that such a re-usable component exists and much of the complex coding is provided as part of the implementation, is merely a bonus for the developer.

We can set the current directory by passing a `String` to its `setCurrentDirectory()` method. `JFileChooser` also has the ability to use special `FileFilters` (discussed below) to only allow navigation of certain types of files. Several properties control whether directories and/or files can be navigated and selected, and how the typical "Open" (approve) and "Cancel" (cancel) buttons are represented (see the API docs for more on these straightforward methods.)

To use this component we normally create an instance of it, set the desired options, and call `showDialog()` to place it in an active modal dialog. This method takes the parent component and the text to display for its approve button, as parameters. Calling `showOpenDialog()` or `showSaveDialog()` will show a modal dialog with "Open" or "Save" for the approve button text respectively.

Note: `JFileChooser` can take a significant amount of time to instantiate. Please consider storing an instance as a variable and performing instantiation in a separate thread at startup time, as discussed in a note above.

The following code instantiates a `JFileChooser` in an "Open" file dialog, verifies that a valid file is selected, and retrieves that file as a `File` instance:

```
JFileChooser chooser = new JFileChooser();
```

```
chooser.setCurrentDirectory(".");
if (chooser.showOpenDialog(myComponent) !=
  JFileChooser.APPROVE_OPTION)
    return;
File file = chooser.getSelectedFile();
```

JFileChooser generates PropertyChangeEvents when any of its properties change state. The approve and cancel buttons generate ActionEvents when pressed. We can register PropertyChangeListeners and ActionListeners to receive these events respectively. As any well-defined JavaBean should, JFileChooser defines several static String constants corresponding to each property name, e.g. JFileChooser.FILE_FILTER_CHANGED_PROPERTY (see API docs for a full listing). We can use these constants in determining which property a JFileChooser-generated PropertyChangeEvent corresponds to.

JFileChooser also supports the option of inserting an accessory component. This component can be any component we like and will be placed to the right of the JList. In constructing such a component, we are normally expected to implement the PropertyChangeListener interface. This way the component can be registered with the associated JFileChooser to receive notification of property state changes. The component should use these events to update its state accordingly. We use the setAccessory() method to assign an accessory component to a JFileChooser, and addPropertyChangeListener() to register it for receiving property state change notification.

Reference: For a good example of an accessory component used to preview selected images, see the FileChooserDemo example that ships with Java 2. In the final example of this chapter we will show how to customize JFileChooser in a more direct manner.

Several classes and interfaces related to JFileChooser are grouped into javax.swing.filechooser package.

Note: JFileChooser is still fairly immature. For example, multi-selection mode is specified but has not been implemented yet. Later in this chapter we will show how to work around this, as well as how to build our own accessory-like component in a different location than that of a normal accessory as described above.

14.1.9 FileFilter

abstract class javax.swing.filechooser.FileFilter

This abstract class is used to implement a filter for displaying only certain file types in JFileChooser. Two methods must be implemented in concrete sub-classes:

- boolean accept(File f): returns true if the given file should be displayed, false otherwise.
- String getDescription(): returns a description of the filter used in the JComboBox at the bottom of JFileChooser.

To manage FileFilters we can use several methods in JFileChooser, including:

- addChoosableFileFilter(FileFilter f) to add a new filter.
- removeChoosableFileFilter(FileFilter f) to remove an existing filter.
- setFileFilter(FileFilter f) to set a filter as currently active (and append it, if necessary).

By default JFileChooser receives a filter accepting all files. Special effort must be made to remove this

filter if we do not desire our application to accept all files:

```
FileFilter ft = myChooser.getAcceptAllFileFilter();
myChooser.removeChoosableFileFilter(ft);
```

So how do we create a simple file filter instance to only allow navigation and selection of certain file types? The following class can be used as template for defining most of our own filters, and we will see it used in future chapters:

```
class SimpleFilter extends FileFilter
{
  private String m_description = null;
  private String m_extension = null;

  public SimpleFilter(String extension, String description) {
    m_description = description;
    m_extension = "." + extension.toLowerCase();
  }

  public String getDescription() {
    return m_description;
  }

  public boolean accept(File f) {
    if (f == null)
      return false;
    if (f.isDirectory())
      return true;
    return f.getName().toLowerCase().endsWith(m_extension);
  }
}
```

Note that we always return true for directories because we normally always want to be able to navigate any directory. This filter only shows files matching the given extension `String` passed into our constructor and stored as variable `m_extension`. In more robust, multi-purpose filters we might store an array of legal extensions, and check for each in the `accept()` method. Also note that the description `String` passed into the constructor, and stored as variable `m_description`, is the `String` shown in the combo box at the bottom of `JFileChooser` representing the corresponding file type. `JFileChooser` can maintain multiple filters, all added using the `addChoosableFileFilter()` method, and removable with its `removeChoosableFileFilter()` method.

14.1.10 FileSystemView

abstract class javax.swing.filechooser.FileSystemView

This class encapsulates functionality which extracts information about files, directories, and partitions, and supplies this information to the `JFileChooser` component. This class is used to make `JFileChooser` independent from both platform-specific file system information, as well as the JDK/Java 2 release version (since the JDK 1.1 `File` API doesn't allow access to some more specific file information available in Java 2). We can provide our own `FileSystemView` sub-class and assign it to a `JFileChooser` instance using the `setFileSystemView(FileSystemView fsv)` method. Four abstract methods must be implemented:

- `createNewFolder(File containingDir)`: creates a new folder (directory) within the given folder.
- `getRoots()`: returns all root partitions. The notion of a root differs significantly from platform to platform.
- `isHiddenFile(File f)`: returns whether or not the given `File` is hidden.
- `isRoot(File f)`: returns whether or not the given `File` is a partition or drive.

These methods are called by JFileChooser and FileFilter implementations and we will, in general, have no need to extend this class unless we need to tweak the way JFileChooser interacts with our OS. The static getFileSystemView() method currently returns a Unix or Windows specific instance for use by JFileChooser in the most likely event that one of these platform types is detected. Otherwise, a generic instance is used. Support for Macintosh, OS2, and several other operating systems is expected to be provided here in future releases.

14.1.11 FileView

abstract class javax.swing.filechooser.FileView

This abstract class can be used to provide customized information about files and their types (typically determined by the file extension), including icons and a string description. Each L&F provides its own sub-class of FileView, and we can construct our own fairly easily. Each of the five methods in this class is abstract and must be implemented by sub-classes. The following generalized template can be used when creating our own FileViews:

```
class MyExtView extends FileView
{
  // Store icons to use for list cell renderer.
  protected static ImageIcon MY_EXT_ICON =
    new ImageIcon("myexticon.gif");
  protected static ImageIcon MY_DEFAULT_ICON =
    new ImageIcon("mydefaulticon.gif");

  // Return the name of a given file. "" corresponds to
  // a partition, so in this case we must return the path.
  public String getName(File f) {
    String name = f.getName();
    return name.equals("") ? f.getPath() : name;
  }

  // Return the description of a given file.
  public String getDescription(File f) {
    return getTypeDescription(f);
  }

  // Return the String to use for representing each specific
  // file type.  (Not used by JFileChooser in Java 2 FCS)
  public String getTypeDescription(File f) {
    String name = f.getName().toLowerCase();
    if (name.endsWith(".ext"))
      return "My custom file type";
    else
      return "Unrecognized file type";
  }

  // Return the icon to use for representing each specific
  // file type in JFileChooser's JList cell renderer.
  public Icon getIcon(File f) {
    String name = f.getName().toLowerCase();
    if (name.endsWith(".ext"))
      return MY_EXT_ICON;
    else
      return MY_DEFAULT_ICON;
  }

  // Normally we should return true for directories only.
```

```
    public Boolean isTraversable(File f) {
      return (f.isDirectory() ? Boolean.TRUE : Boolean.FALSE);
    }
  }
```

We will see how to build a custom `FileView` for JAR and ZIP archive files in the final example of this chapter.

14.2 Adding an "About" dialog

Most GUI applications have at least one "About" dialog, usually modal, which often displays copyright, company, and other important information such as product name, version number, authors, etc. The following example illustrates how to add such a dialog to our text editor example developed in chapter 12. We build a sub-class of `JDialog`, populate it with some simple components, and store it as a variable which can be shown and hidden indefinitely without having to instantiate a new dialog each time it is requested. We also implement centering so that whenever it is shown it will appear in the center of our application's frame.

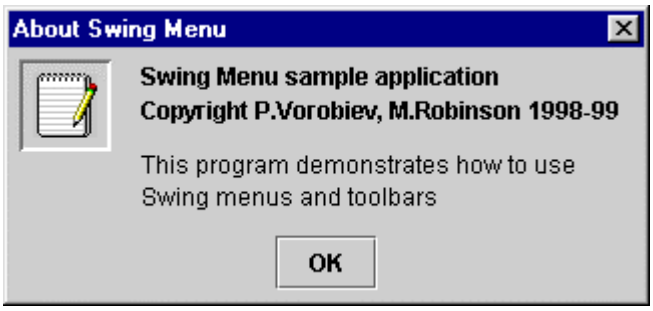


Figure 14.5 A typical "About" custom `JDialog` with dynamic centering.
<<file figure14-5.gif>>

The Code: BasicTextEditor.java
see \Chapter14\1

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

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class BasicTextEditor extends JFrame
{
  // Unchanged code from section 12.5
  protected AboutBox m_dlg;

  public BasicTextEditor() {
    super("\"About\" BasicTextEditor");
    setSize(450, 350);
    ImageIcon icon = new ImageIcon("smallIcon.gif");
    setIconImage(icon.getImage());

    // Unchanged code from section 12.5

    updateMonitor();
    m_dlg = new AboutBox(this);
    setVisible(true);
  }
```

```
  protected JMenuBar createMenuBar() {
    // Unchanged code from section12.5

    JMenu mHelp = new JMenu("Help");
    mHelp.setMnemonic('h');
    Action actionAbout = new AbstractAction("About") {
      public void actionPerformed(ActionEvent e) {
        Dimension d1 = m_dlg.getSize();
        Dimension d2 = BasicTextEditor.this.getSize();
        int x = Math.max((d2.width-d1.width)/2, 0);
        int y = Math.max((d2.height-d1.height)/2, 0);
        m_dlg.setBounds(x+BasicTextEditor.this.getX(),
          y+ BasicTextEditor.this.getY(),
          d1.width, d1.height);
        m_dlg.show();
      }
    };
    item =  mHelp.add(actionAbout);
    item.setMnemonic('a');
    menuBar.add(mHelp);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);
    return menuBar;
  }
  // Unchanged code
}

class AboutBox extends JDialog
{
  public AboutBox(Frame owner) {
    super(owner, "About Swing Menu", true);

    JLabel lbl = new JLabel(new ImageIcon("icon.gif"));
    JPanel p = new JPanel();
    Border b1 = new BevelBorder(BevelBorder.LOWERED);
    Border b2 = new EmptyBorder(5, 5, 5, 5);
    lbl.setBorder(new CompoundBorder(b1, b2));
    p.add(lbl);
    getContentPane().add(p, BorderLayout.WEST);

    String message = "Swing Menu sample application\n"+
      "Copyright P.Vorobiev, M.Robinson 1998-99";
    JTextArea txt = new JTextArea(message);
    txt.setBorder(new EmptyBorder(5, 10, 5, 10));
    txt.setFont(new Font("Helvetica", Font.BOLD, 12));
    txt.setEditable(false);
    txt.setBackground(getBackground());
    p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));
    p.add(txt);

    message = "This program demonstrates how to use\n"+
      "Swing menus and toolbars";
    txt = new JTextArea(message);
    txt.setBorder(new EmptyBorder(5, 10, 5, 10));
    txt.setFont(new Font("Arial", Font.PLAIN, 12));
    txt.setEditable(false);
    txt.setBackground(getBackground());
    p.add(txt);

    getContentPane().add(p, BorderLayout.CENTER);
```

```
    JButton btOK = new JButton("OK");
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        setVisible(false);
      }
    };
    btOK.addActionListener(lst);
    p = new JPanel();
    p.add(btOK);
    getContentPane().add(p, BorderLayout.SOUTH);

    pack();
    setResizable(false);
  }
}
```

Understanding the Code

Class BasicTextEditor

New instance variable:

`AboutBox m_dialog`: used to reference an instance of our custom "About" dialog class.

The constructor now assigns a custom icon to the application frame, instantiates an `AboutBox`, and directs our `m_dlg` reference to the resulting instance. In this way we can simply show and hide the dialog as necessary, without having to build a new instance each time it is requested. Notice that we use a `this` reference for its parent because `BasicTextEditor` extends `JFrame`.

We also modify the `createMenuBar()` method by adding a new "Help" menu containing an "About" menu item. This menu item is created as an `Action` implementation, and its `actionPerformed()` method determines the dimensions of our dialog as well as where it should be placed on the screen so that it will appear centered relative to its parent.

Class AboutBox

This class extends `JDialog` to implement our custom "About" dialog. The constructor creates a modal `JDialog` instance titled "About Swing Menu," and populates it with some simple components. A large icon is placed in the left side and two `JTextAreas` are placed in the center to display multiline text messages with different fonts. A push button titled "OK" is placed at the bottom. Its `ActionListener`'s `actionPerformed()` method invokes `setVisible(false)` when pressed.

Note: We could have constructed a similar "About" dialog using a `JOptionPane` message dialog. However, the point of this example is to demonstrate the basics of custom dialog creation, which we will be using later in chapter 20 to create several complex custom dialogs that could not be derived from `JOptionPane`.

Running the Code

Select the "About" menu item which brings up the dialog shown in figure 14.5. This dialog serves only to display information, and has no functionality other than the "OK" button which hides it. Note that no matter where the parent frame lies on the screen, when the dialog is invoked it appears centered. Also note that displaying the dialog is very fast because we are working with the same instance throughout the application's lifetime. This is, in general, a common practice that should be adhered to.

14.3 JOptionPane message dialogs

Message dialogs provided by the `JOptionPane` class can be used for many purposes in Swing apps: to post a message, ask a question, or get simple user input. The following example brings up several message boxes of different types with a common Shakespeare theme. Both internal and regular dialogs are constructed, demonstrating how to use the convenient `showXXDialog()` methods (see 14.1.2), as well as how to manually create a `JOptionPane` component and place it in a dialog or internal frame for display.

Each dialog is instantiated as needed and we perform no caching here (for purposes of demonstration). A more professional implementation might instantiate each dialog at startup and store them as variables for use throughout the application's lifetime.

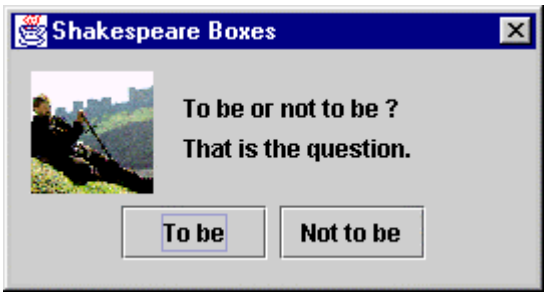


Figure 14.6 JOptionPane with custom icon, message, and option button strings in a JDialog.
<<file figure14-6.gif>>

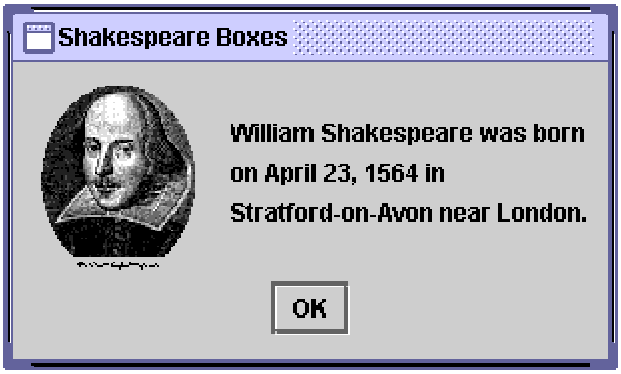


Figure 14.7 JOptionPane with custom icon and message in a JInternalFrame.
<<file figure14-7.gif>>

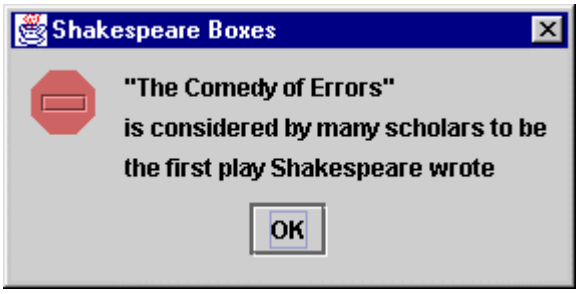


Figure 14.8 JOptionPane ERROR_MESSAGE message dialog with multi-line message.
<<file figure14-8.gif>>

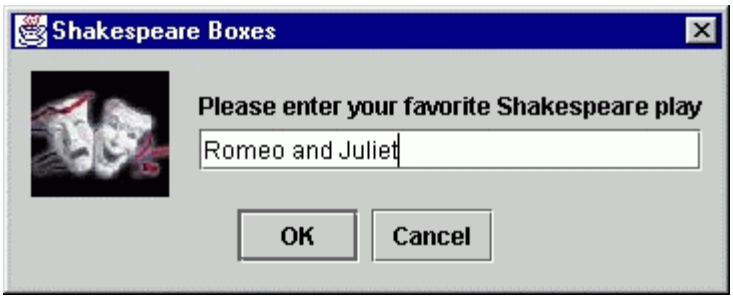


Figure 14.9 JOptionPane INFORMATION_MESSAGE input dialog with custom icon, message, text field input, and initial selection.

<<file figure14-9.gif>>

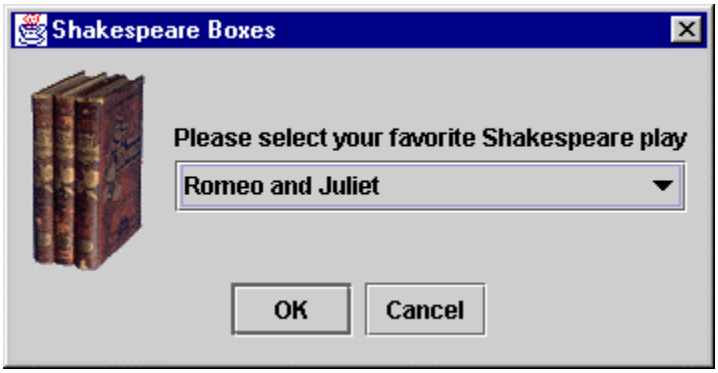


Figure 14.10 JOptionPane INFORMATION_MESSAGE input dialog with custom icon, message, combo box input, and initial selection.

<<file figure14-10.gif>>

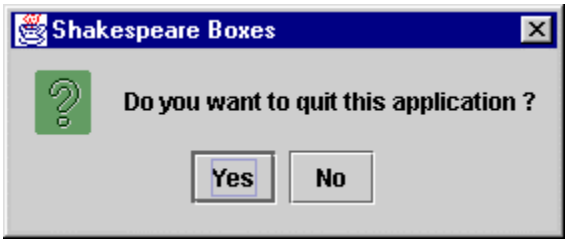


Figure 14.11 JOptionPane YES_NO_OPTION confirm dialog.

<<file figure14-11.gif>>

The Code: DialogBoxes.java
see \Chapter14\2

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

import javax.swing.*;

public class DialogBoxes extends JFrame
{
  static final String BOX_TITLE = "Shakespeare Boxes";

  public DialogBoxes() {
    super(BOX_TITLE);
    setSize(400,300);
    setLayeredPane(new JDesktopPane());

    JMenuBar menuBar = createMenuBar();
    setJMenuBar(menuBar);
```

```
    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  protected JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();

    JMenu mFile = new JMenu("File");
    mFile.setMnemonic('f');

    JMenuItem mItem = new JMenuItem("Ask Question");
    mItem.setMnemonic('q');
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        JOptionPane pane = new JOptionPane(
          "To be or not to be ?\nThat is the question.");
        pane.setIcon(new ImageIcon("Hamlet.gif"));
        Object[] options =
          new String[] {"To be", "Not to be"};
        pane.setOptions(options);
        JDialog dialog = pane.createDialog(
        DialogBoxes.this, BOX_TITLE);
        dialog.show();
        Object obj = pane.getValue();
        int result = -1;
        for (int k=0; k<options.length; k++)
          if (options[k].equals(obj))
        result = k;
        System.out.println("User's choice: "+result);
      }
    };
    mItem.addActionListener(lst);
    mFile.add(mItem);

    mItem = new JMenuItem("Info Message");
    mItem.setMnemonic('i');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String message = "William Shakespeare was born\n"+
          "on April 23, 1564 in\n"
          +"Stratford-on-Avon near London";
        JOptionPane pane = new JOptionPane(message);
        pane.setIcon(new ImageIcon("Shakespeare.gif"));
        JInternalFrame frame = pane.createInternalFrame(
          (DialogBoxes.this).getLayeredPane(), BOX_TITLE);
        getLayeredPane().add(frame);
      }
    };
    mItem.addActionListener(lst);
    mFile.add(mItem);

    mItem = new JMenuItem("Error Message");
    mItem.setMnemonic('e');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String message = "\"The Comedy of Errors\"\n"+
```

```
      "is considered by many scholars to be\n"+
      "the first play Shakespeare wrote";
    JOptionPane.showMessageDialog(
      DialogBoxes.this, message,
      BOX_TITLE, JOptionPane.ERROR_MESSAGE);
  }
};
mItem.addActionListener(lst);
mFile.add(mItem);

mFile.addSeparator();

mItem = new JMenuItem("Text Input");
mItem.setMnemonic('t');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    String input = (String) JOptionPane.showInputDialog(
      DialogBoxes.this,
      "Please enter your favorite Shakespeare play",
      BOX_TITLE, JOptionPane.INFORMATION_MESSAGE,
      new ImageIcon("Plays.jpg"), null,
      "Romeo and Juliet");
    System.out.println("User's input: "+input);
  }
};
mItem.addActionListener(lst);
mFile.add(mItem);

mItem = new JMenuItem("Combobox Input");
mItem.setMnemonic('c');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    String[] plays = new String[] {
      "Hamlet", "King Lear", "Otello", "Romeo and Juliet" };
    String input = (String) JOptionPane.showInputDialog(
      DialogBoxes.this,
      "Please select your favorite Shakespeare play",
      BOX_TITLE, JOptionPane.INFORMATION_MESSAGE,
      new ImageIcon("Books.gif"), plays,
      "Romeo and Juliet");
    System.out.println("User's input: "+input);
  }
};
mItem.addActionListener(lst);
mFile.add(mItem);

mFile.addSeparator();

mItem = new JMenuItem("Exit");
mItem.setMnemonic('x');
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    if (JOptionPane.showConfirmDialog(
     DialogBoxes.this,
     "Do you want to quit this application ?",
     BOX_TITLE, JOptionPane.YES_NO_OPTION)
     == JOptionPane.YES_OPTION)
      System.exit(0);
  }
};
mItem.addActionListener(lst);
mFile.add(mItem);
menuBar.add(mFile);
```

```
    return menuBar;
  }

  public static void main(String argv[]) {
    new DialogBoxes();
  }
}
```

Understanding the Code

Class DialogBoxes

This class represents a simple frame which contains a menu bar created with our `createMenuBar()` method, and a `JDesktopPane` (see chapter 16) which is used as the frame's layered pane. The menu bar contains a single menu titled "File," which holds several menu items.

The `createMenuBar()` method is responsible for populating our frame's menu bar with seven menu items, each with an `ActionListener` to invoke the display of a `JOptionPane` in either a `JDialog` or `JInternalFrame`. The first menu item titled "Ask Question" creates an instance of `JOptionPane`, assigns it a custom icon using its `setIcon()` method, and custom option button `String`s using `setOptions()`. A `JDialog` is created to hold this message box, and the `show()` method displays this dialog on the screen and waits until it is dismissed. At that point the `getValue()` method retrieves the user's selection as an `Object`, which may be `null` or one of the option button `String`s assigned to this message box. The resulting dialog is shown in figure 14.6.

UI Guideline: Affirmative Text

The use of the affirmative and unambiguous text "To Be" and "Not to be" greatly enhances the usability of the option dialog. For example, were the text to have read, "To be or not to be? That is the question", "Yes" or "No", this would have been somewhat ambiguous and may have confused some users. The explicit text, "To Be", "Not to be" is much clearer.

This is another example of improved usability with just a little more extra coding effort.

The second menu item titled "Info Message" creates a `JOptionPane` with a multi-line message `String` and a custom icon. The `createInternalFrame()` method is used to create a `JInternalFrame` holding the resulting `JOptionPane` message box. This internal frame is then added to the layered pane, which is now a `JDesktopPane` instance. The resulting internal frame is shown on the figure 14.7.

The third menu item titled "Error Message" produces a standard error message box using `JOptionPane`'s static `showMessageDialog()` method and the `ERROR_MESSAGE` message type. The resulting dialog is shown in figure 14.8. Recall that `JOptionPane` dialogs appear, by default, centered with respect to the parent if the parent is a frame. This is why we don't do any manual positioning here.

The next two menu items titled "Text Input" and "Combobox Input" produce `INFORMATION_MESSAGE` `JOptionPanes` which take user input in a `JTextField` and `JComboBox` respectively. The static `showInputDialog()` is used to display these `JOptionPanes` in `JDialogs`. Figures 14.9 and 14.10 illustrate. The "Text Input" pane takes the initial text to display in its text field as a `String` parameter. The "Combobox Input" pane takes an array of `String`s to display in the combo box as possible choices, as well as a `String` to display as initialy selected.

UI Guideline: Added Usability with Constrained Lists

Figures 14.9 and 14.10 clearly highlight how usability can be improved through effective component choice. The

combobox with a constrained list of choices is clearly the better tool for the task at hand.

The Domain Problem in this example has a fixed number of choices. Shakespeare is clearly dead and the plays attributed to him are known. Thus the combobox in Fig 14.10 is a better choice. It should be populated with a list of all the known plays.

The Option Pane in Fig 14.9 would be better used for an unknown data entry such as "Please enter your name".

The final menu item titled "Exit" brings up a `YES_NO_OPTION` confirmation `JOptionPane` in a `JDialog` (shown in figure 14.11) by calling `showConfirmDialog()`. The application is terminated if the user answers "Yes."

14.4 Customizing JColorChooser

In chapter 12 we developed a custom menu item allowing quick and easy selection of a color for the background and foreground of a `JTextArea`. In section 14.1 we built off of this example to add a simple "About" dialog. In this section we'll build off of it further, and construct a customized `JColorChooser` allowing a much wider range of color selection. Our implementation includes a preview component, `PreviewPanel`, that illustrates how text will appear with chosen background and foreground colors. Note that we have to return both background and foreground selection values when the user dismisses the color chooser in order to update the text component properly.

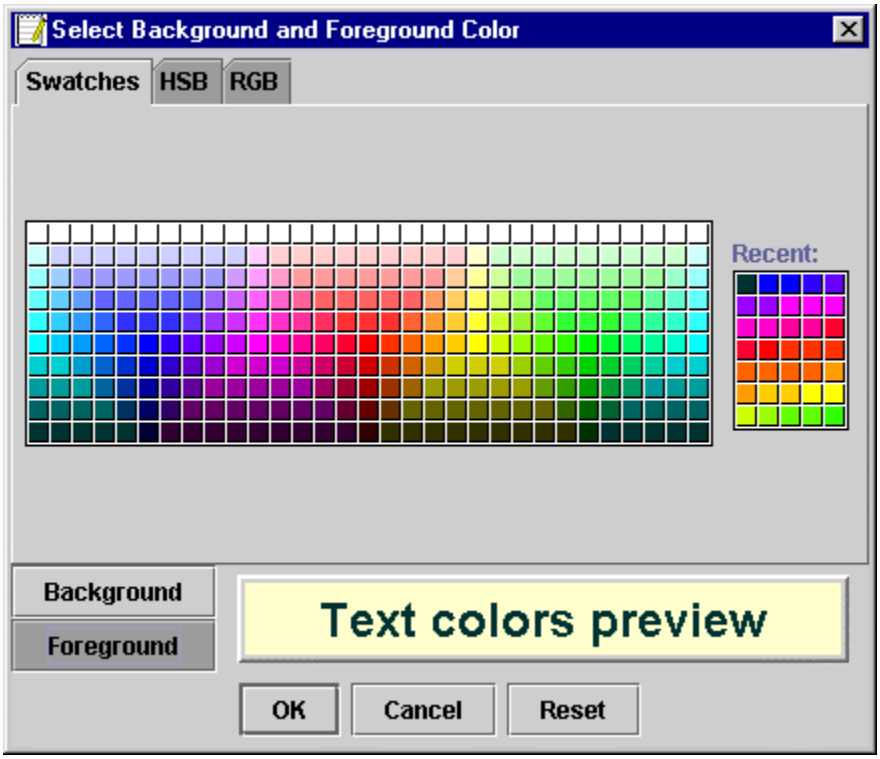


Figure 14.12 `JColorChooser` with custom `PreviewPanel` component capable of returning two `Color` selections.
<<file figure14-12.gif>>

UI Guideline: Preview Improves Usability

In this example, the User may have a goal of "Select suitable colours for a banner headline." By allowing the User to view a WYSIWYG preview, usability is improved. The user doesn't have to experiment with selection, which involves opening and closing the dialog several times. Instead, she can achieve the goal on a single visit to the color chooser dialog.

The Code: BasicTextEditor.java
see \Chapter14\3

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

import javax.swing.*;

public class BasicTextEditor extends JFrame
{
  // Unchanged code from section 14.2

  protected JColorChooser m_colorChooser;
  protected PreviewPanel m_previewPanel;
  protected JDialog m_colorDialog;

  public BasicTextEditor() {
    super("BasicTextEditor with JColorChooser");
    setSize(450, 350);
    ImageIcon icon = new ImageIcon("smallIcon.gif");
    setIconImage(icon.getImage());

    m_colorChooser = new JColorChooser();
    m_previewPanel = new PreviewPanel(m_colorChooser);
    m_colorChooser.setPreviewPanel(m_previewPanel);

    // Unchanged code from section 14.2
  }

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 14.2

    Action actionChooser = new AbstractAction("Color Chooser") {
      public void actionPerformed(ActionEvent e) {
        BasicTextEditor.this.repaint();
        if (m_colorDialog == null)
        m_colorDialog = JColorChooser.createDialog(
          BasicTextEditor.this,
          "Select Background and Foreground Color",
          true, m_colorChooser, m_previewPanel, null);
        m_previewPanel.setTextForeground(
          m_monitor.getForeground());
        m_previewPanel.setTextBackground(
          m_monitor.getBackground());
        m_colorDialog.show();

        if (m_previewPanel.isSelected()) {
          m_monitor.setBackground(
            m_previewPanel.getTextBackground());
          m_monitor.setForeground(
            m_previewPanel.getTextForeground());
        }
      }
    };
    mOpt.addSeparator();
    item =  mOpt.add(actionChooser);
    item.setMnemonic('c');

    menuBar.add(mOpt);

    // Unchanged code from section 14.2
  }
```

```
}

// Unchanged code from section 14.2

class PreviewPanel extends JPanel
 implements ChangeListener, ActionListener
{
  protected JColorChooser m_chooser;
  protected JLabel m_preview;
  protected JToggleButton m_btBack;
  protected JToggleButton m_btFore;
  protected boolean m_isSelected = false;

  public PreviewPanel(JColorChooser chooser) {
    this(chooser, Color.white, Color.black);
  }

  public PreviewPanel(JColorChooser chooser,
   Color background, Color foreground) {
    m_chooser = chooser;
    chooser.getSelectionModel().addChangeListener(this);

    setLayout(new BorderLayout());
    JPanel p = new JPanel(new GridLayout(2, 1, 0, 0));
    ButtonGroup group = new ButtonGroup();
    m_btBack = new JToggleButton("Background");
    m_btBack.setSelected(true);
    m_btBack.addActionListener(this);
    group.add(m_btBack);
    p.add(m_btBack);
    m_btFore = new JToggleButton("Foreground");
    m_btFore.addActionListener(this);
    group.add(m_btFore);
    p.add(m_btFore);
    add(p, BorderLayout.WEST);

    p = new JPanel(new BorderLayout());
    Border b1 = new EmptyBorder(5, 10, 5, 10);
    Border b2 = new BevelBorder(BevelBorder.RAISED);
    Border b3 = new EmptyBorder(2, 2, 2, 2);
    Border cb1 = new CompoundBorder(b1, b2);
    Border cb2 = new CompoundBorder(cb1, b3);
    p.setBorder(cb2);

    m_preview = new JLabel("Text colors preview",
      JLabel.CENTER);
    m_preview.setBackground(background);
    m_preview.setForeground(foreground);
    m_preview.setFont(new Font("Arial",Font.BOLD, 24));
    m_preview.setOpaque(true);
    p.add(m_preview, BorderLayout.CENTER);
    add(p, BorderLayout.CENTER);

    m_chooser.setColor(background);
  }

  protected boolean isSelected() {
    return m_isSelected;
  }

  public void setTextBackground(Color c) {
    m_preview.setBackground(c);
  }
```

```
  public Color getTextBackground() {
    return m_preview.getBackground();
  }

  public void setTextForeground(Color c) {
    m_preview.setForeground(c);
  }

  public Color getTextForeground() {
    return m_preview.getForeground();
  }

  public void stateChanged(ChangeEvent evt) {
    Color c = m_chooser.getColor();
    if (c != null) {
      if (m_btBack.isSelected())
        m_preview.setBackground(c);
      else
        m_preview.setForeground(c);
    }
  }

  public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() == m_btBack)
      m_chooser.setColor(getTextBackground());
    else if (evt.getSource() == m_btFore)
      m_chooser.setColor(getTextForeground());
    else
      m_isSelected = true;
  }
}
```

Understanding the Code

Class BasicTextEditor

New instance variables:

`JColorChooser m_colorChooser`: stored `JColorChooser` to avoid unnecessary instantiation.

`PreviewPanel m_previewPanel`: instance of our custom color previewing component.

`JDialog m_colorDialog`: stored `JDialog` acting as the parent of `m_colorChooser`.

The constructor instantiates `m_colorChooser` and `m_previewPanel`, assigning `m_previewPanel` as `m_colorChooser`'s preview component using the `setPreviewPanel()` method.

The menu bar receives a new menu item titled "Color Chooser" set up in the `createMenuBar()` method as an `Action` implementation. When selected, this item first repaints our application frame to ensure that the area covered by the popup menu is refreshed properly. Then it checks to see if our `m_colorDialog` has been instantiated yet. If not we call `JColorChooser`'s static `createDialog()` method to wrap `m_colorChooser` in a dialog, and use `m_previewPanel` as an `ActionListener` for the "OK" button (see 14.1.3). Note that this instantiation only occurs once.

We then assign the current colors of `m_monitor` to `m_previewPanel` (recall that `m_monitor` is the `JTextArea` central to this application). The reason we do this is because the foreground and background can also be assigned by our custom menu color choosers. If this occurs `m_previewPanel` is not notified, so we update the selected colors each time the dialog is invoked.

The dialog is then shown and the main application thread waits for it to be dismissed. When the dialog is dismissed m_previewPanel is checked for whether or not new colors have been selected, using its isSelected() method (see below). If new colors have been chosen they are assigned to m_monitor.

Note: We have purposely avoided updating the selected colors in our custom color menu components. The reason we did this is because in a more professional implementation we would most likely not offer both methods for choosing text component colors. If we did want to support both methods we would need to determine the closest color in our custom color menus that matches the corresponding color selected with JColorChooser (because JColorChooser offers a much wider range of choices).

Class PreviewPanel

This class represents our custom color preview component designed for use with JColorChooser. It extends JPanel and implements two listener interfaces, ChangeListener and ActionListener. It displays selected foreground and background colors in a label, and includes two JToggleButtons used to switch between background color and foreground color selection modes. Instance variables:

- JColorChooser m_chooser: a reference to the hosting color chooser.
- JLabel m_preview: label to preview background and foreground colors.
- JToggleButton m_btBack: toggle button to switch to background color selection.
- JToggleButton m_btFore: toggle button to switch to foreground color selection.
- boolean m_isSelected: flag indicating a selection has taken place.

The first PreviewPanel constructor takes a JColorChooser as parameter and delegates its work to the second constructor, passing it the JColorChooser as well as white and black Colors for the initial background and foreground colors respectively. As we discussed in the beginning of this chapter, JColorChooser's ColorSelectionModel fires ChangeEvent's when the selected Color changes. So we start by registering this component as a ChangeListener with the given color chooser's model.

A BorderLayout is used to manage this container and two toggle buttons are placed in a 2x1 GridLayout, which is added to the WEST region. Both buttons receive a this reference as an ActionListener. A label with a large font is then placed in the CENTER region. This label is surrounded by a decorative, doubly-compounded border consisting of an EmptyBorder, BevelBorder, and another EmptyBorder. The foreground and background colors of this label are assigned as those values passed to the constructor.

Several methods are used to set and get the selected colors and do not require any special explanation. The stateChanged() method will be called when the color chooser model fires ChangeEvents. Depending on which toggle button is selected, this method updates the background or foreground color of the preview label.

The actionPerformed() method will be called when one of the toggle buttons is pressed. It assignes the stored background or foreground, depending which button is pressed, as the color of the hosting JColorChooser. This method is also called when the "OK" button is pressed, in which case the m_isSelected flag is set to true.

Running the Code

Select the "Color Chooser" menu item to bring up our customized JColorChooser shown in figure 14.12. Select a background and foreground color using any of the available color panes. Verify that the preview label is updated accordingly to reflect the current color selection, and the currently selected toggle button. Press the "OK" button to dismiss the dialog and note that both the selected foreground and background colors are

assigned to our application's text area. Also note that pressing the "Cancel" button dismisses the dialog without making any color changes.

14.5 Customizing JFileChooser

Examples using `JFileChooser` to load and save files are scattered throughout the whole book. In this section we'll take a closer look at the more advanced features of this component through building a powerful JAR and ZIP archive creation, viewing, and extraction tool. We will see how to implement a custom `FileView` and `FileFilter`, and how to access and manipulate the internals of `JFileChooser` to allow multiple file selection and add our own components. Since this example deals with Java 2 archive functionality, we will first briefly summarize the classes from the `java.util.zip` and `java.util.jar` packages we will be using.

Note: The interface presented in this section is extremely basic and professional implementations would surely construct a more ellaborate GUI. We have purposely avoided this here due to the complex nature of the example, and to avoid straying from the `JFileChooser` topics central to its construction.

14.5.1 ZipInputStream

class java.util.zip.ZipInputStream

This class represents a filtered input stream which uncompresses ZIP archive data. The constructor takes an instance of `InputStream` as parameter. Before reading data from this stream we need to find a ZIP file entry using the `getNextEntry()` method. Each entry corresponds to an archived file. We can `read()` an an array of `bytes` from an entry, and then close it using the `closeEntry()` method when reading is complete.

14.5.2 ZipOutputStream

class java.util.zip.ZipOutputStream

This class represents a filtered output stream which takes binary data and writes them into an archive in the compressed (default) or uncompressed (optional) form. The constructor of this class takes an instance of `OutputStream` as parameter. Before writing data to this stream we need to create a new `ZipEntry` using the `putNextEntry()` method. Each `ZipEntry` corresponds to an archived file. We can `write()` an array of bytes to a `ZipEntry`, and close it using the `closeEntry()` method when writing is complete. We can also specify the compression method for storing `ZipEntrys` using `ZipOutputStream`'s `setMethod()` method.

14.5.3 ZipFile

class java.util.zip.ZipFile

This class encapsulates a collection of `ZipEntrys` and represents a read-only ZIP archive. We can fetch an `Enumeration` of the contained `ZipEntrys` using the `entries()` method. The `size()` method tells us how many files are contained and `getName()` returns the archive's full path name. We can retrieve an `InputStream` for reading the contents of a contained `ZipEntry` using its `getInputStream()` method. When we are done reading we are expected to call the `close()` method to close the archive.

14.5.4 ZipEntry

class java.util.zip.ZipEntry

This class represents a single archived file or directory within a ZIP archive. It allows retrieval of its name and

can be cloned using the `clone()` method. Using typical set/get accessors, we can access a `ZipEntry`'s compression method, CRC-32 checksum, size, modification time, compression method, and a comment attachment. We can also query whether or not a `ZipEntry` is a directory using its `isDirectory()` method.

14.5.5 The java.util.jar package

This package contains a set of classes for managing Java archives (JAR files). The relevant classes that we will be dealing with (`JarEntry`, `JarFile`, `JarInputStream`, and `JarOutputStream`) are direct subclasses of the `zip` package counterparts, and thus inherit the functionality described above.

14.5.6 Manifest

class java.util.jar.Manifest

This class represents a JAR `Manifest` file. A `Manifest` contains a collection of names and their associated attributes specific both for the archive in whole and for a particular `JarEntry` (i.e. a file or directory in the archive). We are not concerned with the details of JAR manifest files in this chapter, and it suffices to say that the `JarOutputStream` constructor takes a `Manifest` instance as parameter, along with an `OutputStream`.

In this example we create a simple, two-button GUI with a status bar (a label). One button corresponds to creating a ZIP or JAR archive, and another corresponds to decompressing an archive. In each case two `JFileChoosers` are used to perform the operation. The first chooser is used to allow entry of an archive name to use, or selection of an archive to decompress. The second chooser is used to allow selection of files to compress or decompress. (As noted above, more professional implementations would most likely include a more elaborate GUI.) A custom `FileView` class is used to represent ZIP and JAR archives using a custom icon, and a `FileFilter` class is constructed to only allow viewing of ZIP (".zip") and JAR (".jar") files. We also work with `JFileChooser` as a container by adding our own custom component, taking advantage of the fact that it uses a y-oriented `BoxLayout` to organize its children. Using similar tactics we show how to gain access to its `JList` (used for the display and selection of files and directories) to allow multiple selection (an unimplemented feature in `JFileChooser` as of Java 2 FCS).

Figure 14.12 ZIP/JAR Manager JFileChooser example at startup.
<<file figure14-12.gif>>

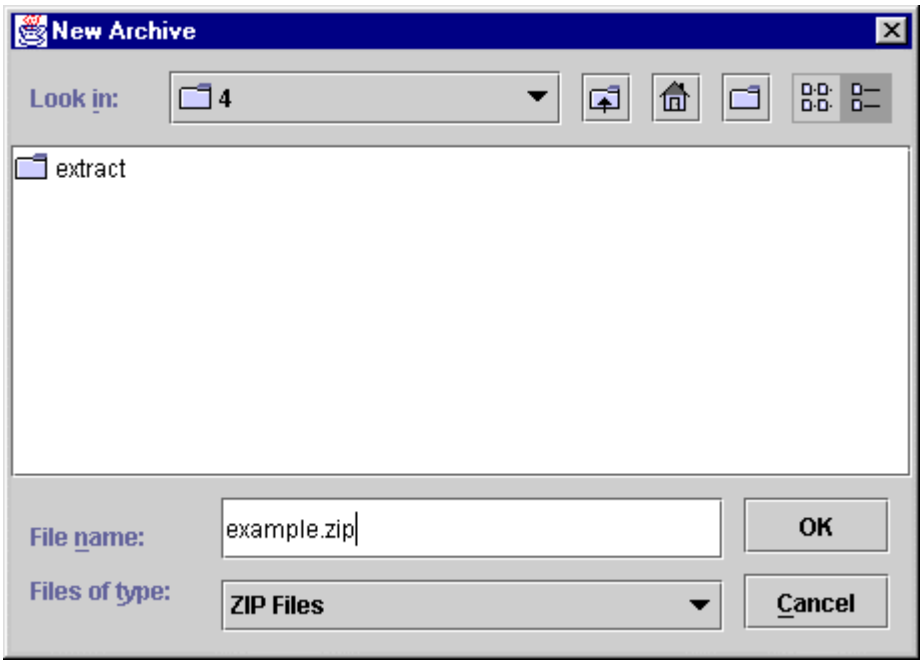


Figure 14.13 First step in creating an archive; using JFileChooser to select an archive location and name.
<<file figure14-13.gif>>

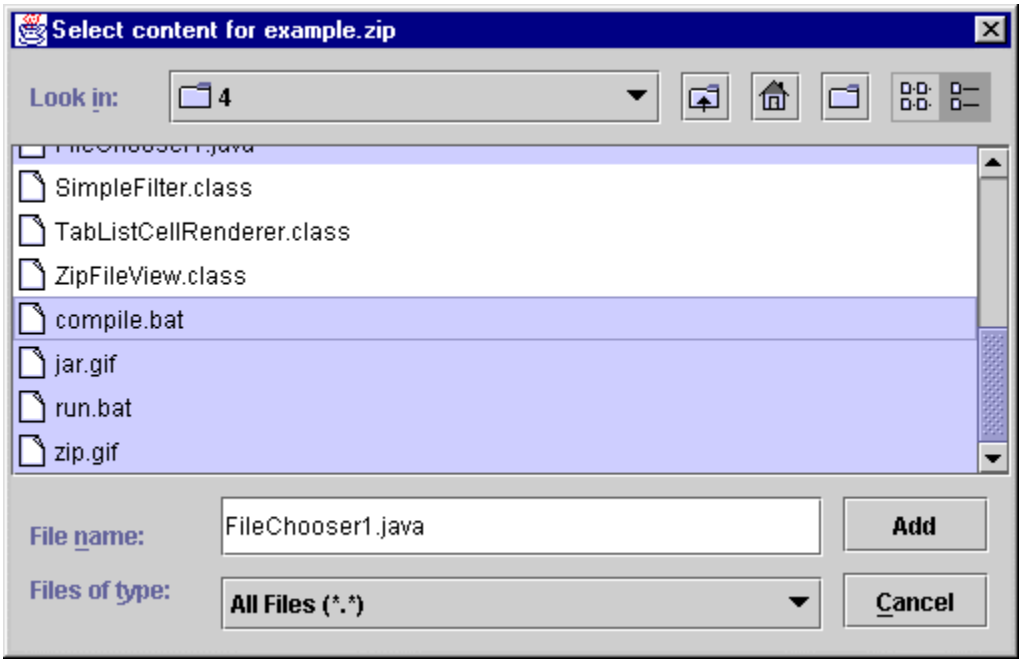


Figure 14.14 Second step in creating an archive; using JFileChooser to select archive content.
<<file figure14-14.gif>>

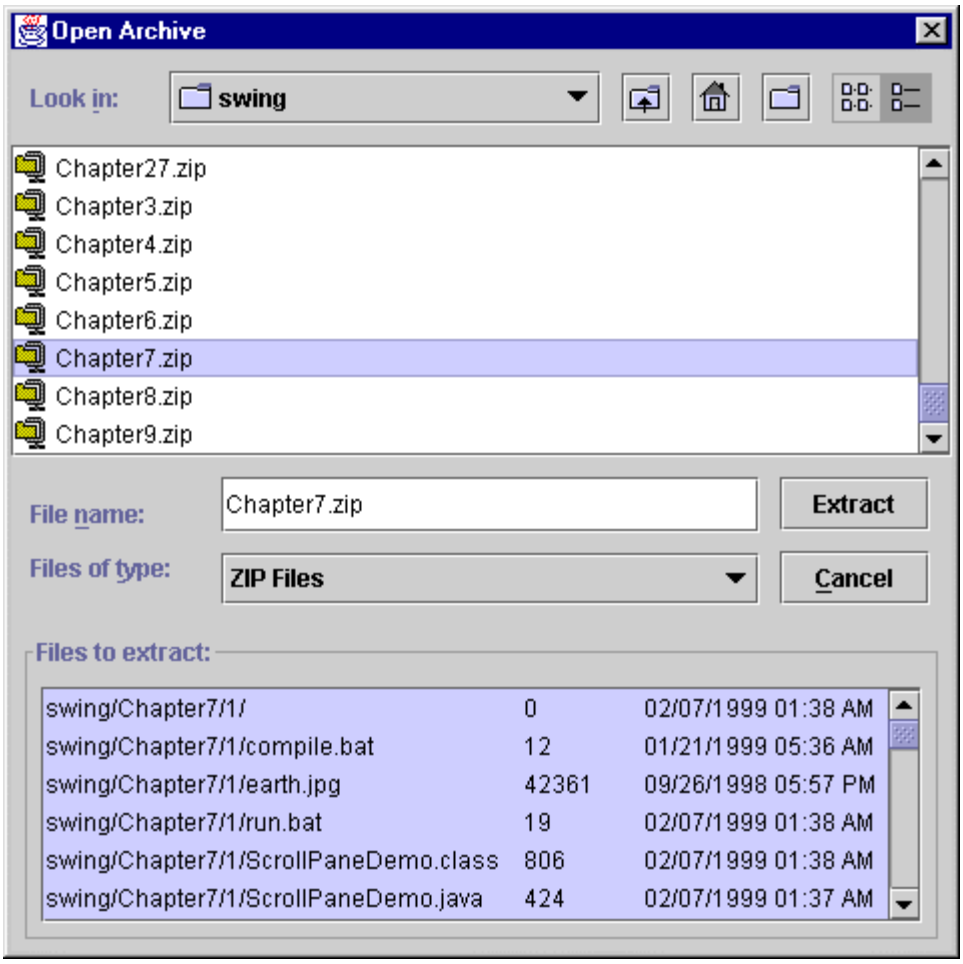


Figure 14.15 First step in uncompressing an archive; using a custom component in JFileChooser.
<<file figure14-15.gif>>

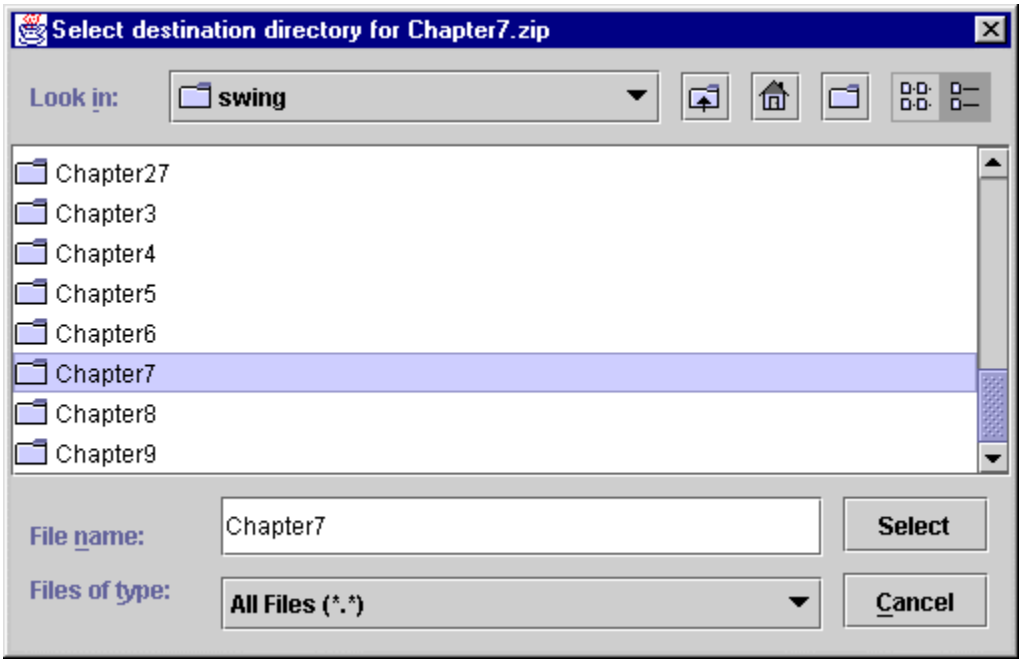


Figure 14.16 Second step in uncompressing an archive; using JFileChooser to select a destination directory.
<<file figure14-16.gif>>

The Code: ZipJarManager.java
see \Chapter14\4

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.util.zip.*;
import java.util.jar.*;
import java.beans.*;
import java.text.SimpleDateFormat;

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

public class ZipJarManager extends JFrame
{
  public static int BUFFER_SIZE = 10240;

  protected JFileChooser chooser1;
  protected JFileChooser chooser2;

  protected File  m_currentDir = new File(".");
  protected SimpleFilter m_zipFilter;
  protected SimpleFilter m_jarFilter;
  protected ZipFileView  m_view;

  protected JButton m_btCreate;
  protected JButton m_btExtract;
  protected JLabel  m_status;

  protected JList m_zipEntries;
  protected File  m_selectedFile;

  public ZipJarManager()
  {
    super("ZIP/JAR Manager");

    JWindow jwin = new JWindow();
    jwin.getContentPane().add(new JLabel(
      "Loading ZIP/JAR Manager...", SwingConstants.CENTER));
    jwin.setBounds(200,200,200,100);
    jwin.setVisible(true);

    chooser1 = new JFileChooser();
    chooser2 = new JFileChooser();

    JPanel p = new JPanel(new GridLayout(3, 1, 10, 10));
    p.setBorder(new EmptyBorder(10, 10, 10, 10));

    m_btCreate = new JButton("Create new Archive");
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_btCreate.setEnabled(false);
        m_btExtract.setEnabled(false);
        createArchive();
        m_btCreate.setEnabled(true);
        m_btExtract.setEnabled(true);
      }
    };
```

```
    m_btCreate.addActionListener(lst);
    m_btCreate.setMnemonic('c');
    p.add(m_btCreate);

    m_btExtract = new JButton("Extract from Archive");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_btCreate.setEnabled(false);
        m_btExtract.setEnabled(false);
        extractArchive();
        m_btCreate.setEnabled(true);
        m_btExtract.setEnabled(true);
      }
    };
    m_btExtract.addActionListener(lst);
    m_btExtract.setMnemonic('e');
    p.add(m_btExtract);

    m_status = new JLabel();
    m_status.setBorder(new BevelBorder(BevelBorder.LOWERED,
      Color.white, Color.gray));
    p.add(m_status);

    m_zipFilter = new SimpleFilter("zip", "ZIP Files");
    m_jarFilter = new SimpleFilter("jar", "JAR Files");
    m_view = new ZipFileView();

    chooser1.addChoosableFileFilter(m_zipFilter);
    chooser1.addChoosableFileFilter(m_jarFilter);
    chooser1.setFileView(m_view);
    chooser1.setMultiSelectionEnabled(false);
    chooser1.setFileFilter(m_jarFilter);

    javax.swing.filechooser.FileFilter ft =
      chooser1.getAcceptAllFileFilter();
    chooser1.removeChoosableFileFilter(ft);

    getContentPane().add(p, BorderLayout.CENTER);
    setBounds(0,0,300,150);
    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    jwin.setVisible(false);
    jwin.dispose();

    setVisible(true);
  }

  public void setStatus(String str) {
    m_status.setText(str);
    m_status.repaint();
  }

  protected void createArchive() {
    chooser1.setCurrentDirectory(m_currentDir);
    chooser1.setDialogType(JFileChooser.SAVE_DIALOG);
    chooser1.setDialogTitle("New Archive");
    chooser1.setPreferredSize(new Dimension(450,300));
```

```
    if (chooser1.showDialog(this, "OK") !=
      JFileChooser.APPROVE_OPTION)
        return;
    m_currentDir = chooser1.getCurrentDirectory();

    final File archiveFile = chooser1.getSelectedFile();
    if (!isArchiveFile(archiveFile))
      return;

    // Show chooser to select entries.
    chooser2.setCurrentDirectory(m_currentDir);
    chooser2.setDialogType(JFileChooser.OPEN_DIALOG);
    chooser2.setDialogTitle("Select content for "
      + archiveFile.getName());
    chooser2.setMultiSelectionEnabled(true);
    chooser2.setFileSelectionMode(JFileChooser.FILES_ONLY);

    if (chooser2.showDialog(this, "Add") !=
      JFileChooser.APPROVE_OPTION)
        return;

    m_currentDir = chooser2.getCurrentDirectory();
    final File[] selected = getSelectedFiles(chooser2);

    String name = archiveFile.getName().toLowerCase();
    if (name.endsWith(".zip")) {
      Thread runner = new Thread() {
        public void run() {
          createZipArchive(archiveFile, selected);
        }
      };
      runner.start();
    }
    else if (name.endsWith(".jar")) {
      Thread runner = new Thread() {
        public void run() {
          createJarArchive(archiveFile, selected);
        }
      };
      runner.start();
    }
  }

  protected void createZipArchive(File archiveFile,
   File[] selected) {
    try {
      byte buffer[] = new byte[BUFFER_SIZE];

      // Open archive file.
      FileOutputStream stream =
        new FileOutputStream(archiveFile);
      ZipOutputStream out = new ZipOutputStream(stream);

      for (int k=0; k<selected.length; k++) {
        if (selected[k]==null || !selected[k].exists() ||
          selected[k].isDirectory())
            continue;  // Just in case...
        setStatus("Adding "+selected[k].getName());

        // Add archive entry.
        ZipEntry zipAdd = new ZipEntry(selected[k].getName());
        zipAdd.setTime(selected[k].lastModified());
        out.putNextEntry(zipAdd);
```

```
      // Read input & write to output.
      FileInputStream in = new FileInputStream(selected[k]);
      while (true) {
        int nRead = in.read(buffer, 0, buffer.length);
        if (nRead <= 0)
          break;
        out.write(buffer, 0, nRead);
      }
      in.close();
    }

    out.close();
    stream.close();
    setStatus("Adding completed OK");
  }
  catch (Exception e) {
    e.printStackTrace();
    setStatus("Error: "+e.getMessage());
    return;
  }
}

protected void createJarArchive(
 File archiveFile, File[] selected) {
  try {
    byte buffer[] = new byte[BUFFER_SIZE];

    // Open archive file.
    FileOutputStream stream =
      new FileOutputStream(archiveFile);
    JarOutputStream out = new JarOutputStream(stream,
      new Manifest());

    for (int k=0; k<selected.length; k++) {
      if (selected[k]==null || !selected[k].exists() ||
        selected[k].isDirectory())
          continue;  // Just in case...
      setStatus("Adding "+selected[k].getName());

      // Add archive entry.
      JarEntry jarAdd = new JarEntry(selected[k].getName());
      jarAdd.setTime(selected[k].lastModified());
      out.putNextEntry(jarAdd);

      // Write file to archive
      FileInputStream in = new FileInputStream(selected[k]);
      while (true) {
        int nRead = in.read(buffer, 0, buffer.length);
        if (nRead <= 0)
          break;
        out.write(buffer, 0, nRead);
      }
      in.close();
    }

    out.close();
    stream.close();
    setStatus("Adding completed OK");
  }
  catch (Exception ex) {
    ex.printStackTrace();
    setStatus("Error: "+ex.getMessage());
```

```
    }
  }

  protected void extractArchive() {
    chooser1.setCurrentDirectory(m_currentDir);
    chooser1.setDialogType(JFileChooser.OPEN_DIALOG);
    chooser1.setDialogTitle("Open Archive");
    chooser1.setPreferredSize(new Dimension(470,450));

    // Construct a JList to show archive entries.
    m_zipEntries = new JList();
    m_zipEntries.setSelectionMode(
      ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
    TabListCellRenderer renderer = new TabListCellRenderer();
    renderer.setTabs(new int[] {240, 300, 360});
    m_zipEntries.setCellRenderer(renderer);

    // Place entries list in a scroll pane and add it to chooser.
    JPanel p = new JPanel(new BorderLayout());
    p.setBorder(new CompoundBorder(
      new CompoundBorder(
        new EmptyBorder(5, 5, 5, 5),
        new TitledBorder("Files to extract:")),
      new EmptyBorder(5,5,5,5)));
    JScrollPane ps = new JScrollPane(m_zipEntries);
    p.add(ps, BorderLayout.CENTER);
    chooser1.add(p);

    PropertyChangeListener lst = new PropertyChangeListener() {
      SimpleDateFormat m_sdf = new SimpleDateFormat(
        "MM/dd/yyyy hh:mm a");
      DefaultListModel m_emptyModel = new DefaultListModel();

      public void propertyChange(PropertyChangeEvent e) {
        if (e.getPropertyName() ==
         JFileChooser.FILE_FILTER_CHANGED_PROPERTY) {
          m_zipEntries.setModel(m_emptyModel);
          return;
        }
        else if (e.getPropertyName() ==
         JFileChooser.SELECTED_FILE_CHANGED_PROPERTY) {
          File f = chooser1.getSelectedFile();
          if (f != null) {
            if (m_selectedFile!=null && m_selectedFile.equals(f)
             && m_zipEntries.getModel().getSize() > 0) {
              return;
            }

            String name = f.getName().toLowerCase();
            if (!name.endsWith(".zip") && !name.endsWith(".jar")) {
              m_zipEntries.setModel(m_emptyModel);
              return;
            }

            try {
              ZipFile zipFile = new ZipFile(f.getPath());
              DefaultListModel model = new DefaultListModel();
              Enumeration en = zipFile.entries();
              while (en.hasMoreElements()) {
                ZipEntry zipEntr = (ZipEntry)en.nextElement();
                Date d = new Date(zipEntr.getTime());
                String str = zipEntr.getName()+'\t'+
                  zipEntr.getSize()+'\t'+m_sdf.format(d);
```

```
            model.addElement(str);
          }
          zipFile.close();
          m_zipEntries.setModel(model);
          m_zipEntries.setSelectionInterval(0,
            model.getSize()-1);
        }
        catch(Exception ex) {
          ex.printStackTrace();
          setStatus("Error: "+ex.getMessage());
        }
      }
    }
    else {
      m_zipEntries.setModel(m_emptyModel);
      return;
    }
  }
};
chooser1.addPropertyChangeListener(lst);
chooser1.cancelSelection();

if (chooser1.showDialog(this, "Extract") !=
 JFileChooser.APPROVE_OPTION) {
  chooser1.remove(p);
  chooser1.removePropertyChangeListener(lst);
  return;
}
m_currentDir = chooser1.getCurrentDirectory();
final File archiveFile = chooser1.getSelectedFile();

if (!archiveFile.exists() || !isArchiveFile(archiveFile)) {
  chooser1.remove(p);
  chooser1.removePropertyChangeListener(lst);
  return;
}

Object[] selObj = m_zipEntries.getSelectedValues();
if (selObj.length == 0) {
  setStatus("No entries have been selected for extraction");
  chooser1.removePropertyChangeListener(lst);
  chooser1.remove(p);
  return;
}
final String[] entries = new String[selObj.length];
for (int k=0; k<selObj.length; k++) {
  String str = selObj[k].toString();
  int index = str.indexOf('\t');
  entries[k] = str.substring(0, index);
}

// Show dialog to select output directory.
chooser2.setCurrentDirectory(m_currentDir);
chooser2.setDialogType(JFileChooser.OPEN_DIALOG);
chooser2.setDialogTitle("Select destination directory for "
  + archiveFile.getName());
chooser2.setMultiSelectionEnabled(false);
chooser2.setFileSelectionMode(JFileChooser.DIRECTORIES_ONLY);

if (chooser2.showDialog(this, "Select") !=
 JFileChooser.APPROVE_OPTION) {
  chooser1.remove(p);
  chooser1.removePropertyChangeListener(lst);
```

```
      return;
    }

    m_currentDir = chooser2.getCurrentDirectory();
    final File outputDir = chooser2.getSelectedFile();

    Thread runner = new Thread() {
      public void run() {
        try {
          byte buffer[] = new byte[BUFFER_SIZE];

          // Open the archive file.
          FileInputStream stream =
            new FileInputStream(archiveFile);
          ZipInputStream in = new ZipInputStream(stream);

          // Find archive entry.
          while (true) {
            ZipEntry zipExtract = in.getNextEntry();
            if (zipExtract == null)
              break;
            boolean bFound = false;
            for (int k=0; k<entries.length; k++) {
              if (zipExtract.getName().equals(entries[k])) {
                bFound = true;
                break;
              }
            }
            if (!bFound) {
              in.closeEntry();
              continue;
            }
            setStatus("Extracting " + zipExtract.getName());

            // Create output file and check required directory.
            File outFile = new File(outputDir,
              zipExtract.getName());
            File parent = outFile.getParentFile();
            if (parent != null && !parent.exists())
              parent.mkdirs();

            // Extract unzipped file.
            FileOutputStream out = new FileOutputStream(outFile);
            while (true) {
              int nRead = in.read(buffer, 0, buffer.length);
              if (nRead <= 0)
                break;
              out.write(buffer, 0, nRead);
            }
            out.close();
            in.closeEntry();
          }

          in.close();
          stream.close();
          setStatus("Extracting completed OK");
        }
        catch (Exception ex) {
          ex.printStackTrace();
          setStatus("Error: "+ex.getMessage());
        }
      }
    };
```

```
    runner.start();
    chooser1.removePropertyChangeListener(lst);
    chooser1.remove(p);
  }

  public static File[] getSelectedFiles(JFileChooser chooser) {
    // Although JFileChooser won't give us this information,
    // we need it...
    Container c1 = (Container)chooser.getComponent(3);
    JList list = null;
    while (c1 != null) {
      Container c = (Container)c1.getComponent(0);
      if (c instanceof JList) {
        list = (JList)c;
        break;
      }
      c1 = c;
    }
    Object[] entries = list.getSelectedValues();
    File[] files = new File[entries.length];
    for (int k=0; k<entries.length; k++) {
      if (entries[k] instanceof File)
        files[k] = (File)entries[k];
    }
    return files;
  }

  public static boolean isArchiveFile(File f) {
    String name = f.getName().toLowerCase();
    return (name.endsWith(".zip") || name.endsWith(".jar"));
  }

  public static void main(String argv[]) {
    new ZipJarManager();
  }
}

class SimpleFilter extends javax.swing.filechooser.FileFilter
{
  private String m_description = null;
  private String m_extension = null;

  public SimpleFilter(String extension, String description) {
    m_description = description;
    m_extension = "."+extension.toLowerCase();
  }

  public String getDescription() {
    return m_description;
  }

  public boolean accept(File f) {
    if (f == null)
      return false;
    if (f.isDirectory())
      return true;
    return f.getName().toLowerCase().endsWith(m_extension);
  }
}

class ZipFileView extends javax.swing.filechooser.FileView
{
  protected static ImageIcon ZIP_ICON = new ImageIcon("zip.gif");
```

```
  protected static ImageIcon JAR_ICON = new ImageIcon("jar.gif");

  public String getName(File f) {
    String name = f.getName();
    return name.equals("") ? f.getPath() : name;
  }

  public String getDescription(File f) {
    return getTypeDescription(f);
  }

  public String getTypeDescription(File f) {
    String name = f.getName().toLowerCase();
    if (name.endsWith(".zip"))
      return "ZIP Archive File";
    else if (name.endsWith(".jar"))
      return "Java Archive File";
    else
      return "File";
  }

  public Icon getIcon(File f) {
    String name = f.getName().toLowerCase();
    if (name.endsWith(".zip"))
      return ZIP_ICON;
    else if (name.endsWith(".jar"))
      return JAR_ICON;
    else
      return null;
  }

  public Boolean isTraversable(File f) {
    return (f.isDirectory() ? Boolean.TRUE : Boolean.FALSE);
  }
}

// Class TabListCellRenderer is taken from Chapter 10,
// section 10.3 without modification.
```

Understanding the Code

Class ZipJarManager

This class extends `JFrame` to provide a very simple GUI for our ZIP/JAR archive manager application. One class variable is defined:

`int BUFFER_SIZE`: used to define the size of an array of bytes for reading and writing files.

Instance variables:

`JFileChooser chooser1`: file chooser used to select an archive for creation or extraction.

`JFileChooser chooser2`: file chooser used to select files to include or in an archive or files to extract from an archive.

`File m_currentDir`: the currently selected directory.

`SimpleFilter m_zipFilter`: filter for files with a "zip" extension.

`SimpleFilter m_jarFilter`: filter for files with a ".jar" extension.

`ZipFileView m_view`: a custom `FileView` implementation for JAR and ZIP files.

`JButton m_btCreate`: initiates the creation of an archive.

`JButton m_btExtract`: initiates extraction from an archive.

`JLabel m_status`: label do display status messages.

`JList m_zipEntries`: list component to display an array of entries in an archive.

`File m_selectedFile`: the currently selected archive file.

The `ZipJarManager` constructor first creates and displays a simple `JWindow` splash screen to let the user know that the application is loading (a progress bar might be more effective here, but we want to stick to the point). Our two `JFileChoosers` are then instantiated, which takes a significant amount of time, and then the buttons and label are created, encapsulated in a `JPanel` using a `GridLayout`, and added to the content pane. The first button titled "Create new Archive" is assigned an `ActionListener` which invokes `createArchive()`. The second button titled "Extract from Archive" is assigned an `ActionListener` which invokes `extractArchive()`. Our custom `SimpleFilters` and `FileView` are then instantiated, and assigned to `chooser1`. We then set `chooser1`'s `multiSelectionEnabled` property to `false`, tell it to use the JAR filter initially, and remove the "AllFiles" filter. Finally, the splash window is disposed before `FileChooser1` is made visible.

The `setStatus()` method simply assigns a given `String` to the `m_status` label.

The `createArchive()` method is used to create a new archive file using both `JFileChoosers`. First we set `chooser1`'s title to "New Archive", its type to `SAVE_DIALOG`, and assign it a new preferred size (the reason we change its size will become clear below). Then we show `chooser1` in a dialog to prompt the user for a new archive name. If the dialog is dismissed by pressing "Cancel" or the close button we do nothing and return. Otherwise we store the current directory in our `m_currentDir` instance variable and create a `File` instance corresponding to the file specified in the chooser.

Interestingly, `JFileChooser` does not check whether the filename entered in its text field is valid with respect to its filters when the approve button pressed. So we are forced to check if our `File`'s name has a ".zip" or ".jar" extension manually using our custom `isArchiveFile()` method. If this method returns `false` we do nothing and return. Otherwise we setup `chooser2` to allow multiple selections to make up the content of the archive, and only allow file selections (by setting the `fileSelectionMode` property to `FILES_ONLY`) to avoid overcomplicating our archive processing scheme. Also note that we set the dialog title to specify the name of the archive we are creating.

We use `JFileChooser`'s `showDialog()` method to display `chooser2` in a `JDialog` and assign "Add" as its approve button text. If the approve button is not pressed we do nothing and return. Otherwise we create an array of `Files` to be placed in the specified archive using our custom `getSelectedFiles()` method (see below). Finally, we invoke our `createZipArchive()` method if the selected archive file has a ".zip" extension, or `createJarArchive()` if it has a ".jar" extension. These method calls are wrapped in separate threads to avoid clogging up the event dispatching thread.

The `createZipArchive()` method takes two parameters: a ZIP archive file and an array of the files to be added to the archive. It creates a `ZipOutputStream` to write the selected archive file. Then for each of files in the given array it creates a `ZipEntry` instance, places it in the `ZipOutputStream`, and performs standard read/write operations until all data has been written into the archive. The status label is updated, using our `setStatus()` method, each time a file is written and when the operation completes, to provide feedback during long operations.

The `createJarArchive()` method works almost identically to `createZipArchive()`, using the corresponding `java.util.jar` classes. Note that a default `Manifest` instance is supplied to the `JarOutputStream` constructor.

The `extractArchive()` method extracts data from an archive file using both `JFileChoosers`. First we assign `chooser1` a preferred size of 470x450 because we will be adding a custom component which requires a bit more space than `JFileChooser` normally offers (this is also why we set the preferred size of `chooser1` in our `createArchive()` method). Since `JFileChooser` is derived from `JComponent`, we can add our own components to it just like any other container. A quick look at the source code shows that `JFileChooser` uses a y-oriented `BoxLayout`. This implies that new components added to a `JFileChooser` will be placed below all other existing components (see chapter 4 for more about `BoxLayout`).

We take advantage of this knowledge and add a `JList` component, `m_zipEntries`, to allow selection of compressed entries to be extracted from the selected archive. This `JList` component receives an instance of our custom `TabListCellRenderer` as its cell renderer to process `Strings` with tabs (see chapter 10, section 10.3). The location of `String` segments between tabs are assigned using its `setTabs()` method. Finally, this list is placed in a `JScrollPane` to provide scrolling capabilities, and added to the bottom of the `JFileChooser` component.

A `PropertyChangeListener` is added to `chooser1` to process the user's selection. This anonymous class maintains two instance variables:

`SimpleDateFormat m_sdf`: used to format file time stamps.

`DefaultListModel m_emptyModel`: assigned to `m_zipEntries` when non-archive files are selected, or when the file filter is changed.

This listener's `propertyChange()` method will receive a `PropertyChangeEvent` when, among other things, `chooser1`'s selection changes. The selected file can then be retrieved using `PropertyChangeEvent`'s `getNewValue()` method. If this file represents a ZIP or JAR archive, our implementation creates a `ZipFile` instance to read it's content, and retrieves an `Enumeration` of `ZipEntries` in this archive (recall that `JarFile` and `JarEntry` are subclasses of `ZipFile` and `ZipEntry`, allowing us to display the contents of a JAR or a ZIP archive identically). For each entry we form a `String` containing that entry's name, size, and time stamp. This `String` is added to a `DefaultListModel` instance. After each entry has been processed, this model is assign to our `JList`, and all items are initially selected. The user can then modify the selection to specify entries to be extracted from the archive.

Once the `PropertyChangeListener` is added, `chooser1` is displayed with "Extract" as its approve button text. If it is dismissed we remove both the panel containing our list component and the `PropertyChangeListener`, as they are only temporary additions (remember that we use this same chooser to initiate creation of an archive in the `createArchive()` method). Otherwise, if the approve button is pressed we check whether the selected file exists and represents an archive. We then create an array of `Objects` corresponding to each selected item in our list (which are `Strings`). If no items are selected we report an error in our status label, remove our temporary component and listener, and return. Then we form an array of entry names corresponding to each selected item (which is the portion of each `String` before the appearance of the first tab character).

Now we need to select a directory to be used to extract the selected archive entries. We use `chooser2` for this purpose. We set its `fileSelectionMode` property to `DIRECTORIES_ONLY`, and allow only single selection by setting its `multiSelectionEnabled` property to `false`. We then show `chooser2` using "Select" for its approve button text. If it is dismissed we remove our our temporary component and listener, and return. Otherwise we start the extraction process in a separate thread.

To begin the extraction process we create a `ZipInputStream` to read from the selected archive file. Then we process each entry in the archive by retrieving a corresponding `ZipEntry` and verifying whether each

ZipEntry's name matches a String in our previously obtained array of selected file names (from our list component that was added to chooser1). If a match is found we create a File instance to write that entry to. If a ZipEntry includes sub-directories, we create these sub-directories using File's mkdirs() method. Finally we perform standard read/write operations until all files have been extracted from the archive. Note that we update the status label each time a file is extracted and when the opertion completes.

The getSelectedFiles() method takes a JFileChooser instance as parameter and returns an array of Files selected in the given chooser's list. Interestingly JFileChooser provides the getSelectedFiles() method which does not work properly as of Java 2 FCS (it always returns null). In order to work around this problem, we use java.awt.Container functionality (which, as we know, all Swing components inherit) to gain access to JFileChooser's components. With a little detective work we found that the component with index 3 represents the central part of the chooser. This component contains several children nested inside one another. One of these child components is the JList component we need access to in order to determine the current selection state. So we can simply loop through these nested containers until we find the JList. As soon as we have gained access to this component we can retrieve an array of the selected objects using JList's getSelectedValues() method. As expected, these objects are instances of the File class, so we need only upcast each and return them in a File[] array.

Note: This solution should be considered a temporary bug workaround. We expect the getSelectedFiles() method to be implemented correctly in future releases, and suggest that you try substituting this method here to determine whether it has been fixed in the release you are working with.

Class SimpleFilter

This class represents a basic FileFilter that accepts files with a given String extension, and displays a given String description in JFileChooser's "Files of Type" combo box. We have already seen and discussed this filter in section 14.1.9. It is used here to create our JAR and ZIP filters in the FileChooser1 constructor.

Class ZipFileView

This class extends FileView to provide a more user-friendly graphical representation of ZIP and JAR files in JFileChooser. Two instance variables, ImageIcon ZIP_ICON and ImageIcon JAR_ICON, represent small images corresponding to each archive type:  , jar.gif &  , zip.gif. This class is a straightforward adaptation of the sample FileView class presented in section 14.1.11.

Running the Code

Note that a JWindow is displayed as a simple splash screen because instantiation JFileChooser takes a significant amount of time. Press the "Create new Archive" button and select a name and location for the new archive file in the first file chooser that appears. Press its "OK" button and then select files to be added to that archive in the second chooser. Figure 14.12 shows FileChooser1 in action, and figures 14.13 and 14.14 show the first and second choosers that appear during the archive creation process.

Try uncompressing an existing archive. Press the "Extract from Archive" button and select an existing archive file in the first chooser that appears. Note the custom list component displayed in the bottom of this chooser, figure 14.15 illustrates. Each time an archive is selected its contents are displayed in this list. Select entries to extract and press the "Extract" button. A second chooser will appear, shown in figure 14.16, allowing selection of a destination directory.

Bug Alert!: As of Java 2 FCS, PropertyChangeEvents are not always fired as expected when JFileChooser's selection changes. This causes the updating of our custom list component to fail

occasionaly. We expect this to be fixed in a future release.

Part III - Advanced Topics

In chapters 15 through 21 we discuss the most advanced Swing components and the classes and interfaces that support them. We start with `JLayeredPane` in chapter 15, and implement our own MDI internal frame component from scratch. Chapter 16 is about `JDesktopPane` and `JIntenalFrame`, the MDI components that ship with Swing. This chapter culminates with the implementation of a multiuser networked desktop environment. Chapters 17 and 18 discuss the powerful and intricate tree and table components. Among other examples, we show how to build a directory browser using the tree component, and a sortable, JDBC-aware stocks application using the table component. Chapter 19 continues with text component coverage where chapter 11 left off, and discusses them at a much lower level. Chapter 20 is the most advanced chapter in this book and presents a complete RTF word processor application using `JTextPane` and several powerful custom dialogs used to manage fonts, paragraph formatting, find and replace, and spell checking. Chapter 21 discusses the pluggable look-and-feel architecture in detail and presents the contstruction of our own custom `LookAndFeel` implementation. This chapter includes examples showing how to implement custom component support for existing and third party L&Fs, and custom L&F support for both existing and custom components.

Chapter 15. Layered Panes and custom MDI

In this chapter:

- JLayeredPane
- Using JLayeredPane to enhance interfaces
- Creating a custom MDI: part I - Dragging panels
- Creating a custom MDI: part II - Resizability
- Creating a custom MDI: part III - Enhancements
- Creating a custom MDI: part IV - Selection and management
- Creating a custom MDI: part V - JavaBeans compliance

15.1 JLayeredPane

class javax.swing.JLayeredPane

`JLayeredPane` is one of the most powerful and robust components in the Swing package. It is a container with a practically infinite number of layers in which components can reside. Not only is there no limit to the number or type of components in each layer, but components can overlap one another.

Components within each layer of a `JLayeredPane` are organized by position. When overlapping is necessary those components with a higher valued position are displayed under those with a lower valued position. However, components in higher layers are displayed over all components residing in lower layers. It is important to get this overlapping hierarchy down early, as it can often be confusing.

Position is numbered from -1 to the number of components in the layer minus one. If we have N components in a layer then the component at position 0 will overlap the component at position 1, and the component at position 1 will overlap the component at position 2, etc. The lowest position is N-1. Note that position -1 represents the same position as N-1. Figure 15.1 illustrates position within a layer.

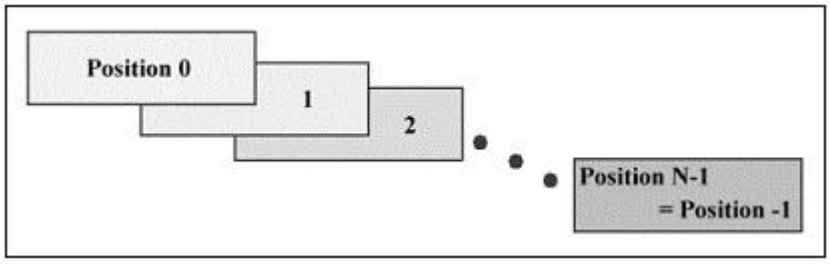


Figure 15.1 Position of components within a layer.
<<file figure15-1.gif>>

The layer a component resides at is often referred to as its depth. (Note that heavyweight components cannot conform to this notion of depth--see chapter 1 for more about this.) Each layer is represented by an `Integer` object (whereas the position of a component within each layer is represented by an `int` value). The `JLayeredPane` class defines six different `Integer` object constants representing, what are intended to be, commonly used layers: `FRAME_CONTENT_LAYER`, `DEFAULT_LAYER`, `PALETTE_LAYER`, `MODAL_LAYER`, `POPUP_LAYER`, and `DRAG_LAYER`.

Figure 15.2 illustrates the six standard layers and their overlap hierarchy.

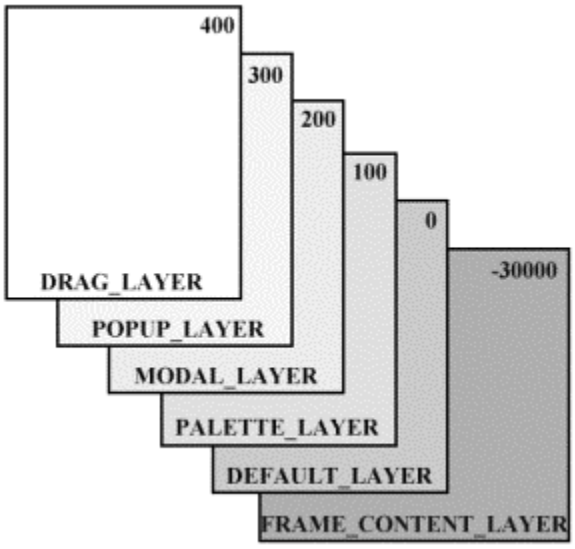


Figure 15.2. The JLayeredPane standard layers
<<file figure15-2.gif>>

We have discussed a component's layer and position within a layer. There is also another value associated with each component within a `JLayeredPane`. This value is called the index. The index is the same as position if we were to ignore layers. That is, components are assigned indices by starting from position 0 in the highest

layer and counting upward in position and downward in layer until all layers have been exhausted. The lowest component in the lowest layer will have index M-1, where M is the total number of components in the JLayeredPane. Similar to position, an index of -1 means the bottom-most component. (Note that the index is really a combination a component's layer and position. As such there are no methods to directly change a component's index within JLayeredPane. Although we can always query a component for its current index.)

There are three ways to add a component to a JLayeredPane. (Note that there is no add method defined within JLayeredPane itself.) Each method used to add a component to a JLayeredPane is defined within the Container class (see API docs).

1. We can add a component to a JLayeredPane using the add(Component component) method. This places the component in the layer represented by the Integer object with value 0, the DEFAULT_LAYER.
2. To add a component to a specific layer of a JLayeredPane we use the add(Component component, Object obj) method. We pass this method our component and an Integer object representing the desired layer. For layer 10 we would pass it: new Integer(10). If we wanted to place it on one of the standard layers, for instance the POPUP_LAYER, we could instead pass it JLayeredPane.POPUP_LAYER.
3. To add a component to a specific position within a specific layer we would use the add(Component component, Object obj, int index) method. Where the object is specified as above, and the int is the value representing the component's position with the layer.

15.2 Using JLayeredPane to enhance interfaces

As we mentioned early in chapter 4, JLayeredPane can sometimes come in handy when we want to manually position and size components. Because it's layout is null, it is not prone to the effects of resizing. Thus, when its parent is resized, a layered pane's children will stay in the same position and maintain the same size. (We saw a simple example of this in the beginning of chapter 6.) However, there are other ways to use JLayeredPane in typical interfaces. For instance, we can easily place a nice background image behind all of our components, giving life to an otherwise dull-looking panel.

Figure 15.3. The JLayeredPane standard layers
<<file figure15-3.gif>>

The Code: TestFrame.java
see \Chapter15\1

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

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

public class TestFrame extends JFrame
{
  public TestFrame() {
    super("JLayeredPane Demo");
    setSize(256,256);

    JPanel content = new JPanel();
    content.setLayout(new BoxLayout(content, BoxLayout.Y_AXIS));
    content.setOpaque(false);

    JLabel label1 = new JLabel("Username:");
    label1.setForeground(Color.white);
    content.add(label1);

    JTextField field = new JTextField(15);
    content.add(field);

    JLabel label2 = new JLabel("Password:");
    label2.setForeground(Color.white);
    content.add(label2);

    JPasswordField fieldPass = new JPasswordField(15);
    content.add(fieldPass);

    getContentPane().setLayout(new FlowLayout());
    getContentPane().add(content);
    ((JPanel)getContentPane()).setOpaque(false);

    ImageIcon earth = new ImageIcon("earth.jpg");
    JLabel backlabel = new JLabel(earth);
    getLayeredPane().add(backlabel,
      new Integer(Integer.MIN_VALUE));
    backlabel.setBounds(0,0,earth.getIconWidth(),
      earth.getIconHeight());

    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(l);

    setVisible(true);
  }

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

Most of this code should look familiar. We extend `JFrame` and create a new `JPanel` with a y-oriented `BoxLayout`. We make this panel non-opaque so our background image will show through, then we add four simple components: two `JLabels`, a `JTextField`, and a `JPasswordField`. We then set the layout of the `contentPane` to `FlowLayout` (remember that the `contentPane` has a `BorderLayout` by default), and add our panel to it. We also set the `contentPane`'s opaque property to false ensuring that our background will show through this panel as well. Finally we create a `JLabel` containing our background image, add it to our `JFrame`'s `layeredPane`, and set its bounds based on the background image's size.

15.3 Creating a custom MDI: part I - Dragging panels

JDesktopPane, Swing's version of a multiple document interface (MDI), is the prime example of a complicated layering environment derived from JLayeredPane. One of the best ways to fully leverage the power of JLayeredPane is through the construction of our own MDI environment from scratch. We will end up with a powerful desktop environment that we understand from the inside out and have complete control over. The central component we will be developing, InnerFrame, will ultimately imitate the functionality of JInternalFrame (discussed in the next chapter). In chapter 21, we will continue to develop this component by adding support for all the major look-and-feels, as well as our own custom look-and-feel. The next five sections are centered around this component's construction and proceed in a stepwise fashion.

Note: Due to the complexity of these examples, we suggest you have some time on your hands before attempting to understand each part. Some fragments are too mathematically dense (although not rigorous by any means) to warrant exaustive explanation. In these cases we expect the reader to either trust that it does what we say it does, or bite the bullet and plunge through the code.

We start our construction of InnerFrame by implementing movable, closeable, and iconifiable functionality. LayeredPaneDemo is a simple class we will use throughout each stage of development (with very few modifications) to load several InnerFrames and place them in a JLayeredPane.

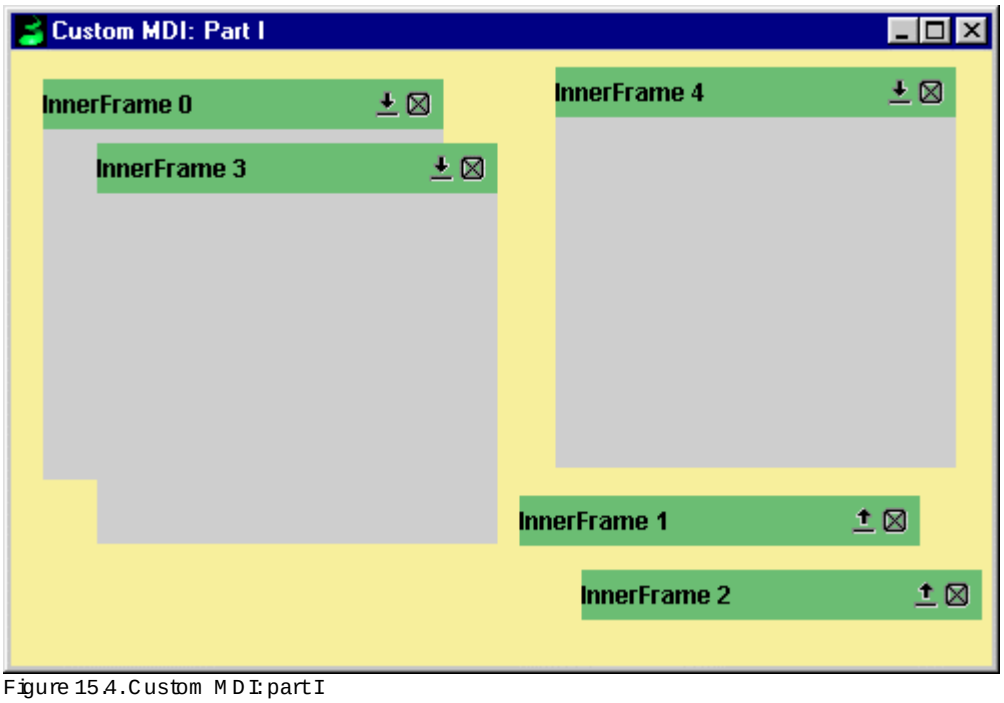


Figure 15.4. Custom MDI: part I
<<file figure15-4.gif>>

The Code: LayeredPaneDemo.java
see \Chapter15\2

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

import mdi.*;

public class LayeredPaneDemo extends JFrame
{
```

```
  public LayeredPaneDemo()
  {
    super("Custom MDI: Part I");
    setSize(570,400);
    getContentPane().setBackground(new Color(244,232,152));

    getLayeredPane().setOpaque(true);

    InnerFrame[] frames = new InnerFrame[5];
    for(int i=0; i<5; i++) {
      frames[i] = new InnerFrame("InnerFrame " + i);
      frames[i].setBounds(50+i*20, 50+i*20, 200, 200);
      getLayeredPane().add(frames[i]);
    }

    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };

    Dimension dim = getToolkit().getScreenSize();
    setLocation(dim.width/2-getWidth()/2,
      dim.height/2-getHeight()/2);

    ImageIcon image = new ImageIcon("spiral.gif");
    setIconImage(image.getImage());
    addWindowListener(l);
    setVisible(true);
  }

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

The Code: InnerFrame.java
see \Chapter15\2\mdi

```
package mdi;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.EmptyBorder;

public class InnerFrame
extends JPanel
{
  private static String IMAGE_DIR = "mdi" + java.io.File.separator;
  private static ImageIcon ICONIZE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"iconize.gif");
  private static ImageIcon RESTORE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"restore.gif");
  private static ImageIcon CLOSE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"close.gif");
  private static ImageIcon PRESS_CLOSE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"pressclose.gif");
  private static ImageIcon PRESS_RESTORE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"pressrestore.gif");
  private static ImageIcon PRESS_ICONIZE_BUTTON_ICON =
```

```
    new ImageIcon(IMAGE_DIR+"pressiconize.gif");
  private static final int WIDTH = 200;
  private static final int HEIGHT = 200;
  private static final int TITLE_BAR_HEIGHT = 25;
  private static Color TITLE_BAR_BG_COLOR =
    new Color(108,190,116);

  private String m_title;
  private JLabel m_titleLabel;

  private boolean m_iconified;

  private JPanel m_titlePanel;
  private JPanel m_contentPanel;
  private JPanel m_buttonPanel;
  private JPanel m_buttonWrapperPanel;

  private InnerFrameButton m_iconize;
  private InnerFrameButton m_close;

  public InnerFrame(String title) {
    m_title = title;
    setLayout(new BorderLayout());
    createTitleBar();
    m_contentPanel = new JPanel();
    add(m_titlePanel, BorderLayout.NORTH);
    add(m_contentPanel, BorderLayout.CENTER);
  }

  public void toFront() {
    if (getParent() instanceof JLayeredPane)
      ((JLayeredPane) getParent()).moveToFront(this);
  }

  public void close() {
    if (getParent() instanceof JLayeredPane) {
      JLayeredPane jlp = (JLayeredPane) getParent();
      jlp.remove(InnerFrame.this);
      jlp.repaint();
    }
  }

  public void setIconified(boolean b) {
    m_iconified = b;
    if (b) {
      setBounds(getX(), getY(), WIDTH, TITLE_BAR_HEIGHT);
      m_iconize.setIcon(RESTORE_BUTTON_ICON);
      m_iconize.setPressedIcon(PRESS_RESTORE_BUTTON_ICON);
    }
    else {
      setBounds(getX(), getY(), WIDTH, HEIGHT);
      m_iconize.setIcon(ICONIZE_BUTTON_ICON);
      m_iconize.setPressedIcon(PRESS_ICONIZE_BUTTON_ICON);
      revalidate();
    }
  }

  public boolean isIconified() {
    return m_iconified;
  }

  ////////////////////////////////////////
  /////////////// Title Bar /////////////////
```

```
//////////////////////////////////////////

// create the title bar m_titlePanel
public void createTitleBar() {
  m_titlePanel = new JPanel() {
    public Dimension getPreferredSize() {
      return new Dimension(InnerFrame.WIDTH,
        InnerFrame.TITLE_BAR_HEIGHT);
    }
  };
  m_titlePanel.setLayout(new BorderLayout());
  m_titlePanel.setOpaque(true);
  m_titlePanel.setBackground(TITLE_BAR_BG_COLOR);

  m_titleLabel = new JLabel(m_title);
  m_titleLabel.setForeground(Color.black);

  m_close = new InnerFrameButton(CLOSE_BUTTON_ICON);
  m_close.setPressedIcon(PRESS_CLOSE_BUTTON_ICON);
  m_close.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      InnerFrame.this.close();
    }
  });

  m_iconize = new InnerFrameButton(ICONIZE_BUTTON_ICON);
  m_iconize.setPressedIcon(PRESS_ICONIZE_BUTTON_ICON);
  m_iconize.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      InnerFrame.this.setIconified(
        !InnerFrame.this.isIconified());
    }
  });

  m_buttonWrapperPanel = new JPanel();
  m_buttonWrapperPanel.setOpaque(false);
  m_buttonPanel = new JPanel(new GridLayout(1,2));
  m_buttonPanel.setOpaque(false);
  m_buttonPanel.add(m_iconize);
  m_buttonPanel.add(m_close);
  m_buttonPanel.setAlignmentX(0.5f);
  m_buttonPanel.setAlignmentY(0.5f);
  m_buttonWrapperPanel.add(m_buttonPanel);

  m_titlePanel.add(m_titleLabel, BorderLayout.CENTER);
  m_titlePanel.add(m_buttonWrapperPanel, BorderLayout.EAST);

  InnerFrameTitleBarMouseAdapter iftbma =
    new InnerFrameTitleBarMouseAdapter(this);
  m_titlePanel.addMouseListener(iftbma);
  m_titlePanel.addMouseMotionListener(iftbma);
}

// title bar mouse adapter for frame dragging
class InnerFrameTitleBarMouseAdapter
extends MouseInputAdapter
{
  InnerFrame m_if;
  int m_XDifference, m_YDifference;
  boolean m_dragging;

  public InnerFrameTitleBarMouseAdapter(InnerFrame inf) {
    m_if = inf;
```

```
    }

    public void mouseDragged(MouseEvent e) {
      if (m_dragging)
        m_if.setLocation(e.getX()-m_XDifference + getX(),
          e.getY()-m_YDifference + getY());
    }

    public void mousePressed(MouseEvent e) {
      m_if.toFront();
      m_XDifference = e.getX();
      m_YDifference = e.getY();
      m_dragging = true;
    }

    public void mouseReleased(MouseEvent e) {
      m_dragging = false;
    }
  }

  // custom button class for title bar
  class InnerFrameButton extends JButton
  {
    Dimension m_dim;

    public InnerFrameButton(ImageIcon ii) {
      super(ii);
      m_dim = new Dimension(
        ii.getIconWidth(), ii.getIconHeight());
      setOpaque(false);
      setContentAreaFilled(false);
      setBorder(null);
    }

    public Dimension getPreferredSize() {
      return m_dim;
    }

    public Dimension getMinimumSize() {
      return m_dim;
    }

    public Dimension getMaximumSize() {
      return m_dim;
    }
  }
}
```

Understanding the Code:

Class LayeredPaneDemo

We are familiar with most of the code in this class. Note that the `mdi` package is imported, which is the package `InnerFrame` resides in. The only purpose of `LayeredPaneDemo` is to provide a container for several `InnerFrames`. An array of five `InnerFrames` is created and they are placed in our `JFrame`'s layered pane, at the default layer, in a cascading fashion. Note that the layered pane's opaque property has been explicity enabled (`setOpaque(true)`). This is to ensure smooth repainting when dragging our `InnerFrames` by guaranteeing that all layered pane pixels will be painted (see chapter 2 for more about opacity).

Class mdi.InnerFrame

This is our custom internal frame we will be expanding on throughout the remainder of this chapter. It is defined within the `mdi` package and extends `JPanel`.

Class variables:

`String IMAGE_DIR`: directory header string used to locate images in the `mdi` package

`ImageIcon ICONIZE_BUTTON_ICON`: icon used for the iconify button =  [black interior]

`ImageIcon RESTORE_BUTTON_ICON`: icon used for the iconfy button when in the iconified state (representing de-iconify) =  [black interior]

`ImageIcon CLOSE_BUTTON_ICON`: icon used for the close button = [black interior]

`ImageIcon PRESS_CLOSE_BUTTON_ICON`: icon used for the close button when it is in the pressed state = [dark green interior]

`ImageIcon PRESS_RESTORE_BUTTON_ICON`: icon used for the iconfy button in the iconified state (representing de-iconify) when also in the pressed state = [dark green interior]

`ImageIcon PRESS_ICONIZE_BUTTON_ICON`: icon used for the iconify button when in the pressed state = [dark green interior]

`Color C_BUTTONBACK`: the titlebar button background color

Several instance variables are also necessary:

`int WIDTH`: InnerFrame's width.

`int HEIGHT`: InnerFrame's height.

`int TITLE_BAR_HEIGHT`: default height of the title bar.

`Color TITLE_BAR_BG_COLOR`: default `Color` the title bar.

Instance variables

`String m_title`: title bar `String`.

`JLabel m_titleLabel`: label contained in the title bar displaying the title string, `m_title`.

`boolean m_iconified`: true when in the iconified state.

`JPanel m_titlePanel`: panel representing the title bar.

`JPanel m_contentPanel`: the central `InnerFrame` container.

`JPanel m_buttonPanel`: small panel for frame buttons.

`JPanel m_buttonWrapperPanel`: panel to surround `m_buttonPanel` to allow correct alignment.

`InnerFrameButton m_iconize`: custom frame button for iconification/de-iconification.

`InnerFrameButton m_close`: custom frame button for closing.

The `InnerFrame` constructor takes a title `String` as its only parameter and sets the title variable accordingly. We use a `BorderLayout` and call our `createTitleBar()` method to initialize `m_titlePanel`, which represents our title bar (see below). We then initialize `m_contentPanel` which acts as our frame's central container. We add these components to `InnerFrame` in the `NORTH` and `CENTER` regions respectively.

The `toFront()` method is responsible for moving `InnerFrame` to the frontmost position in its layer. This is called from `m_titlePanel`'s mouse adapter when the title bar is pressed, as we will see below. This will

only function if InnerFrame is within a JLayeredPane (or JLayeredPane subclass--such as JDesktopPane).

The close() method is responsible for removing an InnerFrame from its parent container. This is called when the close button, m_close, is clicked within the title bar. We specifically only allow removal to work from within a JLayeredPane. If this component is added to any other container we are assuming that it is not meant for removal in this fashion. We also keep a reference to the parent layered pane so that it can be repainted after we have been removed.

The setIconified() method controls iconification and deiconification (often called restore) and reduces the height of InnerFrame to that of the title bar. When an iconification occurs we change the m_iconize button's regular and pressed state icons to those representing de-iconification. Similarly, when a de-iconify occurs we restore the original height of InnerFrame and replace the icons with those representing iconification.

The createTitleBar() method is responsible for populating and laying out m_titlePanel, our title bar. It starts by initializing m_titlePanel with a preferred size according to our WIDTH and TITLE_BAR_HEIGHT constants. m_titlePanel uses a BorderLayout and its background is set to TITLE_BAR_BG_COLOR. (Note that these hard-coded values will be replaced by variables in future stages of development.) m_titleLabel is created with the m_title String that was passed to the InnerFrame constructor.

The m_close button is an instance of InnerFrameButton (see below) and uses custom ImageIcon's for normal and pressed states. An ActionListener is added to simply invoke close() when this button is pressed. The m_iconize button is created in a similar fashion, except its ActionListener is constructed to invoke the setIconified() method. Once these buttons are constructed, we place them in m_buttonPanel using a GridLayout, and place this panel within our m_buttonWrapperPanel which uses a FlowLayout. This allows for perfect button alignment, using JComponent's setAlignmentX/Y() methods, while enforcing equal size among buttons no greater than their preferred sizes.

Finally, the createTitleBar() method ends by creating an InnerFrameTitleBarMouseAdapter and attaches it to m_titlePanel.

Class mdi.InnerFrame.InnerFrameTitleBarMouseAdapter

This class extends javax.swing.event.MouseInputAdapter and holds a reference to the InnerFrame it is associated with. The function of mousePressed() is to obtain the base coordinate offsets, m_XDifference and m_YDifference, for use in the mouseDragged method, and set the m_dragging flag to true (because a mouse press always precedes a mouse drag). The mouseDragged() method is called whenever an InnerFrame is dragged. This method starts by testing whether the m_dragging flag is set to true. If it is InnerFrame is moved to a new location. To calculate the new position as it is dragged this method uses the current InnerFrame coordinates, getX() and getY(), plus the difference of the mouseDragged() event coordinates, e.getX() and e.getY(), and the base offset coordinates, m_XDifference and m_YDifference. These base offset coordinates, initialized within the mousePressed() method, are necessary to keep mouse position with regard to the upper left-hand corner of InnerFrame remains constant. The mouseReleased() method's only function is to set the m_dragging flag to false when the mouse button is released.

Class mdi.InnerFrame.InnerFrameButton

The InnerFrameButton class extends JButton and its constructor takes an ImageIcon parameter. It uses this icon to construct a Dimension instance which is in turn used for the button's minimum, maximum, and preferred sizes. The button is then set to non-opaque, and we call setContentAreaFilled(false) to guarantee that its background will never be filled. We also remove its border to give it an embedded look.

Running the Code

Figure 15.4 shows `LayeredPaneDemo` in action. Experiment with moving, iconifying, de-iconifying and closing our custom frames. Note that a frame can be selected even when its title bar lies behind the content of another frame. Mouse events pass right through our frame's content panels. This is just one of the many flaws that need to be accounted for. Also notice that we can drag any frame completely outside of the layered pane. The most immediately noticeable functionality that `InnerFrame` lacks is resizability. This is the single most difficult feature to implement, and is the subject of the next section.

15.4 Creating a custom MDI: part II - Resizability

In this stage, four distinct components are added to `InnerFrame`'s `NORTH`, `SOUTH`, `EAST`, and `WEST` regions. Each of these components, defined by the `NorthResizeEdge`, `SouthResizeEdge`, `EastResizeEdge`, and `WestResizeEdge` inner classes respectively, provide the functionality needed to allow proper resizing in all directions. The title bar and content panel of `InnerFrame` are now wrapped inside another panel that is placed in `InnerFrame`'s `CENTER` region. Additionaly, we add a label containing an icon to the `WEST` region of the title bar, and allow dynamic title bar height based on the size of the icon used.

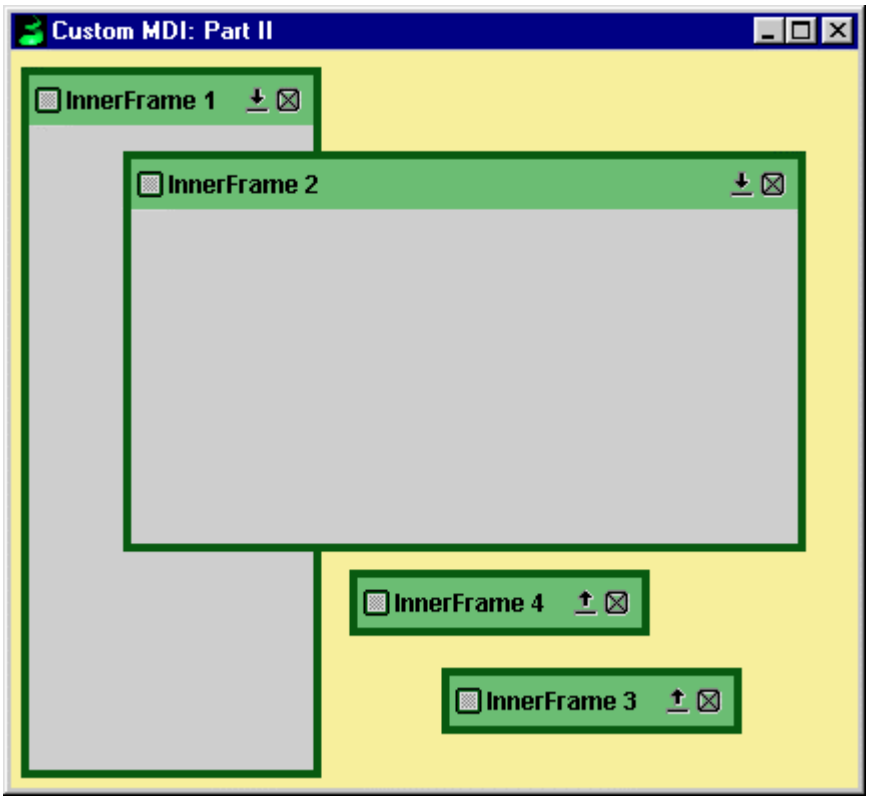


Figure 15.6. Custom MDI: part II with default icon

<<file figure15-6.gif>>

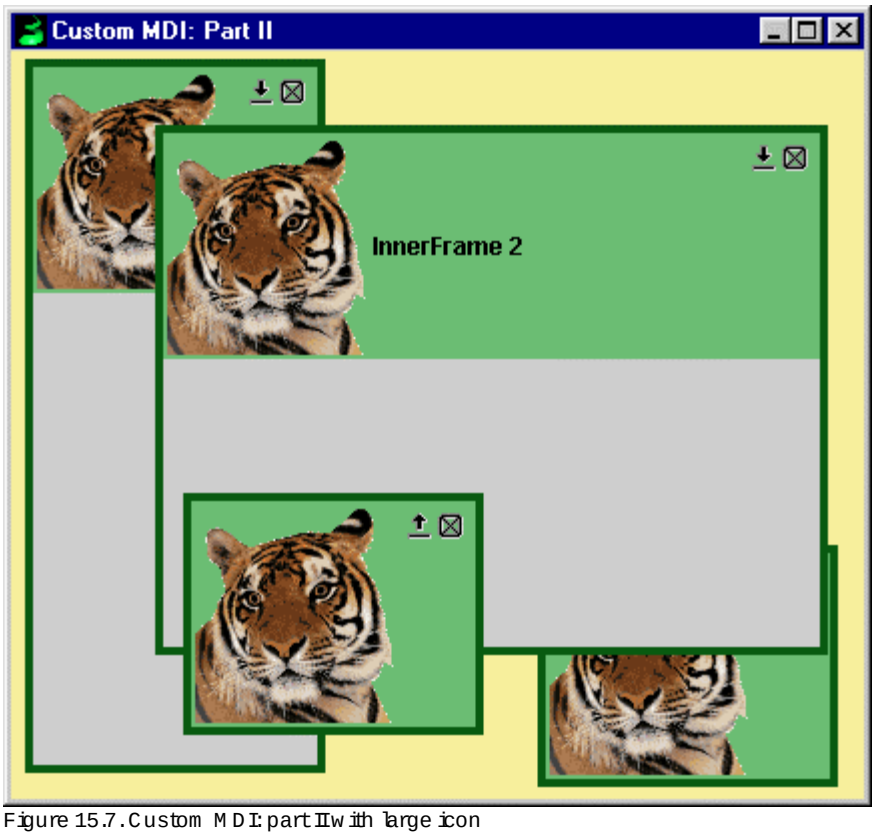


Figure 15.7. Custom MDI: part II with large icon
<<file figure15-7.gif>>

The Code: InnerFrame.java
see \Chapter15\3\mdi

```
package mdi;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.EmptyBorder;

public class InnerFrame
extends JPanel
{
  // Unchanged code

  private static ImageIcon DEFAULT_FRAME_ICON =
    new ImageIcon(IMAGE_DIR+"default.gif");
  private static int BORDER_THICKNESS = 4;
  private static int WIDTH = 200;
  private static int HEIGHT = 200;
  private static int TITLE_BAR_HEIGHT = 25;
  private static int FRAME_ICON_PADDING = 2;
  private static int ICONIZED_WIDTH = 150;
  private static Color TITLE_BAR_BG_COLOR =
    new Color(108,190,116);
  private static Color BORDER_COLOR = new Color(8,90,16);

  private int m_titleBarHeight = TITLE_BAR_HEIGHT;
  private int m_width = WIDTH;
```

```
  private int m_height = HEIGHT;
  private int m_iconizedWidth = ICONIZED_WIDTH;

  private String m_title;
  private JLabel m_titleLabel;
  private JLabel m_iconLabel;

  private boolean m_iconified;

  private boolean m_iconizeable;
  private boolean m_resizeable;
  private boolean m_closeable;

  // used to wrap title bar and contentPanel
  private JPanel m_frameContentPanel;

  private JPanel m_titlePanel;
  private JPanel m_contentPanel;
  private JPanel m_buttonPanel;
  private JPanel m_buttonWrapperPanel;

  private InnerFrameButton m_iconize;
  private InnerFrameButton m_close;

  private ImageIcon m_frameIcon = DEFAULT_FRAME_ICON;

  private NorthResizeEdge m_northResizer;
  private SouthResizeEdge m_southResizer;
  private EastResizeEdge m_eastResizer;
  private WestResizeEdge m_westResizer;

  public InnerFrame() {
    this("");
  }

  public InnerFrame(String title) {
    this(title, null);
  }

  public InnerFrame(String title, ImageIcon frameIcon) {
    this(title, frameIcon, true, true, true);
  }

  public InnerFrame(String title, ImageIcon frameIcon,
   boolean resizeable, boolean iconizeable, boolean closeable) {
    super.setLayout(new BorderLayout());
    attachNorthResizeEdge();
    attachSouthResizeEdge();
    attachEastResizeEdge();
    attachWestResizeEdge();
    populateInnerFrame();

    setTitle(title);
    setResizeable(resizeable);
    setIconizeable(iconizeable);
    setCloseable(closeable);
    if (frameIcon != null)
      setFrameIcon(frameIcon);
  }

  protected void populateInnerFrame() {
    m_frameContentPanel = new JPanel();
    m_frameContentPanel.setLayout(new BorderLayout());
```

```
    createTitleBar();
    m_contentPanel = new JPanel();
    m_frameContentPanel.add(m_titlePanel, BorderLayout.NORTH);
    m_frameContentPanel.add(m_contentPanel, BorderLayout.CENTER);
    super.add(m_frameContentPanel, BorderLayout.CENTER);
  }

  public Component add(Component c) {
    return((m_contentPanel == null)
      ? null : m_contentPanel.add(c));
  }

  public void setLayout(LayoutManager mgr) {
    if (m_contentPanel != null)
      m_contentPanel.setLayout(mgr);
  }

  // Unchanged code

  public boolean isIconizeable() {
    return m_iconizeable;
  }

  public void setIconizeable(boolean b) {
    m_iconizeable = b;
    m_iconize.setVisible(b);
    m_titlePanel.revalidate();
  }

  public boolean isCloseable() {
    return m_closeable;
  }

  public void setCloseable(boolean b) {
    m_closeable = b;
    m_close.setVisible(b);
    m_titlePanel.revalidate();
  }

  public boolean isIconified() {
    return m_iconified;
  }

  public void setIconified(boolean b) {
    m_iconified = b;
    if (b) {
      m_width = getWidth();     // remember width
      m_height = getHeight();   // remember height
      setBounds(getX(), getY(), ICONIZED_WIDTH,
        m_titleBarHeight + 2*BORDER_THICKNESS);
      m_iconize.setIcon(RESTORE_BUTTON_ICON);
      m_iconize.setPressedIcon(PRESS_RESTORE_BUTTON_ICON);
      setResizeable(false);
    }
    else {
      setBounds(getX(), getY(), m_width, m_height);
      m_iconize.setIcon(ICONIZE_BUTTON_ICON);
      m_iconize.setPressedIcon(PRESS_ICONIZE_BUTTON_ICON);
      setResizeable(true);
    }
    revalidate();
  }
```

```
  ////////////////////////////////////////
  ///////////////// Title Bar /////////////////
  ////////////////////////////////////////

  public void setFrameIcon(ImageIcon fi) {
    m_frameIcon = fi;

    if (fi != null) {
      if (m_frameIcon.getIconHeight() > TITLE_BAR_HEIGHT)
        setTitleBarHeight(
          m_frameIcon.getIconHeight() + 2*FRAME_ICON_PADDING);
      m_iconLabel.setIcon(m_frameIcon);
    }
    else setTitleBarHeight(TITLE_BAR_HEIGHT);
    revalidate();
  }

  public ImageIcon getFrameIcon() {
    return m_frameIcon;
  }

  public void setTitle(String s) {
    m_title = s;
    m_titleLabel.setText(s);
    m_titlePanel.repaint();
  }

  public String getTitle() {
    return m_title;
  }

  public void setTitleBarHeight(int h) {
    m_titleBarHeight = h;
  }

  public int getTitleBarHeight() {
    return m_titleBarHeight;
  }

  // create the title bar: m_titlePanel
  protected void createTitleBar() {
    m_titlePanel = new JPanel() {
      public Dimension getPreferredSize() {
        return new Dimension(InnerFrame.this.getWidth(),
          m_titleBarHeight);
      }
    };
    m_titlePanel.setLayout(new BorderLayout());
    m_titlePanel.setOpaque(true);
    m_titlePanel.setBackground(TITLE_BAR_BG_COLOR);

    m_titleLabel = new JLabel();
    m_titleLabel.setForeground(Color.black);

    // Unchanged code

    m_iconLabel = new JLabel();
    m_iconLabel.setBorder(new EmptyBorder(
      FRAME_ICON_PADDING, FRAME_ICON_PADDING,
      FRAME_ICON_PADDING, FRAME_ICON_PADDING));
    if (m_frameIcon != null)
      m_iconLabel.setIcon(m_frameIcon);
```

```
    m_titlePanel.add(m_titleLabel, BorderLayout.CENTER);
    m_titlePanel.add(m_buttonWrapperPanel, BorderLayout.EAST);
    m_titlePanel.add(m_iconLabel, BorderLayout.WEST);

    InnerFrameTitleBarMouseAdapter iftbma =
      new InnerFrameTitleBarMouseAdapter(this);
    m_titlePanel.addMouseListener(iftbma);
    m_titlePanel.addMouseMotionListener(iftbma);
  }

  // Unchanged code

  ////////////////////////////////////////////
  ///////////////// Resizability /////////////////
  ////////////////////////////////////////////

  public boolean isResizeable() {
    return m_resizeable;
  }

  public void setResizeable(boolean b) {
    if (!b && m_resizeable == true) {
      m_northResizer.removeMouseListener(m_northResizer);
      m_northResizer.removeMouseMotionListener(m_northResizer);
      m_southResizer.removeMouseListener(m_southResizer);
      m_southResizer.removeMouseMotionListener(m_southResizer);
      m_eastResizer.removeMouseListener(m_eastResizer);
      m_eastResizer.removeMouseMotionListener(m_eastResizer);
      m_westResizer.removeMouseListener(m_westResizer);
      m_westResizer.removeMouseMotionListener(m_westResizer);
    }
    else if (m_resizeable == false) {
      m_northResizer.addMouseListener(m_northResizer);
      m_northResizer.addMouseMotionListener(m_northResizer);
      m_southResizer.addMouseListener(m_southResizer);
      m_southResizer.addMouseMotionListener(m_southResizer);
      m_eastResizer.addMouseListener(m_eastResizer);
      m_eastResizer.addMouseMotionListener(m_eastResizer);
      m_westResizer.addMouseListener(m_westResizer);
      m_westResizer.addMouseMotionListener(m_westResizer);
    }
    m_resizeable = b;
  }

  protected void attachNorthResizeEdge() {
    m_northResizer = new NorthResizeEdge(this);
    super.add(m_northResizer, BorderLayout.NORTH);
  }

  protected void attachSouthResizeEdge() {
    m_southResizer = new SouthResizeEdge(this);
    super.add(m_southResizer, BorderLayout.SOUTH);
  }

  protected void attachEastResizeEdge() {
    m_eastResizer = new EastResizeEdge(this);
    super.add(m_eastResizer, BorderLayout.EAST);
  }

  protected void attachWestResizeEdge() {
    m_westResizer = new WestResizeEdge(this);
    super.add(m_westResizer, BorderLayout.WEST);
  }
```

```
class EastResizeEdge extends JPanel
implements MouseListener, MouseMotionListener {
  private int WIDTH = BORDER_THICKNESS;
  private int MIN_WIDTH = ICONIZED_WIDTH;
  private boolean m_dragging;
  private JComponent m_resizeComponent;

  protected EastResizeEdge(JComponent c) {
    m_resizeComponent = c;
    setOpaque(true);
    setBackground(BORDER_COLOR);
  }

  public Dimension getPreferredSize() {
    return new Dimension(WIDTH, m_resizeComponent.getHeight());
  }

  public void mouseClicked(MouseEvent e) {}
  public void mouseMoved(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {
    m_dragging = false;
  }

  public void mouseDragged(MouseEvent e) {
    if (m_resizeComponent.getWidth() + e.getX() >= MIN_WIDTH)
      m_resizeComponent.setBounds(m_resizeComponent.getX(),
        m_resizeComponent.getY(),
        m_resizeComponent.getWidth() + e.getX(),
        m_resizeComponent.getHeight());
    else
      m_resizeComponent.setBounds(m_resizeComponent.getX(),
        m_resizeComponent.getY(),
        MIN_WIDTH, m_resizeComponent.getHeight());
    m_resizeComponent.validate();
  }

  public void mouseEntered(MouseEvent e) {
    if (!m_dragging)
      setCursor(Cursor.getPredefinedCursor(
        Cursor.E_RESIZE_CURSOR));
  }

  public void mouseExited(MouseEvent e) {
    if (!m_dragging)
      setCursor(Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
  }

  public void mousePressed(MouseEvent e) {
    toFront();
    m_dragging = true;
  }
}

class WestResizeEdge extends JPanel
implements MouseListener, MouseMotionListener {
  private int WIDTH = BORDER_THICKNESS;
  private int MIN_WIDTH = ICONIZED_WIDTH;
  private int m_dragX, m_rightX;
  private boolean m_dragging;
  private JComponent m_resizeComponent;
```

```
    protected WestResizeEdge(JComponent c) {
      m_resizeComponent = c;
      setOpaque(true);
      setBackground(BORDER_COLOR);
    }

    public Dimension getPreferredSize() {
      return new Dimension(WIDTH, m_resizeComponent.getHeight());
    }

    public void mouseClicked(MouseEvent e) {}
    public void mouseMoved(MouseEvent e) {}

    public void mouseReleased(MouseEvent e) {
      m_dragging = false;
    }

    public void mouseDragged(MouseEvent e) {
      if (m_resizeComponent.getWidth()-
       (e.getX()-m_dragX) >= MIN_WIDTH)
        m_resizeComponent.setBounds(
          m_resizeComponent.getX() + (e.getX()-m_dragX),
          m_resizeComponent.getY(),
          m_resizeComponent.getWidth()-(e.getX()-m_dragX),
          m_resizeComponent.getHeight());
      else
        if (m_resizeComponent.getX() + MIN_WIDTH < m_rightX)
          m_resizeComponent.setBounds(m_rightX-MIN_WIDTH,
            m_resizeComponent.getY(),
            MIN_WIDTH, m_resizeComponent.getHeight());
        else
          m_resizeComponent.setBounds(m_resizeComponent.getX(),
            m_resizeComponent.getY(),
            MIN_WIDTH, m_resizeComponent.getHeight());
      m_resizeComponent.validate();
    }

    public void mouseEntered(MouseEvent e) {
      if (!m_dragging)
        setCursor(Cursor.getPredefinedCursor(
          Cursor.W_RESIZE_CURSOR));
    }

    public void mouseExited(MouseEvent e) {
      if (!m_dragging)
        setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
    }

    public void mousePressed(MouseEvent e) {
      toFront();
      m_rightX = m_resizeComponent.getX() +
        m_resizeComponent.getWidth();
      m_dragging = true;
      m_dragX = e.getX();
    }
  }

  class NorthResizeEdge extends JPanel
  implements MouseListener, MouseMotionListener {
    private static final int NORTH = 0;
    private static final int NORTHEAST = 1;
    private static final int NORTHWEST = 2;
```

```
private int CORNER = 10;
private int HEIGHT = BORDER_THICKNESS;
private int MIN_WIDTH = ICONIZED_WIDTH;
private int MIN_HEIGHT = TITLE_BAR_HEIGHT+(2*HEIGHT);
private int m_width, m_dragX, m_dragY, m_rightX, m_lowerY;
private boolean m_dragging;
private JComponent m_resizeComponent;
private int m_mode;

protected NorthResizeEdge(JComponent c) {
  m_resizeComponent = c;
  setOpaque(true);
  setBackground(BORDER_COLOR);
}

public Dimension getPreferredSize() {
  return new Dimension(m_resizeComponent.getWidth(), HEIGHT);
}

public void mouseClicked(MouseEvent e) {}

public void mouseMoved(MouseEvent e) {
  if (!m_dragging) {
    if (e.getX() < CORNER) {
      setCursor(Cursor.getPredefinedCursor(
        Cursor.NW_RESIZE_CURSOR));
    }
    else if(e.getX() > getWidth()-CORNER) {
      setCursor(Cursor.getPredefinedCursor(
        Cursor.NE_RESIZE_CURSOR));
    }
    else {
      setCursor(Cursor.getPredefinedCursor(
        Cursor.N_RESIZE_CURSOR));
    }
  }
}

public void mouseReleased(MouseEvent e) {
  m_dragging = false;
}

public void mouseDragged(MouseEvent e) {
  int h = m_resizeComponent.getHeight();
  int w = m_resizeComponent.getWidth();
  int x = m_resizeComponent.getX();
  int y = m_resizeComponent.getY();
  int ex = e.getX();
  int ey = e.getY();
  switch (m_mode) {
    case NORTH:
      if (h-(ey-m_dragY) >= MIN_HEIGHT)
        m_resizeComponent.setBounds(x, y + (ey-m_dragY),
          w, h-(ey-m_dragY));
      else
          m_resizeComponent.setBounds(x,
            m_lowerY-MIN_HEIGHT, w, MIN_HEIGHT);
      break;
    case NORTHEAST:
      if (h-(ey-m_dragY) >= MIN_HEIGHT
      && w + (ex-(getWidth()-CORNER)) >= MIN_WIDTH)
        m_resizeComponent.setBounds(x,
          y + (ey-m_dragY), w + (ex-(getWidth()-CORNER)),
```

```
            h-(ey-m_dragY));
        else if (h-(ey-m_dragY) >= MIN_HEIGHT
        && !(w + (ex-(getWidth()-CORNER)) >= MIN_WIDTH))
          m_resizeComponent.setBounds(x,
            y + (ey-m_dragY), MIN_WIDTH, h-(ey-m_dragY));
        else if (!(h-(ey-m_dragY) >= MIN_HEIGHT)
        && w + (ex-(getWidth()-CORNER)) >= MIN_WIDTH)
          m_resizeComponent.setBounds(x,
            m_lowerY-MIN_HEIGHT, w + (ex-(getWidth()-CORNER)),
              MIN_HEIGHT);
        else
          m_resizeComponent.setBounds(x,
            m_lowerY-MIN_HEIGHT, MIN_WIDTH, MIN_HEIGHT);
        break;
      case NORTHWEST:
        if (h-(ey-m_dragY) >= MIN_HEIGHT
        && w-(ex-m_dragX) >= MIN_WIDTH)
          m_resizeComponent.setBounds(x + (ex-m_dragX),
            y + (ey-m_dragY), w-(ex-m_dragX),
              h-(ey-m_dragY));
        else if (h-(ey-m_dragY) >= MIN_HEIGHT
        && !(w-(ex-m_dragX) >= MIN_WIDTH)) {
          if (x + MIN_WIDTH < m_rightX)
            m_resizeComponent.setBounds(m_rightX-MIN_WIDTH,
              y + (ey-m_dragY), MIN_WIDTH, h-(ey-m_dragY));
          else
            m_resizeComponent.setBounds(x,
              y + (ey-m_dragY), w, h-(ey-m_dragY));
        }
        else if (!(h-(ey-m_dragY) >= MIN_HEIGHT)
        && w-(ex-m_dragX) >= MIN_WIDTH)
          m_resizeComponent.setBounds(x + (ex-m_dragX),
            m_lowerY-MIN_HEIGHT, w-(ex-m_dragX), MIN_HEIGHT);
        else
          m_resizeComponent.setBounds(m_rightX-MIN_WIDTH,
            m_lowerY-MIN_HEIGHT, MIN_WIDTH, MIN_HEIGHT);
        break;
    }
    m_rightX = x + w;
    m_resizeComponent.validate();
  }

  public void mouseEntered(MouseEvent e) {
    mouseMoved(e);
  }

  public void mouseExited(MouseEvent e) {
    if (!m_dragging)
      setCursor(Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
  }

  public void mousePressed(MouseEvent e) {
    toFront();
    m_dragging = true;
    m_dragX = e.getX();
    m_dragY = e.getY();
    m_lowerY = m_resizeComponent.getY()
      + m_resizeComponent.getHeight();
    if (e.getX() < CORNER) {
      m_mode = NORTHWEST;
    }
    else if(e.getX() > getWidth()-CORNER) {
```

```
      m_mode = NORTHEAST;
    }
    else {
      m_mode = NORTH;
    }
  }
}

class SouthResizeEdge extends JPanel
implements MouseListener, MouseMotionListener {
  private static final int SOUTH = 0;
  private static final int SOUTHEAST = 1;
  private static final int SOUTHWEST = 2;
  private int CORNER = 10;
  private int HEIGHT = BORDER_THICKNESS;
  private int MIN_WIDTH = ICONIZED_WIDTH;
  private int MIN_HEIGHT = TITLE_BAR_HEIGHT+(2*HEIGHT);
  private int m_width, m_dragX, m_dragY, m_rightX;
  private boolean m_dragging;
  private JComponent m_resizeComponent;
  private int m_mode;

  protected SouthResizeEdge(JComponent c) {
    m_resizeComponent = c;
    setOpaque(true);
    setBackground(BORDER_COLOR);
  }

  public Dimension getPreferredSize() {
    return new Dimension(m_resizeComponent.getWidth(), HEIGHT);
  }

  public void mouseClicked(MouseEvent e) {}

  public void mouseMoved(MouseEvent e) {
    if (!m_dragging) {
      if (e.getX() < CORNER) {
        setCursor(Cursor.getPredefinedCursor(
          Cursor.SW_RESIZE_CURSOR));
      }
      else if(e.getX() > getWidth()-CORNER) {
        setCursor(Cursor.getPredefinedCursor(
          Cursor.SE_RESIZE_CURSOR));
      }
      else {
        setCursor(Cursor.getPredefinedCursor(
          Cursor.S_RESIZE_CURSOR));
      }
    }
  }

  public void mouseReleased(MouseEvent e) {
    m_dragging = false;
  }

  public void mouseDragged(MouseEvent e) {
    int h = m_resizeComponent.getHeight();
    int w = m_resizeComponent.getWidth();
    int x = m_resizeComponent.getX();
    int y = m_resizeComponent.getY();
    int ex = e.getX();
    int ey = e.getY();
    switch (m_mode) {
```

```
      case SOUTH:
        if (h+(ey-m_dragY) >= MIN_HEIGHT)
         m_resizeComponent.setBounds(x, y, w, h+(ey-m_dragY));
        else
          m_resizeComponent.setBounds(x, y, w, MIN_HEIGHT);
        break;
      case SOUTHEAST:
        if (h+(ey-m_dragY) >= MIN_HEIGHT
          && w + (ex-(getWidth()-CORNER)) >= MIN_WIDTH)
          m_resizeComponent.setBounds(x, y,
            w + (ex-(getWidth()-CORNER)), h+(ey-m_dragY));
        else if (h+(ey-m_dragY) >= MIN_HEIGHT
          && !(w + (ex-(getWidth()-CORNER)) >= MIN_WIDTH))
          m_resizeComponent.setBounds(x, y,
            MIN_WIDTH, h+(ey-m_dragY));
        else if (!(h+(ey-m_dragY) >= MIN_HEIGHT)
          && w + (ex-(getWidth()-CORNER)) >= MIN_WIDTH)
          m_resizeComponent.setBounds(x, y,
            w + (ex-(getWidth()-CORNER)), MIN_HEIGHT);
        else
          m_resizeComponent.setBounds(x,
            y, MIN_WIDTH, MIN_HEIGHT);
        break;
      case SOUTHWEST:
        if (h+(ey-m_dragY) >= MIN_HEIGHT
          && w-(ex-m_dragX) >= MIN_WIDTH)
          m_resizeComponent.setBounds(x + (ex-m_dragX), y,
            w-(ex-m_dragX), h+(ey-m_dragY));
        else if (h+(ey-m_dragY) >= MIN_HEIGHT
          && !(w-(ex-m_dragX) >= MIN_WIDTH)) {
          if (x + MIN_WIDTH < m_rightX)
            m_resizeComponent.setBounds(m_rightX-MIN_WIDTH, y,
              MIN_WIDTH, h+(ey-m_dragY));
          else
            m_resizeComponent.setBounds(x, y, w,
              h+(ey-m_dragY));
        }
        else if (!(h+(ey-m_dragY) >= MIN_HEIGHT)
          && w-(ex-m_dragX) >= MIN_WIDTH)
          m_resizeComponent.setBounds(x + (ex-m_dragX), y,
            w-(ex-m_dragX), MIN_HEIGHT);
        else
          m_resizeComponent.setBounds(m_rightX-MIN_WIDTH,
            y, MIN_WIDTH, MIN_HEIGHT);
        break;
    }
    m_rightX = x + w;
    m_resizeComponent.validate();
  }

  public void mouseEntered(MouseEvent e) {
    mouseMoved(e);
  }

  public void mouseExited(MouseEvent e) {
    if (!m_dragging)
      setCursor(Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
  }

  public void mousePressed(MouseEvent e) {
    toFront();
    m_dragging = true;
```

```
      m_dragX = e.getX();
      m_dragY = e.getY();
      if (e.getX() < CORNER) {
        m_mode = SOUTHWEST;
      }
      else if(e.getX() > getWidth()-CORNER) {
        m_mode = SOUTHEAST;
      }
      else {
        m_mode = SOUTH;
      }
    }
  }
}
```

Understanding The Code:

Class InnerFrame

New class variables:

`ImageIcon DEFAULT_FRAME_ICON`: default image used for the frame icon = 

`int BORDER_THICKNESS`: default thickness of resize edges (borders).

`int FRAME_ICON_PADDING`: default thickness of padding around the title bar icon label.

`int ICONIZED_WIDTH`: default width in the iconified state.

`Color BORDER_COLOR`: default resize border background color.

New instance variables:

`int m_titleBarHeight`: title bar height.

`int m_width`: used for recalling the frame's previous width when deiconifying.

`int m_height`: used for recalling the frame's previous height when deiconifying.

`int m_iconizedWidth`: frame width in the iconified state.

`JLabel m_iconLabel`: label used to display the frame icon in the title bar.

`boolean m_iconizeable`: determines whether the frame can be iconified.

`boolean m_resizeable`: determines whether the frame is resizable.

`boolean m_closeable`: determines whether the frame is closeable.

`JPanel m_frameContentPanel`: used to wrap the title bar and `contentPanel` for placement in `InnerFrame`'s `CENTER` region.

`ImageIcon m_frameIcon`: the frame icon displayed by `m_iconLabel` in the title bar.

`NorthResizeEdge m_northResizer`: Used for north, northeast, and northwest resizing.

`SouthResizeEdge m_southResizer`: Used for south, southeast, and southwest resizing,

`EastResizeEdge m_eastResizer`: Used for east resizing.

`WestResizeEdge m_westResizer`: Used for west reszing.

There are noew four `InnerFrame` constructors. The first creates an `InnerFrame` with no title and default frame icon. The second creates an `InnerFrame` with a title and default icon, and the third creates one with both a title and a specified frame icon. The first three constructors all end up calling the fourth to do the actual

work.

The fourth `InnerFrame` constructor calls four methods to attach our custom resize components to its edges. Then our `populateInnerFrame()` method is called which is responsible for encapsulating the title bar and content panel in `m_frameContentPanel`, which is then added to the `CENTER` of `InnerFrame`. The constructor ends by calling methods to set the title, frame icon, and resizeable, iconizeable, and closeable properties (discussed below).

The `add()` and `setLayout()` methods of `JComponent` are overridden to perform these actions on the content panel contained inside `m_frameContentPanel`. Thus, anything we add to `InnerFrame` will be placed in `m_contentPanel`, and any layout we assign to `InnerFrame` will actually be assigned to `m_contentPanel`.

Note: This functionality is not on par with fundamental Swing containers such as `JFrame`, and `JInternalFrame` which use a `JRootPane` to manage their contents. We will fix this in the next section by implementing the `RootPaneContainer` interface.

The `setIconizeable()` and `setCloseable()` methods set their respective properties and hide or show the corresponding title bar buttons using `setVisible()`. We call `revalidate()` to perform any layout changes that may be necessary after a button is shown or hidden.

The `setIconified()` method is modified store the width and height of `InnerFrame` before it is iconified. The size of an iconified `InnerFrame` is now determined by the title bar height, which is in turn determined by the frame icon size, and the thickness of the resize edges:

```
      setBounds(getX(), getY(), ICONIZED_WIDTH,
        m_titleBarHeight + 2*BORDER_THICKNESS);
```

Whenever an iconification occurs we call `setResizable(false)` to remove mouse listeners from each resize edge. If a deiconification occurs we call `setResizable(true)` to add mouse listeners back to each resize edge. Also, the width and height saved in an iconification is used to set the size of `InnerFrame` when it is deiconified.

The title bar code has several methods added to it. `setFrameIcon()` is responsible for replacing the icon in the title bar icon label, `m_iconLabel`, and calculating the new title bar height based on this change:

```
  public void setFrameIcon(ImageIcon fi) {
    m_frameIcon = fi;

    if (fi != null) {
      if (m_frameIcon.getIconHeight() > TITLE_BAR_HEIGHT)
        setTitleBarHeight(m_frameIcon.getIconHeight()
          + 2*FRAME_ICON_PADDING);
      m_iconLabel.setIcon(m_frameIcon);
    }
    else setTitleBarHeight(TITLE_BAR_HEIGHT);
    revalidate();
  }
```

The title bar height will never be smaller than `TITLE_BAR_HEIGHT` (25). If the icon's height is greater than this default value the title bar height will then be based on that icon's height, plus the default icon padding, `FRAME_ICON_PADDING`, above and below the icon. This is necessary because when the frame icon is added to the title bar within our `createTitleBar()` method, it is placed in a `JLabel` surrounded by an `EmptyBorder` as follows:

```
    m_iconLabel.setBorder(new EmptyBorder(
      FRAME_ICON_PADDING, FRAME_ICON_PADDING,
      FRAME_ICON_PADDING, FRAME_ICON_PADDING));
```

This label is then added to the `WEST` region of `m_titlePanel` (the title bar).

The custom resize components are the key to making `InnerFrame` resizable. They are built as inner classes inside `InnerFrame` and are discussed below. Before we discuss them in detail, it is helpful to clarify `InnerFrame`'s structure. Figure 15.5 illustrates this and shows which cursor will appear when the mouse pointer is placed over different portions of each resize edge (this functionality is something we have come to expect from frames in any modern desktop environment):

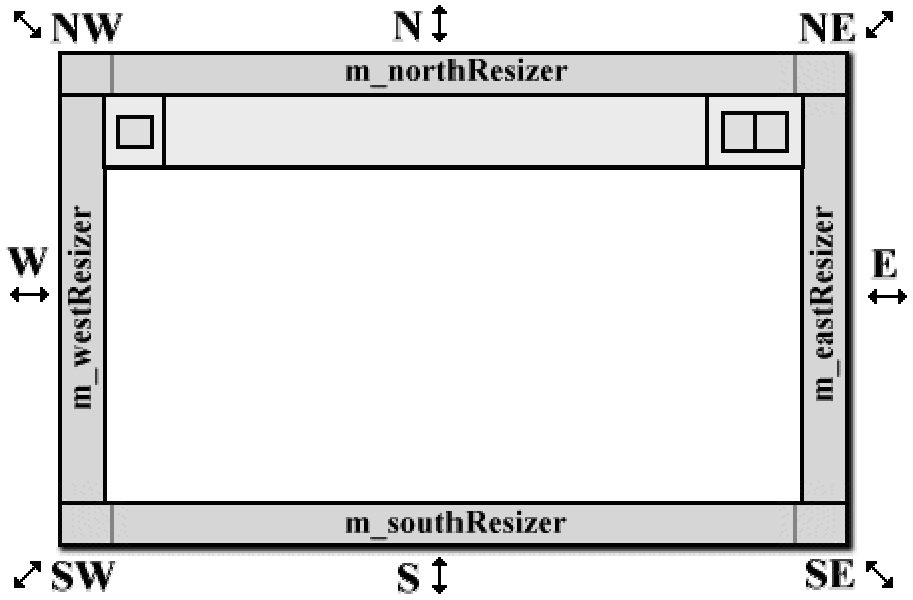


Figure 15.5. InnerFrame structure and resize edge cursor regions
<<file figure15-5.gif>>

The `Cursor` class defines several class fields representing pre-defined cursor icons that we use in the resize classes discussed below:

```
N_RESIZE_CURSOR, NE_RESIZE_CURSOR, NW_RESIZE_CURSOR, S_RESIZE_CURSOR,
SE_RESIZE_CURSOR, SW_RESIZE_CURSOR, E_RESIZE_CURSOR, W_RESIZE_CURSOR,
DEFAULT_CURSOR
```

Note: Upon first inspection you might think it would be easier to implement your own border and build resizable functionality into that. The problem is that borders are not part of the `Component` hierarchy. That is, they do not inherit from the `Component` class. In fact, every border is a subclass of `AbstractBorder` which is a direct subclass of `Object`. There is no way to associate mouse events with a border.

Class InnerFrame.EastResizeEdge

`EastResizeEdge` is the component that is placed in the `east` portion of `InnerFrame`'s `BorderLayout`. This component allows resizing `InnerFrame` horizontally. This is the simplest of the four edges because when it is used to resize, `InnerFrame` always stays in the same location (defined by its northwest corner) and the height doesn't change. Two class variables and two instance variables are necessary:

`int WIDTH`: constant thickness.

`int MIN_WIDTH`: minimum width of `m_resizeComponent`.

`boolean m_dragging`: true when dragging.

`JComponent m_resizeComponent`: the component to resize.

The constructor, as with all of the resize edge component constructors, takes a `JComponent` as a parameter. This `JComponent` is the component that is resized whenever the edge detects a mouse drag event. The `setPreferredSize()` method ensures that our edges will have a constant thickness.

Note: We have designed these components to be significantly generic, while still being encapsulated as inner classes. They can easily be modified for use within other classes or as stand-alone resizable utility components. In the next chapter we create a package called `resize` which contains each of these classes as separate entities that can be wrapped around any `JComponent`. See section 16.4. (For the `resize` package source code, refer to \Chapter16\2\resize.)

The `mouseEntered()` method is used to detect whenever the cursor is over this edge. When invoked it changes the cursor to `E_RESIZE_CURSOR`. The `mouseExited()` method changes the cursor back to the normal `DEFAULT_CURSOR` when the mouse leaves this component. In both methods cursor changes only occur when the `m_dragging` flag is `false` (i.e. when the border is not being dragged). We do not want the cursor to change while we are resizing.

The `mousePressed()` method sets the `m_dragging` flag and moves the component associated with this resize edge to the foremost position in its `JLayeredPane` layer (in other words it calls `toFront()`). The `mouseDragged` method actually handles the resizing and defines two cases to ensure that `m_resizeComponent`'s width is never made smaller than `MIN_WIDTH`.

Class InnerFrame.WestResizeEdge

`WestResizeEdge` works very similar to `EastResizeEdge` except that it must handle an extra case in resizing because the position, as well as width, of `m_resizeComponent` changes. An extra variable, `m_rightX`, is used to keep track of the coordinate of the northeast corner of `m_resizeComponent` to ensure that the right side of `m_resizeComponent` never moves during a resize.

Class InnerFrame.NorthResizeEdge, InnerFrame.SouthResizeEdge

`NorthResizeEdge` and `SouthResizeEdge` are very complicated because there are three regions in each. The left-most and right-most portions are reserved for resizing in the northwest, southwest, and northeast, southeast directions. The most complicated cases are resizing from the northwest and southwest corners because the height, width, as well as both the x and y coordinates of `m_resizeComponent` can change. Thus, the `mouseDragged` method in each of these classes is quite extensive.

Note: You are encouraged to work through this code and understand each case. We will not explain it in detail because of its repetitive and mathematical nature.

After each mouse drag, `validate()` is called on `m_resizeComponent` to lay out its contents again. If we did not include this we would see that as we resized an `InnerFrame` the size change and rendering of its contents would not occur properly. (Try this by commenting out these lines in each of the resize edge classes.) Validation forces the container to lay out its contents based on the current size. Normally we would call `revalidate()` rather than `validate()` for the sake thread safety. However, there are often performance bottlenecks associated with `revalidate()` when performing fast resizes because these requests get forwarded, queued, and coalesced by the `RepaintManager` service (see chapter 2 for more about validation and painting). By making the direct call to `validate()` we ensure that the layout will be performed

immediately, and with rapid resizing this does make a difference.

Running The Code

Figure 15.6 shows `LayeredPaneDemo` in action. First try resizing frames from all directions to see that this works as expected. Now try replacing the frame icon with one of a different size (you can use the tiger.gif image included in this example's directory). You will see that the title bar and the minimum height of our frames changes accordingly. Figure 15.7 shows our `InnerFrames` with a much larger frame icon.

15.5 Creating a custom MDI: part III - Enhancements

There are a few things about the previous example to take special note of at this stage. You may have noticed that mouse events are still propagating right through our frames. We can also still drag `InnerFrames` completely outside the layered pane view. In this section we address these issues, implement maximizable functionality, and take the final step in making `InnerFrame` a fundamental Swing container by implementing the `RootPaneContainer` interface.

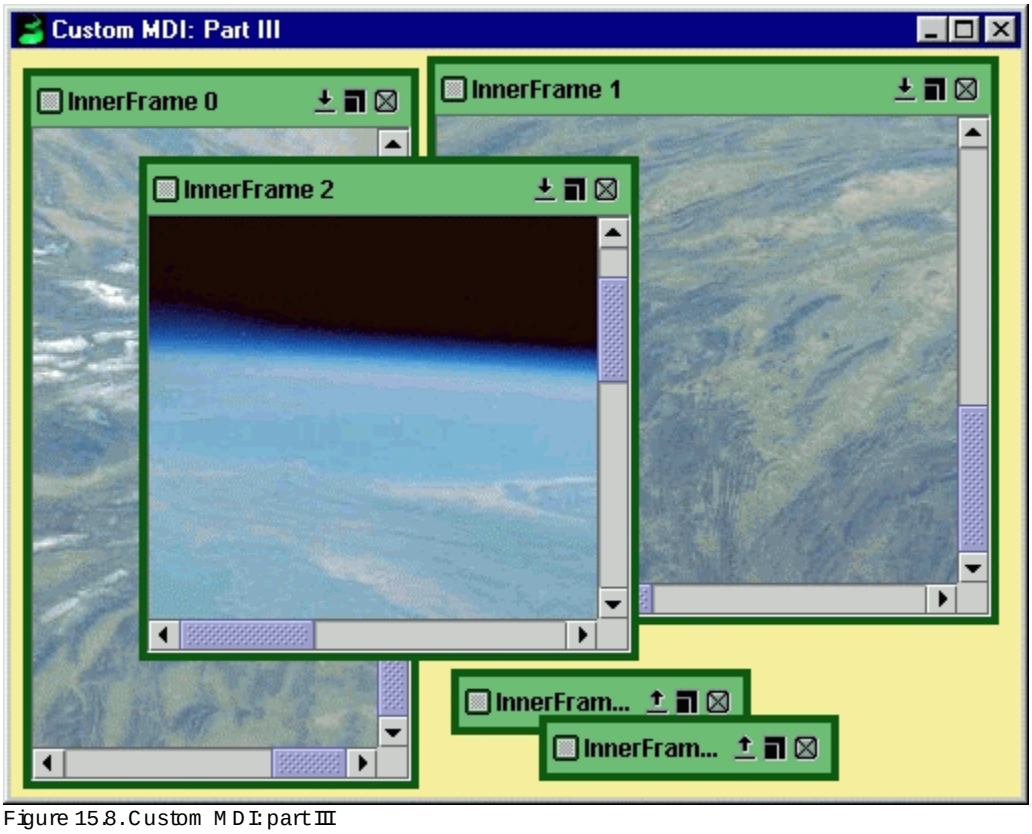


Figure 15.8. Custom MDI: part III
<<file figure15-8.gif>>

The Code: InnerFrame.java
see \Chapter15\4\mdi

```
package mdi;

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

```
public class InnerFrame
extends JPanel implements RootPaneContainer
{
  private static String IMAGE_DIR = "mdi" + java.io.File.separator;
  private static ImageIcon ICONIZE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"iconize.gif");
  private static ImageIcon RESTORE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"restore.gif");
  private static ImageIcon CLOSE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"close.gif");
  private static ImageIcon MAXIMIZE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"maximize.gif");
  private static ImageIcon MINIMIZE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"minimize.gif");
  private static ImageIcon PRESS_CLOSE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"pressclose.gif");
  private static ImageIcon PRESS_RESTORE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"pressrestore.gif");
  private static ImageIcon PRESS_ICONIZE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"pressiconize.gif");
  private static ImageIcon PRESS_MAXIMIZE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"pressmaximize.gif");
  private static ImageIcon PRESS_MINIMIZE_BUTTON_ICON =
    new ImageIcon(IMAGE_DIR+"pressminimize.gif");
  private static ImageIcon DEFAULT_FRAME_ICON =
    new ImageIcon(IMAGE_DIR+"default.gif");
  private static int BORDER_THICKNESS = 4;
  private static int WIDTH = 200;
  private static int HEIGHT = 200;
  private static int TITLE_BAR_HEIGHT = 25;
  private static int FRAME_ICON_PADDING = 2;
  private static int ICONIZED_WIDTH = 150;
  private static Color TITLE_BAR_BG_COLOR =
    new Color(108,190,116);
  private static Color BORDER_COLOR = new Color(8,90,16);

  private int m_titleBarHeight = TITLE_BAR_HEIGHT;
  private int m_width = WIDTH;
  private int m_height = HEIGHT;
  private int m_iconizedWidth = ICONIZED_WIDTH;
  private int m_x;
  private int m_y;

  private String m_title;
  private JLabel m_titleLabel;
  private JLabel m_iconLabel;

  private boolean m_iconified;
  private boolean m_maximized;

  private boolean m_iconizeable;
  private boolean m_resizeable;
  private boolean m_closeable;
  private boolean m_maximizeable;

  // only false when maximized
  private boolean m_draggable = true;

  private JRootPane m_rootPane;

  // used to wrap m_titlePanel and m_rootPane
  private JPanel m_frameContentPanel;
```

```
  private JPanel m_titlePanel;
  private JPanel m_contentPanel;
  private JPanel m_buttonPanel;
  private JPanel m_buttonWrapperPanel;

  private InnerFrameButton m_iconize;
  private InnerFrameButton m_close;
  private InnerFrameButton m_maximize;

  // Unchanged code

  public InnerFrame(String title, ImageIcon frameIcon) {
    this(title, frameIcon, true, true, true, true);
  }

  public InnerFrame(String title, ImageIcon frameIcon,
   boolean resizeable, boolean iconizeable,
   boolean maximizeable, boolean closeable) {
    super.setLayout(new BorderLayout());
    attachNorthResizeEdge();
    attachSouthResizeEdge();
    attachEastResizeEdge();
    attachWestResizeEdge();
    populateInnerFrame();

    setTitle(title);
    setResizeable(resizeable);
    setIconizeable(iconizeable);
    setCloseable(closeable);
    setMaximizeable(maximizeable);
    if (frameIcon != null)
      setFrameIcon(frameIcon);
  }

  protected void populateInnerFrame() {
    m_rootPane = new JRootPane();
    m_frameContentPanel = new JPanel();
    m_frameContentPanel.setLayout(new BorderLayout());
    createTitleBar();
    m_contentPanel = new JPanel(new BorderLayout());
    m_rootPane.setContentPane(m_contentPanel);
    m_frameContentPanel.add(m_titlePanel, BorderLayout.NORTH);
    m_frameContentPanel.add(m_rootPane, BorderLayout.CENTER);
    setupCapturePanel();
    super.add(m_frameContentPanel, BorderLayout.CENTER);
  }

  protected void setupCapturePanel() {
    CapturePanel mouseTrap = new CapturePanel();
    m_rootPane.getLayeredPane().add(mouseTrap,
      new Integer(Integer.MIN_VALUE));
    mouseTrap.setBounds(0,0,10000,10000);
  }

  // don't allow this in root pane containers
  public Component add(Component c) {
    return null;
  }

  // don't allow this in root pane containers
  public void setLayout(LayoutManager mgr) {
  }
```

```
  public JMenuBar getJMenuBar() {
    return m_rootPane.getJMenuBar();
  }

  public JRootPane getRootPane() {
    return m_rootPane;
  }

  public Container getContentPane() {
    return m_rootPane.getContentPane();
  }

  public Component getGlassPane() {
    return m_rootPane.getGlassPane();
  }

  public JLayeredPane getLayeredPane() {
    return m_rootPane.getLayeredPane();
  }

  public void setJMenuBar(JMenuBar menu) {
    m_rootPane.setJMenuBar(menu);
  }

  public void setContentPane(Container content) {
    m_rootPane.setContentPane(content);
  }

  public void setGlassPane(Component glass) {
    m_rootPane.setGlassPane(glass);
  }

  public void setLayeredPane(JLayeredPane layered) {
    m_rootPane.setLayeredPane(layered);
  }

  // Unchanged code

  public boolean isMaximizeable() {
    return m_maximizeable;
  }

  public void setMaximizeable(boolean b) {
    m_maximizeable = b;
    m_maximize.setVisible(b);
    m_titlePanel.revalidate();
  }

  public boolean isIconified() {
    return m_iconified;
  }

  public void setIconified(boolean b) {
    m_iconified = b;
    if (b) {
      if (isMaximized())
        setMaximized(false);
      toFront();
      m_width = getWidth();     // remember width
      m_height = getHeight();   // remember height
      setBounds(getX(), getY(), ICONIZED_WIDTH,
        m_titleBarHeight + 2*BORDER_THICKNESS);
      m_iconize.setIcon(RESTORE_BUTTON_ICON);
```

```
      m_iconize.setPressedIcon(PRESS_RESTORE_BUTTON_ICON);
      setResizeable(false);
    }
    else {
      toFront();
      setBounds(getX(), getY(), m_width, m_height);
      m_iconize.setIcon(ICONIZE_BUTTON_ICON);
      m_iconize.setPressedIcon(PRESS_ICONIZE_BUTTON_ICON);
      setResizeable(true);
    }
    revalidate();
  }

  public boolean isMaximized() {
    return m_maximized;
  }

  public void setMaximized(boolean b) {
    m_maximized = b;
    if (b)
    {
      if (isIconified())
        setIconified(false);
      toFront();
      m_width = getWidth();     // remember width
      m_height = getHeight();   // remember height
      m_x = getX();             // remember x
      m_y = getY();             // remember y
      setBounds(0, 0, getParent().getWidth(),
        getParent().getHeight());
      m_maximize.setIcon(MINIMIZE_BUTTON_ICON);
      m_maximize.setPressedIcon(PRESS_MINIMIZE_BUTTON_ICON);
      setResizeable(false);
      setDraggable(false);
    }
    else {
      toFront();
      setBounds(m_x, m_y, m_width, m_height);
      m_maximize.setIcon(MAXIMIZE_BUTTON_ICON);
      m_maximize.setPressedIcon(PRESS_MAXIMIZE_BUTTON_ICON);
      setResizeable(true);
      setDraggable(true);
    }
    revalidate();
  }

  // Unchanged code

  public boolean isDraggable() {
    return m_draggable;
  }

  private void setDraggable(boolean b) {
    m_draggable = b;
  }

  // create the title bar: m_titlePanel
  protected void createTitleBar() {

    // Unchanged code

    m_maximize = new InnerFrameButton(MAXIMIZE_BUTTON_ICON);
```

```
    m_maximize.setPressedIcon(PRESS_MAXIMIZE_BUTTON_ICON);
    m_maximize.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        InnerFrame.this.setMaximized(
          !InnerFrame.this.isMaximized());
      }
    });

    m_buttonWrapperPanel = new JPanel();
    m_buttonWrapperPanel.setOpaque(false);
    m_buttonPanel = new JPanel(new GridLayout(1,3));
    m_buttonPanel.setOpaque(false);
    m_buttonPanel.add(m_iconize);
    m_buttonPanel.add(m_maximize);

    // Unchanged code
  }

  // title bar mouse adapter for frame dragging
  class InnerFrameTitleBarMouseAdapter
  extends MouseInputAdapter
  {
    // Unchanged code

    // don't allow dragging outside of parent
    public void mouseDragged(MouseEvent e) {
      int ex = e.getX();
      int ey = e.getY();
      int x = m_if.getX();
      int y = m_if.getY();
      int w = m_if.getParent().getWidth();
      int h = m_if.getParent().getHeight();
      if (m_dragging & m_if.isDraggable()) {
        if((ey + y > 0 && ey + y < h) &&
          (ex + x > 0 && ex + x < w))
        {
          m_if.setLocation(ex-m_XDifference+x, ey-m_YDifference+y);
        }
        else if (!(ey + y > 0 && ey + y < h) &&
          (ex + x > 0 && ex + x < w))
        {
          if (!(ey + y > 0) && ey + y < h) {
            m_if.setLocation(ex-m_XDifference+x, 0-m_YDifference);
          else if (ey + y > 0 && !(ey + y < h))
            m_if.setLocation(ex-m_XDifference+x, h-m_YDifference);
        }
        else if ((ey + y > 0 && ey + y < h) &&
          !(ex + x > 0 && ex + x < w))
        {
          if (!(ex + x > 0) && ex + x < w)
            m_if.setLocation(0-m_XDifference, ey-m_YDifference+y);
          else if (ex + x > 0 && !(ex + x < w))
            m_if.setLocation(w-m_XDifference, ey-m_YDifference+y);
        }
        else if (!(ey + y > 0) && ey + y < h
         && !(ex + x > 0) && ex + x < w)
          m_if.setLocation(0-m_XDifference, 0-m_YDifference);
        else if (!(ey + y > 0) && ey + y < h
         && ex + x > 0 && !(ex + x < w))
          m_if.setLocation(w-m_XDifference, 0-m_YDifference);
        else if (ey + y > 0 && !(ey + y < h)
         && !(ex + x > 0) && ex + x < w)
```

```
          m_if.setLocation(0-m_XDifference, h-m_YDifference);
        else if (ey + y > 0 && !(ey + y < h)
         && ex + x > 0 && !(ex + x < w))
          m_if.setLocation(w-m_XDifference, h-m_YDifference);
      }
    }

    // Unchanged code
  }

  // Unchanged code

  ////////////////////////////////////////////
  /////////// Mouse Event Capturing //////////
  ////////////////////////////////////////////

  class CapturePanel extends JPanel
  {
    public CapturePanel() {
      MouseInputAdapter mia = new MouseInputAdapter() {};
      addMouseListener(mia);
      addMouseMotionListener(mia);
    }
  }

  // Unchanged code
}
```

Understanding The Code:

Class InnerFrame

New class variables:

`ImageIcon MAXIMIZE_BUTTON_ICON`: icon used for the maximize button =  [black interior]

`ImageIcon PRESS_MAXIMIZE_BUTTON_ICON`: icon used for the maximize button in the pressed state = [dark green interior]

`ImageIcon MINIMIZE_BUTTON_ICON`: icon used for the maximize button to represent minimization = [black interior]

`ImageIcon PRESS_MINIMIZE_BUTTON_ICON`: icon used for the maximize button in the pressed state, representing minimization =  [dark green interior]

New instance variables:

`int m_x`: used to record the location of `InnerFrame` before a maximize occurs

`int m_y`: used to record the location of `InnerFrame` before a maximize occurs

`boolean m_maximizable`: true when frame can be maximized.

`boolean m_resizable`: True when `InnerFrame` is not iconified or maximized.

`JRootPane m_rootPane`: central `InnerFrame` container--all external access is restricted to this container and its panes.

`InnerFrameButton m_maximize`: the maximize title bar button.

The `InnerFrame` constructors now support a fourth boolean parameter specifying whether the frame will be maximizable or not.

The `populateInnerFrame()` method is now responsible for creating a `JRootPane` to be used as `InnerFrame`'s central container. Since `InnerFrame` now implements the `RootPaneContainer` interface, we are required to implement access to this `JRootPane` and its `contentPane`, `layeredPane`, `glassPane` and `JMenuBar` just as a `JFrame` or `JInternalFrame`. Thus `get()` and `set()` methods have been implemented for each of these constituents.

The `setupCapturePanel()` method places an instance of our mouse-event-consuming panel, `CapturePanel` (see below), in the lowest possible layer of of our `rootPane`'s `layeredPane`:

```
protected void setupCapturePanel() {
  CapturePanel mouseTrap = new CapturePanel();
  m_rootPane.getLayeredPane().add(mouseTrap,
    new Integer(Integer.MIN_VALUE));
  mouseTrap.setBounds(0,0,10000,10000);
}
```

We set the bounds of this `CapturePanel` to be extremely large so we are, for all practical purposes, guaranteed that mouse events will not pass through the 'back' of `InnerFrame`.

The `add()` and `setLayout()` methods we had redirected to another panel in the last section, have been modified to return null and do nothing respectively. This enforces `InnerFrame` container access through its `JRootPane` constituents, similar to all other primary Swing containers.

The `setMaximizable()` method has been added to control the state of the `m_maximizeable` property and the visibility of the `m_maximize` button in the title bar.

Method `setMaximized()` has also been added for maximize and minimize functionality. When `InnerFrame` is told to maximize it first checks to see if it is iconified. If it is it deiconifies itself. Then it records its dimensions and location and resizes itself to be the size of its parent container. It swaps the maximize button icon for that representing minimize, and `setResizable(false)` is called to remove mouse listeners (we should not be able to resize a maximized frame). Finally a new method called `setDraggable()` is called and passed a `false` value. This method controls a flag that the title bar's `mouseDragged()` method checks before along `InnerFrame` to be dragged. If we set this flag to `false` `InnerFrame` will not be draggable. In the maximized state this is desirable.

```
public void setMaximized(boolean b) {
  m_maximized = b;
  if (b)
  {
    if (isIconified())
      setIconified(false);
    toFront();
    m_width = getWidth();     // remember width
    m_height = getHeight();   // remember height
    m_x = getX();             // remember x
    m_y = getY();             // remember y
    setBounds(0, 0, getParent().getWidth(),
      getParent().getHeight());
    m_maximize.setIcon(MINIMIZE_BUTTON_ICON);
    m_maximize.setPressedIcon(PRESS_MINIMIZE_BUTTON_ICON);
    setResizeable(false);
    setDraggable(false);
  }
```

When a minimize occurs, `InnerFrame` is moved to the recorded location, set to its stored width and height,

and the maximize/minimize button icons are swapped again. `setResizable(true)` and `setDraggable(true)` restore full resizeable and draggable functionality:

```
    else {
      toFront();
      setBounds(m_x, m_y, m_width, m_height);
      m_maximize.setIcon(MAXIMIZE_BUTTON_ICON);
      m_maximize.setPressedIcon(PRESS_MAXIMIZE_BUTTON_ICON);
      setResizeable(true);
      setDraggable(true);
    }
```

The `setIconified()` method has been modified to take into account the possibility that `InnerFrame` may be iconified from within the maximized state. In this case we call `setMaximized(false)` before proceeding with the iconfication.

The `m_maximize` button is created for placement in the title bar, and an `ActionListener` is attached with an `actionPerformed()` method that invokes `setMaximized()`. The title bar's button panel then allocates an additional cell (it uses `GridLayout`) for `m_maximize`, and it is added between the iconify and close buttons.

Class InnerFrame.InnerFrameTitleBarMouseAdapter

This class's `mouseDragged()` method is now much more involved. It is somewhat overwhelming at first, but all of this code is actually necessary to smoothly stop the selected `InnerFrame` from being dragged outside of the visible region of its parent. This code handles all mouse positions allowing vertical movement when horizontal is not possible, horizontal movement when vertical is not possible, and all combinations of possible dragging, while making sure that `InnerFrame` never leaves the visible region of its parent. It is not necessary that you work through the details, but it is encouraged (similar to the code for the `XXResizeEdge` classes), as it will provide an appreciation for how complicated situations such as this can be dealt with in an organized manner.

Reference: Similar code will be used in Chapter 16, section 16.5, where we build an X windows style pager that is not allowed to leave the `JDesktopPane` view.

Note: We have not implemented code to stop the user from resizing an `InnerFrame` so that its title bar lies outside of the layered pane view. This can result in a lost frame. In order to provide a solution to this we would have to add a considerable amount of code to the `NorthResizeEdge mouseDragged()` method. It can be done but we will avoid it here because other issues deserve more attention. In a commercial implementation we would want to include code to watch for this. It is interesting that this is not handled in the `JDesktopPane/JInternalFrame` MDI.

Class InnerFrame.CapturePanel

As we noticed in the past two stages of development, mouse events would pass right through our `InnerFrames`. By constructing a component to capture mouse events and placing it in our `rootPane`'s `layeredPane`, we can stop this from happening. This is the purpose of `CapturePanel`. It is a simple `JPanel` with an empty `MouseInputAdapter` shell attached as a `MouseListener` and `MouseMotionListener`. This adapter will consume any mouse events passed to it. When an `InnerFrame` is constructed, as we discussed above, a `CapturePanel` instance is added at the lowest possible layer of its `layeredPane`. Thus, mouse events that don't get handled by a component in a higher layer, such as its `contentPane`, will get trapped here.

Running The Code

Note that we've added one line to `LayeredPaneDemo` in this section that we didn't mention yet:

```
frames[i].getContentPane().add(new JScrollPane(new JLabel(ii)));
```

This places a `JScrollPane` containing a `JLabel` with an image in `InnerFrame`'s `contentPane`.

Figure 15.8 shows `LayeredPaneDemo` in action. Experiment with maximizing, iconifying and restoring. Drag frames all around the layered pane and ensure that they cannot be lost from view. Now resize a frame and notice that we can lose the title bar if it is resized above our `JFrame` title bar. (This is a flaw that should be accounted for in any commercial MDI.)

Now try maximizing an `InnerFrame` and changing the size of the `JFrame`. You will notice that the maximized `InnerFrame` does not change size along with its parent. In the next section we show how to implement this as well as other important features.

15.6 Creating a custom MDI: part IV - Selection and management

When we resize our `JFrame` it would be nice if iconified frames lined up and stayed at the bottom. This would prevent them from being lost from the layered pane view, and would also increase the organized feel of our MDI. Similarly, when an `InnerFrame` is maximized it should always fill the entire viewable region of its parent. Thus, it would be nice to have some way of controlling the layout of our `InnerFrames` when the parent is resized. We can do this by extending `JLayeredPane` and implementing the `java.awt.event.ComponentListener` interface to listen for resize `ComponentEvents`. We can capture and handle events sent to the `componentResized()` method to lay out iconified frames and resize maximized frames in a manner similar to most modern MDI environments. (Note that we could also extend `ComponentAdapter` and use an instance of the resulting class as a `ComponentListener` attached to our `JLayeredPane`. As is often the case, there is more than one way to implement the functionality we are looking for for.) In this section we'll build a `JLayeredPane` subclass that implements `ComponentListener`, and add it to our `mdi` package.

Another limiting characteristic of `InnerFrame` is that the only way to get the focus, and to move an `InnerFrame` to the front of the layered pane view, is to click on its title bar. Ideally, clicking anywhere on an `InnerFrame` should move it to the front of the layered pane view. This is where `JRootPane`'s `glassPane` comes in handy. Since our `InnerFrames` now contain a `JRootPane` as their main container we can use the `glassPane` to intercept mouse events and move the selected `InnerFrame` to the front. We will need to make our `glassPane` completely transparent, and it should only be active (i.e. receiving mouse events) within an `InnerFrame` when that `InnerFrame` is not in the foremost position of its layer (see section 15.1). Thus, only one `InnerFrame` per layer should have an inactive `glassPane`. This one frame is the selected `InnerFrame` -- the one with the current user focus.

It is customary to visually convey to the user which frame is selected. We will do this by adding a new boolean property to our `InnerFrame` representing whether or not it is selected. We will manipulate this property in such a way that there can be only one selected `InnerFrame` per layer. A selected `InnerFrame` will be characterised by unique border and title bar colors.

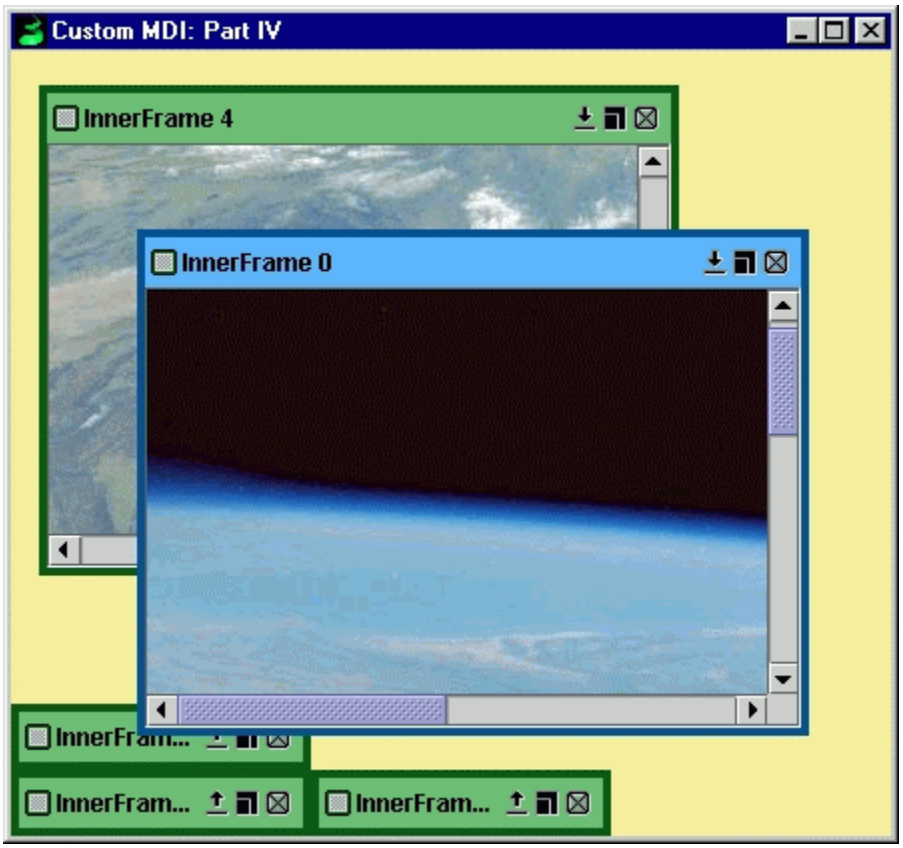


Figure 15.9. Custom MDI: part IV
<<file figure15-9.gif>>

The Code: InnerFrame.java
see \Chapter15\6\mdi

```
package mdi;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.EmptyBorder;

public class InnerFrame
extends JPanel implements RootPaneContainer
{
  // Unchanged code

  private static Color DEFAULT_TITLE_BAR_BG_COLOR =
    new Color(108,190,116);
  private static Color DEFAULT_BORDER_COLOR =
    new Color(8,90,16);
  private static Color DEFAULT_SELECTED_TITLE_BAR_BG_COLOR =
    new Color(91,182,249);
  private static Color DEFAULT_SELECTED_BORDER_COLOR =
    new Color(0,82,149);

  private Color m_titleBarBackground =
    DEFAULT_TITLE_BAR_BG_COLOR;
  private Color m_titleBarForeground = Color.black;
  private Color m_BorderColor = DEFAULT_BORDER_COLOR;
```

```
  private Color m_selectedTitleBarBackground =
    DEFAULT_SELECTED_TITLE_BAR_BG_COLOR;
  private Color m_selectedBorderColor =
    DEFAULT_SELECTED_BORDER_COLOR;

  private boolean m_selected;

  // Unchanged code

  protected void setupCapturePanel() {
    CapturePanel mouseTrap = new CapturePanel();
    m_rootPane.getLayeredPane().add(mouseTrap,
      new Integer(Integer.MIN_VALUE));
    mouseTrap.setBounds(0,0,10000,10000);
    setGlassPane(new GlassCapturePanel());
    getGlassPane().setVisible(true);
  }

  // Unchanged code

  public void toFront() {
    if (getParent() instanceof JLayeredPane)
      ((JLayeredPane) getParent()).moveToFront(this);
    if (!isSelected())
      setSelected(true);
  }

  // Unchanged code

  public boolean isSelected() {
    return m_selected;
  }

  public void setSelected(boolean b) {
    if (b)
    {
      if (m_selected != true &&
        getParent() instanceof JLayeredPane)
      {
        JLayeredPane jlp = (JLayeredPane) getParent();
        int layer = jlp.getLayer(this);
        Component[] components = jlp.getComponentsInLayer(layer);
        for (int i=0; i<components.length; i++) {
          if (components[i] instanceof InnerFrame) {
            InnerFrame tempFrame = (InnerFrame) components[i];
            if (!tempFrame.equals(this))
              tempFrame.setSelected(false);
          }
        }
        m_selected = true;
        updateBorderColors();
        updateTitleBarColors();
        getGlassPane().setVisible(false);
        repaint();
      }
    }
    else
    {
      m_selected = false;
      updateBorderColors();
      updateTitleBarColors();
      getGlassPane().setVisible(true);
```

```
    repaint();
  }
}

//////////////////////////////////////////
//////////////// Title Bar /////////////////
//////////////////////////////////////////

public void setTitleBarBackground(Color c) {
  m_titleBarBackground = c;
  updateTitleBarColors();
}

public Color getTitleBarBackground() {
  return m_titleBarBackground;
}

public void setTitleBarForeground(Color c) {
  m_titleBarForeground = c;
  m_titleLabel.setForeground(c);
  m_titlePanel.repaint();
}

public Color getTitleBarForeground() {
  return m_titleBarForeground;
}

public void setSelectedTitleBarBackground(Color c) {
  m_titleBarBackground = c;
  updateTitleBarColors();
}

public Color getSelectedTitleBarBackground() {
  return m_selectedTitleBarBackground;
}

protected void updateTitleBarColors() {
  if (isSelected())
    m_titlePanel.setBackground(m_selectedTitleBarBackground);
  else
    m_titlePanel.setBackground(m_titleBarBackground);
}

// Unchanged code

protected void createTitleBar() {
  // Unchanged code
  m_titleLabel.setForeground(m_titleBarForeground);
}

// Unchanged code

//////////////////////////////////////////////
////////////// GlassPane Selector ////////////
//////////////////////////////////////////////

class GlassCapturePanel extends JPanel
{
  public GlassCapturePanel() {
    MouseInputAdapter mia = new MouseInputAdapter() {
      public void mousePressed(MouseEvent e) {
        InnerFrame.this.toFront();
```

```
        }
      };
      addMouseListener(mia);
      addMouseMotionListener(mia);
      setOpaque(false);
    }
  }

  ////////////////////////////////////////////
  ///////////////// Resizability /////////////////
  ////////////////////////////////////////////

  public void setBorderColor(Color c) {
    m_BorderColor = c;
    updateBorderColors();
  }

  public Color getBorderColor() {
    return m_BorderColor;
  }

  public void setSelectedBorderColor(Color c) {
    m_selectedBorderColor = c;
    updateBorderColors();
  }

  public Color getSelectedBorderColor() {
    return m_selectedBorderColor;
  }

  protected void updateBorderColors() {
    if (isSelected()) {
      m_northResizer.setBackground(m_selectedBorderColor);
      m_southResizer.setBackground(m_selectedBorderColor);
      m_eastResizer.setBackground(m_selectedBorderColor);
      m_westResizer.setBackground(m_selectedBorderColor);
    } else {
      m_northResizer.setBackground(m_BorderColor);
      m_southResizer.setBackground(m_BorderColor);
      m_eastResizer.setBackground(m_BorderColor);
      m_westResizer.setBackground(m_BorderColor);
    }
  }

  // Unchanged code
}
```

The Code: MDIPane.java
see \Chapter15\6\mdi

```
package mdi;

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

public class MDIPane
extends JLayeredPane
implements ComponentListener
{
  public MDIPane() {
    addComponentListener(this);
```

```
    setOpaque(true);

    // default background color
    setBackground(new Color(244,232,152));
  }

  public void componentHidden(ComponentEvent e) {}
  public void componentMoved(ComponentEvent e) {}
  public void componentShown(ComponentEvent e) {}
  public void componentResized(ComponentEvent e) { lineup(); }
  public void lineup() {
    int frameHeight, frameWidth, currentX,
      currentY, lheight, lwidth;
    lwidth = getWidth();
    lheight = getHeight();
    currentX = 0;
    currentY = lheight;
    Component[] components = getComponents();
    for (int i=components.length-1; i>-1; i--) {
      if (components[i] instanceof InnerFrame) {
        InnerFrame tempFrame = (InnerFrame) components[i];
        frameHeight = tempFrame.getHeight();
        frameWidth = tempFrame.getWidth();
        if (tempFrame.isMaximized()) {
          tempFrame.setBounds(0,0,getWidth(),getHeight());
          tempFrame.validate();
          tempFrame.repaint();
        }
        else if (tempFrame.isIconified()) {
          if (currentX+frameWidth > lwidth) {
            currentX = 0;
            currentY -= frameHeight;
          }
          tempFrame.setLocation(currentX, currentY-frameHeight);
          currentX += frameWidth;
        }
      }
    }
  }
}
```

Understanding The Code:

Class LayeredPaneDemo

The only changes that have been made to this class is the replacement of the default `JLayeredPane` in our application frame with an instance of our custom `MDIPane` (see below).

Class InnerFrame

The following class variables have been added:

`Color DEFAULT_TITLE_BAR_BG_COLOR`: default title bar background.

`Color DEFAULT_BORDER_COLOR`: default border background.

`Color DEFAULT_SELECTED_TITLE_BAR_BG_COLOR`: default selected title bar background.

`Color DEFAULT_SELECTED_BORDER_COLOR`: default selected frame border.

New instance variables:

`Color m_titleBarBackground`: current title bar background.

`Color m_titleBarForeground`: current title bar foreground.

`Color m_BorderColor`: current border.

`Color m_selectedTitleBarBackground`: current selected title bar background.

`Color m_selectedBorderColor`: current selected border background.

`boolean m_selected`: true when frame is selected.

The `setupCapturePanel()` method now adds a call to `setInnerFrame`'s `glassPane` to an instance of our custom class `GlassCapturePanel` (see below). This allows selection via clicking on any region of an inactive `InnerFrame`.

We've inserted an additional check in the `toFront()` method to call `setSelected(true)` if that frame is not already selected:

```
    if (!isSelected())
      setSelected(true);
```

The `isSelected()` method has been added to simply return the current value of `m_selected`, and method `setSelected()` is what actually controls this property.

Method `setSelected()` takes a boolean value representing whether the frame should be selected or de-selected. If it is to be selected and it resides in a `JLayeredPane`, this method searches for all other `InnerFrame` siblings in the same layer of that `JLayeredPane` and calls `setSelected(false)` on each one it finds. Then we set the current `InnerFrame`'s selected property, `m_selected`, to true and call `updateBorderColors()` and `updateTitleBarColors()` (see below) to visually convey that this is the selected frame:

```
      m_selected = true;
      updateBorderColors();
      updateTitleBarColors();
      getGlassPane().setVisible(false);
      repaint();
```

The `glassPane` is hidden whenever a frame is selected so that mouse events will no longer be trapped (see `GlassCapturePanel` below). When a frame is de-selected (i.e. `setSelected(false)` has been called), this method disables its selected property, calls the `updateXXColors()` methods, and brings its `glassPane` out of hiding so that it may intercept mouse events for future selection:

```
      m_selected = false;
      updateBorderColors();
      updateTitleBarColors();
      getGlassPane().setVisible(true);
      repaint();
```

This whole scheme provides us with a guarantee that only one `InnerFrame` will be selected per `JLayeredPane` layer.

Methods `setTitleBarBackground()`, `setTitleBarForeground()`, and `setSelectedTitleBarBackground()` have all been added to manage the state of the current title bar color properties. Each of these methods calls `updateTitleBarColors()` so that the changes made are actually applied to the title bar and border components. In the JavaBeans spirit, we've also added `get()` methods to retreive these properties.

Similarly, methods `setBorderColor()`, `setSelectedBorderColor()`, `updateBorderColors()`, and associated `get()` methods, to manage the border color properties. The `updateBorderColors()`

method is responsible for applying these colors to each of the resize edge components.

Class mdi.InnerFrame.GlassCapturePanel

This class is almost identical to our `CapturePanel` inner class. The only difference is that its `MouseInputAdapter` overrides the `mousePressed()` method to call `toFront()` on the associated `InnerFrame`.

```
        public void mousePressed(MouseEvent e) {
          InnerFrame.this.toFront();
        }
```

As we saw above, `toFront()` calls `setSelected()` as necessary. Instances of this class are used as the `glassPane` of each `InnerFrame`'s `JRootPane`. `GlassCapturePanel` is active (visible) when its parent `InnerFrame` is not selected. It is inactive (hidden) when the associated `InnerFrame` is selected. This activation is controlled by the `setSelected()` method, as we saw above. The only function of this component is to provide a means of switching `InnerFrame` selection by clicking on any portion of an unselected `InnerFrame`.

Class mdi.MDIPane

This class extends `JLayeredPane` and implements the `java.awt.event.ComponentListener` interface. Whenever this component is resized the `componentResized()` method is invoked. This method invokes `lineup()` which grabs an array of all `Components` within the `MDIPane`. We then loop through this array, each time checking whether the `Component` at the current index is an instance of `InnerFrame`.

```
    Component[] components = getComponents();
    for (int i=components.length-1; i>-1; i--) {
      if (components[i] instanceof InnerFrame) {
```

If it is we then check if it is maximized or iconified. If it is maximized we reset its bounds to completely fill the visible region of the `MDIPane`. If it is iconified we place it at the bottom of the layered pane. This method locally maintains the position where the next iconified frame should be placed (`currentX` and `currentY`) and places these frames in rows, stacked from bottom up, that completely fit within `MDIPane`'s horizontal visible region (refer back to the code for details).

Running The Code

Figure 5.9 shows `LayeredPaneDemo` in action. Iconify the `InnerFrames` and adjust the size of the application frame to see the layout change. Now maximize an `InnerFrame` and adjust the size of the `JFrame` to see that the `InnerFrame` is resized appropriately. You may also want to experiment with the `LayeredPaneDemo` constructor and add another set or two of `InnerFrames` to different layers. You will see that there can only be one selected `InnerFrame` per layer, as expected.

This method of organizing iconified frames is certainly not adequate for professional implementations. However, developing it any further would take us a bit too far into the details of MDI construction. Ideally we might implement some sort of manager that `InnerFrames` and `MDIPane` can use to communicate with one another. (In the next chapter we will discuss a class called `DesktopManager` which functions as such a communications bridge between `JDesktopPane` and its `JInternalFrame` children. We will also learn that such a manager, as simple as it is, provides us with a great deal of flexibility.)

15.7 Creating a custom MDI: partV - JavaBeans compliance

The functionality of our `InnerFrame` is pretty much complete at this point. However, there is still much to be desired if we plan to use `InnerFrame` in the field. JavaBeans compliance is one feature that not only is

popular, but has come to be expected of each and every Java GUI component. In this section we will enhance `InnerFrame` by implementing the `Externalizable` interface, providing us with full control over its serialization. Although `JComponent` provides a default serialization mechanism for all Swing components, this is far from reliable at the time of this writing. Implementing our own serialization mechanism is not only reliable and safe for both long and short-term persistency, but it is also efficient. The default serialization mechanism tends to store much more information than we actually need.

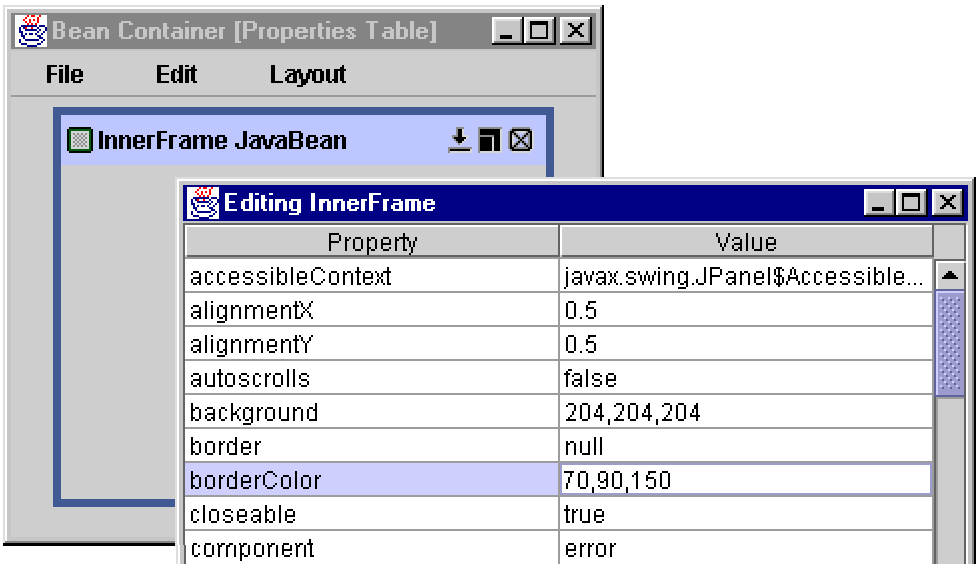


Figure 15.10. Custom MDI part V

<<file figure15-10.gif>>

The Code: InnerFrame.java
see \Chapter15\6\mdi

```
package mdi;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.EmptyBorder;

public class InnerFrame
extends JPanel implements RootPaneContainer, Externalizable
{
  // Unchanged code

  ////////////////////////////////////////////////
  /////////////// Serialization /////////////////
  ////////////////////////////////////////////////

  public void writeExternal(ObjectOutput out)
    throws IOException
  {
    out.writeObject(m_titleBarBackground);
    out.writeObject(m_titleBarForeground);
    out.writeObject(m_BorderColor);
    out.writeObject(m_selectedTitleBarBackground);
    out.writeObject(m_selectedBorderColor);
```

```
    out.writeObject(m_title);

    out.writeBoolean(m_iconizeable);
    out.writeBoolean(m_resizeable);
    out.writeBoolean(m_closeable);
    out.writeBoolean(m_maximizeable);

    out.writeObject(m_frameIcon);
    out.writeObject(getBounds());
  }

  public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException
  {
    setTitleBarBackground((Color)in.readObject());
    setTitleBarForeground((Color)in.readObject());
    setBorderColor((Color)in.readObject());
    setSelectedTitleBarBackground((Color)in.readObject());
    setSelectedBorderColor((Color)in.readObject());

    setTitle((String)in.readObject());

    setIconizeable(in.readBoolean());
    setResizeable(in.readBoolean());
    setCloseable(in.readBoolean());
    setMaximizeable(in.readBoolean());
    setSelected(false);

    setFrameIcon((ImageIcon)in.readObject());
    Rectangle r = (Rectangle)in.readObject();
    r.x = getX();
    r.y = getY();
    setBounds(r);
  }
}
```

Understanding The Code:

The added support for `InnerFrame` serialization here is quite simple. The `readExternal()` method will be invoked when `readObject()` is called on a given `ObjectInput` stream pointing to a previouly serialized InnerFrame. The `writeExternal()` method will be invoked when `writeObject()` is passed an `InnerFrame` and called on a given `ObjectOutput` stream. Refer back to chapter 4, section 4.7 to see how this is implemented in our `BeanContainer` JavaBeans environment.

Running The Code

Figure 5.10 shows an instance of `InnerFrame`, instantiated from a serialized InnerFrame saved to disk, and loaded into our JavaBeans property editing environment. We started the construction of this environment in chapter 4 and it will be completed (as shown) in chapter 18. The point to make here is that `InnerFrame` is now a JavaBean. There are certainly many ways to make `InnerFrame` a better bean. Specifically, many of the class variables would allow greater flexibility as properties, such as the default title bar height, border thickness, frame icon padding, button icons, etc. (Some of these might actually be better off within UI delegate code. Colors and button icons should change with look-and-feel, not be part of the component itself.) We could also add support for communication (which is completely lacking in `InnerFrame` now). For instance, we could make `m_maximized` into a bound or constrained property by sending out `PropertyChangeEvents` or `VetoableChangeEvents` respectively (refer back to chapter 2 for a discussion of JavaBeans and properties.). In this way we could notify interested listeners that a maximization is about to occur (in the case that `m_maximize` is constrained), and give them an opportunity to veto it.

Another major feature lacking in `InnerFrame` is look-and-feel support. The title bar and borders look like standard army-issued components at best. They should respond to look-and-feel changes just like any other Swing component. In chapter 21 we will implement support for all the major look-and-feels (Metal, Motif, Windows) for `InnerFrame`, plus our own custom look-and-feel (Malachite).

Chapter 16. Desktops and Internal Frames

In this Chapter:

- JDesktopPane and JInternalFrame
- Internalizable/Externalizable frames
- Cascading and outline dragging mode
- An X windows style desktop environment
- A networked multi-user desktop using sockets

16.1 JDesktopPane and JInternalFrame

16.1.1 JDesktopPane

class javax.swing.JDesktopPane

`JDesktopPane` is a powerful extension of `JLayeredPane` built specifically to manage `JInternalFrame` children. This is Swing's version of a multiple document interface, a feature common to most modern operating system desktops. In the last chapter we created our own MDI from scratch. Both our MDI and the `JDesktopPane`/`JInternalFrame` prebuilt MDI are quite powerful. This chapter focuses mostly on the latter, but we will relate the discussion of it to our own often.

16.1.2 JInternalFrame

class javax.swing.JInternalFrame

The purpose of `JDesktopPane` is to provide a specialized container for `JInternalFrames`. We can access its contents identically to `JLayeredPane`. There are several additional convenience methods defined in `JDesktopPane` for accessing `JInternalFrame` children (see API docs) and attaching a `DesktopManager` implementation (see below).

`JInternalFrame`s are very similar to our custom `InnerFrames` of chapter 15. They can be dragged, resized, iconified, maximized, and closed. `JInternalFrame` contains a `JRootPane` as its main container and implements the `RootPaneContainer` interface. We can access a `JInternalFrame`'s `rootPane` and its associated `glassPane`, `contentPane`, `layeredPane`, and `menuBar` the same way we access them in `JFrame` and in our `InnerFrame`.

16.1.3 JInternalFrame.JDesktopIcon

class javax.swing.JInternalFrame.JDesktopIcon

This represents a `JInternalFrame` in its iconified state. We are warned against using this class as it will disappear in future versions of Swing: "This API should NOT BE USED by Swing applications, as it will go away in future versions of Swing as its functionality is moved into `JInternalFrame`."[API] Currently when a

JInternalFrame is iconified it is removed from its JDesktopPane and a JDesktopIcon instance is added to represent it. In future versions of Swing JInternalFrame will have JDesktopIcon functionality built into it.

16.1.4 DefaultDesktopManager

class javax.swing.DefaultDesktopManager

This is the concrete default implementation of the DesktopManager interface. An instance of this class is attached to each JDesktopPane if a custom DesktopManager implementation is not specified.

16.1.5 The DesktopManager interface

abstract interface javax.swing.DesktopManager

Each JDesktopPane has a DesktopManager object attached to it whose job it is to manage all operations performed on JInternalFrames within the desktop. DesktopManager methods are automatically called from the associated JDesktopPane when an action is invoked on a JInternalFrame within that desktop. These are usually invoked when the user performs some action on a JInternalFrame with the mouse:

```
activateFrame(JInternalFrame f)
beginDraggingFrame(JComponent f)
beginResizingFrame(JComponent f, int direction)
closeFrame(JInternalFrame f)
deactivateFrame(JInternalFrame f)
deiconifyFrame(JInternalFrame f)
dragFrame(JComponent f, int newX, int newY)
endDraggingFrame(JComponent f)
endResizingFrame(JComponent f)
iconifyFrame(JInternalFrame f)
maximizeFrame(JInternalFrame f)
minimizeFrame(JInternalFrame f)
openFrame(JIntenerlFrame f)
resizeFrame(JComponent f, int newX, int newY, int newWidth, int newHeight)
setBoundsForFrame(JComponent f, int newX, int newY, int newWidth, int
  newHeight)
```

Note that if we want to manually invoke say, iconification, on a JInternalFrame we should do the following:

```
myJInternalFrame.getDesktopPane().getDesktopManager().
  iconifyFrame(myJInternalFrame);
```

We could also directly call setIcon(true) on a JInternalFrame, but we are discouraged from doing so because it is not good practice to bypass the DesktopManager. The reason this is not good practice is that there may be necessary actions defined within the DesktopManager's iconifyFrame() method that would not be invoked. So, in general, all calls to methods of JInternalFrame that have DesktopManager counterparts should be delegated to the DesktopManager.

We have written an animated demo that shows when and how often each DesktopManager method is called. See \Chapter16\4, and execute the DesktopManagerDemo class. Figure 16.1 illustrates.

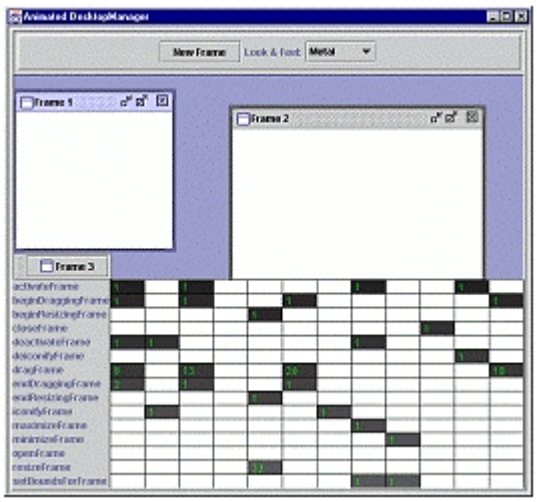


Figure 16.1 DesktopManager animated demo.
<<file figure16-1.gif>>

16.1.6 The WindowConstnats interface

abstract interface javax.swing.WindowConstants

Refer to chapter 3 for a description of this interface.

Bug Alert: using DO_NOTHING_ON_CLOSE with setDefaultCloseOperation() on a JInternalFrame does not work as expected. See bug #4176136 at the Java Developer Connection Bug Parade: http://developer.javasoft.com/developer/bugParade/bugs/4176136.html. This will most likely be fixed in the next release of Java 2.

To capture the closing of a JInternalFrame and display a confirmation dialog we can construct the following JInternalFrame subclass:

```
class ConfirmJInternalFrame extends JInternalFrame
  implements VetoableChangeListener {

   public ConfirmJInternalFrame(String title, boolean resizable,
    boolean closable, boolean maximizable, boolean iconifiable) {
     super(title, resizable, closable, maximizable, iconifiable);
     addVetoableChangeListener(this);
   }

   public void vetoableChange(PropertyChangeEvent pce)
    throws PropertyVetoException {
     if (pce.getPropertyName().equals(IS_CLOSED_PROPERTY)) {
       boolean changed = ((Boolean) pce.getNewValue()).booleanValue();
       if (changed) {
         int confirm = JOptionPane.showOptionDialog(this,
           "Close " + getTitle() + "?",
           "Close Confirmation",
           JOptionPane.YES_NO_OPTION,
           JOptionPane.QUESTION_MESSAGE,
           null, null, null);
         if (confirm == 0) {
           m_desktop.remove(this);
```

```
            m_desktop.repaint();
          }
          else throw new PropertyVetoException("Cancelled",null);
        }
      }
    }
  }
```

Using this class in place of `JInternalFrame` will always display a confirmation dialog when the close button is pressed. This code checks to see if the `closed` property has changed from its previous state. This is a constrained property which we can veto if desired (see chapter 2). Luckily this comes in real handy for working around the `DO_NOTHING_ON_CLOSE` bug.

If the confirmation dialog is displayed and then cancelled (i.e. either the "NO" button or the close dialog button is pressed) a `PropertyVetoException` is thrown which vetos the property change and the internal frame will not be closed. Figure 16.2 illustrates.

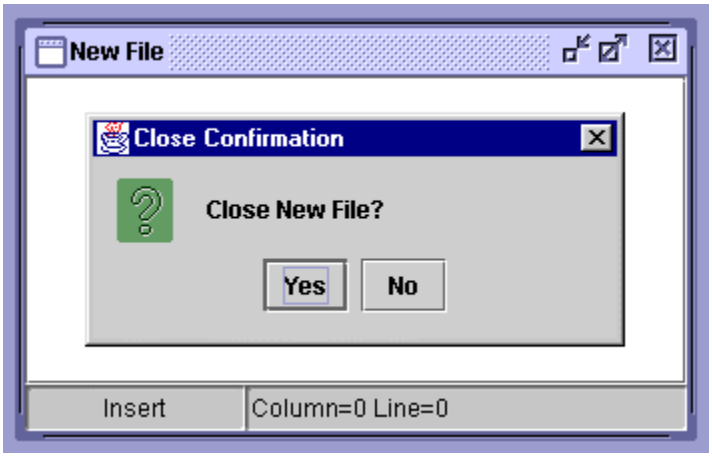


Figure 16.2 Handling internal frame closing with a close confirmation dialog.
<<file figure16-2.gif>>

16.1.7 The InternalFrameListener interface

abstract interface javax.swing.event.InternalFrameListener

Each `JInternalFrame` can have one or more `InternalFrameListener`s attached. An `InternalFrameListener` will receive `InternalFrameEvent`s allowing us to capture and handle them however we like with the following methods:

```
internalFrameActivated(InternalFrameEvente)
internalFrameClosed(InternalFrameEvent e)
internalFrameClosing(InternalFrameEvent e)
internalFrameDeactivated(InternalFrameEvent e)
internalFrameDeiconified(InternalFrameEvent e)
internalFrameIconified(InternalFrameEvent e)
internalFrameOpened(InternalFrameEvent e)
```

`InternalFrameListener` and `DesktopManager` both exist to process changes in a `JInternalFrame`'s state. However, they can both be used to achieve different ends. `DesktopManager` allows us to define internal frame handling methods for all `JInternalFrames` within a given `JDesktopPane`, whereas `InternalFrameListener`

allows us to define InternalFrameEvent handling unique to each individual JInternalFrame. We can attach a different InternalFrameListener implementation to each instance of JInternalFrame, whereas only one DesktopManager implementation can be attached to any instance of JDesktopPane (and thus, each of its children).

We have written an animated demo that shows when and how often each InternalFrameListener method is called. See \Chapter16\5, and execute the InternalFrameListenerDemo class. Figure 16.3 illustrates.

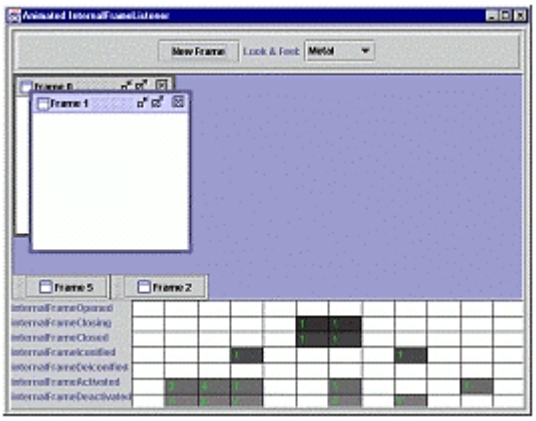


Figure 16.3 InternalFrameListener animated demo.
<<file figure16-3.gif>>

16.1.8 InternalFrameEvent

class javax.swing.event.InternalFrameEvent

InternalFrameEvents are sent to InternalFrameListeners whenever a JInternalFrame is activated, closed, about to close, deactivated, deiconified, iconified, and opened. The following static int IDs designate which type of action an InternalFrameEvent corresponds to:

```
INTERNAL_FRAME_ACTIVATED
INTERNAL_FRAME_CLOSED
INTERNAL_FRAME_CLOSING
INTERNAL_FRAME_DEACTIVATED
INTERNAL_FRAME_DEICONIFIED
INTERNAL_FRAME_ICONIFIED
INTERNAL_FRAME_OPENED
```

InternalFrameEvent extends AWTEvent, and thus encapsulates its source and the associated event ID (retrievable with getSource() and getID() respectively).

16.1.9 InternalFrameAdapter

class javax.swing.event.InternalFrameAdapter

This is a concrete implementation of the InternalFrameListener interface. It is intended to be extended for use by InternalFrameListener implementations that need to define only a subset of the InternalFrameListener methods. All methods defined within this adapter class have empty bodies.

16.1.10 Outline dragging mode

`JDesktopPane` supports an outline dragging mode to help with `JInternalFrame` dragging performance bottlenecks. To enable this mode on any `JDesktopPane` we must set the `JDesktopPane.dragMode` client property:

```
myDesktopPane.putClientProperty(
  "JDesktopPane.dragMode","outline");
```

Instead of actually moving and painting the frame whenever it is dragged, an XOR'd rectangle is drawn in its place until the drag ends. The example in the next section shows outline dragging mode in action.

16.2 Internalizable/externalizable frames

Most often in Java applets and applications we do not work in full-screen mode. Because of this `JDesktopPanes` can often become very cluttered. We may, at some point, want to have the option of bringing an internal frame outside of the desktop. We call this externalizing a frame, for lack of a given name. (Please do not confuse the use of "externalizable" here with Java's `Externalizable` interface, an extension of the `Serializable` interface.) Superficially, externalizing is the process of transforming a `JInternalFrame` into a `JFrame`.

Now consider an application in which a maximized `JFrame` is used. When this maximized frame gains the focus it hides all other existing frames and dialogs behind it. In situations where we need to switch back and forth between frames or dialogs this can be quite annoying. In order to accomodate for this problem we can think of bringing dialogs and frames inside the maximized frame to a `JDesktopPane`. We call this internalizing a frame. Superficially, internalizing is the process of transforming a `JFrame` into a `JInternalFrame`.

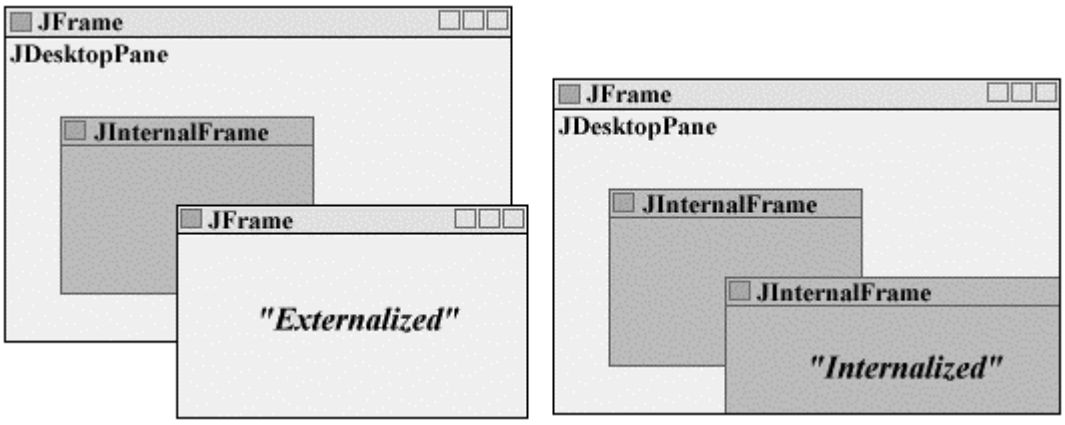


Figure 16.2 Internalizable/externalizable frames
<<file figure16-2.gif>>

Although there is no real transformation that occurs, this is what appears to happen from the user's perspective. Internalizing and externalizing is actually achieved by moving the `contentPane` from a `JFrame` to a `JInternalFrame` and vice versa, respectively. The process is simple to implement:

For externalization do the following:
1. Hide a `JInternalFrame` with `setVisible(false)`.
2. Replace the `contentPane` of a hidden `JFrame` with that of the hidden `JInternalFrame`.
3. Reveal the `JFrame` using `setVisible(true)`.
Internalization is just the opposite.

Reference: We constructed a small demo application showing how to do this in a Swing Connection 'Tips and Tricks' article. See http://java.sun.com/products/jfc/tsc/friends/tips/tips.html

16.3 Cascading and outline dragging mode

You are most likely familiar with the cascading layout that occurs as new windows are opened in MDI environments. In fact, if you have looked at any of the custom MDI examples of chapter 15 you will have seen that when you start each demo the `InnerFrames` are arranged in a cascaded fashion. This example shows how to control cascading for an arbitrary number of internal frames. Additionally, the ability to switch between any pluggable L&F available on your system is added, and outline dragging mode is enabled in our desktop.

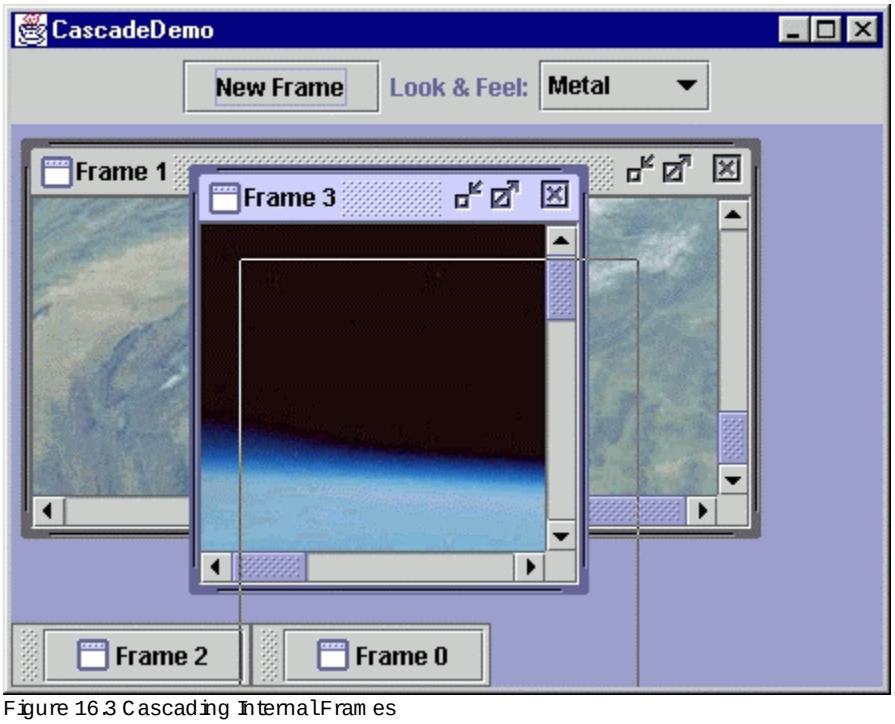


Figure 16.3 Cascading InternalFrames
<<file figure16-3.gif>>

The Code: CascadeDemo.java
see \Chapter16\1

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

public class CascadeDemo extends JFrame implements ActionListener
{
  private static ImageIcon EARTH;
  private int m_count;
  private int m_tencount;
  private JButton m_newFrame;
  private JDesktopPane m_desktop;
  private JComboBox m_UIBox;
  private UIManager.LookAndFeelInfo[] m_infos;

  public CascadeDemo() {
```

```
    super("CascadeDemo");
    EARTH = new ImageIcon("earth.jpg");
    m_count = m_tencount = 0;

    m_desktop = new JDesktopPane();
    m_desktop.putClientProperty(
      "JDesktopPane.dragMode","outline");
    m_newFrame = new JButton("New Frame");
    m_newFrame.addActionListener(this);

    m_infos = UIManager.getInstalledLookAndFeels();
    String[] LAFNames = new String[m_infos.length];
    for(int i=0; i<m_infos.length; i++) {
      LAFNames[i] = m_infos[i].getName();
    }
    m_UIBox = new JComboBox(LAFNames);
    m_UIBox.addActionListener(this);

    JPanel topPanel = new JPanel(true);
    topPanel.add(m_newFrame);
    topPanel.add(new JLabel("Look & Feel:",SwingConstants.RIGHT));
    topPanel.add(m_UIBox);

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add("North", topPanel);
    getContentPane().add("Center", m_desktop);

    setSize(570,400);
    Dimension dim = getToolkit().getScreenSize();
    setLocation(dim.width/2-getWidth()/2,
      dim.height/2-getHeight()/2);
    setVisible(true);
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(l);
  }

  public void newFrame() {
    JInternalFrame jif = new JInternalFrame("Frame " + m_count,
      true, true, true, true);
    jif.setBounds(20*(m_count%10) + m_tencount*80,
      20*(m_count%10), 200, 200);

    JLabel label = new JLabel(EARTH);
    jif.getContentPane().add(new JScrollPane(label));

    m_desktop.add(jif);
    try {
      jif.setSelected(true);
    }
    catch (PropertyVetoException pve) {
      System.out.println("Could not select " + jif.getTitle());
    }

    m_count++;
    if (m_count%10 == 0) {
      if (m_tencount < 3)
        m_tencount++;
      else
        m_tencount = 0;
```

```
    }
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == m_newFrame)
      newFrame();
    else if (e.getSource() == m_UIBox) {
      m_UIBox.hidePopup(); // BUG WORKAROUND
      try {
        UIManager.setLookAndFeel(
          m_infos[m_UIBox.getSelectedIndex()].getClassName());
        SwingUtilities.updateComponentTreeUI(this);
      }
      catch(Exception ex) {
        System.out.println("Could not load " +
          m_infos[m_UIBox.getSelectedIndex()].getClassName());
      }
    }
  }

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

Understanding the Code

Class CascadeDemo

`CascadeDemo` extends `JFrame` to provide the main container for this example. The constructor is responsible for initializing and laying out all GUI components. One class variable, `EARTH`, and several instance variables are needed:

- `ImageIcon EARTH`: image used in each `JLabel`.
- `int m_count`: keeps track of the number of internal frames that exist within the `desktop`.
- `int m_tencount`: incremented every time ten internal frames are added to the `desktop`.
- `JButton m_newFrame`: used to add new `JInternalFrames` to `m_desktop`.
- `JDesktopPane m_desktop`: container for our `JInternalFrames`.
- `JComboBox m_UIBox`: used for L&F selection.
- `UIManager.LookAndFeelInfo[] m_infos`: An array of `LookAndFeelInfo` objects used in changing L&Fs.

The only code that may look unfamiliar to you in the constructor is the following:

```
    m_infos = UIManager.getInstalledLookAndFeels();
    String[] LAFNames = new String[m_infos.length];
    for(int i=0; i<m_infos.length; i++) {
      LAFNames[i] = m_infos[i].getName();
    }
    m_UIBox = new JComboBox(LAFNames);
```

The `UIManager` class is in charge of keeping track of the current look and feel as well as providing us with a way to query information about the different look and feels available on our system. Its static `getInstalledLookAndFeels()` method returns an array of `UIManager.LookAndFeelInfo` objects and we assign this array to `m_infos`.

Each `UIManager.LookAndFeelInfo` object represents a different look-and-feel that is currently installed on our system. Its `getName()` method returns a short name representing its associated look and feel (e.g. "Metal", "CDE/Motif", "Windows", etc.). We create an array of these `Strings`, `LAFNames`, with indices corresponding to those of `m_infos`.

Finally we create a `JComboBox`, `m_UIBox`, using this array of `Strings`. In the `actionPerformed()` method (see below) when an entry in `m_UIBox` is selected we match it with its corresponding `UIManager.LookAndFeelInfo` object in `m_infos` and load the associated look-and-feel.

The `newFrame` method is invoked whenever `m_NewButton` is pressed. First this method creates a new `JInternalFrame` with resizable, closable, maximizable, and iconifiable properties, and a unique title based on the current frame count:

```
JInternalFrame jif = new JInternalFrame("Frame " + m_count,
  true, true, true, true);
```

The frame is then sized to 200 x 200 and its initial position within the our desktop is calculated based on the value of `m_count` and `m_tencount`. The value of `m_tencount` is periodically reset so that each new internal frame lies within our desktop view (assuming we do not resize our desktop to have a smaller width than the maximum of `20*(m_count%10) + m_tencount*80`, and a smaller height than the maximum of `20*(m_count%10)`. This turns out to be 420 x 180, where the maximum of `m_count%10` is 9 and the maximum of `m_tencount` is 3).

```
jif.setBounds(20*(m_count%10) + m_tencount*80,
  20*(m_count%10), 200, 200);
```

Note: You might imagine a more flexible cascading scheme that positions internal frames based on the current size of the desktop. In general a rigid cascading routine is sufficient, but we are certainly not limited to this.

A `JLabel` with an image is added to a `JScrollPane`, which is then added to the `contentPane` of each internal frame. Each frame is added to the desktop in layer `0` (the default layer when none is specified). Note that adding an internal frame to the desktop does not automatically place that frame at the frontmost position within the specified layer, and it is not automatically selected. To force both of these things to occur we use the `JInternalFrame setSelected()` method (which requires us to catch a `java.beans.PropertyVetoException`).

Finally the `newFrame()` method increments `m_count` and determines whether to increment `m_tencount` or reset it to `0`. `m_tencount` is only incremented after a group of 10 frames has been added (`m_count%10 == 0`) and is only reset after it has reached a value of 3. So 40 internal frames are created for each cycle of `m_tencount` (10 for `m_tencount` = 0, 1, 2, and 3).

```
m_count++;
if (m_count%10 == 0) {
  if (m_tencount < 3)
    m_tencount++;
  else
    m_tencount = 0;
}
```

The `actionPerformed()` method handles `m_newFrame` button presses and `m_UIBox` selections. The `m_newFrame` button invokes the `newFrame()` method and selecting a look and feel from `m_UIBox` changes the application to use that L&F. Look-and-feel switching is done by calling the `UIManager setLookAndFeel()` method and passing it the classname of the L&F to use (which we stored in the `m_infos` array in the constructor). Calling `SwingUtilities.updateComponentTreeUI(this)` changes the look

and feel of everything contained within the `CascadeDemo` frame (refer back to chapter 2).

Bug Alert: The call to `m_UIBox.hidePopup()` is added to avoid a null pointer exception bug that is caused when changing the look-and-feel of an active `JComboBox`. We expect this to be fixed in a future Java 2 release.

Running the Code:

Figure 16.2 shows `CascadeDemo` in action. This figure shows a `JInternalFrame` in the process of being dragged in outline dragging mode. Try creating plenty of frames to make sure that cascading is working properly. Experiment with different L&Fs. As a final test comment out the `m_UIBox.hidePopup()` call to check if this bug has been fixed in your version of Java.

16.4 An X-like desktop environment

Some X windows systems (specifically fvwm, Panorama (SCO), and TED (TriTeal)) provide what is referred to as a pager. This is a small window that sits in the desktop (usually at the top of the screen) and shows the positions of all windows contained in the desktop. By clicking or dragging the mouse inside the pager the user's view is moved to the associated location within the desktop. This is very helpful to X windows users because these systems often support very large desktops. Often they are larger than four times the actual size of the screen. Figure 16.4 shows a pager running on a Linux system with a desktop nine times the size of the screen.

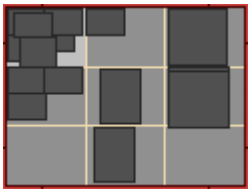


Figure 16.4 A Linux pager
<<file figure16-4.gif>>

In this example we develop our own partial implementation of a pager for use with `JDesktopPane` and its `JInternalFrame` children. We show it in use with a fairly large `JDesktopPane` (1600x1200). This desktop is scrollable and the pager will always stay in the user's current view, even when scrolling.

Note: In this example we use a custom package called `resize`. The classes contained in this package were introduced in chapter 15 as our `XXResizeEdge` components. Refer back to chapter 15 if you have any questions regarding this code. We do not explain how it works in this chapter. However, you should know that the classes contained in this package can be wrapped around any `JComponent` in a `BorderLayout` to make that component resizable. The thickness and mininum dimension properties associated with each class have been hard-coded as constants. If you plan to work with these classes we suggest adding a pair of `set()`/`get()` accessors to modify and retrieve these values.

The main purpose of presenting this example here is to show how `DesktopManager` can be customized to our needs. In the next section we will expand on this example to build a networked, multi-user desktop environment.

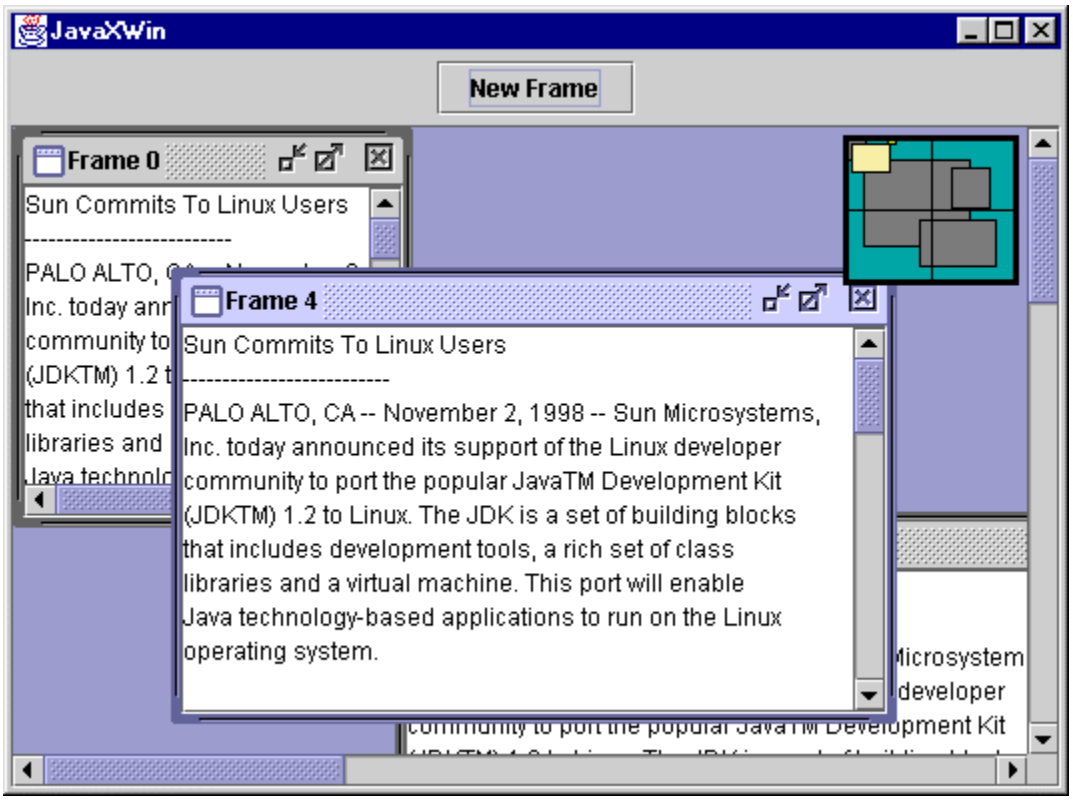


Figure 16.5 JavaXWin with WindowWatcher
<<file figure16-5.gif>>

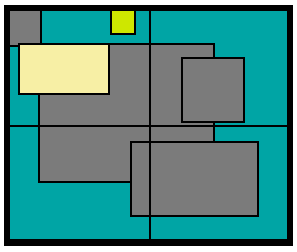


Figure 16.6 WindowWatcher with XXResizeEdges
<<file figure16-6.gif>>

The Code: JavaXWin.java
see \Chapter16\2

```
import java.beans.PropertyVetoException;
import javax.swing.*;
import java.awt.event.*;
import java.io.*;
import java.awt.*;

public class JavaXWin extends JFrame
{
  protected int m_count;
  protected int m_tencount;
  protected int m_wmX, m_wmY;
  protected JButton m_newFrame;
  protected JDesktopPane m_desktop;
  protected WindowManager m_wm;
  protected JViewport viewport;
```

```
public JavaXWin() {
  setTitle("JavaXWin");
  m_count = m_tencount = 0;
  m_desktop = new JDesktopPane();

  JScrollPane scroller = new JScrollPane();
  m_wm = new WindowManager(m_desktop);
  m_desktop.setDesktopManager(m_wm);
  m_desktop.add(m_wm.getWindowWatcher(),
    JLayeredPane.PALETTE_LAYER);
  m_wm.getWindowWatcher().setBounds(555,5,200,150);

  viewport = new JViewport() {
    public void setViewPosition(Point p) {
      super.setViewPosition(p);
      m_wm.getWindowWatcher().setLocation(
        m_wm.getWindowWatcher().getX() +
          (getViewPosition().x-m_wmX),
        m_wm.getWindowWatcher().getY() +
          (getViewPosition().y-m_wmY));
      m_wmX = getViewPosition().x;
      m_wmY = getViewPosition().y;
    }
  };
  viewport.setView(m_desktop);
  scroller.setViewport(viewport);

  ComponentAdapter ca = new ComponentAdapter() {
    JViewport view = viewport;
    public void componentResized(ComponentEvent e) {
      m_wm.getWindowWatcher().setLocation(
        view.getViewPosition().x + view.getWidth()-
          m_wm.getWindowWatcher().getWidth()+5,
        view.getViewPosition().y + 5);
    }
  };
  viewport.addComponentListener(ca);

  m_newFrame = new JButton("New Frame");
  m_newFrame.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      newFrame();
    }
  });

  JPanel topPanel = new JPanel(true);
  topPanel.setLayout(new FlowLayout());

  getContentPane().setLayout(new BorderLayout());
  getContentPane().add("North", topPanel);
  getContentPane().add("Center", scroller);

  topPanel.add(m_newFrame);

  Dimension dim = getToolkit().getScreenSize();
  setSize(800,600);
  setLocation(dim.width/2-getWidth()/2,
    dim.height/2-getHeight()/2);
  m_desktop.setPreferredSize(new Dimension(1600,1200));
  setVisible(true);
  WindowListener l = new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
```

```
        System.exit(0);
      }
    };
    addWindowListener(l);
  }

  public void newFrame() {
    JInternalFrame jif = new JInternalFrame("Frame " + m_count,
      true, true, true, true);
    jif.setBounds(20*(m_count%10) + m_tencount*80,
      20*(m_count%10), 200, 200);

    JTextArea text = new JTextArea();
    JScrollPane scroller = new JScrollPane();
    scroller.getViewport().add(text);
    try {
      FileReader fileStream = new FileReader("JavaLinux.txt");
      text.read(fileStream, "JavaLinux.txt");
    }
    catch (Exception e) {
      text.setText("* Could not read JavaLinux.txt *");
    }
    jif.getContentPane().add(scroller);

    m_desktop.add(jif);
    try {
      jif.setSelected(true);
    }
    catch (PropertyVetoException pve) {
      System.out.println("Could not select " + jif.getTitle());
    }

    m_count++;
    if (m_count%10 == 0) {
      if (m_tencount < 3)
        m_tencount++;
      else
        m_tencount = 0;
    }
  }

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

class WindowManager extends DefaultDesktopManager
{
  protected WindowWatcher ww;

  public WindowManager(JDesktopPane desktop) {
    ww = new WindowWatcher(desktop);
  }

  public WindowWatcher getWindowWatcher() { return ww; }

  public void activateFrame(JInternalFrame f) {
    super.activateFrame(f);
    ww.repaint();
  }
  public void beginDraggingFrame(JComponent f) {
    super.beginDraggingFrame(f);
    ww.repaint();
```

```
  }
  public void beginResizingFrame(JComponent f, int direction) {
    super.beginResizingFrame(f,direction);
    ww.repaint();
  }
  public void closeFrame(JInternalFrame f) {
    super.closeFrame(f);
    ww.repaint();
  }
  public void deactivateFrame(JInternalFrame f) {
    super.deactivateFrame(f);
    ww.repaint();
  }
  public void deiconifyFrame(JInternalFrame f) {
    super.deiconifyFrame(f);
    ww.repaint();
  }
  public void dragFrame(JComponent f, int newX, int newY) {
    f.setLocation(newX, newY);
    ww.repaint();
  }
  public void endDraggingFrame(JComponent f) {
    super.endDraggingFrame(f);
    ww.repaint();
  }
  public void endResizingFrame(JComponent f) {
    super.endResizingFrame(f);
    ww.repaint();
  }
  public void iconifyFrame(JInternalFrame f) {
    super.iconifyFrame(f);
    ww.repaint();
  }
  public void maximizeFrame(JInternalFrame f) {
    super.maximizeFrame(f);
    ww.repaint();
  }
  public void minimizeFrame(JInternalFrame f) {
    super.minimizeFrame(f);
    ww.repaint();
  }
  public void openFrame(JInternalFrame f) {
    super.openFrame(f);
    ww.repaint();
  }
  public void resizeFrame(JComponent f,
   int newX, int newY, int newWidth, int newHeight) {
    f.setBounds(newX, newY, newWidth, newHeight);
    ww.repaint();
  }
  public void setBoundsForFrame(JComponent f,
   int newX, int newY, int newWidth, int newHeight) {
    f.setBounds(newX, newY, newWidth, newHeight);
    ww.repaint();
  }
}
```

The Code: WindowWatcher.java
see \Chapter16\2

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

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

import resize.*;

public class WindowWatcher extends JPanel
{
  protected static final Color C_UNSELECTED =
    new Color(123, 123, 123);
  protected static final Color C_SELECTED =
    new Color(243, 232, 165);
  protected static final Color C_BACKGROUND =
    new Color(5,165,165);
  protected static final Color C_WWATCHER =
    new Color(203,226,0);
  protected float m_widthratio, m_heightratio;
  protected int m_width, m_height, m_XDifference, m_YDifference;
  protected JDesktopPane m_desktop;
  protected NorthResizeEdge m_northResizer;
  protected SouthResizeEdge m_southResizer;
  protected EastResizeEdge m_eastResizer;
  protected WestResizeEdge m_westResizer;

  public WindowWatcher(JDesktopPane desktop) {
    m_desktop = desktop;
    setOpaque(true);

    m_northResizer = new NorthResizeEdge(this);
    m_southResizer = new SouthResizeEdge(this);
    m_eastResizer = new EastResizeEdge(this);
    m_westResizer = new WestResizeEdge(this);

    setLayout(new BorderLayout());
    add(m_northResizer, BorderLayout.NORTH);
    add(m_southResizer, BorderLayout.SOUTH);
    add(m_eastResizer, BorderLayout.EAST);
    add(m_westResizer, BorderLayout.WEST);

    MouseInputAdapter ma = new MouseInputAdapter() {
      public void mousePressed(MouseEvent e) {
        m_XDifference = e.getX();
        m_YDifference = e.getY();
      }
      public void mouseDragged(MouseEvent e) {
        int vx = 0;
        int vy = 0;
        if (m_desktop.getParent() instanceof JViewport) {
          vx = ((JViewport)
            m_desktop.getParent()).getViewPosition().x;
          vy = ((JViewport)
            m_desktop.getParent()).getViewPosition().y;
        }
        int w = m_desktop.getParent().getWidth();
        int h = m_desktop.getParent().getHeight();
        int x = getX();
        int y = getY();
        int ex = e.getX();
        int ey = e.getY();
        if ((ey + y > vy && ey + y < h+vy) &&
            (ex + x > vx && ex + x < w+vx))
        {
          setLocation(ex-m_XDifference + x, ey-m_YDifference + y);
        }
```

```
      else if (!(ey + y > vy && ey + y < h+vy) &&
                (ex + x > vx && ex + x < w+vx))
      {
        if (!(ey + y > vy) && ey + y < h+vy)
          setLocation(ex-m_XDifference + x, vy-m_YDifference);
        else if (ey + y > vy && !(ey + y < h+vy))
          setLocation(ex-m_XDifference + x, (h+vy)-m_YDifference);
      }
      else if ((ey + y >vy && ey + y < h+vy) &&
              !(ex + x > vx && ex + x < w+vx))
      {
        if (!(ex + x > vx) && ex + x < w+vx)
          setLocation(vx-m_XDifference, ey-m_YDifference + y);
        else if (ex + x > vx && !(ex + x < w))
          setLocation((w+vx)-m_XDifference, ey-m_YDifference + y);
      }
      else if (!(ey + y > vy) && ey + y < h+vy &&
               !(ex + x > vx) && ex + x < w+vx)
        setLocation(vx-m_XDifference, vy-m_YDifference);
      else if (!(ey + y > vy) && ey + y < h+vy &&
               ex + x > vx && !(ex + x < w+vx))
        setLocation((w+vx)-m_XDifference, vy-m_YDifference);
      else if (ey + y > vy && !(ey + y < h+vy) &&
               !(ex + x > vx) && ex + x < w+vx)
        setLocation(vx-m_XDifference, (h+vy)-m_YDifference);
      else if (ey + y > vy && !(ey + y < h+vy) &&
               ex + x > vx && !(ex + x < w+vx))
        setLocation((w+vx)-m_XDifference, (h+vy)-m_YDifference);
    }
    public void mouseEntered(MouseEvent e) {
      setCursor(Cursor.getPredefinedCursor(
        Cursor.MOVE_CURSOR));
    }
    public void mouseExited(MouseEvent e) {
      setCursor(Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
    }
  };
  addMouseListener(ma);
  addMouseMotionListener(ma);
}

public void paintComponent(Graphics g) {
  super.paintComponent(g);
  m_height = getHeight();
  m_width = getWidth();
  g.setColor(C_BACKGROUND);
  g.fillRect(0,0,m_width,m_height);
  Component[] components = m_desktop.getComponents();
  m_widthratio = ((float)
    m_desktop.getWidth())/((float) m_width);
  m_heightratio = ((float)
    m_desktop.getHeight())/((float) m_height);
  for (int i=components.length-1; i>-1; i--) {
    if (components[i].isVisible()) {
      g.setColor(C_UNSELECTED);
      if (components[i] instanceof JInternalFrame) {
        if (((JInternalFrame) components[i]).isSelected())
          g.setColor(C_SELECTED);
      }
      else if(components[i] instanceof WindowWatcher)
        g.setColor(C_WWATCHER);
      g.fillRect(
```

```
          (int)(((float)components[i].getX())/m_widthratio),
          (int)(((float)components[i].getY())/m_heightratio),
          (int)(((float)components[i].getWidth())/m_widthratio),
          (int)(((float)components[i].getHeight())/m_heightratio));
        g.setColor(Color.black);
        g.drawRect(
          (int)(((float)components[i].getX())/m_widthratio),
          (int)(((float)components[i].getY())/m_heightratio),
          (int)(((float)components[i].getWidth())/m_widthratio),
          (int)(((float)components[i].getHeight())/m_heightratio));
      }
    }
    g.drawLine(m_width/2,0,m_width/2,m_height);
    g.drawLine(0,m_height/2,m_width,m_height/2);
  }
}
```

Understanding the Code

Class JavaXWin

`JavaXWin` extends `JFrame` and provides the main container for this example. Several instance variables are needed:

`int m_count,int m_tencount`: used for cascading

`JButton m_newFrame`: used to create new frames.

`JDesktopPane m_desktop`: our desktop pane.

`int m_wmX`: keeps track of the most recent x coordinate of the desktop scrollpane's view position.

`int m_wmY`: keeps track of the most recent y coordinate of the desktop scrollpane's view position.

`WindowManager m_wm`: our custom `DesktopManager` implementation that updates `WindowWatcher` whenever any of its methods are called.

`JViewport viewport`: The viewport of the scrollpane that will contain our desktop.

In the `JavaXWin` constructor we create a new `JDesktopPane` and place it inside a `JScrollPane`. Then we create a new `WindowManager` (see below) and pass it a reference to our desktop. We then tell our desktop that this `WindowManager` is a `DesktopManager` implementation and it should be used to manage our internal frames. This is done with `JDesktopPane`'s `setDesktopManager()` method. We then place our `WindowManager`'s `WindowWatcher` in our desktop's `PALETTE_LAYER`. This will guarantee that it is always displayed over all internal frames:

```
    m_wm = new WindowManager(m_desktop);
    m_desktop.setDesktopManager(m_wm);
    m_desktop.add(m_wm.getWindowWatcher(),
      JLayeredPane.PALETTE_LAYER);
```

A custom `JViewport` is constructed with an overriden `setViewPosition()` method. This method is responsible for keeping our `WindowWatcher` in the same place as we scroll the desktop. Each time the view is changed we reposition the `WindowWatcher` to give the impression that it lies completely above our desktop and is unaffected by scrolling. Basically, this code just computes the difference between the current and most recent viewport position, and adds this difference to the coordinates of the `WindowWatcher`. We then use this `JViewport` as the viewport for the `JScrollPane` our desktop is contained in using `JScrollPane`'s `setView()` method.

Next we construct a `ComponentAdapter` and attach it to our viewport. We override its

componentResized() method to move WindowWatcher to the top whenever the viewport is resized. This is done so that WindowWatcher will never disappear from our view when the application is resized.

The newFrame() method is almost identical to that of CascadeDemo. The only difference is that we place a JTextArea in each internal frame and load a text file into it (the original press release from Sun announcing a Linux port of JDK 1.2!)

Class WindowManager

The WindowManager class is a simple extension of DefaultDesktopManager which overrides all JInternalFrame related methods. Only one instance variable is necessary:

WindowWatcher ww: our custom pager component.

Each of the methods overriden from DefaultDesktopManager call their superclass counterparts by using super, and then call repaint on ww. So each time the user performs an action on an internal frame, WindowManager basically just tells our WidowWatcher to repaint itself.

The WindowManager constructor takes a reference to the desktop it manages, and in turn passes this reference to the WindowWatcher constructor. WindowWatcher uses this reference to find out all the information it needs to know about our desktop's contents to paint itself correctly (see below).

The getWindowWatcher() method just returns a reference to the WindowWatcher object, ww, and is used when the desktop is scrolled as discussed above.

Class WindowWatcher

This class is our version of an X windows pager. It uses the XXResizeEdge components in our custom resize package to allow full resizability. Four class variables are necessary:

Color C_UNSELECTED: used to represent all components but selected JInternalFrames and WindowWatcher itself.

Color C_SELECTED: used to represent selected JInternalFrames.

Color C_BACKGROUND: used for the WindowWatcher background.

Color C_WWATCHER: used to represent the WindowWatcher itself.

Instance variables:

float m_widthratio: Keeps the ratio of desktop width to WindowWatcher width.

float m_heightratio: Keeps the ratio of desktop height to WindowWatcher height.

int m_width: The current WindowWatcher width.

int m_height: The current WindowWatcher height.

int m_XDifference: Used for dragging the WindowWatcher horizontally.

int m_YDifference: Used for dragging the WindowWatcher vertically.

NorthResizeEdge m_northResizer: north resize component

SouthResizeEdge m_southResizer: south resize component

EastResizeEdge m_eastResizer: east resize component

WestResizeEdge m_westResizer: west resize component

JDesktopPane m_desktop: Reference to the desktop the WindowWatcher is watching over.

The constructor is passed a JDesktopPane reference which is assigned to m_desktop. We use a BorderLayout for this component and add instances of our resize package's XXResizeEdge classes to each outer region, allowing WindowWatcher to be fully resizable.

Note: See the resize package source code for details about these components. They were introduced and discussed in chapter 15. We encourage you to add more accessors to these classes to allow such things as setting thickness and color.

We then construct custom MouseInputAdapter. This adapter overrides the mousePressed(), mouseDragged(), mouseEntered(), and mouseExited() events. The mousePressed() method stores the location of the mouse press in our m_XDifference and m_YDifference class variables. These are used in the mouseDragged() method to allow WindowWatcher to be continuously dragged from any point within its bounds.

The mouseDragged() method allows the user to drag WindowManager anywhere within the visible region of the desktop. In order to enforce this and still allow smooth movement we need handle many different cases depending on mouse position and, possibly, the current JViewport position that the desktop is contained within. Note that we do not assume that WindowWatcher and its associated desktop are contained within a JViewport. However, in such a case we have to handle WindowWatcher's movement differently.

Reference: The mouseDragged code is a straight-forward adaptation of the code we used to control dragging our InnerFrames in chapter 15. See section 15.5.

The mouseEntered() method just changes the cursor to MOVE_CURSOR and mouseExited changes the cursor back to DEFAULT_CURSOR.

Finally we add this adapter with both addMouseListener() and addMouseMotionListener(). (Note that MouseInputAdapter implements both of the MouseListener and MouseMotionListener interfaces.)

The paintComponent() method starts by filling the background, getting the current dimensions, and retrieving an array of components contained in the desktop. The ratios of desktop size to WindowWatcher size are computed and then we enter a loop which is executed for each component in the array. This loop starts by setting the color to C_UNSELECTED. We then check if the component under consideration is a JInternalFrame. If it is we check if it is selected. If it is selected we set the current color to C_SELECTED. If it the component is not a JInternalFrame we check if it is the WindowWatcher itself. If so we set the current color to C_WWATCHER.

```
    for (int i=components.length-1; i>-1; i--) {
      if (components[i].isVisible()) {
        g.setColor(C_UNSELECTED);
        if (components[i] instanceof JInternalFrame) {
          if (((JInternalFrame) components[i]).isSelected())
            g.setColor(C_SELECTED);
        }
        else if(components[i] instanceof WindowWatcher)
          g.setColor(C_WWATCHER);
        g.fillRect((int) (((float)
        .
        .
        .
      }
    }
```

Once the color is selected we paint a filled, scaled rectangle representing that component. We scale this rectangle based on the ratios we computed earlier, making sure to use `floats` to avoid otherwise large rounding errors. We then paint a black outline around this rectangle and move on to the next component in our array until it has been exhausted. Note that we cycle through this array from the highest index down to 0 so that the rectangles are painted in the same order that the components appear in the `JDesktopPane` (the appearance of layering is consistent).

Running the code:

Figure 16.5 shows `JavaXWin` in action and figure 16.6 is a snapshot of the `WindowWatcher` itself. Try moving frames around and resizing them. Note that `WindowWatcher` smoothly captures and displays each component as it changes position and size. Try moving `WindowWatcher` and note that you cannot move it outside the visible region of the desktop. Now try scrolling to a different position within the desktop and note that `WindowWatcher` follows us and remains in the same position within our view. Also note that `WindowWatcher` can be resized because we've taken advantage of the classes in our custom `resize` package. In the next example we will build on top of `JavaXWin` and `WindowManager` to construct a multi-user, networked desktop environment.

`WindowWatcher` does not fully implement the functionality of most pagers. Usually clicking on an area of the pager repositions the view of our desktop. This may be an interesting and useful feature to implement in `WindowWatcher`.

16.5 A basic multi-user desktop environment using sockets

Collaborative environments are becoming more commonplace as the internet flourishes. They will no doubt continue to grow in popularity. Imagine a class taught using an interactive whiteboard or a set of whiteboards each contained in an internal frame.

In this section we show how to construct a basic multi-user `JDesktopPane` using `sockets`. We support a server and only one client. Both the server and client-side users can move, resize, and close frames, as well as chat in a console window. (We only allow the client to create frames.) All `JInternalFrame` actions invoked by one user are sent to the other user's desktop using a lightweight message-passing scheme. What we end up with is the beginnings of a true multi-user desktop environment.

Note: We've tested this environment between South Carolina and New York with satisfatory response times (using 28.8 modems dialed in to typical ISPs).

Before we present the code, it is helpful here to briefly summarize the network-centric classes that this example takes advantage of (see the API docs or the Java tutorial for more thorough coverage).

16.5.1 Socket

class java.net.Socket

A `Socket` is a connection to a single remote machine. Each `Socket` has an `InputStream` and an `OutputStream` associated with it. Data can be sent to the remote machine by writing to the `OutputStream` and data is retrieved from the remote machine via the `InputStream`. Each `Socket` also has an `InetAdress` instance associated with it which encapsulates the IP address that it is connected to.

16.5.2 SocketServer

class java.net.SocketServer

A `SocketServer` is used to establish `Socket`-to-`Socket` connections. Usually a `SocketServer` calls its `accept()` method to wait for a client to connect. Once a client connects a `ServerSocket` can return a `Socket` that the host machine uses for communication with that client.

16.5.3 InetAddress

class java.net.InetAddress

Encapsulates information about an IP address, such as the hostname and the address.

This example works by sending messages back and forth between the client and server. Each message is received and processed identically on each end. Two types of messages can be sent:

Chat messages: messages of this type can be of any length and always begin with "cc".

Internal frame messages: messages of this type are always 29 characters long, are represeted only by numeric characters, and have a distinct six field structure.

Figure 16.7 illustrates our internal frame message structure.

Figure 16.7 Internal frame message structure
<<file figure16-7.gif>>

ID represents the `WindowManager` method to invoke.

`TAG` is a unique identifier returned by each internal frame's overridden `toString()` method.

new X is the new x coordinate of the internal frame (if applicable).

new X is the new y coordinate of the internal frame (if applicable).

new width is the new width of the internal frame (if applicable).

new height is the new height of the internal frame (if applicable).

We will discuss how and when both types of messages are constructed and interpreted after we present the code. The server is implemented as class `JavaXServer`, the client as `JavaXClient`. `JavaXClient` was largely built from `JavaXServer`. We have highlighted the changes below, and inserted comments to denote where code has been modifed or unchanged.

The `WindowManager` class from the last section has been completely rebuilt and defined in a separate class file. The `WindowWatcher` class remains unchanged, using the `XXResizeEdge` classes in our resize package to allow full resizability. Both `JavaXServer` and `JavaXClient` use instances of the new `WindowManager` to manage their desktop, send messages, and interpret calls invoked by their message receiving mechanism (the `processMessage()` method).

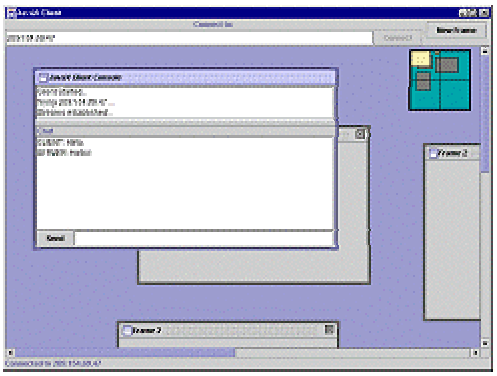


Figure 16.8 JavaXClient with established connection
<<file figure16-8.gif>>

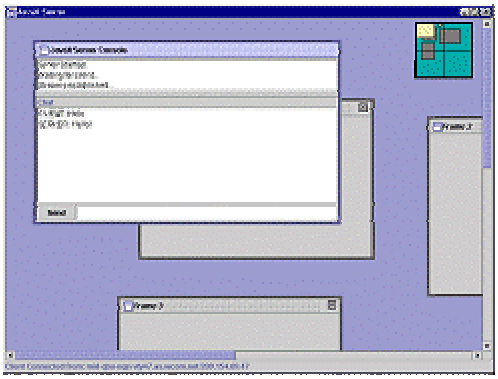


Figure 16.9 JavaXServer with established connection
<<file figure16-9.gif>>

The Code: JavaXServer.java
see \Chapter16\3

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

public class JavaXServer extends JFrame implements Runnable
{
  protected int m_count;
  protected int m_tencount;
  protected int m_wmX, m_wmY;
  protected JDesktopPane m_desktop;
  protected WindowManager m_wm;
  protected JViewport viewport;

  protected JTextArea m_consoletext, m_consolechat;
  protected JTextField m_chatText;
  protected boolean m_connected;
  protected JLabel m_status;
  protected DataInputStream m_input;
  protected DataOutputStream m_output;
  protected Socket m_client;
  protected ServerSocket m_server;
  protected Thread m_listenThread;
  protected ConThread m_conthread;
```

```
  public JavaXServer() {
    setTitle("JavaX Server");
    m_count = m_tencount = 0;
    m_desktop = new JDesktopPane();

    m_status = new JLabel("No Client");

    JScrollPane scroller = new JScrollPane();
    m_wm = new WindowManager(m_desktop);
    m_desktop.setDesktopManager(m_wm);
    m_desktop.add(m_wm.getWindowWatcher(),
      JLayeredPane.PALETTE_LAYER);
    m_wm.getWindowWatcher().setBounds(555,5,100,100);

    viewport = new JViewport() {
      public void setViewPosition(Point p) {
        super.setViewPosition(p);
        m_wm.getWindowWatcher().setLocation(
          m_wm.getWindowWatcher().getX() +
            (getViewPosition().x-m_wmX),
          m_wm.getWindowWatcher().getY() +
            (getViewPosition().y-m_wmY));
        m_wmX = getViewPosition().x;
        m_wmY = getViewPosition().y;
      }
    };
    viewport.setView(m_desktop);
    scroller.setViewport(viewport);

    ComponentAdapter ca = new ComponentAdapter() {
      JViewport view = viewport;
      public void componentResized(ComponentEvent e) {
        m_wm.getWindowWatcher().setLocation(
          view.getViewPosition().x + view.getWidth()-
            m_wm.getWindowWatcher().getWidth()-15,
          view.getViewPosition().y + 5);
      }
    };
    viewport.addComponentListener(ca);

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add("Center", scroller);
    getContentPane().add("South", m_status);

    setupConsole();

    Dimension dim = getToolkit().getScreenSize();
    setSize(800,600);
    setLocation(dim.width/2-getWidth()/2,
      dim.height/2-getHeight()/2);
    m_desktop.setPreferredSize(new Dimension(1600,1200));
    WindowListener l = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(l);

    setVisible(true);
  }

  public void setupConsole() {
```

```
  JInternalFrame console = new JInternalFrame(
   "JavaX Server Console",
   false, false, false, false) {
    int TAG = m_count;
    public String toString() {
      return "" + TAG;
    }
  };
  m_count++;
  console.setBounds(20, 20, 500, 300);

  JPanel chatPanel = new JPanel();
  JLabel chatLabel = new JLabel(" Chat");
  chatPanel.setLayout(new BorderLayout());

  m_consoletext = new JTextArea();
  m_consoletext.setPreferredSize(new Dimension(500,50));
  m_consoletext.setLineWrap(true);
  m_consoletext.setText("Server Started." +
    "\nWaiting for client...");
  m_consoletext.setEditable(false);

  m_consolechat = new JTextArea();
  m_consolechat.setLineWrap(true);
  m_consolechat.setEditable(false);

  m_chatText = new JTextField();
  m_chatText.addActionListener(new ChatAdapter());

  JButton chatSend = new JButton("Send");
  chatSend.addActionListener(new ChatAdapter());

  JPanel sendPanel = new JPanel();

  sendPanel.setLayout(new BorderLayout());
  sendPanel.add("Center", m_chatText);
  sendPanel.add("West", chatSend);

  JScrollPane cscroller1 = new JScrollPane(m_consoletext);
  JScrollPane cscroller2 = new JScrollPane(m_consolechat);

  chatPanel.add("North", chatLabel);
  chatPanel.add("Center", cscroller2);
  chatPanel.add("South", sendPanel);

  JSplitPane splitter = new JSplitPane(
    JSplitPane.VERTICAL_SPLIT, true, cscroller1, chatPanel);

  console.getContentPane().add(splitter);
  m_desktop.add(console);

  m_wm.getWindowWatcher().repaint();

  try {
    m_server = new ServerSocket(5000,500);
  }
  catch (IOException e) {
    m_consoletext.append("\n" + e);
  }
  m_conthread = new ConThread();
}

public void run() {
```

```
    while (m_connected) {
      try {
        processMessage(m_input.readUTF());
      }
      catch (IOException e) {
        m_consoletext.append("\n" + e);
        m_connected = false;
      }
    }
  }

  public void newFrame() {
    JInternalFrame jif = new JInternalFrame("Frame " + m_count,
     true, true, false, false) {
      int TAG = m_count;
      public String toString() {
        return "" + TAG;
      }
    };
    jif.setBounds(20*(m_count%10) + m_tencount*80,
      20*(m_count%10), 200, 200);

    m_desktop.add(jif);
    try {
      jif.setSelected(true);
    }
    catch (PropertyVetoException pve) {
      System.out.println("Could not select " + jif.getTitle());
    }

    m_count++;
    if (m_count%10 == 0) {
      if (m_tencount < 3)
        m_tencount++;
      else
        m_tencount = 0;
    }
  }

  public void processMessage(String s) {
    if (s.startsWith("cc")) {
      m_consolechat.append("CLIENT: " + s.substring(2) + "\n");
      m_consolechat.setCaretPosition(
        m_consolechat.getText().length());
    }
    else {
      int id = (Integer.valueOf(s.substring(0,2))).intValue();
      m_wm.setPropagate(false);
      if (id == 16) {
        newFrame();
      }
      else {
        Component[] components = m_desktop.getComponentsInLayer(0);
        int index = 0;
        int tag = (Integer.valueOf(s.substring(2,5))).intValue();
        int param1 =
          (Integer.valueOf(s.substring(5,11))).intValue();
        int param2 =
          (Integer.valueOf(s.substring(11,17))).intValue();
        int param3 =
          (Integer.valueOf(s.substring(17,23))).intValue();
        int param4 =
          (Integer.valueOf(s.substring(23))).intValue();
```

```
boolean found = false;
for (int i=components.length-1; i>-1;i--) {
  if (components[i] instanceof JInternalFrame) {
    if (Integer.valueOf(
     components[i].toString()).intValue() == tag) {
      try {
        ((JInternalFrame) components[i]).setSelected(true);
        ((JInternalFrame) components[i]).toFront();
        index = i;
        found = true;
        break;
      }
      catch (PropertyVetoException pve) {
        System.out.println(
         "Could not select JInternalFrame with tag " + tag);
      }
    }
  }
}
if (found == false) return;
switch (id)
{
  case 1:
    m_wm.activateFrame((JInternalFrame) components[index]);
    break;
  case 2:
    m_wm.beginDraggingFrame((JComponent) components[index]);
    break;
  case 3:
    m_wm.beginResizingFrame(
     (JComponent) components[index], param1);
    break;
  case 4:
    m_wm.closeFrame((JInternalFrame) components[index]);
    break;
  // case 5: not implemented
  // case 6: not implemented
  case 7:
    m_wm.dragFrame(
      (JComponent)components[index], param1, param2);
    break;
  case 8:
    m_wm.endDraggingFrame((JComponent) components[index]);
    break;
  case 9:
    m_wm.endResizingFrame((JComponent) components[index]);
    break;
  // case 10: not implemented
  // case 11: not implemented
  // case 12: not implemented
  case 13:
    m_wm.openFrame((JInternalFrame) components[index]);
    break;
  case 14:
    m_wm.resizeFrame(
      (JComponent) components[index], param1,
        param2, param3, param4);
    break;
  case 15:
    m_wm.setBoundsForFrame(
      (JComponent) components[index], param1,
        param2, param3, param4);
    break;
```

```
        }
      }
      m_wm.setPropagate(true);
    }
    m_desktop.repaint();
  }

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

  class ChatAdapter implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      m_wm.sendMessage("cc" + m_chatText.getText());
      m_consolechat.append("SERVER: " +
        m_chatText.getText() + "\n");
      m_chatText.setText("");
    }
  }

  class ConThread extends Thread
  {
    ConThread() { start(); }
    public void run() {
      while(true) {
        try {
          m_client = m_server.accept();
          m_connected = true;
          m_input = new DataInputStream(m_client.getInputStream());
          m_output = new DataOutputStream(
            m_client.getOutputStream());
          m_wm.setOutputStream(m_output);
          m_consoletext.append("\nStreams established...");
          m_listenThread = new Thread(JavaXServer.this);
          m_listenThread.start();
          m_status.setText("Client Connected from: "
            + m_client.getInetAddress());
          m_consoletext.append("\nClient Connected from: "
            + m_client.getInetAddress());
        }
        catch (Exception ex) {
          m_consoletext.append("\n" + ex);
        }
      }
    }
  }
}
```

The Code: WindowManager.java
see \Chapter16\3

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

public class WindowManager extends DefaultDesktopManager
{
  //ID 16 means new Frame
  protected static final int ACIVATE_ID = 1;
  protected static final int BEGINDRAG_ID = 2;
  protected static final int BEGINRESIZE_ID = 3;
  protected static final int CLOSE_ID = 4;
```

```
protected static final int DEACTIVATE_ID = 5;
protected static final int DEICONIFY_ID = 6;
protected static final int DRAG_ID = 7;
protected static final int ENDDRAG_ID = 8;
protected static final int ENDRESIZE_ID = 9;
protected static final int ICONIFY_ID = 10;
protected static final int MAXIMIZE_ID = 11;
protected static final int MINIMIZE_ID = 12;
protected static final int OPEN_ID = 13;
protected static final int RESIZE_ID = 14;
protected static final int SETBOUNDS_ID = 15;
protected WindowWatcher ww;
protected DataOutputStream m_output;
protected JDesktopPane m_desktop;
protected boolean m_prop;

public WindowManager(JDesktopPane desktop) {
  m_desktop = desktop;
  m_prop = true;
  ww = new WindowWatcher(desktop);
}

public WindowWatcher getWindowWatcher() { return ww; }

public void setOutputStream(DataOutputStream output) {
  m_output = output;
}

public void sendMessage(String s) {
  try {
    if (m_output != null)
      m_output.writeUTF(s);
  }
  catch (IOException e) {}
}

public void setPropagate(boolean b) {
  m_prop = b;
}

public String getStringIndex(Component f) {
  String s = f.toString();
  while (s.length() < 3)
    s = ("0").concat(s);
  return s;
}

public String getString(int number) {
  String s;
  if(number < 0)
    s = "" + (-number);
  else
    s = "" + number;
  while (s.length() < 6)
    s = ("0").concat(s);
  if (number < 0)
    s = "-" + s.substring(1,6);
  return s;
}

public void activateFrame(JInternalFrame f) {
  String index = getStringIndex(f);
  super.activateFrame(f);
```

```
    ww.repaint();
    if (m_prop)
      sendMessage("01" + index + "000000000000000000000000");
  }
  public void beginDraggingFrame(JComponent f) {
    String index = getStringIndex(f);
    super.beginDraggingFrame(f);
    ww.repaint();
    if (m_prop)
      sendMessage("02" + index + "000000000000000000000000");
  }
  public void beginResizingFrame(JComponent f, int direction) {
    String index = getStringIndex(f);
    String dir = getString(direction);
    super.beginResizingFrame(f,direction);
    ww.repaint();
    if (m_prop)
      sendMessage("03" + index + dir + "0000000000000000");
  }
  public void closeFrame(JInternalFrame f) {
    String index = getStringIndex(f);
    super.closeFrame(f);
    ww.repaint();
    if (m_prop)
      sendMessage("04" + index + "000000000000000000000000");
  }
  public void deactivateFrame(JInternalFrame f) {
    super.deactivateFrame(f);
    ww.repaint();
    // ID 05 - not implemented
  }
  public void deiconifyFrame(JInternalFrame f) {
    super.deiconifyFrame(f);
    ww.repaint();
    // ID 06 - not implemented
  }
  public void dragFrame(JComponent f, int newX, int newY) {
    String index = getStringIndex(f);
    String x = getString(newX);
    String y = getString(newY);
    f.setLocation(newX, newY);
    ww.repaint();
    if (m_prop)
      sendMessage("07" + index + x + y +"000000000000");
  }
  public void endDraggingFrame(JComponent f) {
    String index = getStringIndex(f);
    super.endDraggingFrame(f);
    ww.repaint();
    if (m_prop)
      sendMessage("08" + index + "000000000000000000000000");
  }
  public void endResizingFrame(JComponent f) {
    String index = getStringIndex(f);
    super.endResizingFrame(f);
    ww.repaint();
    if (m_prop)
      sendMessage("09" + index + "000000000000000000000000");
  }
  public void iconifyFrame(JInternalFrame f) {
    super.iconifyFrame(f);
    ww.repaint();
    // ID 10 - not implemented
```

```
  }
  public void maximizeFrame(JInternalFrame f) {
    String index = getStringIndex(f);
    super.maximizeFrame(f);
    ww.repaint();
    // ID 11 - not implemented
  }
  public void minimizeFrame(JInternalFrame f) {
    super.minimizeFrame(f);
    ww.repaint();
    // ID 12 - not implemented
  }
  public void openFrame(JInternalFrame f) {
    String index = getStringIndex(f);
    super.openFrame(f);
    ww.repaint();
    if (m_prop)
      sendMessage("13" + index + "000000000000000000000000");
  }
  public void resizeFrame(JComponent f,
   int newX, int newY, int newWidth, int newHeight) {
    String index = getStringIndex(f);
    String x = getString(newX);
    String y = getString(newY);
    String w = getString(newWidth);
    String h = getString(newHeight);
    f.setBounds(newX, newY, newWidth, newHeight);
    ww.repaint();
    if (m_prop)
      sendMessage("14" + index + x + y + w + h);
  }
  public void setBoundsForFrame(JComponent f,
   int newX, int newY, int newWidth, int newHeight) {
    String index = getStringIndex(f);
    String x = getString(newX);
    String y = getString(newY);
    String w = getString(newWidth);
    String h = getString(newHeight);
    if (newWidth > m_desktop.getWidth())
      newWidth = m_desktop.getWidth();
    if (newHeight > m_desktop.getHeight())
      newHeight = m_desktop.getHeight();
    f.setBounds(newX, newY, newWidth, newHeight);
    ww.repaint();
    if (m_prop)
      sendMessage("15" + index + x + y + w + h);
  }
}
```

Code: JavaXClient.java
see \Chapter16\3

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

public class JavaXClient extends JFrame implements Runnable
{
  protected int m_count;
```

```
  protected int m_tencount;
  protected int m_wmX, m_wmY;
  protected JButton m_newFrame;
  protected JDesktopPane m_desktop;
  protected WindowManager m_wm;
  protected JViewport viewport;

  protected JTextArea m_consoletext, m_consolechat;
  protected JTextField m_text, m_chatText;
  protected boolean m_connected;
  protected JLabel m_status;
  protected DataInputStream m_input;
  protected DataOutputStream m_output;
  protected Socket m_client;
  protected Thread m_listenThread;

  // ServerSocket and ConThread code removed.

  protected JButton m_connect;

  public JavaXClient() {
    setTitle("JavaX Client");
    m_count = m_tencount = 0;
    m_desktop = new JDesktopPane();

    m_status = new JLabel("Not Connected");

    JScrollPane scroller = new JScrollPane();
    m_wm = new WindowManager(m_desktop);
    m_desktop.setDesktopManager(m_wm);
    m_desktop.add(m_wm.getWindowWatcher(),
      JLayeredPane.PALETTE_LAYER);
    m_wm.getWindowWatcher().setBounds(555,5,100,100);

    viewport = new JViewport() {
      //...identical in JavaXServer
    };
    viewport.setView(m_desktop);
    scroller.setViewport(viewport);

    ComponentAdapter ca = new ComponentAdapter() {
      //...identical in JavaXServer
    };
    viewport.addComponentListener(ca);

    m_newFrame = new JButton("New Frame");
    m_newFrame.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_wm.setPropagate(false);
        newFrame();
        if (m_connected)
          m_wm.sendMessage("160000000000000000000000000000");
        m_wm.setPropagate(true);
      }
    });
    m_newFrame.setEnabled(false);

    JPanel topPanel = new JPanel(true);
    topPanel.add(m_newFrame);

    m_connect = new JButton("Connect");
    m_connect.addActionListener(new ActionListener() {
```

```
      public void actionPerformed(ActionEvent e) {
        if (m_listenThread == null) {
          Thread connector = new Thread() {
            public void run() {
              try
              {
                m_consoletext.append(
                  "\nTrying " + m_text.getText() + " ...");
                m_client = new Socket(
                  InetAddress.getByName(m_text.getText()),5000);
                m_input = new DataInputStream(
                  m_client.getInputStream());
                m_output = new DataOutputStream(
                  m_client.getOutputStream());
                m_connected = true;
                m_listenThread = new Thread(JavaXClient.this);
                m_listenThread.start();
                m_wm.setOutputStream(m_output);
                m_consoletext.append("\nStreams established...");
                m_status.setText("Connected to " + m_text.getText());
                m_connect.setEnabled(false);
                m_newFrame.setEnabled(true);
              }
              catch (Exception ex) {
                m_consoletext.append("\n" + ex);
                m_newFrame.setEnabled(false);
              }
            }
          };
          connector.start();
        }
      }
    });
```

```
    JPanel XPanel = new JPanel();
    XPanel.setLayout(new BorderLayout());
    JLabel hl = new JLabel("Connect to: ", SwingConstants.CENTER);
    m_text = new JTextField(15);
    XPanel.add("North", hl);
    XPanel.add("Center", m_text);
    XPanel.add("East", m_connect);

    JPanel upperPanel = new JPanel();
    upperPanel.setLayout(new BorderLayout());
    upperPanel.add("Center", XPanel);
    upperPanel.add("East",topPanel);

    getContentPane().setLayout(new BorderLayout());
    getContentPane().add("North", upperPanel);
    getContentPane().add("Center", scroller);

    // Unchanged code
  }

  public void setupConsole() {
    JInternalFrame console = new JInternalFrame(
     "JavaX Client Console",
     false, false, false, false) {
      int TAG = m_count;
      public String toString() {
        return "" + TAG;
      }
    };
```

```
    m_count++;
    console.setBounds(20, 20, 500, 300);

    JPanel chatPanel = new JPanel();
    JLabel chatLabel = new JLabel(" Chat");
    chatPanel.setLayout(new BorderLayout());

    m_consoletext = new JTextArea();
    m_consoletext.setPreferredSize(new Dimension(500,50));
    m_consoletext.setLineWrap(true);
    m_consoletext.setText("Client Started...");
    m_consoletext.setEditable(false);

    // The remainder of this method is identical
    // to JavaXServer's setupConsole() method.
    // However, we have removed the ServerSocket
    // code from the end.
  }

  public void run() {
    // ...identical to JavaXServer's run() method.
  }

  public void newFrame() {
    // ...identical to JavaXServer's newFrame() method.
  }

  public void processMessage(String s) {
    if (s.startsWith("cc")) {
      m_consolechat.append("SERVER: " + s.substring(2) + "\n");
      m_consolechat.setCaretPosition(
        m_consolechat.getText().length());
    }
    else {

    // With the exception of the highlighted code
    // above, this method is identical to JavaXServer's
    // processMessage() method.
  }

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

  class ChatAdapter implements ActionListener {
    public void actionPerformed(ActionEvent e) {
      m_wm.sendMessage("cc" + m_chatText.getText());
      m_consolechat.append("CLIENT: " + m_chatText.getText() + "\n");
      m_chatText.setText("");
    }
  }

  // ConThread inner class removed.
}
```

Understanding the Code

Class JavaXServer

`JavaXServer` implements the `Runnable` interface allowing us to define a separate thread of execution. Several instance variables are necessary:

`int m_count,int m_tencount`: used for cascading

`int m_wmX`: keeps track of the most recent x coordinate of the desktop scrollpane's view position.

`int m_wmY`: keeps track of the most recent y coordinate of the desktop scrollpane's view position.

`JDesktopPane m_desktop`: our desktop pane.

`WindowManager m_wm`: our custom `DesktopManager` implementation.

`JViewport viewport`: The viewport of the scrollpane that will contain our desktop.

`JTextArea m_consoletext`: The console textarea used to display server status information.

`JTextArea m_consolechat`: The console textarea used for chatting between server and client.

`boolean m_connected`: Flag specifying whether a client is connected or not.

`JLabel m_status`: Status bar used to display the IP address of the connected client.

`DataInputStream m_input`: The `DataInputStream` of the client connection `Socket`.

`DataOutputStream m_output`: The `DataOutputStream` of the client connection `Socket`.

`Socket m_client`: The `Socket` created when a client connects.

`ServerSocket m_server`: Used to wait for an incoming client and establish the client `Socket`.

`Thread m_listenThread`: A handle used for creating and starting the `JavaXServer` thread.

`ConThread m_conthred`: An instance of our custom `Thread` extended inner class used to allow the `ServerSocket` to wait for client connections without hogging our application's thread.

The `JavaXServer` constructor performs familiar GUI layout tasks, and is very similar to the `JavaXWin` constructor we studied in the last section. Before the frame is made visible our `setupConsole()` method is called. This method is responsible for contructing our chat console internal frame (it also acts as a message log for the server). We override this `JInternalFrame`'s `toString()` method to return a unique `TAG` which is the value of the `m_count` variable at the time of its creation. Since the console is the first frame created it will have a `TAG` of `0`. We then increment `m_count` so that the next frame, created in the `newFrame()` method (see below), will have a different `TAG` value. This is how we identify frames when sending internal frame messages.

The console contains two textareas, a "Send" button and a textfield. The send button and textfield are used for chatting, the upper textarea is used to display server status information, and the lower textfield is used to display chat text. We attach an instance of our custom `ChatAdapater` class (see below) to both the send button, `chatSend`, and the textfield, `m_chatText`.

The `setupConsole()` method ends by actually starting the server. We create a new `ServerSocket` on port `5000` with queue length `500`. A queue length of `500` represents the maximum amount of messages that can be buffered at any given time by the `ServerSocket`.

Note: This example uses a fixed port. In professional applications the server would most likely provide the operator with a field to enter the desired port number. Also, note that a maximum queue length of `500` is not really necessary here, as we are only expecting one client to connect through this `ServerSocket`. However, it does not add any extra overhead, and if for some reason this port gets bombarded with extraneous messages this will give our client a better chance of getting in.

The `run()` method defines the separate thread of execution that is created and called within the `ConThread` class (see below). When a client is connected this method continuously reads data from the client `Socket`,

m_client, and sends it to our custom processMessage() method (see below).

Our newFrame() method looks familiar, however, each JInternalFrame created gets assigned a unique TAG, which is the value of the m_count variable at the time of its creation, and its toString() method is overriden to return this tag. This is how we identify destinations of internal frame messages.

The processMessage() method takes a String parameter representing a message from the client. First we check if it is a chat message (remember that all chat messages start with "cc"). If so we simply append the this message (minus the "cc" header) to our chat text area and set the cursor position accordingly.

If it is not a chat message then we process it as an internal frame message. First we get the method ID and check if it is 16. (See discussion of the WindowManager class below for an explanation of what method each ID corresponds to):

1. If the id is 16 all we need to do is create a new frame, so the newFrame() method is invoked. (Note that only the client can send 'create new frame' messages.)

2. If the ID is not 16 we first call our WindowManager's custom setPropagate() message to stop processed messages sent to our WindowManager from being sent back to the client (this is effectively how we consume each message). Then we grab an array of all the components in layer 0 and check each JInternalFrame's TAG until we find a match. (Note that all frames are contained at layer 0 in this example. In a more complete example we would search through all components in the desktop, regardless of layer.) We then extract the internal frame message parameters and if a TAG match is found we send a message to our WindowManager, m_wm, based on the id and extracted parameters (if applicable).

Class JavaXServer.ChatAdapter

ChatAdapter is used as an ActionListener attached to our console's chat text field and send button (see the setupConsole() method). Whenever the user presses enter in the chat input text field or clicks the console's "Send" button, this adapter sends the text field contents as a message to the client, appends it to the console chat text area, m_consolechat, and clears this text field.

Class JavaXServer.ConThread

Inner class ConThread extends Thread. Its constructor calls its start() method, and its start() method calls its run() method. We override the run() method to create an endless loop that starts by invoking accept() on our m_server ServerSocket. This method blocks execution until a client connects. When a client does connect our ServerSocket returns from the accept() method with a Socket, m_client, representing the connection with the client. We then set our m_connected flag to true so that when our main JavaXServer thread is started, it can receive and process messages from the client, as discussed above. We assign m_client's DataInputStream to m_input and its DataOutputStream to m_output. Then we pass m_output to our WindowManager's setOutputStream() method (see discussion of WindowManager below). We print a message in the console to inform the server operator that a connection has been established and then start() the JavaXServer thread (see the run() method--discussed above). Finally we display the IP address the client connected from in the status bar and append this information to our console.

Class WindowManager

The WindowManager class starts by defining 15 self-explanitory int id fields, each corresponding to one of the internal frame methods defined within this class. Four instance variables are used:

WindowWatcher ww: our custom pager component.

DataOutputStrean m_output: The clientSocket's outputstream used to send messages.

`JDesktopPane m_desktop`: The desktop we are managing.

`boolean m_prop`: Used to block messages from being sent back to the sender when being processed.

The `setOutputStream()` method is called when a client connects. This is used to provide `WindowManager` with a reference to the client `Socket`'s `DataOutputStream` for sending messages.

The `sendMessage()` method takes a `String` parameter representing a message to be sent to the client and attempts to write it to the cleint's `DataOutputStream`:

```
public void sendMessage(String s) {
  try {
    if (m_output != null) {
      m_output.writeUTF(s);
    }
  }
  catch (IOException e) {}
}
```

The `setPropagate()` method takes a `boolean` parameter which is used to block messages from being sent back to the client when being processed. This method is called from `JavaXServer`'s `processMessage()` method.

The `getStringIndex()` method takes an `int` parameter, representing the `TAG` of an internal frame, and converts it to a `String` 3 characters long by concatenating `0`s in front if necessary. This is used to build the `TAG` field in an internal frame message (which, as we know from our discussion in the beginning of this section, is always 3 characters long) which is returned.

```
public String getStringIndex(Component f) {
  String s = f.toString();
  while (s.length() < 3)
    s = ("0").concat(s);
  return s;
}
```

In the `getString()` method we take an `int` parameter which can represent any of four possible parameters passed to one of the internal frame methods defined within this class. We then convert this value to a `String`. If this value was negative we remove the "-" sign. Then we concatenate a number of `0`s to the front of the string forcing the length to be 6. We then check if the value passed in was negative one more time. If it was we replace the first character with a "-" sign:

```
public String getString(int number) {
  String s;
  if (number < 0)
    s = "" + (-number);
  else
    s = "" + number;
  while (s.length() < 6)
    s = ("0").concat(s);
  if (number < 0)
    s = "-" + s.substring(1,6);
  return s;
}
```

Note: This example assumes that no frame coordinate or dimension will ever be larger than 999999 or smaller than -99999. For all practical purposes, this is a completely safe assumption to make!

Whenever any of the internal frame methods are invoked we first get the TAG of the frame associated with the method call. Then we call the superclass counterpart of that method, repaint our WindowWatcher, and check our message propagation flag, m_prop. If it is set to true we go ahead and construct our message, using the getString() method where applicable, and pass it to our sendMessage method to send to the client. If it is false, no message is sent.

Class JavaXClient

JavaXClient functions very similar to JavaXServer. It sends, receives, and interprets messages identically to JavaXServer. However, unlike JavaXServer, JavaXClient can create new frames and send new frame messages to the server (new frame messages have ID 16) These messages are constructed and sent within JavaXClient's actionPerformed method:

```
    if (e.getSource() == m_newFrame) {
      m_wm.setPropagate(false);
      newFrame();
      if (m_connected)
        m_wm.sendMessage("16000000000000000000000000000000");
      m_wm.setPropagate(true);
    }
```

JavaXClient also has some additional GUI components added to the top of its frame. A text field for entering the server's IP address, a "New Frame" button, and a connect button. The ActionListener for this button is wrapped in a thread to allow connection attempts while not blocking the main thread of execution. It attempts to connect to the address specified in the text field by creating a new Socket:

```
                m_client = new Socket(
                  InetAddress.getByName(m_text.getText()),5000);
```

If this works it establishes input and output data streams and starts the JavaXClient thread (see its run() method--identical to JavaXServer's run() method) to listen for messages. We then append text to the console, update the status bar, and enable the "New Frame" button. (Note that the client can only create new frames after a connection has been established.)

Running the Code:

Figure 16.8 and 16.9 show JavaXClient and JavaXServer during a collaborative session. Ideally you test this example out with a friend in a remote, far-away place. If this is not possible try using two machines in your network. (If you are not networked you can run both client and the server on your machine by connecting to 127.0.0.1 which is always used as a pointer to your own machine.)

Try chatting and resizing each other's frames. Now is the time to think of other possible applications of such a multi-user desktop environment. Clearly we will begin to see more and more remote interaction and collaboration as the web and its surrounding technologies continue to grow.

Chapter 17. Trees

In this chapter:

- JTree

- Basic JTree example
- Directories tree: part I - Dynamic node retrieval
- Directories tree: part II - Popup menus and programmatic navigation
- Directories tree: part III - Tooltips
- JTree and XML documents
- Custom editors and renderers

17.1 JTree

`JTree` is a perfect tool for the display, navigation, and editing of hierarchical data. Because of its complex nature, `JTree` has a whole package devoted to it: `javax.swing.tree`. This package consists of a set of classes and interfaces which we will briefly review before moving on to several examples. But first, what is a tree?

17.1.1 Tree concepts and terminology

The tree is a very important and heavily used data structure throughout computer science (e.g. compiler design, graphics, artificial intelligence, etc.). This data structure consists of a logically arranged set of nodes, which are containers for data. Each tree contains one root node, which serves as that tree's top-most node. Any node can have an arbitrarty number of child (descendant) nodes. In this way, each descendant node is the root of a subtree.

Each node is connected by an edge. An edge signifies the relationship between two nodes. A node's direct predecessor is called its parent, and all predecesors (above and including the parent) are called its ancestors. A node that has no descendants is called a leaf node. All direct child nodes of a given node are siblings.

A path from one node to another is a sequence of nodes with edges from one node to the next. The level of a node is the number of nodes visited in the path between the root and that node. The height of a tree is its largest level--the length of its longest path.

17.1.2 Tree traversal

It is essential that we be able to systematically visit each and every node of a tree. (The term 'visit' here refers to performing some task before moving on.) There are three common traversal orders used for performing such an operation: preoder, inorder, and postorder. Each is recursive and can be summarized as follows:

Preorder
  Recursively do the following:
    If the tree is not empty, visit the root and then traverse all subtrees in ascending order.

Inorder (often referred to as breadth first):
  Start the traversal by visiting the main tree root. Then, in ascending order, visit the root of each subtree. Continue visiting the roots of all subtrees in this manner, in effect visiting the nodes at each level of the tree in ascending order.

Postorder (often referred to as depth first):
  Recursively do the following:
    If the tree is not empty, traverse all subtrees in ascending order, and then visit the root.

17.1.3 JTree

class javax.swing.JTree

So how does Swing's `JTree` component deal with all this structure? Implementations of the `TreeModel` interface encapsulate all tree nodes, which are implementations of the `TreeNode` interface. The `DefaultMutabeTreeNode` class (an implementation of `TreeNode`) provides us with the ability to perform preorder, inorder, and postorder tree traversals.

Note: There is nothing stopping us from using `TreeModel` as a data structure class without actually displaying it in a GUI. However, since this book, and the Swing library, is devoted to GUI, we will not discuss these possibilities further.)

`JTree` graphically displays each node similar to how `JList` displays its elements: in a vertical column of cells. Similarly, each cell can be rendered with a custom renderer (an implementation of `TreeCellRenderer`) and can be edited with a custom `TreeCellEditor`. Each tree cell shows a non-leaf node as being expanded or collapsed, and can represent node relationships (i.e. edges) in various ways. Expanded nodes show their subtree nodes, and collapsed nodes hide this information.

The selection of tree cells is similar to `JList`'s selection mechanism, and is controlled by a `TreeSelectionModel`. Selection also involves keeping track of paths between nodes as instances of `TreeNode`. Two kinds of events are used specifically with trees and tree selections: `TreeModelEvent` and `TreeExpansionEvent`. Other AWT and Swing events also apply to `JTree`. For instance, we can use `MouseListeners` to intercept mouse presses and clicks. Also note that `JTree` implements the `Scrollable` interface (see chapter 7) and is intended to be placed in a `JScrollPane`.

A `JTree` can be constructed using either the default constructor, by providing a `TreeNode` to use for the root node, providing a `TreeModel` containing all constituent nodes, or by providing a one-dimensional array, `Vector`, or `Hashtable` of objects. In the latter case, if any element in the given structure is a multielement structure itself, it is recursively used to build a subtree (this functionality is handled by an inner class called `DynamicUtilTreeNode`).

We will see how to construct and work with all aspects of a `JTree` soon enough. But first we need to develop a more solid understanding of its underlying constituents and how they interact.

17.1.4 The TreeModel interface

abstract interface javax.swing.tree.TreeModel

This model handles the data to be used in a `JTree`, assuming that each node maintains an array of child nodes. Nodes are represented as `Objects`, and a separate root node accessor is defined. A set of methods is intended to retrieve a node based on a given parent node and index, return the number of children of a given node, return the index of a given node based on a given parent, check if a given node is a leaf node (has no children), and a method to notify `JTree` that a node which is the destination of a given `TreePath` has been modified. It also provides method declarations for adding and removing `TreeModelListeners` which should be notified when any nodes are added, removed, or changed. A `JTree`'s `TreeModel` can be retrieved and assigned with its `getModel()` and `setModel()` methods respectively.

17.1.5 DefaultTreeModel

class javax.swing.tree.DefaultTreeModel

`DefaultTreeModel` is the default concrete implementation of the `TreeModel` interface. It defines the root and each node of the tree as `TreeNode` instances. It maintains an `EventListenerList` of `TreeModelListeners` and provides several methods for firing `TreeModelEvents` when anything in the tree changes. It defines the `asksAllowedChildren` flag which is used to confirm whether or not a node allows children to be added before actually attempting to add them. `DefaultTreeModel` also defines methods for returning an array of nodes from a given node to the root node, inserting and removing nodes, and a method to reload/refresh a tree from a specified node. We normally build off of this class when implementing a `JTree` component.

17.1.6 The TreeNode interface

abstract interface javax.swing.tree.TreeNode

`TreeNode` describes the base interface which all tree nodes must conform to in a `DefaultTreeModel`. Implementations of this class represent the basic building block of `JTree`'s model. It declares properties for specifying whether a node is a leaf, a parent, allows addition of child nodes, determining the number of children, obtaining a `TreeNode` child at a given index or the parent node, and obtaining an `Enumeration` of all child nodes.

17.1.7 The MutableTreeNode interface

abstract interface javax.swing.tree.MutableTreeNode

Thic interface extends `TreeNode` to describe a more sophisticated tree node which can carry a user object. This is the object that represents the data of a given tree node. The `setUserObject()` method declares how the user object should be assigned (it is assumed that implementations of this interface will provide the equivalent of a `getUserObject()` method, even though none is included here). This interface also provides method declarations for inserting and removing nodes from a given node, and changing its parent node.

17.1.8 DefaultMutableTreeNode

class javax.swing.tree.DefaultMutableTreeNode

`DefaultMutableTreeNode` is a concrete implementation of the `MutableTreeNode` interface. Method `getUserObject()` returns a the data object encapsulated by this node. It stores all child nodes in a `Vector` called children, accessible with the `children()` method which returns an `Enumeration` of all child nodes. We can also use the `getChildAt()` method to retreive the node corresponding to the given index. There are many methods for, among other things, retrieving and assigning tree nodes, and they are all self-explanitory (or can be understood through simple reference of the API docs). The only methods that deserve special mention here are the overriden `toString()` method, which returns the `String` given by the user object's `toString()` method, and the tree traversal methods which return an `Enumeration` of nodes in the order they were visited. As discussed above, there are three types of traversal supported: preorder, inorder, and postorder. The corresponding methods are `preorderEnumeration()`, `breadthFirstEnumeration()`, `depthFirstEnumeration()` and `postorderEnumeration()` (note that the last two methods do the same thing).

17.1.9 TreePath

class javax.swing.tree.TreePath

A `TreePath` represents the path to a node as a set of nodes starting from the root. (Recall that nodes are `Objects`, not necessarily `TreeNodes`.) `TreePaths` are read-only objects and provide functionality for comparison between other `TreePaths`. The `getLastPathComponent()` gives us the final node in the path, `equals()` compares two paths, `getPathCount()` gives the number of nodes in a path, `isDescendant()` checks whether a given path is a descendant of (i.e. is completely contained in) a given path, and `pathByAddingChild()` returns a new `TreePath` instance resulting from adding the given node to the path. See the example of section 17.2 for more about working with `TreePaths`.

17.1.10 The TreeCellRenderer interface

abstract interface javax.swing.tree.TreeCellRenderer

This interface describes the component used to render a cell of the tree. The `getTreeCellRendererComponent()` method is called to return the component to render corresponding to a given cell and that cell's selection, focus, and tree state (i.e. whether it is a leaf or a parent, and whether it is expanded or collapsed). This works similar to custom cell rendering in `JList` and `JComboBox` (see chapters 9 and 10). To assign a renderer to `JTree` we use its `setCellRenderer()` method. Recall that renderer components are not at all interactive and simply act as "rubber stamps" for display purposes only.

17.1.11 DefaultTreeCellRenderer

class javax.swing.tree.DefaultTreeCellRenderer

`DefaultTreeCellRenderer` is the default concrete implementation of the `TreeCellRenderer` interface. It extends `JLabel` and maintains several properties used to render a tree cell based on its current state, as described above. These properties include `Icons` used to represent the node in any of its possible states (leaf, parent collapsed, parent expanded) and background and foreground colors to use based on whether the node is selected or unselected. Each of these properties is self-explanitory and typical get/set accessors are provided.

17.1.12 CellRenderPane

class javax.swing.CellRenderPane

In chapter 2 we discussed the painting and validation process in detail, but we purposely avoided the discussion of how renderers actually work behind the scenes because they are only used be a few specific components. The component returned by a renderer's `getXXRendererComponent()` method is placed in an instance of `CellRenderPane`. The `CellRenderPane` is used to act as the component's parent so that any validation and repaint requests that occur do not propogate up the ancestry tree of the container it resides in. It does this by overriding the `paint()` and `invalidate()` with empty implementations.

Several `paintComponent()` methods are provided to render a given component onto a given graphical context. These are used by the `JList`, `JTree`, and `JTable` UI delegates to actually paint each cell, which results in the "rubber stamp" behavior we have referred to.

17.1.13 The CellEditor interface

abstract javax.swing.CellEditor

Unlike renderers, cell editors for `JTree` and `JTable` are defined from a generic interface. This interface is

CellEditor and it declares methods for controlling when editing will start and stop, retrieving a new value resulting from an edit, and whether or not an edit request changes the component's current selection.

- `Object getCellEditorValue()`: used by JTree and JTable after an accepted edit to retrieve the new value.
- `boolean isCellEditable(EventObject anEvent)`: used to test whether the given event should trigger a cell edit. For instance, to accept a single mouse click as an edit invocation we would override this method to test for an instance of MouseEvent and check its click count. If the click count is 1 return true, otherwise return false.
- `boolean shouldSelectCell(EventObject anEvent)`: used to specify whether the given event causes a cell that is about to be edited to also be selected. This will cancel all previous selection, and for components that want to allow editing during an ongoing selection we would return false here. It is most common to return true, as we normally think of the cell being edited as the currently selected cell.
- `boolean stopCellEditing()`: used to stop a current cell edit. This method can be overriden to perform input validation. If a value is found to be unacceptable we can return false indicating to the component that editing should not be stopped.
- `void cancelCellEditing()`: used to stop a current cell edit and ignore any new input.

This interface also declares methods for adding and removing CellEditorListeners which should recieve ChangeEvents whenever an edit is stopped or canceled. So stopCellEditing() and cancelCellEditing() are responsible for firing ChangeEvents to any registered listeners.

Normally cell editing starts with the user clicking on a cell a specified number of times which can be defined in the isCellEditable() method. The component containing the cell then replaces the current renderer pane with its editor component (JTree's editor component is that returned by TreeCellEditor's getTreeCellEditorComponent() method). If shouldSelectCell() returns true then the component's selection state changes to only contain the cell being edited. A new value is entered using the editor and an appropriate action takes place which invokes either stopCellEditing() or cancelCellEditing(). Finally, if the edit was stopped and not canceled, the component retrieves the new value from the editor, using getCellEditorValue(), and overwrites the old value. The editor is then replaced by the renderer pane which is updated to reflect the new data value.

17.1.14 The TreeCellEditor interface

abstract interface javax.swing.tree.TreeCellEditor

This interface extends CellEditor and describes the behavior of a component to be used in editing the cells of a tree. Method getTreeCellEditorComponent() is called prior to the editing of a new cell to set the initial data for the component it returns as the editor, based on a given cell and that cell's selection, focus, and its expanded/collapsed states. We can use any component we like as an editor. To assign a TreeCellEditor to JTree we use its setCellEditor() method.

17.1.15 DefaultCellEditor

class javax.swing.DefaultCellEditor

This is a concrete implementation of the TreeCellEditor interface as well as the TableCellEditor interface (see 18.1.11). This editor allows the use of JTextField, JComboBox, or JCheckBox components to edit data. It defines a protected inner class called EditorDelegate which is responsible for returning the current value of the editor component in use when the getCellEditorValue() method is invoked. DefaultCellEditor is limited to three constructors for creating a JTextField, JComboBox, or

a JCheckBox editor.

Note: The fact that the only constructors provided are component-specific makes DefaultCellEditor a bad candidate for extensibility.

DefaultCellEditor maintains an int property called clickCountToStart which specifies how many mouse click events should trigger an edit. By default this is 2 for JTextFields and 1 for JComboBox and JCheckBox editors. As expected ChangeEvents are fired when stopCellEditing() and cancelCellEditing() are invoked.

17.1.16 DefaultTreeCellEditor

class javax.swing.tree.DefaultTreeCellEditor

DefaultTreeCellEditor extends DefaultCellEditor, and is the default concrete implementation of the TreeCellEditor interface. It uses a JTextField for editing a node's data (an instance of DefaultTreeCellEditor.DefaultTextField). stopCellEditing() is called when ENTER is pressed in this textfield.

An instance of DefaultTreeCellRenderer is needed to construct this editor, allowing renderer icons to remain visible while editing (accomplished by embedding the editor in an instance of DefaultTreeCellEditor.EditorContainer), and fires ChangeEvents when editing begins and ends. As expected, we can add CellEditorListeners to intercept and process these events.

By default, editing starts (if it is enabled) when a cell is triple-clicked or a pause of 1200ms occurs between two single mouse clicks (the latter is accomplished using an internal Timer). We can set the click count requirement using the setClickCountToStart() method, or check for it directly by overriding isCellEditable().

17.1.17 The RowMapper interface

abstract interface javax.swing.text.RowMapper

RowMapper describes a single method, getRowsForPaths(), which maps an array of tree paths to array of tree rows. A tree row corrsponds to a tree cell, and as we discussed, these are organized similar to JList cells. JTree selections are based on rows and tree paths, and we can choose which to deal with depending on the needs of our application. We aren't expected to implement this interface unless we decide to build our own JTree UIdelegate.

17.1.18 The TreeSelectionModel interface

abstract interface javax.swing.tree.TreeSelectionModel

The TreeSelectionModel interface describes a base interface for a tree's selection model. Three modes of selection are supported, similar to JList (see chapter 10), and implementations allow setting this mode through the setSelectionMode() method: SINGLE_TREE_SELECTION, DISCONTIGUOUS_TREE_SELECTION, and CONTIGUOUS_TREE_SELECTION. Implementations are expected to maintain a RowMapper instance. The getSelectionPath() and getSelectionPaths() methods are intended to return a TreePath and an array of TreePaths respectively, allowing access to the currently selected paths. The getSelectionRows() method should return an int array representing the indices of all rows currently selected. The lead selection refers to the most recently added path to the current selection. Whenever the selection changes, implementations of this interface should fire

TreeSelectionEvents. Appropriately, add/remove TreeSelectionListener methods are also declared. All other methods are, for the most part, self explanitory (see API docs). The tree selection model can be retrieved using JTree's getSelectionModel() method.

Note: JTree defines the inner class EmptySelectionModel which does not allow any selection at all.

17.1.19 DefaultTreeSelectionModel

class javax.swing.tree.DefaultTreeSelectionModel

DefaultTreeSelectionModel is the default concrete implementation of the TreeSelectionModel interface. This model supports TreeSelectionListener notification when changes are made to a tree's path selection. Several methods are defined for, among other things, modifying a selection, testing if it can be modified, and firing TreeSelectionEvents when a modification occurs.

17.1.20 The TreeModelListener interface

abstract interface javax.swing.event.TreeModelListener

The TreeModelListener interface describes a listener which receives notifications about changes in a tree's model. TreeModelEvents are normally fired from a TreeModel when nodes are modified, added, or removed. We can register/unregsiter a TreeModelListener with a JTree's model using TreeModel's addTreeModelListener() and removeTreeModelListener() methods respectively.

17.1.21 The TreeSelectionListener interface

abstract interface javax.swing.event.TreeSelectionListener

The TreeSelectionListener interface describes a listener which receives notifications about changes in a tree's selection. It declares only one method, valueChanged(), accepting a TreeSelectionEvent. These events are normally fired whenever a tree's selection changes. We can register/unregister a TreeSelectionListener with a tree's selection model using JTree's addTreeSelectionListener() and removeTreeSelectionListener() methods respectively.

17.1.22 The TreeExpansionListener interface

abstract interface javax.swing.event.TreeExpansionListener

The TreeExpansionListener interface describes a listener which receives notifications about tree expansions and collapses. Implementations must define treeExpanded() and treeCollapsed() events, which take a TreeExpansionEvent as parameter. We can register/unregister a TreeExpansionListener with a tree using JTree's addTreeExpansionListener() and removeTreeExpansionListener() methods respectively.

17.1.23 The TreeWillExpandListener interface

abstract interface javax.swing.event.TreeWillExpandListener

The TreeWillExpandListener interface describes a listener which receives notifications when a tree is about to expand or collapse. Unlike TreeExpansionListener this listener will be notified before the actual change occurs. Implementations are expected to throw an ExpandVetoException if it is determined that a pending expansion or collapse should not be carried out. Its two methods, treeWillExpand() and

`treeWillCollapse()`, take a `TreeExpansionEvent` as parameter. We can register/unregister a `TreeWillExpandListener` with a tree using `JTree`'s `addTreeWillExpandListener()` and `removeTreeWillExpandListener()` methods respectively.

17.1.24 TreeModelEvent

class javax.swing.event.TreeModelEvent

`TreeModelEvent` is used to notify `TreeModelListeners` that all or part of a `JTree`'s data has changed. This event encapsulates a reference to the source component, and a single `TreePath` or an array of path `Objects` leading to the top-most affected node. We can extract the source as usual, using `getSource()`, and we can extract the path(s) using either of the `getPath()` or `getTreePath()` methods (the former returns an array of `Objects`, the latter returns a `TreePath`). Optionally, this event can also carry an `int` array of node indices and an array of child nodes. These can be extracted using the `getChildIndices()` and `getChildren()` methods respectively.

17.1.25 TreeSelectionEvent

class javax.swing.event.TreeSelectionEvent

`TreeSelectionEvent` is used to notify `TreeSelectionListeners` that the selection of a `JTree` has changed. One variant of this event encapsulates a reference to the source component, the selected `TreePath`, a flag specifying whether the tree path is a new addition to the selection (true if so), and the new and old lead selection paths (remember that the lead selection path is the newest path added to a selection). The second variant of this event encapsulates a reference to the source component, an array of selected `TreePaths`, an array of flags specifying whether each path is a new addition or not, and the new and old lead selection paths. Typical `getXX()` accessor methods allow extraction of this data.

Note: An interesting and unusual method defined in this class is `cloneWithSource()`. By passing it a component, this method returns a clone of the event, but with a reference to the given component as the event source.

17.1.26 TreeExpansionEvent

class javax.swing.event.TreeExpansionEvent

`TreeExpansionEvent` is used to encapsulate a `TreePath` corresponding to a recently, or possibly pending, expanded or collapsed tree path. This path can be extracted with the `getPath()` method.

17.1.27 ExpandVetoException

class javax.swing.tree.ExpandVetoException

`ExpandVetoException` may be thrown by `TreeWillExpandListener` methods to indicate that a tree path expansion or collapse is prohibited, and should be vetoed.

17.1.28 JTree client properties and UI defaults

When using the `MetalL&F`, `JTree` uses a specific line style to represent the edges between nodes. The default is no edges, but we can set `JTree`'s `lineStyle` client property so that each parent node appears connected to each of its child nodes by an angled line:

```
myJTree.putClientProperty("JTree.lineStyle", "Angled");
```

We can also set this property such that each tree cell is separated by a horizontal line:

```
myJTree.putClientProperty("JTree.lineStyle", "Horizontal");
```

To disable the line style:

```
myJTree.putClientProperty("JTree.lineStyle", "None");
```

As with any Swing component, we can also change the UI resource defaults used for all instances of the JTree class. For instance, to change the color of the lines used for rendering the edges between nodes as described above, we can modify the entry in the UIDefaults table for this resource as follows:

```
UIManager.put("Tree.hash",
  new ColorUIResource(Color.lightGray));
```

To modify the open node icons used by all trees when a node's children are shown:

```
UIManager.put("Tree.openIcon", new IconUIResource(
  new ImageIcon("myOpenIcon.gif")));
```

We can do a similar thing for the closed, leaf, expanded, and collapsed icons using `Tree.closedIcon`, `Tree.leafIcon`, `Tree.expandedIcon`, and `Tree.collapsedIcon` respectively. (See the `BasicLookAndFeel` source code for a complete list of UI resource defaults.)

Note: We used the `ColorUIResource` and `IconUIResource` wrapper classes found in the javax.swing.plaf package to wrap our resources before placing them in the UIDefaults table. If we do not wrap our resources in `UIResource` objects they will persist through L&F changes (which may or may not be desirable). See chapter 21 for more about L&F and resource wrappers.

17.1.29 Controlling JTree appearance

Though we haven't concentrated heavily on UI delegate customization for each component throughout this book, Swing certainly provides us with a high degree of flexibility in this area. It is particularly useful with JTree because no methods are provided in the component itself to control the indentation spacing of tree cells (note that the row height can be specified with JTree's `setRowHeight()` method). The JTree UI delegate also provides methods for setting expanded and collapsed icons, allowing us to assign these on a per-component basis rather than a global basis (which is done using `UIManager` -- see 17.1.28). The following BasicTreeUI methods provide this control, and figure 17.1 illustrates:

- `void setCollapsedIcon(Icon newG)`: icon used to specify that a child icon is in the collapsed state.
- `void setExpandedicon(Icon newG)`: icon used to specify that a child icon is in the expanded state.
- `void setLeftChildIndent(int newAmount)`: used to assign a distance between the left side of a parent node and the center of an expand/collapse box of a child node.
- `void setRightChildIndent(int newAmount)`: used to assign a distance between the center of the expand/collapse box of a child node to the left side of that child node's cell renderer.

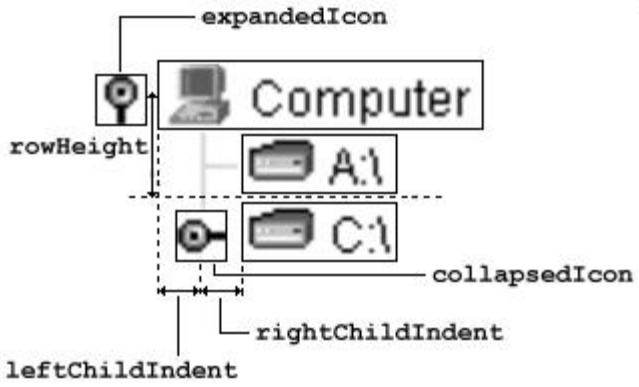


Figure 17.1 JTree UIdelegate icon and indentation properties.
<<file figure17-1.gif>>

To actually use these methods we first have to obtain the target tree's UIdelegate. For example, to assign a left indent of 8 and a right indent of 10:

```
BasicTreeUI basicTreeUI = (BasicTreeUI) myJTree.getUI();
basicTreeUI.setRightChildIndent(10);
basicTreeUI.setLeftChildIndent(8);
```

UIGuideline: When to use a tree As a selection device

The tree component was invented as a mechanism for selection from large hierarchical data sets without having to resort to a "Search" mechanism. As such, JTree falls between listing and search data as a component which can improve usability by easing the process of finding something, providing that the item to be found (or selected) is hidden within a hierarchical data set.

For example, finding an employee by name. For a small data set, a simple list may be sufficient. As the data set grows, it may be easier for the user if you sort the names alphabetically, or by department in which they work. By doing so, you have introduced a hierarchy and may now use a tree component. Use of the tree component may help and speed random selection from the data set, providing that the hierarchical structure used exists within the domain. i.e. don't introduce artificial hierarchies and expect users to understand them. More explicitly, if you can put the hierarchy into your analysis model and users accept it then its fine. If you can't then don't and consequently, don't select a tree component as the solution.

As a data set rises to become very large, tree component may again be of little value and you will need to introduce a full search facility.

As a general rule, when using a tree as a selection device, you would start with the tree collapsed and allow the user to expand it as they "search" for the item they are looking for. If there is a default selection or a current selection then you may expand that part of the tree to show that selection.

As a Visual Layering Device

Even with a small data set, you may find it advantagous to display a visual hierarchy to aid visual comprehension and visual searching. With the employee example, even for a small data set, you may prefer to layer by department or by alphabetical order. Even if selection is being used in the UI, it is important to understand when you chose to use the tree component for improved visual communication i.e. selection would still have been perfectly possible using a list but for visual communication reasons a tree was chosen. When a tree is selected for display only i.e. no selection is taking place, then you are definitely using the tree as a visual layering device.

As a general rule, when you use a tree as a visual layering device, you will be default expand the tree in full, revealing the full hierarchy.

How you use a tree and which options to select amongst the many selection and display variants, can be affected by the usage as we will demonstrate later.

17.2 Basic JTree example - network object IDs

As we know very well by now, `JTree` is suitable for the display and editing of a hierarchical set of objects. To demonstrate this in an introductory-level example, we will consider a set of Object Identifiers (OIDs) used in the Simple Network Management Protocol (SNMP). In the following example, we show how to build a simple `JTree` displaying the initial portion of the OID tree.

SNMP is used extensively to manage network components, and is particularly important in managing internet routers and hosts. Every object managed by SNMP must have a unique OID. An OID is built from a sequence of numbers separated by periods. Objects are organized heirarchicaly and have an OID with a sequence of numbers equal in length to their level (see 17.1.1) in the OID tree. The International Organization of Standards (ISO) establishes rules for building OIDs.

Note that understanding SNMP is certainly not necessary to understand this example. The purpose is to show how to construct a tree using:

- A `DefaultTreeModel` with `DefaultMutableTreeNodes` containing custom user objects.
- A customized `DefaultTreeCellRenerer`.
- A `TreeSelectionListener` which displays information in a status bar based on the `TreePath` encapsulated in the `TreeSelectionEvents` it receives.

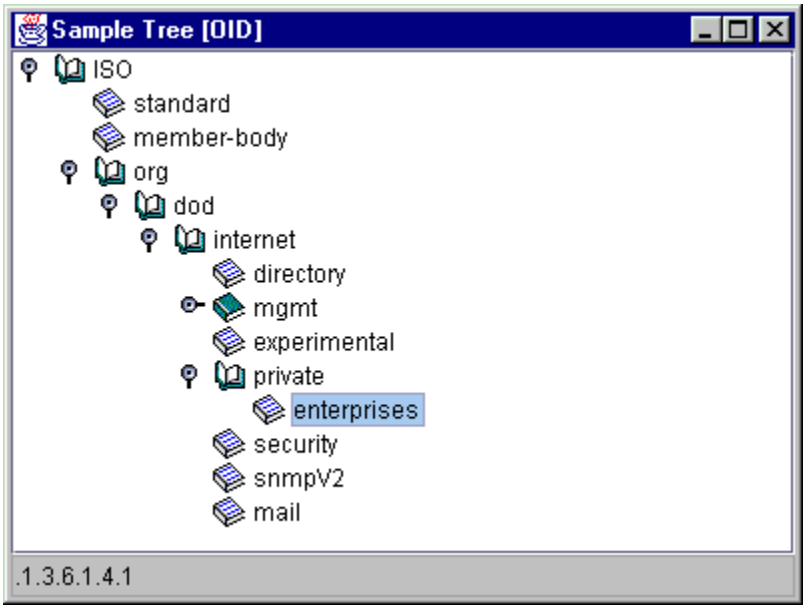


Figure 17.2 JTree with custom cell renderer icons, selection listener, and visible root handles.
<<file figure17-2.gif>>

The Code: Tree1.java
see \Chapter17\1

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

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
```

```
public class Tree1 extends JFrame
{
  protected JTree  m_tree = null;
  protected DefaultTreeModel m_model = null;
  protected JTextField m_display;

  public Tree1() {
    super("Sample Tree [OID]");
    setSize(400, 300);

    Object[] nodes = new Object[5];
    DefaultMutableTreeNode top = new DefaultMutableTreeNode(
      new OidNode(1, "ISO"));
    DefaultMutableTreeNode parent = top;
      nodes[0] = top;

    DefaultMutableTreeNode node = new DefaultMutableTreeNode(
      new OidNode(0, "standard"));
    parent.add(node);
    node = new DefaultMutableTreeNode(new OidNode(2,
      "member-body"));
    parent.add(node);
    node = new DefaultMutableTreeNode(new OidNode(3, "org"));
    parent.add(node);
    parent = node;
    nodes[1] = parent;

    node = new DefaultMutableTreeNode(new OidNode(6, "dod"));
    parent.add(node);
    parent = node;
    nodes[2] = parent;

    node = new DefaultMutableTreeNode(new OidNode(1, "internet"));
    parent.add(node);
    parent = node;
    nodes[3] = parent;

    node = new DefaultMutableTreeNode(new OidNode(1, "directory"));
    parent.add(node);
    node = new DefaultMutableTreeNode(new OidNode(2, "mgmt"));
    parent.add(node);
    nodes[4] = node;
    node.add(new DefaultMutableTreeNode(new OidNode(1, "mib-2")));
    node = new DefaultMutableTreeNode(new OidNode(3,
      "experimental"));
    parent.add(node);
    node = new DefaultMutableTreeNode(new OidNode(4, "private"));
    node.add(new DefaultMutableTreeNode(new OidNode(1,
      "enterprises")));
    parent.add(node);
    node = new DefaultMutableTreeNode(new OidNode(5, "security"));
    parent.add(node);
    node = new DefaultMutableTreeNode(new OidNode(6, "snmpV2"));
    parent.add(node);
    node = new DefaultMutableTreeNode(new OidNode(7,
      "mail"));
    parent.add(node);

    m_model = new DefaultTreeModel(top);
    m_tree = new JTree(m_model);

    DefaultTreeCellRenderer renderer = new
      DefaultTreeCellRenderer();
```

```
    renderer.setOpenIcon(new ImageIcon("opened.gif"));
    renderer.setClosedIcon(new ImageIcon("closed.gif"));
    renderer.setLeafIcon(new ImageIcon("leaf.gif"));
    m_tree.setCellRenderer(renderer);

    m_tree.setShowsRootHandles(true);
    m_tree.setEditable(false);
    TreePath path = new TreePath(nodes);
    m_tree.setSelectionPath(path);

    m_tree.addTreeSelectionListener(new
      OidSelectionListener());

    JScrollPane s = new JScrollPane();
    s.getViewport().add(m_tree);
    getContentPane().add(s, BorderLayout.CENTER);

    m_display = new JTextField();
    m_display.setEditable(false);
    getContentPane().add(m_display, BorderLayout.SOUTH);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public static void main(String argv[]) {
    new Tree1();
  }

  class OidSelectionListener implements TreeSelectionListener
  {
    public void valueChanged(TreeSelectionEvent e) {
      TreePath path = e.getPath();
      Object[] nodes = path.getPath();
      String oid = "";
      for (int k=0; k<nodes.length; k++) {
        DefaultMutableTreeNode node =
          (DefaultMutableTreeNode)nodes[k];
        OidNode nd = (OidNode)node.getUserObject();
        oid += "."+nd.getId();
      }
      m_display.setText(oid);
    }
  }
}

class OidNode
{
  protected int    m_id;
  protected String m_name;

  public OidNode(int id, String name) {
    m_id = id;
    m_name = name;
  }

  public int getId() { return m_id; }
```

```
  public String getName() { return m_name; }

  public String toString() { return m_name; }
}
```

Understanding the Code

Class Tree1

This class extends `JFrame` to implement the frame container for our `JTree`. Three instance variables are declared:

`JTree m_tree`: our OID tree.

`DefaultTreeModel m_model`: tree model to manage data.

`JTextField m_display`: used as a status bar to display the selected object's OID.

The constructor first initializes the parent frame object. Then a number of `DefaultMutableTreeNodes` encapsulating `OidNodes` (see below) are created. These objects form a hierarchical structure with `DefaultMutableTreeNode top` at the root. Note that during the construction of these nodes, the `Object[] nodes` array is populated with a path of nodes leading to the "mgmt" node.

`DefaultTreeModel m_model` is created with the `top` node as the root, and `JTree m_tree` is created to manage this model. Then some specific options are set for this tree component. First, we replace the default icons for opened, closed and leaf icons with our custom icons, using a `DefaultTreeCellRenderer` as our tree's cell renderer:

```
        DefaultTreeCellRenderer renderer = new
            DefaultTreeCellRenderer();
        renderer.setOpenIcon(new ImageIcon("opened.gif"));
        renderer.setClosedIcon(new ImageIcon("closed.gif"));
        renderer.setLeafIcon(new ImageIcon("leaf.gif"));
        m_tree.setCellRenderer(renderer);
```

Then we set the `showsRootHandles` property to `true`, the `Editable` property to `false`, and select the path determined by the `nodes` array formed above:

```
        m_tree.setShowsRootHandles(true);
        m_tree.setEditable(false);
        TreePath path = new TreePath(nodes);
        m_tree.setSelectionPath(path);
```

Our custom `OidSelectionListener` (see below) `TreeSelectionListener` is added to the tree to receive notification when our tree's selection changes.

A `JScrollPane` is created to provide scrolling capabilities, and our tree is added to its `JViewport`. This `JScrollPane` is then added to the center of our frame. A non-editable `JTextField m_display` is created and added to the south region of our frame's content pane to display the currently selected OID.

Class Tree1.OidSelectionListener

This inner class implements the `TreeSelectionListener` interface to receive notifications about when our tree's selection changes. Our `valueChanged()` implementation extracts `TreePath` for the current selection and visits each node, starting from the root, accumulating the OID in .N.N.N form as it goes (where N is a digit). This method ends by displaying the resulting OID in our text field status bar.

Class OidNode

This class encapsulates a single object identifier as a number and a `String` name describing the associated object. Both values are passed to the `OidNode` constructor. Instances of this class are passed directly to the `DefaultMutableTreeNode` constructor to act as a node's user object. The overridden `toString()` method is used to return the name `String` so that our tree's cell renderer will display each node correctly. Recall that, by default, `DefaultTreeCellRenderer` will call a node's user object `toString()` method for rendering.

Running the Code

Figure 17.1 shows our OID tree in action. Try selecting various tree nodes and note how the selected OID is displayed at the bottom of the frame.

`UI Guideline: Icons & Root Handles`

In this example, we are visually re-inforcing the data hierarchy with icons. The icons are overloading on the root handles to communicate whether an element is a document or a container and whether that container is open or closed. The book icon has two variants to communicate "open book" and "closed book". The icons are communicating the same information as the root handles. Therefore, it is technically possible to remove the root handles. In some problem domains, hidden root handles may be more appropriate, providing that the Users are comfortable with interpreting the book icons and realise that a "closed book" icon means that the node can be expanded.

17.3 Directories tree: part I - dynamic node retrieval

The example in this section uses the `JTree` component to display and navigate through a tree of directories located on drives accessible from the user's machine. We will show how to build a custom tree cell renderer as well as create and insert tree nodes dynamically.

The main problem encountered in building this application is the fact that it is not practical to read all directories for all accessible drives before displaying our tree component. This would take an extremely long time. To deal with this issue we initially display only the roots (i.e. disk partitions or network drives), and then dynamically expand the tree as the user navigates through it. This requires the use of threads and `SwingUtilities.invokeLater()` for thread-safe updating of our tree.

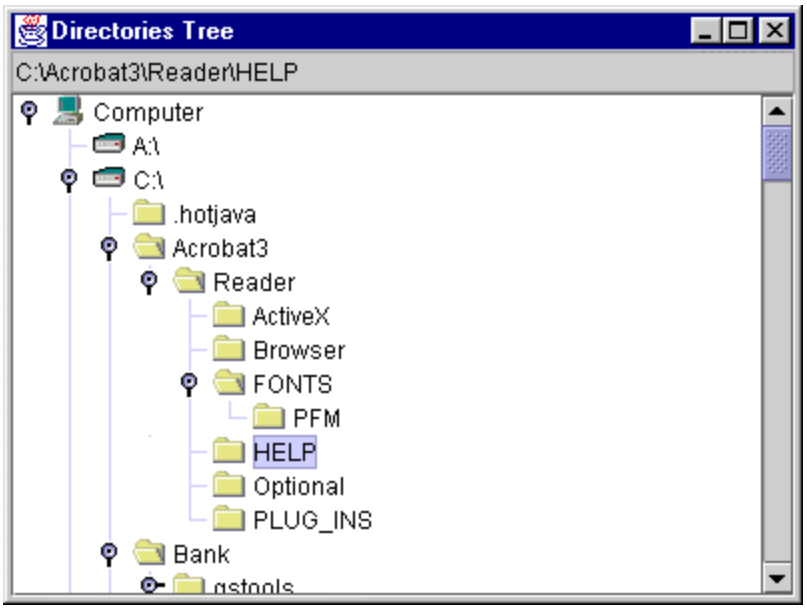


Figure 17.3 Dynamic, threaded directories tree with custom cell renderer and angled line style.

<<file figure17-3.gif>>

The Code: FileTree1.java

see \Chapter17\2

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

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;

public class FileTree1 extends JFrame
{
  public static final ImageIcon ICON_COMPUTER =
    new ImageIcon("computer.gif");
  public static final ImageIcon ICON_DISK =
    new ImageIcon("disk.gif");
  public static final ImageIcon ICON_FOLDER =
    new ImageIcon("folder.gif");
  public static final ImageIcon ICON_EXPANDEDFOLDER =
    new ImageIcon("expandedfolder.gif");

  protected JTree  m_tree;
  protected DefaultTreeModel m_model;
  protected JTextField m_display;

  public FileTree1() {
    super("Directories Tree");
    setSize(400, 300);

    DefaultMutableTreeNode top = new DefaultMutableTreeNode(
      new IconData(ICON_COMPUTER, null, "Computer"));

    DefaultMutableTreeNode node;
      File[] roots = File.listRoots();
    for (int k=0; k<roots.length; k++) {
      node = new DefaultMutableTreeNode(
```

```
      new IconData(ICON_DISK,
      null, new FileNode(roots[k])));
    top.add(node);
    node.add( new DefaultMutableTreeNode(
    new Boolean(true)));
  }

  m_model = new DefaultTreeModel(top);
  m_tree = new JTree(m_model);
  m_tree.getSelectionModel().setSelectionMode(
    TreeSelectionModel.SINGLE_TREE_SELECTION);
  m_tree.putClientProperty("JTree.lineStyle", "Angled");
  TreeCellRenderer renderer = new IconCellRenderer();
  m_tree.setCellRenderer(renderer);
  m_tree.addTreeExpansionListener(new DirExpansionListener());
  m_tree.addTreeSelectionListener(new DirSelectionListener());
  m_tree.setShowsRootHandles(true);
  m_tree.setEditable(false);

  JScrollPane s = new JScrollPane();
  s.getViewport().add(m_tree);
  getContentPane().add(s, BorderLayout.CENTER);

  m_display = new JTextField();
  m_display.setEditable(false);
  getContentPane().add(m_display, BorderLayout.NORTH);

  WindowListener wndCloser = new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  };
  addWindowListener(wndCloser);

  setVisible(true);
}

DefaultMutableTreeNode getTreeNode(TreePath path) {
  return (DefaultMutableTreeNode) (path.getLastPathComponent());
}

FileNode getFileNode(DefaultMutableTreeNode node) {
  if (node == null)
    return null;
  Object obj = node.getUserObject();
  if (obj instanceof IconData)
    obj = ((IconData)obj).getObject();
  if (obj instanceof FileNode)
    return (FileNode)obj;
  else
    return null;
}

// Make sure expansion is threaded and updating the tree model
// only occurs within the event dispatching thread.
class DirExpansionListener implements TreeExpansionListener
{
  public void treeExpanded(TreeExpansionEvent event) {
    final DefaultMutableTreeNode node = getTreeNode(
      event.getPath());
    final FileNode fnode = getFileNode(node);

    Thread runner = new Thread() {
```

```
        public void run() {
          if (fnode != null && fnode.expand(node)) {
            Runnable runnable = new Runnable() {
              public void run() {
                m_model.reload(node);
              }
            };
            SwingUtilities.invokeLater(runnable);
          }
        }
      };
      runner.start();
    }

    public void treeCollapsed(TreeExpansionEvent event) {}
  }

  class DirSelectionListener implements TreeSelectionListener
  {
    public void valueChanged(TreeSelectionEvent event) {
      DefaultMutableTreeNode node = getTreeNode(event.getPath());
        FileNode fnode = getFileNode(node);
        if (fnode != null)
          m_display.setText(fnode.getFile().getAbsolutePath());
        else
          m_display.setText("");
    }
  }

  public static void main(String argv[]) { new FileTree1(); }
}

class IconCellRenderer extends JLabel implements TreeCellRenderer
{
  protected Color m_textSelectionColor;
  protected Color m_textNonSelectionColor;
  protected Color m_bkSelectionColor;
  protected Color m_bkNonSelectionColor;
  protected Color m_borderSelectionColor;

  protected boolean m_selected;

  public IconCellRenderer() {
    super();
    m_textSelectionColor = UIManager.getColor(
      "Tree.selectionForeground");
    m_textNonSelectionColor = UIManager.getColor(
      "Tree.textForeground");
    m_bkSelectionColor = UIManager.getColor(
      "Tree.selectionBackground");
    m_bkNonSelectionColor = UIManager.getColor(
      "Tree.textBackground");
    m_borderSelectionColor = UIManager.getColor(
      "Tree.selectionBorderColor");
    setOpaque(false);
  }

  public Component getTreeCellRendererComponent(JTree tree,
   Object value, boolean sel, boolean expanded, boolean leaf,
   int row, boolean hasFocus)
  {
    DefaultMutableTreeNode node = (DefaultMutableTreeNode)value;
    Object obj = node.getUserObject();
```

```
    setText(obj.toString());

    if (obj instanceof Boolean)
      setText("Retrieving data...");

    if (obj instanceof IconData) {
      IconData idata = (IconData)obj;
      if (expanded)
        setIcon(idata.getExpandedIcon());
      else
        setIcon(idata.getIcon());
    }
    else
      setIcon(null);

    setFont(tree.getFont());
    setForeground(sel ? m_textSelectionColor :
      m_textNonSelectionColor);
    setBackground(sel ? m_bkSelectionColor :
      m_bkNonSelectionColor);
    m_selected = sel;
    return this;
  }

  public void paintComponent(Graphics g) {
    Color bColor = getBackground();
    Icon icon = getIcon();

    g.setColor(bColor);
    int offset = 0;
    if(icon != null && getText() != null)
      offset = (icon.getIconWidth() + getIconTextGap());
    g.fillRect(offset, 0, getWidth() - 1 - offset,
      getHeight() - 1);

    if (m_selected) {
      g.setColor(m_borderSelectionColor);
      g.drawRect(offset, 0, getWidth()-1-offset, getHeight()-1);
    }
    super.paintComponent(g);
  }
}

class IconData
{
  protected Icon   m_icon;
  protected Icon   m_expandedIcon;
  protected Object m_data;

  public IconData(Icon icon, Object data) {
    m_icon = icon;
    m_expandedIcon = null;
    m_data = data;
  }

  public IconData(Icon icon, Icon expandedIcon, Object data) {
    m_icon = icon;
    m_expandedIcon = expandedIcon;
    m_data = data;
  }

  public Icon getIcon() { return m_icon; }
```

```
  public Icon getExpandedIcon() {
    return m_expandedIcon!=null ? m_expandedIcon : m_icon;
  }

  public Object getObject() { return m_data; }

  public String toString() { return m_data.toString(); }
}

class FileNode
{
  protected File m_file;

  public FileNode(File file) { m_file = file; }

  public File getFile() { return m_file; }

  public String toString() {
    return m_file.getName().length() > 0 ? m_file.getName() :
      m_file.getPath();
  }

  public boolean expand(DefaultMutableTreeNode parent) {
    DefaultMutableTreeNode flag =
      (DefaultMutableTreeNode)parent.getFirstChild();
    if (flag==null)      // No flag
      return false;
    Object obj = flag.getUserObject();
    if (!(obj instanceof Boolean))
      return false;      // Already expanded

    parent.removeAllChildren();  // Remove Flag

    File[] files = listFiles();
    if (files == null)
      return true;

    Vector v = new Vector();

    for (int k=0; k<files.length; k++) {
      File f = files[k];
      if (!(f.isDirectory()))
        continue;

      FileNode newNode = new FileNode(f);

      boolean isAdded = false;
      for (int i=0; i<v.size(); i++) {
        FileNode nd = (FileNode)v.elementAt(i);
        if (newNode.compareTo(nd) < 0) {
          v.insertElementAt(newNode, i);
          isAdded = true;
          break;
        }
      }
      if (!isAdded)
      v.addElement(newNode);
    }

    for (int i=0; i<v.size(); i++) {
      FileNode nd = (FileNode)v.elementAt(i);
      IconData idata = new IconData(FileTree1.ICON_FOLDER,
        FileTree1.ICON_EXPANDEDFOLDER, nd);
```

```
      DefaultMutableTreeNode node =
        new DefaultMutableTreeNode(idata);
      parent.add(node);

      if (nd.hasSubDirs())
        node.add(new DefaultMutableTreeNode(
          new Boolean(true) ));
    }
    return true;
  }

  public boolean hasSubDirs() {
    File[] files = listFiles();
    if (files == null)
      return false;
    for (int k=0; k<files.length; k++) {
      if (files[k].isDirectory())
        return true;
    }
    return false;
  }

  public int compareTo(FileNode toCompare) {
    return  m_file.getName().compareToIgnoreCase(
      toCompare.m_file.getName() );
  }

  protected File[] listFiles() {
    if (!m_file.isDirectory())
      return null;
    try {
      return m_file.listFiles();
    }
    catch (Exception ex) {
      JOptionPane.showMessageDialog(null,
        "Error reading directory "+m_file.getAbsolutePath(),
        "Warning", JOptionPane.WARNING_MESSAGE);
          return null;
    }
  }
}
```

Understanding the Code

Class FileTree1

Four custom icons are loaded as static `ImageIcon` variables: `ICON_COMPUTER`, `ICON_DISK`, `ICON_FOLDER`, and `ICON_EXPANDEDFOLDER`, and three instance variables are declared:

- `JTree m_tree`: tree component to display directory nodes.
- `DefaultTreeModel m_model`: tree model to manage nodes.
- `JTextField m_display`: component to display selected directory (acts as a status bar).

The `FileTree1` constructor creates and initializes all GUI components. A root node "Computer" hosts child nodes for all disk partitions and network drives in the system. These nodes encapsulate `Files` retrieved with the static `File.listRoots()` method (a valuable addition to the `File` class in Java 2). Note that `IconData` objects (see below) encapsulate `Files` in the tree. Also note that each newly created child node immediately receives a child node containing a `Boolean` user object. This `Boolean` object allows us to display an expanding message for nodes when they are in the process of being expanded. Exactly how we

expand them will be explained soon enough.

We then create a DefaultTreeModel and pass our "Computer" node as the root. This model is used to instantiate our JTree object:

```
m_model = new DefaultTreeModel(top);
m_tree = new JTree(m_model);
```

We then set the lineStyle client property so that angled lines will represent the edges between parent and child nodes:

```
m_tree.putClientProperty("JTree.lineStyle", "Angled");
```

We also use a custom tree cell renderer, as well as a tree exapansion listener and a tree selection listener, (each of which in detail below): IconCellRenderer, DirExpansionListener, and DirSelectionListener respectively.

The actual contents of our tree nodes represent directories. Each node is a DefaultMutableTreeNode with an IconData user object. Each user object is an instance of IconData, and each IconData contains an instance of FileNode. Each FileNode contain a java.io.File object. Thus we have a four layered nested structure:

- DefaultMutableTreeNode is used for each node to represent a directory or disk (as well as the "Computer" root node). When we retrieve a node at the end of a given TreePath, using the getLastPathComponent() method, we are provided with an instance of this class.
- IconData (see below) sits inside DefaultMutableTreeNode and provides custom icons for our tree cell renderer, and encapsulation of a FileNode object. IconData can be retrieved using DefaultMutableTreeNode's getUserObject() method. Note that we need to cast the returned Object to an IconData instance.
- FileNode (see below) sits inside IconData and encapsulates a File objects. A FileNode can be retrieved using IconData's getObject() method, also requireing a subsequent cast.
- A File object sits inside a FileNode and can be retrieved using FileNode's getFile() method.

Figure 17.2 illustrates this structure.

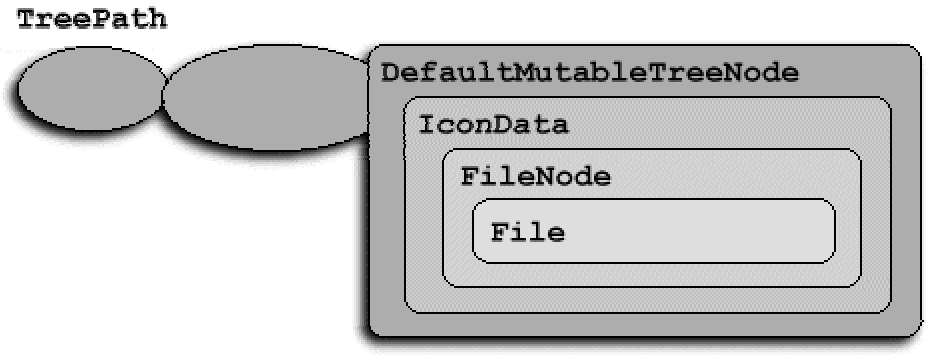


Figure 17.2 Nested structure of our tree nodes.
<<file figure17-2.gif>>

To keep things simple, two helper methods are provided to work with these encapsulated nodes: getTreeNode() retrieves a DefaultMutableTreeNode from a given TreePath, and getFileNode() retrieves the FileNode (or null) from a DefaultMutableTreeNode. We will see where these methods are needed soon enough.

Class FileTree1.DirExpansionListener

This inner class implements `TreeExpansionListener` to listen for tree expansion events. When a node is expanded, method `treeExpanded()` retrieves the `FileNode` instance for that node, and, if it's not `null`, calls the `expand()` method on it (see below). This is wrapped in a separate thread because it can often be a very time consuming process and we do not want the application to freeze. Inside this thread, once expand() has completed, we need to update the tree model with any new nodes retrieved. As we learned in chapter 2, updating the state of a component should only occur within the event dispatching thread. For this reason we wrap the call to `reload()` in a `Runnable` and send it the event dispatching queue using `SwingUtilities.invokeLater()`:

```
Runnable runnable = new Runnable() {
  public void run() {
    m_model.reload(node);
  }
};
SwingUtilities.invokeLater(runnable);
```

As we will see below in our discussion of `IconCellRenderer`, by placing a `Boolean` user object in a dummy child node of each non-expanded node, we allow a certain `String` to be displayed while a node is in the process of being expanded. In our case, "Retreiving data..." is shown below a node until finished expanding.

Class FileTree1.DirSelectionListener

This inner class implements `TreeSelectionListener` to listen for tree selection events. When a node is selected, the `valueChanged()` method extracts the `FileNode` instance contained in that node, and, if it is not `null`, displays the absolute path to that directory in the `m_display` textfield.

Class IconCellRenderer

This class implements the `TreeCellRenderer` interface and extends `JLabel`. The purpose of this renderer is to display custom icons and access `FileNodes` contained in `IconData` instances.

First we declare five `Colors` and retrieve them from the current look-and-feel in use through `UIManager`'s `getColor()` method. The `getTreeCellRendererComponent()` method is then implemented to set the proper text and icon (retrieved from the underlying `IconData` object). If the user object happens to be a `Boolean`, this signifies that a node is in the process of being expanded:

```
if (obj instanceof Boolean)
  setText("Retrieving data...");
```

The reason we do this is slightly confusing. In the `FileNode` `expand()` method (see below), when each new node is added to our tree it receives a node containing a `Boolean` user object only if the corresponding directory has sub-directories. When we click on this node, the `Boolean` child will be immediately shown, and we also generate an expansion event that is received by our `DirExpansionListener`. As we discussed above, this listener extracts the encapsulated `FileNode` and calls the `FileNode` `expand()` method on it. The child node containing the `Boolean` object is removed before all new nodes are added. Until this update occurs, the `JTree` will display the `Boolean` child node, in effect telling us that the expansion is not complete yet. So, if our cell renderer detects a `Boolean` user object, we simply display "Recieving data..." for its text.

The `paintComponent()` method is overridden to fill the text background with the appropriate color set in the `getTreeCellRendererComponent()` method. Fortunately we don't need to explicitly draw the text and icon because we have extended `JLabel`, which can do this for us.

Class IconData

Instances of this class are used as our `DefaultMutableTreeNode` user data objects, and they encapsulate a generic `Object m_data` and two `Icons` for use by `IconCellRenderer`. These icons can be retrieved with our `getIcon()` and `getExpandedIcon()` methods. The icon retrieved with `getExpandedIcon()` represents an expanded folder, and the icon retrieved with `getIcon()` represents a collapsed/non-expanded folder. Note that the `toString()` method invokes `toString()` on the `m_data` object. In our example this object is either a `FileNode`, in the case of an expanded folder, or a `Boolean`, in the case of a non-expanded folder.

Class FileNode

This class encapsulates a `File` object, which is in turn encapsulated in an `IconData` object in a `DefaultMutableTreeNode`.

As we discussed above, the `toString()` method determines the text to be displayed in each tree cell containing a `FileNode`. It returns `File.getName()` for regular directories and `File.getPath()` for partitions.

The most interesting and complex method of this class is `expand()`, which attempts to expand a node by dynamically inserting new `DefaultMutableTreeNode`'s corresponding to each sub-directory. This method returns `true` if nodes are added, and `false` otherwise. First we need to discuss the mechanism of dynamically reading information (of any kind) into a tree:

- Before we add any new node to the tree, we must determine somehow whether or not it has children (we don't need a list of children yet, just a yes or no answer).
- If a newly created node has children, a fake child to be used as a flag will be added to it. This will signify that the parent node has not been expanded.

When a node is expanded, its list of children is examined. Three situations are possible:

- No children. This node is a leaf and cannot be expanded (remember, we've checked previously whether or not any newly created node has children).
- One flag child is present. That node has children which have not been added yet. So we create these children (depending on the nature of the tree) and add new nodes to the parent node.
- One or more non-flag children are present. This node has already been processed, so expand it as usual.

Method `FileNode.expand()` implements this dynamic tree expansion strategy, and takes a parent node as parameter. In the process of expansion it also alphabetically sorts each node for a more organized display structure. Initially this method checks the first child of the given parent node:

```
DefaultMutableTreeNode flag =
  (DefaultMutableTreeNode)parent.getFirstChild();
if (flag==null)      // No flag
  return false;
Object obj = flag.getUserObject();
if (!(obj instanceof Boolean))
  return false;    // Already expanded

parent.removeAllChildren();  // Remove Flag
```

If no child is found, it can only mean that this node was already checked and was found to be a true leaf (a directory with no sub-directories). If this isn't the case then we extract the associated data object and check whether it is an instance of `Boolean`, i.e. it is a flag child. If yes, the flag child is removed and our method proceeds to add nodes corresponding to each sub-directory. Otherwise we conclude that this node has already

been processed and return, allowing it to be expanded as usual.

We process a newly expanded node by retrieving an array of `File` objects representing files contained in the corresponding directory:

```
File[] files = listFiles();
if (files == null)
  return true;
```

If the contents have been successfully read, we check for sub-directories and create new `FileNodes` for each:

```
Vector v = new Vector();

for (int k=0; k<files.length; k++) {
  File f = files[k];
  if (!(f.isDirectory()))
    continue;

  FileNode newNode = new FileNode(f);
```

To perform alphabetical sorting of child nodes we store them in a temporary collection `Vector v`, and iterate through our array of `Files`, inserting them accordingly

```
  boolean isAdded = false;
  for (int i=0; i<v.size(); i++) {
    FileNode nd = (FileNode)v.elementAt(i);
    if (newNode.compareTo(nd) < 0) {
      v.insertElementAt(newNode, i);
      isAdded = true;
       break;
    }
  }
  if (!isAdded)
  v.addElement(newNode);
}
```

We then wrap each newly created `FileNode` object in an `IconData` to bind them with folder icons, and add the sorted nodes to the given parent node. At the same time, flags are added to new nodes if they contain any sub-directories themselves (this is checked by the `hasSubDirs()` method):

```
for (int i=0; i<v.size(); i++) {
  FileNode nd = (FileNode)v.elementAt(i);
  IconData idata = new IconData(FileTree1.ICON_FOLDER,
  FileTree1.ICON_EXPANDEDFOLDER, nd);
  DefaultMutableTreeNode node = new
    DefaultMutableTreeNode(idata);
  parent.add(node);
  if (nd.hasSubDirs())
    node.add(new DefaultMutableTreeNode(
      new Boolean(true) ));
}
return true;
```

The rest of `FileNode` class implements three methods which do not require much explanation at this point:

`boolean hasSubDirs()` returns `true` if this directory has sub-directories and `false` otherwise.

`int compareTo(FileNode toCompare)` returns the result of the alphabetical comparison of this directory with another given as parameter.

`File[] listFiles()` reads a list of contained flies in this directory. If an exception occurs (possible

while reading from a floppy disk or network drive) this method displays a warning message and returns `null`.

Running the Code

Figure 17.3 shows our directory tree at work. Note the use of custom icons for partition roots. Try selecting various directories and note how the selected path is reflected at the top of the frame in our status bar. Also note that when expanding large directories "Retrieving data..." will be displayed underneath the corresponding node. Because we have properly implemented multithreading, we can go off and expand other directories while this one is being processed. Also not that the tree is always updated correctly when the expanding procedure completes because we have made sure to only change its state in the event dispatching thread using `invokeLater()`.

UI Guideline: When to use connecting lines

Angled connecting lines (or edges) add visual noise and clutter to a tree display. Reduced visual clutter aiding recognition and comprehension is a clear advantage to leaving them out of the design. So when is it appropriate to have them included?

Include the line edges when it is likely that:

(a) several nodes may be expanded at one time and/or

(b) the data set is very large and a node may expand off the bottom of the screen and perhaps several screens deep. In this case the introduction of the lines, helps to give the viewer a clear picture of how many layers deep in the hierarchy they are viewing and makes it easier to trace back to the original root node.

17.4 Directories tree: part II - popup menus and programmatic navigation

The example in the previous section can be extended in numerous ways to serve as a framework for a much more flexible application. In this section we'll add simple popup menus to our tree, displayed in response to a right mouse click, with content dependant on the click location. (We discussed popup menus in chapter 12.)

Our popup menu either contains an "Expand" or "Collapse" item depending on the status of the corresponding node nearest to the mouse click. These items will programatically invoke an expand or collapse of the given node. Our popup menu also contains "Delete" and "Rename" dummy items that are not completely implemented, but explicitly illustrate how we might continue to build upon this example to create a more complete directory explorer application.

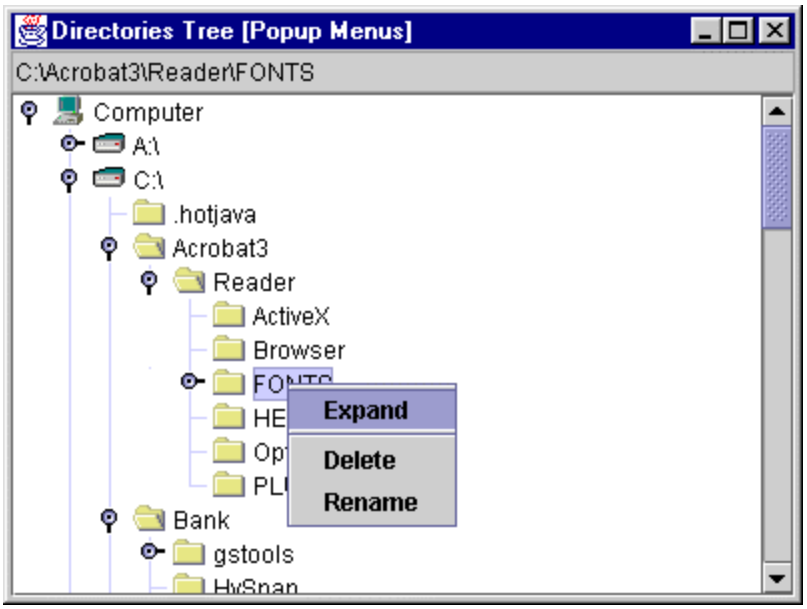


Figure 17.4 Node dependant popup menus allowing programmatic expand and collapse.

<<file figure17-4.gif>>

The Code: FileTree2.java

see \Chapter17\3

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

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;

public class FileTree2 extends JFrame
{
  // Unchanged code from section 17.3

  protected JPopupMenu m_popup;
  protected Action m_action;
  protected TreePath m_clickedPath;

  public FileTree2() {
    super("Directories Tree [Popup Menus]");
    setSize(400, 300);
    getContentPane().setLayout(new BorderLayout());

    // Unchanged code from sectionn 17.3

    m_popup = new JPopupMenu();
    m_action = new AbstractAction() {
      public void actionPerformed(ActionEvent e) {
        if (m_clickedPath==null)
          return;
        if (m_tree.isExpanded(m_clickedPath))
          m_tree.collapsePath(m_clickedPath);
        else
          m_tree.expandPath(m_clickedPath);
      }
```

```
    };
    m_popup.add(m_action);
    m_popup.addSeparator();

    Action a1 = new AbstractAction("Delete") {
      public void actionPerformed(ActionEvent e) {
        m_tree.repaint();
        JOptionPane.showMessageDialog(FileTree2.this,
          "Delete option is not implemented",
          "Info", JOptionPane.INFORMATION_MESSAGE);
      }
    };
    m_popup.add(a1);

    Action a2 = new AbstractAction("Rename") {
      public void actionPerformed(ActionEvent e) {
        m_tree.repaint();
        JOptionPane.showMessageDialog(FileTree2.this,
          "Rename option is not implemented",
          "Info", JOptionPane.INFORMATION_MESSAGE);
      }
    };
    m_popup.add(a2);
    m_tree.add(m_popup);
    m_tree.addMouseListener(new PopupTrigger());

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  // Unchanged code from section 17.3

  class PopupTrigger extends MouseAdapter {
    public void mouseReleased(MouseEvent e) {
      if (e.isPopupTrigger()) {
        int x = e.getX();
        int y = e.getY();
        TreePath path = m_tree.getPathForLocation(x, y);
        if (path != null) {
          if (m_tree.isExpanded(path))
            m_action.putValue(Action.NAME, "Collapse");
          else
            m_action.putValue(Action.NAME, "Expand");
          m_popup.show(m_tree, x, y);
          m_clickedPath = path;
        }
      }
    }
  }

  public static void main(String argv[]) {
    new FileTree2();
  }
}

// Unchanged code from section 17.3
```

Understanding the Code

Class FileTree2

This example adds three new instance variables:

`JPopupMenu m_popup`: popup menu component

`Action m_action`: context sensitive menu action.

`TreePath m_clickedPath`: most recent tree path corresponding to a mouse click.

New code in the base class constructor creates a popup menu component and populates it with three menu items: "Expand", "Delete", and "Rename". The last two items intentionally just display an "option is not implemented" message. Their true implementations would take us too far into file manipulation techniques for this chapter. The first one, on the contrary, is quite meaningful here. The corresponding `actionPerformed()` method uses the recently clicked path (not necessary the currently selected path) which has been set by the `PopupTrigger` instance (see below). This path is collapsed if it currently expanded, or expanded it if this path is currently collapsed.

Finally this newly created popup menu is added to our tree component. An instance of our `PopupTrigger` class is also attatched to our tree as a mouse listener.

Class FileTree2.PopupTrigger

This class extends `MouseAdapter` to trigger the display of our popup menu. This menu should be displayed when the right mouse button is released. So we override the `mouseReleased()` method and check whether `isPopupTrigger()` is true (see `MouseEvent` API docs). In this case we determine the coordinates of the click and retrieve the `TreePath` corresponding to that coordinate with the `getPathForLocation()` method. If a path is not found (i.e. the click does not occur on a tree node or leaf) we do nothing. Otherwise we adjust the title of the first menu item accordingly, display our popup menu with the `show()` method, and store our recently clicked path in the `m_clickedPath` instance variable (for use by the expand/collapse `Action` as discussed above).

Running the Code

Figure 17.4 shows our directories tree application displaying a context sensitive popup menu. Note how the first menu item is changed depending on whether the selected tree node is collapsed or expanded. Also note that the tree can be manipulated (expanded or collapsed) programmatically by choosing the "Collapse" or "Expand" popup menu item.

UI Guideline: Visually re-inforcing variations in behaviour

If you intend to introduce context dependant pop-up menus on tree cells, then this is an ideal time to consider the use of a tree cell renderer which incorporates an icon. The differing icons help to re-inforce the idea that the data on the cell is of different types and consequently when the behaviour is slightly different across nodes, it is less surprising. The icon visually re-inforces the difference in behaviour.

17.5 Directories tree: part III - tooltips

As we discussed in chapter 5, tooltips are commonly used to display helpful information. In this example we will show how to use tooltips specific to each tree cell. The key point (which is mentioned in the `JTree` documentation, but can be easily overlooked) is to register the tree component with the `ToolTipManager` instance:

```
  ToolTipManager.sharedInstance().registerComponent(myTree);
```

Without doing this, no tooltips will appear over our tree (refer back to chapter 2, section 2.5, for more about shared instances and service classes).

The `JTree` component overrides the `getToolTipText(MouseEvent ev)` method inherited from `JComponent`, and delegates this call to the tree's cell renderer component. By implementing the `getToolTipText(MouseEvent ev)` method in our renderer we can allow cell-specific tooltips. Specifically, we can can return the tooltip text as a `String` depending on the last node passed to the `getTreeCellRendererComponent()` method. Alternatively we can subclass our `JTree` component and provide our own `getToolTipText()` implementation. We use the latter method here.

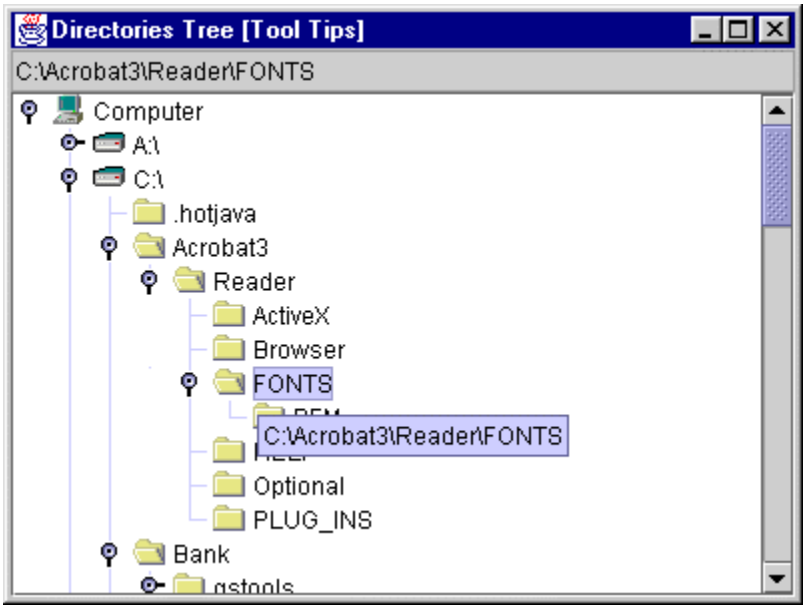


Figure 17.5 JTree with node-specific tooltips.
<<file figure17-5.gif>>

The Code: FileTree2.java
see \Chapter17\4

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

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;

public class FileTree3 extends JFrame
{
  // Unchanged code from section 17.4

  public FileTree3() {
    super("Directories Tree [Tool Tips]");
    setSize(400, 300);
    getContentPane().setLayout(new BorderLayout());

    // Unchanged code
```

```
    m_model = new DefaultTreeModel(top);
    m_tree = new JTree(m_model)
    {
      public String getToolTipText(MouseEvent ev)
      {
        if(ev == null)
          return null;
        TreePath path = m_tree.getPathForLocation(ev.getX(),
          ev.getY());
        if (path != null)
        {
          FileNode fnode = getFileNode(getTreeNode(path));
          if (fnode==null)
            return null;
          File f = fnode.getFile();
          return (f==null ? null : f.getPath());
        }
        return null;
      }
    };
    ToolTipManager.sharedInstance().registerComponent(m_tree);

    // The rest of the code is unchanged
```

Understanding the Code

Class FileTree3

This example anonymously subclasses the `JTree` component to override `getToolTipText(MouseEvent ev)`, which finds the path closest to the current mouse location, determines the `FileNode` at the end of that path, and returns the full file path to that node as a `String` for use as tooltip text. Also note that our `JTree` component is manually registered with the shared instance of `ToolTipManager`, as discussed above.

Running the Code

Figure 17.5 shows our directories tree application displaying a tooltip with text specifying the full path of the directory corresponding to the node nearest to the current mouse location.

UIGuideline: Tooltips as an aid to selection

Tooltips have two really useful advantages for tree cells. Trees have a habit of wandering off to the right hand side of a display, particularly in deep hierarchies. This may result in cell labels being clipped. Using the tooltip to display the full length cell label, will speed selection and prevent the need for scrolling.

The second use is shown clearly in this example. The tooltip is used to unravel the hierarchy. This would be particularly useful when the original root node is off screen. The User can see quickly, the full hierarchical path to the selected cell. This is a very powerful aid to correct selection. Another example of additional coding effort providing improved usability.

17.6 JTree and XML documents

Many see the future of the web in XML™ (Extensible Markup Language). This standard will most likely replace HTML. Unlike HTML, XML allows the definition of custom document tags, allowing the transmition of virtually any type of information over the web. Many sources of information about XML are available, including the standard definition at http://www.w3.org/TR/WD-xml-lang-970331.html.

XML documents have a tree-like structure, and `JTree` can be very useful for constructing an XML structure browser. In this section we'll show how to build a simple implementation of such a browser. We do not intend

to give an introduction to XML, and we will not discuss Sun's API for XML in detail (which contains a fair number of classes and is likely to change significantly in the near future). However, a brief introduction to the XML classes used in this example is appropriate.

Note: Sun's early access XML library is required to run this example. See http://www.javasoft.com/.

17.6.1 XmlDocument

class com.sun.xml.tree.XmlDocument

This class represents a top level XML 1.0 document created with the XML parser built into `XmlDocumentBuilder`. The `getDocumentElement()` method retrieves the top-most node in the document (i.e. the root node). All other nodes can be extracted through this node.

17.6.2 XmlDocumentBuilder

class com.sun.xml.tree.XmlDocumentBuilder

This class builds an XML document using an internal parser. Two overloaded methods, `createXmlDocument(InputStream in)` and `createXmlDocument(String documentURL)`, create new `XmlDocument` instances. (We will use the latter method in this example.)

17.6.3 DataNode

class com.sun.xml.tree.DataNode

This class represents a node in an XML document tree that encapsulates data and has no child nodes (i.e. a leaf node). Method `getData()` retrieves the contained data as a `String`.

17.6.4 ElementNode

class com.sun.xml.tree.ElementNode

This class encapsulates an element in the XML document tree, which is a node with children (i.e. a non-leaf node). The `getLength()` method retrieves the number of child nodes, and `item(int index)` returns the node with the given index.

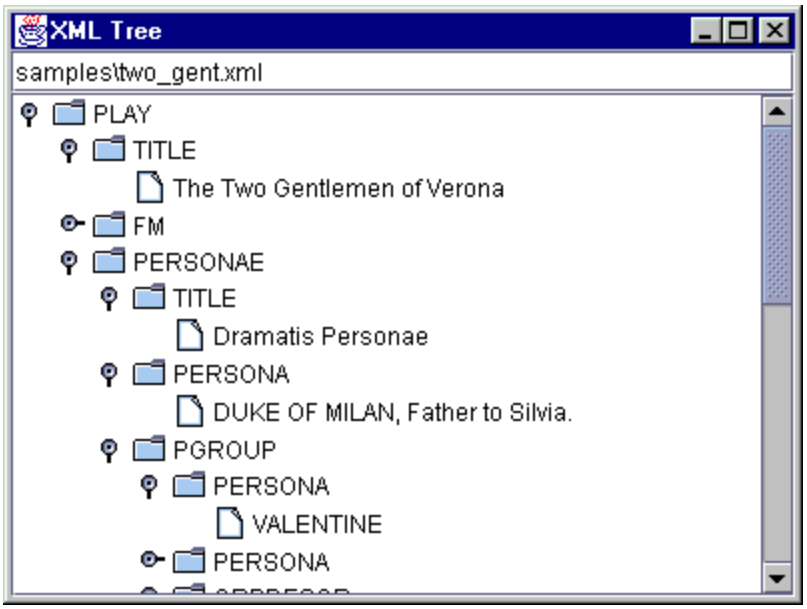


Figure 17.6 An XML document structure tree.

<<file figure17-6.gif>>

The Code: XmlTree.java

see \Chapter17\6

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.net.*;

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;

import com.sun.xml.tree.*;
import com.sun.xml.parser.*;
import org.w3c.dom.*;

public class XmlTree extends JFrame
{
  protected JTree  m_tree;
  protected DefaultTreeModel m_model;
  protected JTextField m_location;

  public XmlTree() {
    super("XML Tree");
    setSize(400, 300);

    m_location = new JTextField();
    m_location.setText("samples\\book-order.xml");
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        readXml(m_location.getText());
      }
    };
    m_location.addActionListener(lst);
    getContentPane().add(m_location, BorderLayout.NORTH);

    DefaultMutableTreeNode top = new DefaultMutableTreeNode(
```

```
      "Empty");
    m_model = new DefaultTreeModel(top);
    m_tree = new JTree(m_model);

    m_tree.getSelectionModel().setSelectionMode(
      TreeSelectionModel.SINGLE_TREE_SELECTION);
    m_tree.setShowsRootHandles(true);
      m_tree.setEditable(false);

    JScrollPane s = new JScrollPane();
    s.getViewport().add(m_tree);
    getContentPane().add(s, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public void readXml(String sUrl) {
    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    Thread runner = new Thread() {
      public void run () {
        try {
          URL source;
          try {
            File f = new File(sUrl);
            source = f.toURL();
          }
          catch (Exception ex) {
            source = new URL(sUrl);
          }

          XmlDocument doc =
            XmlDocumentBuilder.createXmlDocument(
          source.toString());

          ElementNode root =
            (ElementNode)doc.getDocumentElement();
          root.normalize();

          DefaultMutableTreeNode top = createTreeNode(root);
          m_model.setRoot(top);
          m_tree.treeDidChange();
        }
        catch (Exception ex) {
          ex.printStackTrace();
          JOptionPane.showMessageDialog(this,
            ex.toString(), "Warning",
            JOptionPane.WARNING_MESSAGE);
        }
        setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
      }
    }
  }

  protected DefaultMutableTreeNode createTreeNode(ElementNode root) {
    DefaultMutableTreeNode node = new DefaultMutableTreeNode(
```

```
      root.getNodeName());
    for (int k=0; k<root.getLength(); k++) {
      Node nd = root.item(k);
      if (nd instanceof DataNode) {
        DataNode dn = (DataNode)nd;
        String data = dn.getData().trim();
        if (data.equals("\n") || data.equals("\r\n"))
          data = "";
        if (data.length() > 0)
          node.add(new DefaultMutableTreeNode(data));
      }
      else if (nd instanceof ElementNode) {
        ElementNode en = (ElementNode)nd;
        node.add(createTreeNode(en));
      }
    }
    return node;
  }

  public static void main(String argv[]) {
    new XmlTree();
  }
}
```

Understanding the Code

Class XmlTree

Instance variables:

- `JTree m_tree`: used to display an XML document.
- `DefaultTreeModel m_model`: used to store the content of an XML document.
- `JTextField m_location`: used for entry of a file name or URL location of an XML document.

Initially the `JTree` component receives a single node, "Empty". An `ActionListener` is added to the `m_location` textfield which calls our `readXml()` method passing it the current text.

The `readXml()` method loads an XML document, corresponding to the `String` passed as parameter, into our tree model. The body of this method is placed in a separate thread because it can be a very expensive procedure, and we want to make sure not to clog up the event dispatching thread (to retain GUI responsiveness). First the given string is treated as a file name. A `File` instance is created and converted to a `URL`. If this does not succeed, the string is treated as a URL address:

```
          URL source;
          try {
            File f = new File(sUrl);
            source = f.toURL();
          }
          catch (Exception ex) {
            source = new URL(sUrl);
          }
```

The static method `XmlDocumentBuilder.createXmlDocument()` creates an `XmlDocument` corresponding to the resulting `URL`. As soon as this finishes (which may take a while for large documents), the root `ElementNode` is retrieved from that document:

```
          XmlDocument doc =
            XmlDocumentBuilder.createXmlDocument(
```

```
            source.toString());

            ElementNode root =
              (ElementNode)doc.getDocumentElement();
            root.normalize();
```

We then transform our XML document into a structure suitable for addition to our Swing tree model. Our `createTreeNode()` method does this job, returning the top-most node (the root node) as a `DefaultMutableTreeNode`. Finally that node is set as a root of our tree model, and our tree component is notified that it's content has changed:

```
            DefaultMutableTreeNode top = createTreeNode(root);
            m_model.setRoot(top);
            m_tree.treeDidChange();
```

Note: Normally we are expected to avoid calling the `treeDidChange()` method directly, as it should be called by our UI as needed. However, in this case our `JTree` will not update correctly without it.

The `createTreeNode()` method creates a `DefaultMutableTreeNode` from an `ElementNode` provided. First it creates a root node corresponding to the given `ElementNode`, and then all lower level nodes are retrieved and processed in turn to populate the whole tree:

```
    DefaultMutableTreeNode node = new DefaultMutableTreeNode(
      root.getNodeName());
    for (int k=0; k<root.getLength(); k++) {
      Node nd = root.item(k);
```

Two possibilities are reconciled during this procedure. If a newly processed node is an instance of `DataNode`, its text is retrieved and is used as the user data object for a new `DefaultMutableTreeNode` which is then added to the parent node (in this case care should be taken to avoid empty nodes containing only end-of-line symbols). Otherwise, if a newly processed node is an instance of `ElementNode`, method `createTreeNode()` is called recursively.

Running the Code

Figure 17.6 shows our XML Tree example displaying the contents of "The Two Gentlemen of Verona" XML document which can be found at: ftp://sunsite.unc.edu/pub/sun-info/standards/xml/eg/.

17.7 Custom editors and renderers

In this section we'll construct a simple family tree application. We will show how to use a custom cell editor for name entry, as well as a custom cell renderer which displays an icon corresponding to a node's data rather than its state. This example allows dynamic node insertion, and each node can have no more than two children.

Our representation of an ancestor tree is structured differently than how we normally think of structuring trees, even though, technically speaking, both methods are equivalent. Our root tree node represents a child, and child tree nodes represent parents, grandparents, etc., of that child. So a parent node in this `JTree` actually corresponds to a child in the family ancestry. This illustrates that `JTree` is flexible enough to allow adaptation to any type of heirarchical data set, including a dynamically changing one (as we also saw in our file directories tree examples above).

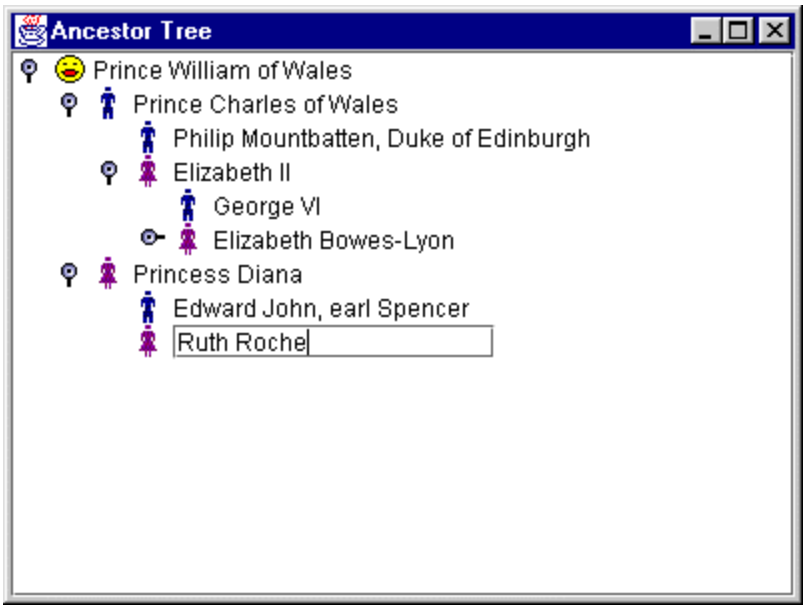


Figure 17.7 JTree with custom editor and cellrenderer enforcing nodes with two children.

<<file figure17-7.gif>>

The Code: AncestorTree.java

see \Chapter17\6

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.tree.*;

public class AncestorTree extends JFrame
{
  public static ImageIcon ICON_SELF =
    new ImageIcon("myself.gif");
  public static ImageIcon ICON_MALE =
    new ImageIcon("male.gif");
  public static ImageIcon ICON_FEMALE =
    new ImageIcon("female.gif");

  protected JTree  m_tree;
  protected DefaultTreeModel m_model;
  protected IconCellRenderer m_renderer;
  protected IconCellEditor m_editor;

  public AncestorTree() {
    super("Ancestor Tree");
    setSize(500, 400);

    DefaultMutableTreeNode top = new DefaultMutableTreeNode(
      new IconData(ICON_SELF, "Myself"));
    addAncestors(top);
    m_model = new DefaultTreeModel(top);
    m_tree = new JTree(m_model);
    m_tree.getSelectionModel().setSelectionMode(
      TreeSelectionModel.SINGLE_TREE_SELECTION);
    m_tree.setShowsRootHandles(true);
```

```
    m_tree.setEditable(true);

    m_renderer = new IconCellRenderer();
    m_tree.setCellRenderer(m_renderer);
    m_editor = new IconCellEditor(m_tree);
    m_tree.setCellEditor(m_editor);
    m_tree.setInvokesStopCellEditing(true);

    m_tree.addMouseListener(new TreeExpander());

    JScrollPane s = new JScrollPane();
    s.getViewport().add(m_tree);
    getContentPane().add(s, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public boolean addAncestors(DefaultMutableTreeNode node) {
    if (node.getChildCount() > 0)
      return false;

    Object obj = node.getUserObject();
    if (obj == null)
      return false;
    node.add(new DefaultMutableTreeNode( new IconData(
      ICON_MALE, "Father of: "+obj.toString()) ));
    node.add(new DefaultMutableTreeNode( new IconData(
      ICON_FEMALE, "Mother of: "+obj.toString()) ));
    return true;
  }

  public static void main(String argv[]) { new AncestorTree(); }

  class TreeExpander extends MouseAdapter
  {
    public void mouseClicked(MouseEvent e) {
      if (e.getClickCount() == 2) {
        TreePath selPath = m_tree.getPathForLocation(
          e.getX(), e.getY());
        if (selPath == null)
          return;
        DefaultMutableTreeNode node =
          (DefaultMutableTreeNode)(selPath.
             getLastPathComponent());
        if (node!=null && addAncestors(node)) {
          m_tree.expandPath(selPath);
          m_tree.repaint();
        }
      }
    }
  }
}

// Classes IconCellRenderer and IconData are
// unchanged from previous examples, and are
```

```
// not listed here to conserve space.

class IconCellEditor extends JLabel
 implements TreeCellEditor, ActionListener
{
  protected JTree m_tree = null;
  protected JTextField m_editor = null;
  protected IconData m_item = null;
  protected int m_lastRow = -1;
  protected long m_lastClick = 0;
  protected Vector m_listeners = null;

  public IconCellEditor(JTree tree) {
    super();
    m_tree = tree;
    m_listeners = new Vector();
  }

  public Component getTreeCellEditorComponent(JTree tree,
   Object value, boolean isSelected, boolean expanded,
   boolean leaf, int row)
  {
    if (value instanceof DefaultMutableTreeNode) {
      DefaultMutableTreeNode node =
        (DefaultMutableTreeNode)value;
      Object obj = node.getUserObject();
      if (obj instanceof IconData) {
        IconData idata = (IconData)obj;
        m_item = idata;
        // Reserve some more space...
        setText(idata.toString()+"     ");
        setIcon(idata.m_icon);
        setFont(tree.getFont());
        return this;
      }
    }
    // We don't support other objects...
    return null;
  }

  public Object getCellEditorValue() {
    if (m_item != null && m_editor != null)
      m_item.m_data = m_editor.getText();
    return m_item;
  }

  public boolean isCellEditable(EventObject evt) {
    if (evt instanceof MouseEvent) {
      MouseEvent mEvt = (MouseEvent)evt;
      if (mEvt.getClickCount() == 1) {
        int row = m_tree.getRowForLocation(mEvt.getX(), mEvt.getY());
        if (row != m_lastRow) {
          m_lastRow = row;
          m_lastClick = System.currentTimeMillis();
          return false;
        }
        else if (System.currentTimeMillis()-m_lastClick > 1000)
        {
          m_lastRow = -1;
          m_lastClick = 0;
          prepareEditor();
          mEvt.consume();
          return true;
```

```
      }
      else
        return false;
    }
  }
  return false;
}

protected void prepareEditor() {
  if (m_item == null)
    return;
  String str = m_item.toString();

  m_editor = new JTextField(str);
  m_editor.addActionListener(this);
  m_editor.selectAll();
  m_editor.setFont(m_tree.getFont());

  add(m_editor);
  revalidate();

  TreePath path = m_tree.getPathForRow(m_lastRow);
  m_tree.startEditingAtPath(path);
}

protected void removeEditor() {
  if (m_editor != null) {
    remove(m_editor);
    m_editor.setVisible(false);
    m_editor = null;
    m_item = null;
  }
}

public void doLayout() {
  super.doLayout();
  if (m_editor != null) {
    int offset = getIconTextGap();
    if (getIcon() != null)
      offset += getIcon().getIconWidth();
    Dimension cSize = getSize();
    m_editor.setBounds(offset, 0, cSize.width - offset,
      cSize.height);
  }
}

public boolean shouldSelectCell(EventObject evt) { return true; }

public boolean stopCellEditing() {
  if (m_item != null)
    m_item.m_data = m_editor.getText();

  ChangeEvent e = new ChangeEvent(this);
  for (int k=0; k<m_listeners.size(); k++) {
    CellEditorListener l = (CellEditorListener)m_listeners.
      elementAt(k);
    l.editingStopped(e);
  }
  removeEditor();
  return true;
}

public void cancelCellEditing() {
```

```
    ChangeEvent e = new ChangeEvent(this);
    for (int k=0; k<m_listeners.size(); k++) {
      CellEditorListener l = (CellEditorListener)m_listeners.
        elementAt(k);
      l.editingCanceled(e);
    }
    removeEditor();
  }

  public void addCellEditorListener(CellEditorListener l) {
    m_listeners.addElement(l);
  }

  public void removeCellEditorListener(CellEditorListener l) {
    m_listeners.removeElement(l);
  }

  public void actionPerformed(ActionEvent e) {
    stopCellEditing();
    m_tree.stopEditing();
  }
}
```

Understanding the Code

Class AncestorTree

Class `AncestorTree` declares and creates three static images representing male and female ancestors, and a root (representing a child whose ancestry is being represented).

Instance variables:

- `JTree m_tree`: ancestors tree component.
- `DefaultTreeModel m_model`: ancestor tree data model.
- `IconCellRenderer m_renderer`: ancestor tree cell renderer.
- `IconCellEditor m_editor`: ancestor tree cell editor.

The `AncestorTree` constructor is similar to the `Tree1` constructor from section 17.2. However, there are some important differences:

- `addAncestors()` method (see below) is called to create initial child nodes.
- The `editable` property is set to `true`.
- Instances of our custom `IconCellRenderer` and `IconCellEditor` classes are set as the renderer and editor for `m_tree`, respectively.
- An instance of our custom `TreeExpander` class is attached as a `MouseListener` to our `m_tree`.

The `addAncestors()` method adds male and female ancestors to a given node as child nodes (representing parents in an ancestry), if this hasn't been done already. Instances of `IconData` (which hold a combination of icons and text, see previous examples) are added as user objects for a newly created node. The initial text of each `IconData` object is assigned as "Father of: <node text>" and "Mother of: <node text>", and appropriate icons are used to distinguish between women and men.

Class AncestorTree.TreeExpander

The `TreeExpander` inner class extends `MouseAdapter` and is used to insert ancestor nodes when a double

click occurs on a node. The most significant aspect of this listener is the call to the `JTree.getPathForLocation()` method, which retrieves the currently selected tree path and allows us to determine the selected node. The `addAncestors()` method is then called on that node, and our tree is repainted to show the newly added nodes (if any).

Class IconCellRenderer and class IconData

Class `IconCellRenderer`, as well as class `IconData`, have recieved no changes from section 17.3. Refer back to this section for more information about their inner workings.

Class IconCellEditor

This class implements the `TreeCellEditor` and `ActionListener` interfaces, and creates a `JTextField` for editing a node's text in-place. This editor is designed in such a way that a cell icon remains in unchanged and in the same location whether in editing mode or not. This explains why `IconCellEditor` extends `JLabel`, and not `JTextField` as we might expect. The underlying `JLabel` component is used to render the icon and reserve any ecessary space. The `JTextField` component is created dynamically and placed above the `JLabel`'s text portion to perform the actual editing. Note that the `JLabel` serves as a `Container` for the `JTextField` component (recall that all Swing components extend the `java.awt.Container` class).

Note: Swing's default editor `DefaultTreeCellEditor` is used similarly, but in a more complex manner. It takes a `DefaultTreeCellRenderer` as parameter and uses it to determine the size of a node's icon. Then it uses a custom inner class as a container which renders the icon and positions the text field. For more details see `DefaultTreeCellEditor.java`.

Instance variables:

- `JTree m_tree`: reference to the parent tree component (must be passed to constructor).
- `JTextField m_editor`: editing component.
- `IconData m_item`: data object to be edited (we limit ourselves to `IconData` instances).
- `int m_lastRow`: tree row where the most recent mouse click occured.
- `long m_lastClick`: time (in ms) that the last mouse click occurred.
- `Vector m_listeners`: collection of `CellEditorListeners`, which we must manage according to the `TreeCellEditor` interface.

The `getTreeCellEditorComponent()` method will be called to initialize a new data object in the editor before we begin editing a node. (Note that the user object from the selected node must be an instance of `IconData`, or editing is not allowed.) The icon and text from the `IconData` object are assigned using inherited `JLabel` functionality. A few spaces are intentionally appended to the end of the text to provide some white space between the icon and the editing area (a very simply way to accomplish this).

The `getCellEditorValue()` method returns the current value of the editor as an `Object`. In our case we return `m_item`, adjusting its `m_data` to `m_editor`'s text:

```
if (m_item != null && m_editor != null)
  m_item.m_data = m_editor.getText();
return m_item;
```

Method `isCellEditable()` is called to determine whether an editor should enter editing mode. It takes an `EventObject` as parameter. In this way, we can use any user activity resulting in an event as a signal for editing, e.g. double mouse click, specific key press combination, etc. We've implemented this method to start

editing mode when two single mouse clicks occur on a cell separated by no less than 1 second (1000 ms). To do this we first filter only single mouse click events:

```
if (evt instanceof MouseEvent) {
  MouseEvent mEvt = (MouseEvent)evt;
  if (mEvt.getClickCount() == 1) {
```

For these events we determine the tree's row that was clicked and store it in our `m_lastRow` instance variable. The current system time is also saved in the `m_lastClick` instance variable. If another click occurs on the same cell after 1000 ms, our custom `prepareEditor()` method is called to prepare the editor for editing, and returns `true`. Otherwise `false` is returned.

Method `shouldSelectCell()` always returns `true` to indicate that a cell's content should be selected at the beginning of editing.

Method `stopCellEditing()` is called to stop editing and store the result of the edit. We simply change `m_data` in the `m_item` object to `m_editor`'s text.

```
if (m_item != null)
  m_item.m_data = m_editor.getText();
```

The `m_item` object also has a reference to our tree's model, so it will affect our tree directly. Thus all registered `CellEditorListeners` are notified in turn by calling their `editingStopped()` method. Finally the editing process is terminated by calling our custom `removeEditor()` method.

The `cancelCellEditing()` method is called to stop editing without storing the result of the edit. Similar to `stopCellEditing()`, all registered `CellEditorListeners` are notified in turn by calling their `editingCanceled()`, and the editing process is terminated by calling `removeEditor()`.

Two methods, `addCellEditorListener()` and `removeCellEditorListener()`, add and remove listeners to/from our `m_listeners` vector.

The `prepareEditor()` method actually starts the editing process. It creates a `JTextField` component and sets its initial text to that of the `m_item` object:

```
m_editor = new JTextField(str);
m_editor.addActionListener(this);
m_editor.selectAll();
m_editor.setFont(m_tree.getFont());

add(m_editor);
revalidate();
```

An `ActionListener` is added to the textfield to enable the ability to stop editing when the user presses the "Enter" key (recall that this class implements `ActionListener`, so we provide a `this` reference for the listener). The most important aspect of this method is the fact that a `JTextField` is added to our base component, and the overridden `doLayout()` method is invoked indirectly (through `revalidate()`) to assign the correct size and position to our editor component. Finally, the base tree is notified by calling our `startEditingAtPath()` method, to allow editing.

The `removeEditor()` method is called to quit the editing process. It removes the editing component from the container, hides and destroys it (dereferences it to allow garbage collection):

```
remove(m_editor);
m_editor.setVisible(false);
m_editor = null;
```

```
        m_item = null;
```

Method `doLayout()` overrides `Component.doLayout()` to set the correct size and location for the editing component.

Method `actionPerformed()` will be called when "Enter" is pressed during editing. It directly calls our `stopCellEditing()` implementation, and notifies our tree by calling `stopEditing()`.

Running the Code

Perform a single click on a tree cell, wait about a second, then click it again to enter editing mode. Note that there is no limit to how far back we can go with this ancestor tree, and all nodes can have either no children, or two children (representing the parents of the individual represented by that node). Try creating your own ancestor tree as far back as you can go (if you end with monkey or gorilla please contact us). Figure 17.7 shows the ancestor tree of Prince William of Wales, the oldest son of the heir of the British monarchy.

UI Guideline: Family Trees and Organisation Charts

The family tree given here is used as an example only. There is still considerable debate in the UI Design field as to whether Tree component is appropriate for displaying and manipulating such data. Generally, Family Trees or Organisation Charts are displayed using a top-down (horizontal orientation), evenly distributed graph. Therefore, the Tree component view with its Left-Right (vertical orientation) is alien for this type of data.

If your User community is particularly technical then you should have no difficulties, however, consider carefully before selecting tree component for a wider user group, in this instance.

You may also like to consider that such a tree component could be used a prototype or "proof of concept". You could later replace the Tree component with an OrganisationChart component (for example) which re-uses the same TableModel interface. Thus the actual domain data and model classes would not need to be changed. The ability to do this, demonstrates the power of the Swing MVC architecture.

Chapter 18. Tables

In this chapter:

- JTable
- StocksTable: part I - Basic JTable example
- StocksTable: part II - Custom renderers
- StocksTable: part III - Data formatting
- StocksTable: part IV - Sorting columns
- StocksTable: part V - JDBC
- StocksTable: part VI - Column addition and removal
- Expense report application
- JavaBeans property editor

18.1 JTable

JTable is extremely useful for the display, navigation, and editing of tabular data. Because of its complex nature, JTable has a whole package devoted just to it: javax.swing.table. This package consists of a set of classes and interfaces which we will review briefly here. In the examples that follow, we construct--in a step-wise fashion--a table-based application used to display stock market data. (In chapters 22 and 26 we enhance this application further to allow printing and print preview, as well as CORBA client-server interaction.) This chapter concludes with an expense report application demonstrating the use of different components as table cell editors and renderers, and the completion of the JavaBeans property editor we started to build in chapter 4.

18.1.1 JTable

class javax.swing.JTable

This class represents Swing's table component and provides a rich API for managing its behavior and appearance. JTable directly extends JComponent and implements the TableModelListener, TableColumnModelListener, ListSelectionListener, CellEditorListener, and the Scrollable interfaces (it is meant to be placed in a JScrollPane). Each JTable has three models: a TableModel, TableColumnModel, and ListSelectionModel. All table data is stored in a TableModel, normally in a 2-dimensional structure such as a 2D array or a Vector of Vectors. TableModel implementations specify how this data is stored, as well as manage the addition, manipulation, and retrieval of this data. TableModel also plays a role in dictating whether or not specific cells can be edited, as well as the data type of each column of data. The location of data in a JTable's TableModel does not directly correspond to the location of that data as displayed by JTable itself. This part is controlled at the lowest level by TableColumnModel.

A TableColumnModel is designed to maintain instances of TableColumn, each of which represents a single column of TableModel data. TableColumn is the class that is responsible for managing the display of a column in the actual JTable GUI. Each TableColumn has an associated cell renderer, cell editor, table header, and a cell renderer for the table header. When a JTable is placed in a JScrollPane these headers are placed in the scroll pane's COLUMN_HEADER viewport and can be dragged and resized to reorder and change the size of columns. A TableColumn's header renderer is responsible for returning a component used to render the column header, and the cell renderer is responsible for returning a component used to render each cell. As with JList and JTree renderers, these renderers also act as "rubber stamps"[API] and are not at all interactive. The component returned by the cell editor, however, is completely interactive. Cell renderers are instances of TableCellRenderer and cell editors are instances of CellEditor. If none are explicitly assigned, default versions will be used based on the Class type of the corresponding TableModel column data.

TableColumnModel's job is to manage all TableColumns, providing control over order, column selections, and margin size. To support several different modes of selection, TableColumnModel maintains a ListSelectionModel which, as we learned in chapter 10, allows single, single-interval, and multiple-interval selections. JTable takes this flexibility even further by providing functionality to customize any row, column, and/or cell-specific selection schemes we can come up with.

We can specify one of several resizing policies which dictate how columns react when another column is resized, as well as whether or not grid lines between rows and/or columns should appear, margin sizes between rows and columns, selected and unselected cell foreground and background colors, the hieght of rows, and the width of each column on a column-by-column basis.

With tables come two new kinds of events in Swing: TableModelEvent and TableColumnModelEvent. Regular Java events apply to JTable as well. For instance, we can use MouseListeners to process double

mouse clicks. ChangeEvents and ListSelectionEvents are also used for communication in TableColumnModel.

Note: Although JTable implements several listener interfaces, it does not provide any methods to register listeners other than those inherited from JComponent. To attach listeners for detecting any of the above events we must first retrieve the appropriate model.

A number of constructors are provided for building a JTable component. We can use the default constructor or pass the table's data and column names each as a separate Vector. We can build an empty JTable with a specified number of rows and columns. We can also pass table data to the constructor as a two-dimensional array of data Objects along with an Object array of column names. Other constructors allow creation of a JTable with specific models. In all cases, if a specific model is not assigned in the constructor, JTable will create default implementations with its protected createDefaultColumnModel(), createDefaultDataModel(), and createDefaultSelectionModel() methods. It will do the same for each TableColumn renderer and editor, as well as its JTableHeader, using createDefaultEditors(), createDefaultRenderers(), and createDefaultTableHeader().

JTable is one of the most complex Swing components and keeping track of its constituents and how they interact is intially a challenge. Before we begin the step-wise construction of our stocks table application, we must make our way through all of these details. The remainder of this section is devoted to a discussion of the classes and interfaces that underly JTable.

18.1.2 The TableModel interface

abstract interface javax.swing.table.TableModel

Instances of TableModel are responsible for storing a table's data in a 2-dimensional structure such as a 2-dimensional array or a vector of vectors. A set of methods is declared for use in retrieving data from a table's cells. The getValueAt() method should retrieve data from a given row and column index as an Object, and setValueAt() should assign the provided data object to the specified location (if valid). getColumnClass() should return the Class describing the data objects stored in the specified column (used to assign a default renderer and editor for that column), and getColumnName() should return the String name associated with the specified column (often used for that column's header). The getColumnCount() and getRowCount() methods should return the number of contained columns and rows respectively.

Note: The getRowCount() is called frequently by JTable for display purposes and should be designed with efficiency in mind because of this.

The isCellEditable() method should return true if the cell at the given row and column index can be edited. The setValueAt() method should be designed so that if isCellEditable() returns false, the object at the given location will not be updated.

This model supports the attachment of TableModelListeners (see below) which should be notified about changes to this model's data. As expected, methods for adding and removing these listeners are provided, addTableModelListener() and removeTableModelListener(), and implementations are responsible for dispatching TableModelEvents to those registered.

Each JTable uses one TableModel instance which can be assigned/retrieved using JTable's setModel() and getModel() methods respectively.

18.1.3 AbstractTableModel

abstract class javax.swing.table.AbstractTableModel

`AbstractTableModel` is an abstract class implementing the `TableModel` interface. It provides default code for firing `TableModelEvents` with the `fireTableRowsDeleted()`, `fireTableCellUpdated()`, and `fireTableChanged()` methods. It also manages all registered `TableModelListeners` in an `EventListenerList` (see chapter 2).

The `findColumn()` method searches for the index of a column with the given `String` name. This search is performed in a linear fashion (referred to as "naive" in the documentation) and should be overriden for large table models for more efficient searching.

Three methods need to be implemented in concrete sub-classes: `getRowCount()`, `getColumnCount()` and `getValueAt(int row, int column)`, and we are expected to use this class as a base for building our own `TableModel` implementations (rather than `DefaultTableModel`--see below).

18.1.4 DefaultTableModel

class javax.swing.table.DefaultTableModel

`DefaultTableModel` is the default concrete `TableModel` implementation used by `JTable` when no model is specified in the constructor. It uses a `Vector` of `Vectors` to manage its data, which is one major reason why extending `AbstractTableModel` is often more desirable (because `AbstractTableModel` allows complete control over how data storage and manipulation is implemented). This `Vector` can be assigned with the overloaded `setDataVector()` method, and retrieved with the `getDataVector()` method. Internally, two overloaded, protected `convertToVector()` methods are used for converting `Object` arrays to `Vectors` when inserting rows, columns, or assigning a new data `Vector`. Methods for adding, inserting, removing, and moving columns and rows of data are also provided.

Note: The position of a row or column in the model does not correspond to `JTable`'s GUI representation of that row or column. Rather, this representation is performed instances of `TableColumn` which map to specific model columns. When a `TableColumn` is moved in the GUI, the associated data in the `TableModel` model stays put, and vice versa (see below).

Along with the `TableModelEvent` functionality inherited from `AbstractTableModel`, this class implements three new event-dispatching methods, each taking a `TableModelEvent` as parameter: `newDataAvailable()`, `newRowsAdded()`, and `rowsRemoved()`. The `newRowsAdded()` method ensures that new rows (see discussion of `TableModelEvent` below) have the correct number of columns by either removing excess elements, or using `null` for each missing cell. If `null` is passed to any of these methods they will construct and fire a default `TableModelEvent` which assumes that all table model data has changed.

18.1.5 TableColumn

class javax.swing.table.TableColumn

`TableColumn` is the basic building block of `JTable`'s visual representation, and provides the main link between the `JTable` GUI and its model. Note that `TableColumn` does not extend `java.awt.Component`, and is thus not a component. Rather, it acts more like a model that maintains all the properties of a column displayed in a `JTable`. An instance of `TableColumn` represents a specific column of data stored in a `TableModel`. `TableColumn` maintains the index of the `TableModel` column it represents as property `modelIndex`. We can get/set this index with the `getModelIndex()` and `setModelIndex()` methods. It

is important to remember that the position of a TableColumn graphically in JTable does not at all correspond to the corresponding TableModel column index.

A TableColumn is represented graphically by a column header renderer, cell renderer, and optionally a cell editor. The renderers must be instances of TableCellRenderer, and the editor must be an instance of TabeCellEditor (see below). A column's header is rendered by a component stored as the headerRenderer property. By default this is an instance of DefaultTableCellRenderer (a JLabel with a beveled border--see below) and is created with TableColumn's protected createDefaultHeaderRenderer() method. This renderer simply renders the String returned by the toString() method of the Object referred to by the headerValue property. The header renderer and value can be assigned/retrieved with setHeaderRenderer()/getHeaderRenderer() and setHeaderValue()/getHeaderValue() methods respectively. Often headerValue directly corresponds to the column name retrived using TableModel's getColumnName() method. If headerValue is not explicitly set it defaults to null.

The column cell renderer and editor also default to null, and unless they are explicitly specified using setCellRenderer() or setCellEditor(), are automatically assigned based on the Class type of the data stored in the associated column in the TableModel (retreived using TableModel's getColumnClass() method). Explicity specified renderers and editors are referred to as column-based, whereas those determined by data type are referred to as class-based (we will discuss renderers and editors in more detail below).

Each TableColumn has an identifier property which also defaults null. This property can be assigned/retrieved using typical set/get accessors, and the getIdentifier() method will return the headerValue property if identifier is null. When searching for a TableColumn by name (using TableColumnModel's getColumnIndex() method--see below--or JTable's getColumn() method), the given Object will be compared, using Object's equals() method, to each TableColumn identifier. Since it is possible that more than one TableColumn will use the same identifier, the first match is returned as the answer.

TableColumn maintains properties minWidth, maxWidth, and width. The first two specify the minimum and maximum allowable widths for column rendering, and the width property stores the current width. Each can be retrieved and assigned with typical get/set methods: getMinWidth()/setMinWidth(), getMaxWith()/setMaxWidth(), and getWidth()/setWidth(). minWidth defaults to 15, maxWidth defaults to Integer.MAX_VALUE, and width defaults to 75. When a JTable is resized it will try to maintain its width, and will never exceeed its maximum or shrink smaller than its minimum.

Note: All other visual aspects of each column are controlled by either JTable or TableColumnModel (see below).

TableColumn also maintains an isResizable property, specifying whether or not its width can be changed by the user (this does not apply to programmatic calls to setWidth()). (We will discuss resizing in more detail below).

An interesting and rarely used property maintained by TableColumn, resizedPostingDisabledCount, is used to enable/disable the posting of PropertyChangeEvents when a TableColumn's width changes. This is an int value that is incremented on each call to disableResizedPosting(), and decremented on each call to enableResizedPosting(). Events will only be fired if this value is less than or equal to 0. The logic behind this is that if two separate sources both disable resize event posting, then two calls should be required to re-enable it.

Big Alert! As of Java 2 FCS the `resizedPostingDisabledCount` property is not actually used anywhere and does not play a role in `PropertyChangeEvent` firing.

`TableColumn` fires `PropertyChangeEvents` when any of the `width`, `cellRenderer`, `headerRenderer`, or `headerValue` bound properties change. Thus we can add and remove `PropertyChangeListeners` to be notified of these changes. The corresponding property names are `COLUMN_WIDTH_PROPERTY`, `COLUMN_RENDERER_PROPERTY`, `HEADER_RENDERER_PROPERTY`, and `HEADER_VALUE_PROPERTY`.

18.1.6 The TableColumnModel interface

abstract interface javax.swing.table.TableColumnModel

This model is designed to maintain a `JTable`'s `TableColumns`, and provides control over column selections and margin size. `TableColumnModel` controls how `JTable` displays its `TableModel` data. The `addColumn()` method should append a given `TableColumn` to the end of the structure used to maintain them (usually a `Vector`), `removeColumn()` should remove a given `TableColumn`, and `moveColumn()` should change the location of a given `TableColumn` within that structure.

Note: When creating a `JTable`, if no `TableColumnModel` is specified, one will automatically be constructed for us containing `TableColumns` displaying `TableModel` data in the same order it appears in the model. This will only occur of `JTable`'s `autoCreateColumnsFromModel` property is set `true`, which it is by default. This is very helpful, but has the often undesirable side-effect of completely rebuilding the `TableColumnModel` whenever `TableModel` changes. Thus it is common to set this property to `false` once a `JTable` has been created or after a new `TableModel` is assigned.

Unlike the location of a column in `TableModel` implementations, the index of a `TableColumn` in a `TableColumnModel`'s storage structure directly corresponds to its position in the `JTable` GUI. Note that the `moveColumn()` method is called whenever the user drags a column to a new position.

The `getColumnCount()` method should return the number of `TableColumns` currently maintained, `getColumns()` should return an `Enumeration` of all contained `TableColumns`, and `getColumn()` returns the `TableColumn` at the given index. The `getColumnIndex()` method should return the index of the `TableColumn` whose `identifier` property is equal to (using `Object`'s `equals()` method) the given `Object`. `getColumnIndexAtX()` should return the index of the `TableColumn` at the given x-corrdinate in table-space (if `getColumnIndexAtX()` is passed a coordinate that maps to the margin space between adjacent columns, or any x-coordinate that does not correspond to a table column, it will return -1). `setColumnMargin()` and `getColumnMargin()` should allow assignment and retrieval of an `int` value for use as the margin space on each side of each table column. The `getTotalColumnWidth()` should return the sum of the current width of all `TableColumns` including all margin space.

Note: The margin size does not correspond to the width of the separating grid lines between columns in `JTable`. In fact, the width of these lines is always 1, and cannot be changed without customizing `JTable`'s UI delegate.

`TableColumnModel` declares methods for controlling the selection of its `TableColumns`, and allows assignment and retrieval of a `ListSelectionModel` implementation used to store information about the current column selection with the methods `setSelectionModel()` and `getSelectionModel()`. The `setColumnSelectionAllowed()` method is intended to turn on/off column selection capabilities, and `getColumnSelectionAllowed()` should return a `boolean` specifying whether slection is currently allowed or not. For convenience, `JTable`'s `setColumnSelectionAllowed()` method delegates its traffic to the method of the same signature in this interface.

TableColumnModel also declares support for TableColumnModelListeners (see below). TableColumnModel implementations are expected to fire a TableColumnModelEvent whenever a TableColumn is added, removed, or moved, a ChangeEvent whenever margin size is changed, and a ListSelectionEvent whenever a change in column selection occurs.

18.1.7 DefaultTableColumnModel

class javax.swing.table.DefaultTableColumnModel

This class is the concrete default implementation of the TableColumnModel interface used by JTable when none is specifically assigned or provided at construction time. All TableColumnModel methods are implemented as expected (see above), and the following protected methods are provided to fire events: fireColumnAdded(), fireColumnRemoved(), fireColumnMoved(), fireColumnSelectionChanged(), and fireColumnMarginChanged(). A valueChanged() method is provided to listen for column selection changes and fire a ListSelectionEvent when necessary. And a propertyChanged() method is used to update the totalColumnWidth property when the width of a contained TableColumn changes.

18.1.8 The TableCellRenderer interface

abstract interface javax.swing.table.TableCellRenderer

This interface describes the renderer used to display cell data in a TableColumn. Each TableColumn has an associated TableCellRender which can be assigned/retrieved with the setCellRenderer()/getCellRenderer() methods. The getTableCellRendererComponent() method is the only method declared by this interface, and is expected to return a Component that will be used to actually render a cell. It takes the following parameters:

- JTable table: the table instance containing the cell to be rendered.
- Object value: the value used to represent the data in the given cell.
- boolean isSelected: whether or not the given cell is selected.
- boolean hasFocus: whether or not the given cell has the focus (true if it was clicked last).
- int row: can be used to return a renderer component specific to row or cell.
- int column: can be used to return a renderer component specific to column or cell.

We are expected to customize or vary the returned component based on the given parameters. For instance, given a value that is an instance of Color, we might return a special JLabel subclass that paints a rectangle in the given color. This method can be used to return different renderer components on a column, row, or cell-specific basis, and is similar to JTree's TreeCellRenderer getTreeCellRendererComponent() method. As with JTree and JList, the renderer component returned acts as a "rubber stamp"[API] used strictly for display purposes.

Note: the row and column parameters refer to the location of data in the TableModel, not a cell location in the TableColumnModel.

When JTable's UI delegate repaints a certain region of a table it must query that table to determine the renderer to use for each cell that it needs to repaint. This is accomplished through JTable's getCellRenderer() method which takes row and column parameters, and returns the component returned by the getTableCellRendererComponent() method of the TableCellRenderer assigned to the appropriate TableColumn. If there is no specific renderer assigned to that TableColumn (recall that this is

the case by default), the `TableModel`'s `getColumnClass()` method is used to recursively determine an appropriate renderer for the given data type. If there is no specific class-based renderer specified for a given class, `getColumnClass()` searches for one corresponding to the super-class. This process will, in the most generic case, stop at `Object`, for which a `DefaultTabelCellRenderer` is used (see below).

A `DefaultTreeCellRenderer` is also used if the class is of type `Number` (subclasses are `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, and `Short`) or `Icon`. If the type happens to be a `Boolean`, a `JCheckBox` is used. We can specify additional class-based renderers with `JTable`'s `setDefaultRenderer()` method. Remember that class-based renderers will only be used if no column-based renderer has been explicitly assigned to the `TableColumn` containing the given cell.

18.1.9 DefaultTableCellRenderer

class javax.swing.table.DefaultTableCellRenderer

This is the concrete default implementation of the `TableCellRenderer` interface. `DefaultTableCellRenderer` extends `JLabel` and is used as the default class-based renderer for `Number`, `Icon`, and `Object` data types. Two private `Color` variables are used to hold selected foreground and background colors which are used to render the cell if it is editable and if it has the current focus. These colors can be assigned with `DefaultTableCellRenderer`'s overridden `setBackground()` and `setForeground()` methods.

A protected `Border` property is used to store the border that is used when the cell does not have the current focus. By default this is an `EmptyBorder` with top and bottom space of 1, and left and right space of 2. Unfortunately `DefaultTableCellRenderer` does not provide a method to change this border.

`DefaultTableCellRenderer` renders the value object passed as parameter to its `getTableCellRenderer()` method by setting its label text to the `String` returned by that object's `toString()` method. Note that all default `JLabel` attributes are used in rendering. Also note that we can do anything to this renderer that we can do to a `JLabel`, such as assign a tooltip or a disabled/enabled state.

Note: `JTable` can have a tooltip assigned to it just as any other Swing component. However, tooltips assigned to renderers take precedence over that assigned to `JTable`, and in the case that both are used the renderer's tooltip text will be displayed when the mouse lies over a cell using it.

18.1.10 The TableCellEditor interface

abstract interface javax.swing.table.TableCellEditor

This interface extends `CellEditor` and describes the editor used to edit cell data in a `TableColumn`. Each `TableColumn` has an associated `TableCellEditor` which can be assigned/retrieved with the `setCellEditor()`/`getCellEditor()` methods. The `getTableCellEditorComponent()` method is the only method declared by this interface, and is expected to return a `Component` that will be used to allow editing of a cell's data value. It takes the following parameters:

- `JTable table`: the table instance containing the cell to be rendered.
- `Object value`: the value used to represent the data in the given cell.
- `boolean isSelected`: whether or not the given cell is selected.
- `int row`: can be used to return a renderer component specific to row or cell.
- `int column`: can be used to return a renderer component specific to column or cell.

We are expected to customize or vary the returned component based on the given parameters. For instance, given a `value` that is an instance of `Color`, we might return a special `JComboBox` which lists several color choices. This method can be used to return different editor components on a column, row, or cell-specific basis, and is similar to `JTree`'s `TreeCellEditor` `getTreeCellEditorComponent()` method.

Note: The `row` and `column` parameters refer to the location of data in the `TableModel`, not a cell location in the `TableColumnModel`.

As with table cell renderers, each `TableColumn` has a column-based editor associated with it. By default this editor is `null` and can be assigned/retrieved with `TableColumn`'s `setCellEditor()`/`getCellEditor()` methods. Unlike renderers, table cell editors are completely interactive and do not simply act as rubber stamps.

`TableCellEditor` implementations must also implement methods defined in the `CellEditor` interface: `addCellEditorListener()`, `removeCellEditorListener()`, `cancelCellEditing()`, `stopCellEditing()`, `isCellEditable()`, `shouldSelectCell()`, and `getCellEditorValue()`. The `isCellEditable()` method is expected to be used in combination with `TableModel`'s `isCellEditable()` method to determine whether a given cell can be edited. Only in the case that both return `true` is editing allowed. (See discussion of the `CellEditor` interface in section 17.1.13 for more about each of these methods.)

To initiate cell editing on a given cell, `JTable` listens for mouse presses and invokes its `editCellAt()` method in response. This method queries both the `TableModel` and the appropriate cell editor to determine if the given cell can be edited. If so the editor component is retrieved with `getTableCellEditorComponent()` and placed in the given cell (its bounds are adjusted so that it will fit within the current cell bounds). Then `JTable` adds itself to the editor component (recall that `JTable` implements the `CellEditorListener` interface) and the same mouse event that sparked the edit gets sent to the editor component. Finally the cell editor's `shouldSelectCell()` method is invoked to determine whether the row containing that cell should become selected.

The default implementation of `TableCellEditor` is provided as `DefaultCellEditor`. Unfortunately `DefaultCellEditor` is not easily extensible and we are often forced to implement all `TableCellEditor` and `CellEditor` functionality ourselves.

18.1.11.1 DefaultCellEditor

class javax.swing.DefaultCellEditor

`DefaultCellEditor` is a concrete implementation of the `TableCellEditor` interface and also the `TreeCellEditor` interface. This editor is designed to return either a `JTextField`, `JComboBox`, or `JCheckBox` for cell editing. It is used by both `JTable` and `JTree` components and is discussed 17.1.15.

18.1.12 The TableModelListener interface

abstract interface javax.swing.event.TableModelListener

This interface describes an object that listens to changes in a `TableModel`. Method `tableChanged()` will be invoked to notify us of these changes. `TableModel`'s `addTableModelListener()` and `removeTableModelListener()` methods are used to add and remove `TableModelListeners` respectively (they are not added directly to `JTable`).

18.1.13 TableModelEvent

class javax.swing.TableModelEvent

This event extends `EventObject` and is used to notify `TableModelListeners` registered with `TableModel` about changes in that model. This class consists of four properties each accessible with typical get methods:

- `int column`: specifies the column affected by the change. `TableModelEvent.ALL_COLUMNS` is used to indicate that more than one column is affected.
- `int firstRow`: specifies the first row affected. `TableModelEvent.HEADER_ROW` can be used here to indicate that the name, type, or order of one or more columns has changed.
- `int lastRow`: the last row affected. This value should always be greater than or equal to `firstRow`.
- `int type`: the type of change that occured. Can be any of `TableModelEvent.INSERT`, `TableModelEvent.DELETE`, or `TableModelEvent.UPDATE`. `INSERT` and `DELETE` indicate the insertion and deletion of rows respectively. `UPDATE` indicates that values have changed but the number of rows and columns has not changed.

As with any `EventObject` we can retrieved the source of a `TableModelEvent` with `getSource()`.

18.1.14 The TableColumnModelListener interface

abstract interface javax.swing.event.TableColumnModelListener

This interface describes an object that listens to changes in a `TableColumnModel`: the adding, removing and movement of columns, as well as changes in margin size and the current selection. `TableColumnModel` provides methods for adding and removing these listeners: `addTableColumnModelListener()` and `removeTableColumnModelListener()`. (As is the case with `TableModelListeners`, `TableColumnModelListeners` are not directly added to `JTable`.)

Five methods are declared in this interface and must be defined by all implementations: `columnAdded(TableColumnModelEvent)`, `columnRemoved(TableColumnModelEvent)`, `columnMoved(TableColumnModelEvent)`, `columnMarginChanged(TableColumnModelEvent)`, and `columnSelectionChanged(ListSelectionEvent)`. `ListSelectionEvents` are forwarded to `TableColumnModel's` `ListSelectionModel`.

18.1.15 TableColumnModelEvent

class javax.swing.event.TableColumnModelEvent

This event extends `EventObject` and is used to notify `TableColumnModel` about changes to a range of columns. These events are passed to `TableColumnModelListeners`. The `fromIndex` property specifies the lowest index of the colmn in the `TableColumnModel` affect by the change. The `toIndex` specifies the highest index. Both can be retrieved with typical get accessors. A `TableColumnModel` fires a `TableColumnModelEvent` whenever a column move, removal, or addition occurs. The event source can be retrieved with `getSource()`.

18.1.16 JTableHeader

class javax.swing.table.JTableHeader

This GUI component (which looks like a set of buttons for each column) is used to display a table's column headers. By dragging these headers the user can rearrange a table's columns dynamically. This component is

used internally by JTable. It can be retrieved with JTable's getTableHeader() method, and assigned with setTableHeader(). When a JTable is placed in a JScrollPane a default JTableHeader corresponding to each column is added to that scroll pane's COLUMN_HEADER viewport (see 7.1.3). Each JTable uses one JTableHeader instance.

JTableHeader extends JComponent and implements TableColumnModelListener. Though JTableHeader is a Swing component, it is not used for display purposes. Instead, each TableColumn maintains a specific TableCellRenderer implementation used to represent its header. By default this is an instance of DefaultTableCellRenderer (see 18.1.8).

Note: It is more common to customize the header renderer of a TableColumn than it is to customize a table's JTableHeader. In most cases the default headers provided by JTable are satisfactory.

The resizingAllowed property specifies whether or not columns can be resized (if this is false it overpowers the isResizable property of each TableColumn). The reorderingAllowed property specifies whether or not columns can be reordered, and updateTableInRealTime property specifies whether or not the whole column is displayed along with the header as it is dragged (this in only applicable if reorderingAllowed is true). All three of these properties are true by default.

UI Guideline: column resizing It is best to try and isolate columns which need to be fixed width, for example the display of monetary amounts which might be 10 significant figures with two decimal places. Such a column requires a fixed width. It doesn't need to be bigger and it doesn't want to be smaller. Allow the other columns to vary in size around the fixed columns.

For example in a two column table displaying Product Description and Price, fix the size of Price and allow Description to resize.

UI Guideline: draggable columns, added flexibility, added complexity If you don't need the flexibility of draggable table columns then it is best to switch them off. If a User accidentally picks up a JHeader component and re-arranges a table, this could be confusing and upsetting. They may not realise what they have done or how to restore the table to its original form.

At any given time during a column drag we can retrieve the distance, in table coordinates, that the column has been dragged with respect to its original position from the draggedDistance property. JTableHeader also maintains a reference to the TableColumn it represents as well as the JTable it is part of -- the tableColumn and table properties respectively.

18.1.17 JTable selection

JTable supports two selection models: one for row selections and one for column selections. JTable also supports selection of individual table cells. Column selections are managed by a ListSelectionModel maintained by a TableColumnModel implementation, and row selections are managed by a ListSelectionModel maintained by JTable itself (both are DefaultListSelectionModels by default). As we learned in chapter 10, ListSelectionModels support three selection modes: SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, and MULTIPLE_INTERVAL_SELECTION. JTable provides the setSelectionMode() methods which will set both selection models to the given mode. Note, however, that getSelectionMode() only returns the current row selection mode.

To assign a specific selection mode to JTable's row ListSelectionModel:

```
myJTable.getSelectionModel().setSelectedMode(
  ListSelectionModel.XX_SELECTION);
```

To assign a specific selection mode to JTable's column ListSelectionModel:

```
myJTable.getColumnModel().getSelectionModel().setSelectionMode(
  ListSelectionModel.XX_SELECTION);
```

Row selection mode defaults to MULTIPLE_INTERVAL_SELECTION, and column selection mode defaults to SINGLE_SELECTION_MODE.

JTable provides control over whether rows and and columns can be selected, and we can query these modes, and turn them on/off, with getRowSelectionAllow(), getColumnSelectionAllowed(), and setRowSelectionAllowed(),setColumnSelectionAllowed() respectively. When row selection is enabled (true by default) and cell selction is disabled (see below), clicking on a cell will select the entire row that cell belongs to. Similarly, when column selection is enabled (false by default) the whole column that cell belongs to will be selected. There is nothing stopping us from having both row and column selection active simultaneously.

JTable also provides control over whether individual cells can be selected with its cellSelectionEnabled property. We can turn this on or off with setCellSelectionEnabled() and query its state using getCellSelectionEnabled(). If cell selection is enabled (false by default), a cell can only be selected if both row selection and column selection are also enabled (see below). If cell selection is not enabled, whenever a row or column containing that cell is selected (assuming that either row and/or column selection is enabled), that cell is also considered selected.

JTable provides several additional methods for querying the state of selection. If there is at least one cell selected:

- getSelectedColumn() returns the index (in the TreeModel) of the most recently seleced column (-1 if no selection exists).
- getSelectedRow() returns the index (in the TreeModel) of the most recently selected row (-1 if no selection exists).
- getSelectedColumns() and getSelectedRows() return the TreeModel indices of all currently selected columns and rows respectively (int[0] if no selection exists).
- getSelectedColumnCount() and getSelectedRowCount() return the current number of selected columns and rows respectively (int[0] if no selection exists).
- isColumnSelected() and isRowSelected() return a boolean specifying whether or not the given column or row is currently selected.
- isCellSelected() returns a boolean specifying whether or not the cell at the given TreeModel row and column index is selected.

The following methods can be used to programatically change JTable's selection, assuming the corresponding selection properties are enabled:

- clearSelection() unselects all rows, columns and cells.
- selectAll() selects all rows, columns, and cells.
- addColumnSelectionInterval() and addRowSelectionInterval() allow programmatic selection of a contiguous group of columns and rows respectively. Note that these can be called repeatedly to build of a multiple-inverval selection if the MULTIPLE_INTERVAL_SELECTION mode is active in the corresponding selection models.
- removeColumnSelectionInterval() and removeRowSelectionInterval() allow

programmatic de-selection of a contiguous interval of columns and rows respectively. These can also be used repeatedly to affect multiple-interval selections.

`setColumnSelectionInterval()` and `setRowSelectionInterval()` clear the current column and row selection respectively, and select the specified contiguous interval.

Interestingly, when cell selection is enabled, `JTable` considers the columns and rows containing selected cells selected themselves (even though they aren't highlighted). For example, if cells (1,5) and (3,6) are selected with row and column selection enabled and cell selection enabled, `getSelectedColumns()` will return {5,6} and `getSelectedRows()` will return {1,3}. Oddly enough, those two cells will be highlighted and considered selected by `JTable`, along with cells (1,6) and (3,5)! This is due to the fact that `JTable` bases cell selection solely on whether or not both the row and column containing a cell are selected. When selected rows and columns intersect, the cells at the intersection points are considered selected.

If these same cells are selected when cell selection is disabled and row and column selection are enabled, all cells in rows 1 and 3, and all cells in columns 5 and 6 will be considered selected. If they are selected with cell selection and only row selection enabled, all cells in rows 1 and 3 will be considered selected. Similarly, if these two cells are selected with cell selection and only column selection enabled, all cells in columns 5 and 6 will be considered selected. If cell selection is not enabled, and row and/or column selection is enabled, a cell will be considered selected if either a column or row containing it is selected.

Note: Multiple single-cell selections can be made by holding down the `CTRL` key and using the mouse for selection.

Typically we are interested in determining cell, row, and/or column selection based on a mouse click. `JTable` supports `MouseListeners` just as any other `JComponent`, and we can used the `getSelectedColumn()` and `getSelectedRow()` methods to determine which cell was clicked in `MouseListener`'s `mouseClicked()` method:

```
myJTable.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent e) {
    // get most recently selected row index
    int row = getSelectedRow();
    // get most recently selected column index
    int column = getSelectedColumn();
    if (row == -1 || column == -1)
      return; // can't determine selected cell
    else
      // do something cell-specific
  }
});
```

This listener is not very robust because it will only give us a cell if both a row and column have recently been selected, which in turn can only occur if both row selection and column selection is enabled. Thankfully, `JTable` provides methods for retrieving a row and column index corresponding to a given `Point`: `rowAtPoint()` and `columnAtPoint()` (these will return `-1` if no row or column is found respectively). Since `MouseEvent` carries a `Point` specifying the location where the event occurred, we can use these methods in place of the `getSelectedRow()` and `getSelectedColumn()` methods. This is particularly useful when row, column, and/or cell selection is not enabled.

As with `JList`, `JTable` does not directly support double-mouse click selections. However, as we learned in chapter 10, we can capture a double click and determine which cell was clicked by adding a listener to `JTable` similar to the following:

```
myJTable.addMouseListener(new MouseAdapter() {
```

```
    public void mouseClicked(MouseEvent e) {
      if (e.getClickCount() == 2) {
        Point origin = e.getPoint();
        int row = myJTable.rowAtPoint(origin);
        int column = myJTable.columnAtPoint(origin);
        if (row == -1 || column == -1)
          return; // no cell found
        else
          // do something cell-specific
      }
    }
  });
```

18.1.18 Column Width and Resizing

When a column's width increases, `JTable` must decide how other columns will react. One or more columns must shrink. Similarly, when a column's width decreases, `JTable` must decide how other columns will react to the newly available amount of space. `JTable`'s `autoResizeMode` property can take on any of five different values that handle these cases differently:

- `JTable.AUTO_RESIZE_ALL_COLUMNS`: All columns gain or lose an equal amount of space corresponding the width lost or gained by the resizing column.
- `JTable.AUTO_RESIZE_LAST_COLUMN`: The right-most column shrinks or grows in direct correspondence with the amount of width lost or gained from the column being resized. All other colmns are not affected.
- `JTable.AUTO_RESIZE_NEXT_COLUMN`: The column to the immediate right of the column being resized shrinks or grows in direct correspondence with the amount of width lost or gained from the resizing column. All other colmns are not affected.
- `JTable.AUTO_RESIZE_OFF`: resizing only affects the column being sized. All columns to the right of the column being resized are shifted right or left accordingly while maintaining their current sizes. Columns to the left are not affected.
- `JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS`: All columns to the right of the column being resized gain or lose an equal amount of space corresponding the width lost or gained by the resizing column. Columns to the left are not affected.

`TableColumn`'s width defaults to 75. Its minimum width defaults to 15 and its maximum width defaults to `Integer.MAX_VALUE`. When a `JTable` is first displayed it attempts to size each `TableColumn` according to its `width` property. If that table's `autoResizeMode` property is set to `AUTO_RESIZE_OFF` this will occur successfully. Otherwise, `TableColumns` are adjusted according to the current `autoResizeMode` property.

A `TableColumn` will never be sized larger than its maximum width or smaller than its minimum. For this reason it is possible that a `JTable` will occupy a larger or smaller area than that available (i.e. visible in the parent `JScrollPane`'s main viewport), which may result in part of the table being clipped from view. If a table is contained in a `JScrollPane` and it occupies more than the available visible width, a horizontal scrollbar will be presented.

At any time we can call `TableColumn`'s `sizeWidthToFit()` method to resize a column to occupy a width corresponding to the preferred width of its table header renderer. This is often used in assigning minimum widths for each `TableColumn`. `JTable`'s `sizeColumnsToFit()` method takes an `int` parameter specifying the index of the `TableColumn` to act as the source of a resize in an attempt to make all columns fit within the available visible space. Also note that `TableColumnModel`'s `getTotalColumnWidth()` method will return the sum of the current width of all `TableColumns` including all margin space.

We can specify the amount of empty space between rows with `JTable`'s `setRowMargin()` method, and we

can assign all rows a specific height with `setRowHeight()`. `JTable`'s `setIntercellSpacing()` method takes a `Dimension` instance and uses it to assign a new width and height to use as margin space between cells (this method will redisplay the table it is invoked on after all sizes have been changed).

18.1.19 JTable Appearance

We can change the background and foreground colors used to highlight selected cells by setting the `selectedBackground` and `setSelectedForeground` properties. The default colors used for each `TableColumn`'s table header renderer are determined from the current `JTableHeader`'s background and foreground colors (recall that `JTableHeader` extends `JComponent`).

We can turn on and off horizontal and vertical grid lines (which are always have a thickness of 1 pixel) by changing the `showHorizontalLines` and `showVerticalLines`. properties. The `showGrid` property will overpower these properties when set with `setShowGrid()` becauuse this method reasigns them to the specified value. So `setShowGrid()` turns on/off both vertical and horizontal lines as specified. The `gridColor` property specifies the `Color` to use for both vertical and horizontal grid lines. `setGridColor()` will assign the specified value to this property and then repaint the whole table.

UI Guideline: Visual Noise Grid Lines add visual noise to the display of a table. Removing some of them can aid the reading of the table data. If you intend the reader to read rows across then switch off the vertical grid lines. If you have columns of figures for example, then you might prefer to switch off the horizontal grid lines, making the columns easier to read.

When switching off the horizontal grid lines on the table, you may wish to use the column cell renderer to change the background color of alternate rows on the table to ease the reading of rows. Using this combination of visual techniques, grid lines to distinguish columns and colour to distinguish rows, helps guide the reader to interpret data as required.

18.1.20 JTable scrolling

`JTable` implements the `Scrollable` interface (see 7.1.4) and is intended to be placed in a `JScrollPane`. `JTableHeaders` will not be displayed otherwise, disabling any column resize capabilities. Among the required `Scrollable` methods, `JTable` implements `getScrollableTracksViewportWidth()` to return `true`, which forces `JTable` to attempt to size itself horizontally to fit within the current scroll pane viewport width. `getScrollableTracksViewportHeight()`, however, returns `false` as it is most common for tables to be vertically scrolled but not horizontally scrolled. Horizontal scrolling is often awkward and we advise avoiding it whenever possible.

`JTable`'s vertical block increment is the number of visible rows less one, and its vertical unit increment is the current row height. The horizontal block inrement is the width of the viewport, and the horizontal unit increment defaults to 100.

UI Guideline: small grids, no column headers If you have a requirement to show two or three pieces of data, grouped and aligned together, consider using a JTable without JScrollPane. This gives you a small grid which is already aligned and neat and tidy for display without column headers.

18.2 Stocks Table: part I - Basic JTable example

This basic example shows how to construct a `JTable` to display information about stock market data for a given day. Despite of its simplicity, it demonstrates the most fundamental features of `JTable`, and serves as a good basis for the more advanced examples that follow.

Stocks and stock trading is characterized by many attributes. The following are selected for display in our example:

Name	Type	Description
Symbol	String	Stock's symbol (NYSE or NASDAQ)
Name	String	Company name
Last	double	Price at the end of the trade day
Open	double	Price at the beginning of the trade day
Change	double	Absolute change in price with respect to previous closing
Change%	double	Percent change in price with respect to previous closing
Volume	long	Day's volume of trade (in $) for this stock

Each stock attribute represents a column in our table, and each row represents a specific company's stock information.

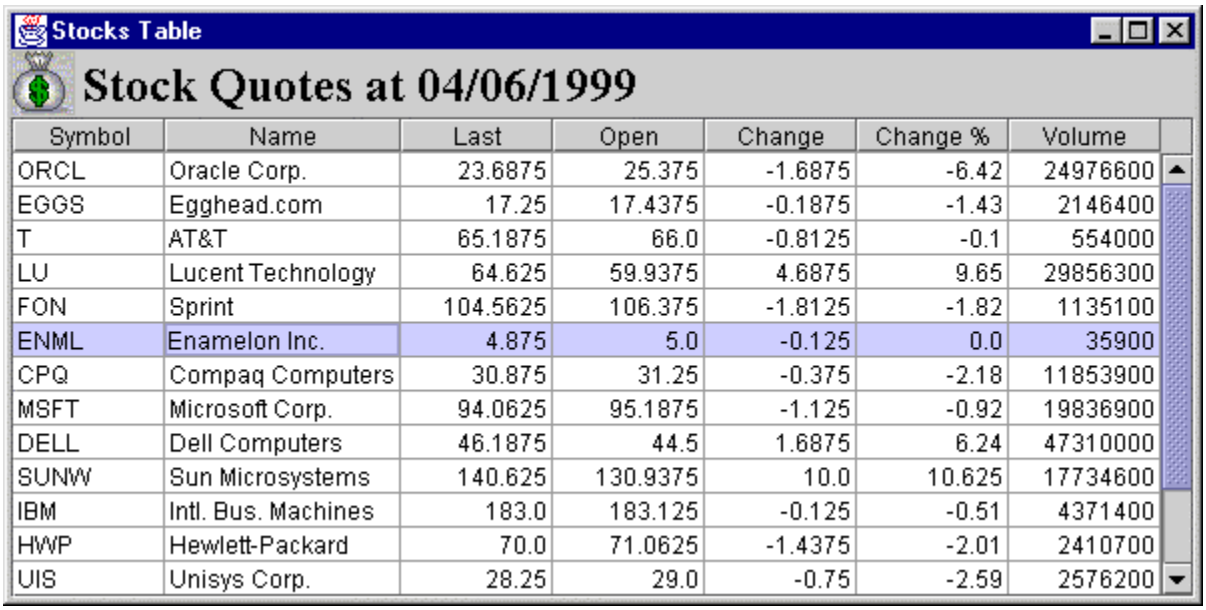


Figure 18.1 JTable in a JScrollPane with 7 TableColumns and 16 rows of data.

<<file figure18-1.gif>>

The Code: StocksTable.java
see \Chapter18\1

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class StocksTable extends JFrame
{
  protected JTable m_table;
  protected StockTableData m_data;
  protected JLabel m_title;
```

```
  public StocksTable() {
    super("Stocks Table");
    setSize(600, 340);

    m_data = new StockTableData();

    m_title = new JLabel(m_data.getTitle(),
      new ImageIcon("money.gif"), SwingConstants.LEFT);
    m_title.setFont(new Font("TimesRoman",Font.BOLD,24));
    m_title.setForeground(Color.black);
    getContentPane().add(m_title, BorderLayout.NORTH);

    m_table = new JTable();
    m_table.setAutoCreateColumnsFromModel(false);
    m_table.setModel(m_data);

    for (int k = 0; k < StockTableData.m_columns.length; k++) {
      DefaultTableCellRenderer renderer = new
        DefaultTableCellRenderer();
      renderer.setHorizontalAlignment(
        StockTableData.m_columns[k].m_alignment);
      TableColumn column = new TableColumn(k,
        StockTableData.m_columns[k].m_width, renderer, null);
      m_table.addColumn(column);
    }

    JTableHeader header = m_table.getTableHeader();
    header.setUpdateTableInRealTime(false);

    JScrollPane ps = new JScrollPane();
    ps.getViewport().add(m_table);
    getContentPane().add(ps, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);
    setVisible(true);
  }

  public static void main(String argv[]) {
    new StocksTable();
  }
}

class StockData
{
  public String m_symbol;
  public String m_name;
  public Double m_last;
  public Double m_open;
  public Double m_change;
  public Double m_changePr;
  public Long m_volume;

  public StockData(String symbol, String name, double last,
   double open, double change, double changePr, long volume) {
    m_symbol = symbol;
    m_name = name;
    m_last = new Double(last);
    m_open = new Double(open);
```

```
    m_change = new Double(change);
    m_changePr = new Double(changePr);
    m_volume = new Long(volume);
  }
}

class ColumnData
{
  public String  m_title;
  public int     m_width;
  public int     m_alignment;

  public ColumnData(String title, int width, int alignment) {
    m_title = title;
    m_width = width;
    m_alignment = alignment;
  }
}

class StockTableData extends AbstractTableModel
{
  static final public ColumnData m_columns[] = {
    new ColumnData( "Symbol", 100, JLabel.LEFT ),
    new ColumnData( "Name", 150, JLabel.LEFT ),
    new ColumnData( "Last", 100, JLabel.RIGHT ),
    new ColumnData( "Open", 100, JLabel.RIGHT ),
    new ColumnData( "Change", 100, JLabel.RIGHT ),
    new ColumnData( "Change %", 100, JLabel.RIGHT ),
    new ColumnData( "Volume", 100, JLabel.RIGHT )
  };

  protected SimpleDateFormat m_frm;
  protected Vector m_vector;
  protected Date   m_date;

  public StockTableData() {
    m_frm = new SimpleDateFormat("MM/dd/yyyy");
    m_vector = new Vector();
    setDefaultData();
  }

  public void setDefaultData() {
    try {
      m_date = m_frm.parse("04/06/1999");
    }
    catch (java.text.ParseException ex) {
      m_date = null;
    }

    m_vector.removeAllElements();
    m_vector.addElement(new StockData("ORCL", "Oracle Corp.",
      23.6875, 25.375, -1.6875, -6.42, 24976600));
    m_vector.addElement(new StockData("EGGS", "Egghead.com",
      17.25, 17.4375, -0.1875, -1.43, 2146400));
    m_vector.addElement(new StockData("T", "AT&T",
      65.1875, 66, -0.8125, -0.10, 554000));
    m_vector.addElement(new StockData("LU", "Lucent Technology",
      64.625, 59.9375, 4.6875, 9.65, 29856300));
    m_vector.addElement(new StockData("FON", "Sprint",
      104.5625, 106.375, -1.8125, -1.82, 1135100));
    m_vector.addElement(new StockData("ENML", "Enamelon Inc.",
      4.875, 5, -0.125, 0, 35900));
    m_vector.addElement(new StockData("CPQ", "Compaq Computers",
```

```
      30.875, 31.25, -0.375, -2.18, 11853900));
    m_vector.addElement(new StockData("MSFT", "Microsoft Corp.",
      94.0625, 95.1875, -1.125, -0.92, 19836900));
    m_vector.addElement(new StockData("DELL", "Dell Computers",
      46.1875, 44.5, 1.6875, 6.24, 47310000));
    m_vector.addElement(new StockData("SUNW", "Sun Microsystems",
      140.625, 130.9375, 10, 10.625, 17734600));
    m_vector.addElement(new StockData("IBM", "Intl. Bus. Machines",
      183, 183.125, -0.125, -0.51, 4371400));
    m_vector.addElement(new StockData("HWP", "Hewlett-Packard",
      70, 71.0625, -1.4375, -2.01, 2410700));
    m_vector.addElement(new StockData("UIS", "Unisys Corp.",
      28.25, 29, -0.75, -2.59, 2576200));
    m_vector.addElement(new StockData("SNE", "Sony Corp.",
      96.1875, 95.625, 1.125, 1.18, 330600));
    m_vector.addElement(new StockData("NOVL", "Novell Inc.",
      24.0625, 24.375, -0.3125, -3.02, 6047900));
    m_vector.addElement(new StockData("HIT", "Hitachi, Ltd.",
      78.5, 77.625, 0.875, 1.12, 49400));
  }

  public int getRowCount() {
    return m_vector==null ? 0 : m_vector.size();
  }

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

  public String getColumnName(int column) {
    return m_columns[column].m_title;
  }

  public boolean isCellEditable(int nRow, int nCol) {
    return false;
  }

  public Object getValueAt(int nRow, int nCol) {
    if (nRow < 0 || nRow >= getRowCount())
      return "";
    StockData row = (StockData)m_vector.elementAt(nRow);
    switch (nCol) {
      case 0: return row.m_symbol;
      case 1: return row.m_name;
      case 2: return row.m_last;
      case 3: return row.m_open;
      case 4: return row.m_change;
      case 5: return row.m_changePr;
      case 6: return row.m_volume;
    }
    return "";
  }

  public String getTitle() {
    if (m_date==null)
      return "Stock Quotes";
    return "Stock Quotes at "+m_frm.format(m_date);
  }
}
```

Understanding the Code

Class StocksTable

This class extends `JFrame` to implement the frame container for our table. Three instance variables are declared (to be used extensively in more complex examples that follow):

`JTable m_table`: table component to display stock data.

`StockTableData m_data`: `TableModel` implementation to manage stock data (see below).

`JLabel m_title`: used to display stocks table title (date which stock prices are referenced).

The `StocksTable` constructor first initializes the parent frame object and builds an instance of `StockTableData`. Method `getTitle()` is invoked to set the text for the title label which is added to the northern region of the content pane. Then a `JTable` is created by passing the `StockTableData` instance to the constructor. Note the the `autoCreateColumnsFromModel` method is set to `false` because we plan on creating our own `TableColumns`.

As we will see below, the static array `m_columns` of the `StockTableData` class describes all columns of our table. It is used here to create each `TableColumn` instance and set their text alignment and width.

Method `setHorizontalAlignment()` (inherited by `DefaultTableCellRenderer` from `JLabel`) is used to set the proper alignment for each `TableColumn`'s cell renderer. The `TableColumn` constructor takes a column index, width, and renderer as parameters. Note that `TableCellEditor` is set to `null` since we don't want to allow editing of stock data. Finally, columns are added to the table's `TableColumnModel` (which `JTable` created by default because we didn't specify one) with the `addColumn()` method.

In the next step, an instance of `JTableHeader` is created for this table, and the `updateTableInRealTime` property is set to `false` (this is done to demonstrate the effect this has on column dragging -- only a column's table header is displayed during a drag).

Lastly a `JScrollPane` instance is used to provide scrolling capabilities, and our table is added to its `JViewport`. This `JScrollPane` is then added to the center of our frame's content pane.

Class StockData

This class encapsulates a unit of stock data as described in the table above. The instance variables defined in this class have the following meaning:

`String m_symbol`: stock's symbol (NYSE or NASDAQ)

`String m_name`: company name

`Double m_last`: the price of the last trade

`Double m_open`: price at the beginning of the trade day

`Double m_change`: absolute change in price with respect to previous closing

`Double m_changePr`: percent change in price with respect to previous closing

`Long m_volume`: day's volume of trade (in $) for this stock

Note that all numerical data are encapsulated in `Object`-derived classes. This design decision simplifies data exchange with the table (as we will see below). The only constructor provided assigns each of these variables from the data passed as parameters.

Note: We use `public` instance variables in this and several other classes in this chapter to avoid overcomplication. In most professional apps these would either be protected or private.

Class ColumnData

This class encapsulates data describing the visual characteristics of a single `TableColumn` of our table. The instance variables defined in this class have the following meaning:

- `String m_title`: column title
- `int m_width`: column width in pixels
- `int m_alignment`: text alignment as defined in `JLabel`

The only constructor provided assigns each of these variables the data passed as parameters.

Class StockTableData

This class extends `AbstractTableModel` to serve as the data model for our table. Recall that `AbstractTableModel` is an abstract class, and three methods must be implemented to instantiate it:

- `public int getRowCount()`: returns the number of rows in the table.
- `public int getColumnCount()`: returns the number of columns in the table.
- `public Object getValueAt(int row, int column)`: returns data in the specified cell as an `Object` instance.

Note: An alternative approach is to extend the `DefaultTableModel` class which is a concrete implementation of `AbstractTableModel`. However, this is not recommended, as the few abstract methods in `AbstractTableModel` can be easily implemented. Usage of `DefaultTableModel` often creates unnecessary overhead.

By design, this class manages all information about our table, including the title and column data. A static array of `ColumnData`, `m_columns`, is provided to hold information about our table's columns (it is used in the `StocksTable` constructor, see above). Three instance variables have the following meaning:

- `SimpleDateFormat m_frm`: used to format dates
- `Date m_date`: date of currently stored market data
- `Vector m_vector`: collection of `StockData` instances for each row in the table

The only constructor of the `StockTableData` class initializes two of these variables and calls the `setDefaultData()` method to assign the pre-defined default data to `m_date` and `m_vector` (in a later example we'll see how to use JDBC to retrieve data from a database rather than using hard-coded data as we do here).

As we discussed above, the `getRowCount()` and `getColumnCount()` methods should return the number of rows and columns, respectively. So their implementation is fairly obvious. The only catch is that they may be called by the `AbstractTableModel` constructor before any member variable is initialized. So we have to check for a `null` instance of `m_vector`. Note that `m_columns`, as a static variable, will be initialized before any non-static code is executed (so we don't have to check `m_columns` against `null`).

The remainder of the `StockTableData` class implements the following methods:

- `getColumnName()`: returns the column title.
- `isCellEditable()`: always returns `false`, because we want to disable all editing.
- `getValueAt()`: retrieves data for a given cell as an `Object`. Depending on the column index, one of the

`StockData` fields is returned.

`getTitle()`: returns our table's title as a `String` to be used in a `JLabel` in the northern region of our frame's content pane.

Running the Code

Figure 18.1 shows `StocksTable` in action displaying our hard-coded stock data. Note that the `TableColumns` resize properly in response to the parent frame size. Also note that the selected row in our table can be moved by changed with the mouse or arrow keys, but no editing is allowed.

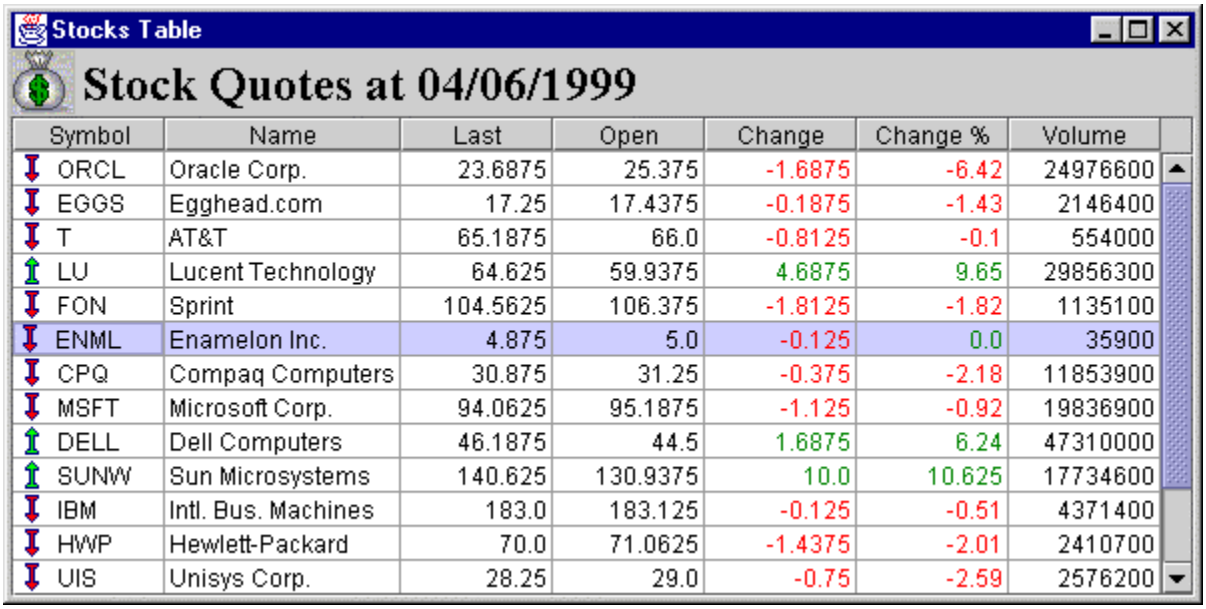
18.3 StocksTable: part II - Custom renderers

Now we'll extend `StocksTable` to use color and small icons in rendering our table cells. To enhance data visibility, we'll make the following two enhancements:

Render absolute and percent changes in green for positive values and red for negative values.

Add an icon next to each stock symbol: arrow up for positive changes and arrow down for negative.

To do this we need to build our own custom `TreeCellRenderer`.



Symbol	Name	Last	Open	Change	Change %	Volume
ORCL	Oracle Corp.	23.6875	25.375	-1.6875	-6.42	24976600
EGGS	Egghead.com	17.25	17.4375	-0.1875	-1.43	2146400
T	AT&T	65.1875	66.0	-0.8125	-0.1	554000
LU	Lucent Technology	64.625	59.9375	4.6875	9.65	29856300
FON	Sprint	104.5625	106.375	-1.8125	-1.82	1135100
ENML	Enamelon Inc.	4.875	5.0	-0.125	0.0	35900
CPQ	Compaq Computers	30.875	31.25	-0.375	-2.18	11853900
MSFT	Microsoft Corp.	94.0625	95.1875	-1.125	-0.92	19836900
DELL	Dell Computers	46.1875	44.5	1.6875	6.24	47310000
SUNW	Sun Microsystems	140.625	130.9375	10.0	10.625	17734600
IBM	Intl. Bus. Machines	183.0	183.125	-0.125	-0.51	4371400
HWP	Hewlett-Packard	70.0	71.0625	-1.4375	-2.01	2410700
UIS	Unisys Corp.	28.25	29.0	-0.75	-2.59	2576200

Figure 18.2 `JTable` using a custom cell renderer.
<<file figure18-2.gif>>

The Code: StocksTable.java
see \Chapter18\2

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;
```

```
public class StocksTable extends JFrame
{
  // Unchanged code from section 18.2

  public StocksTable() {
    // Unchanged code from section 18.2

    for (int k = 0; k < StockTableData.m_columns.length; k++) {
      DefaultTableCellRenderer renderer = new
        ColoredTableCellRenderer();
      renderer.setHorizontalAlignment(
        StockTableData.m_columns[k].m_alignment);
      TableColumn column = new TableColumn(k,
        StockTableData.m_columns[k].m_width, renderer, null);
      m_table.addColumn(column);
    }

    // Unchanged code from section 18.2
  }

  public static void main(String argv[]) {
    new StocksTable();
  }
}

class ColoredTableCellRenderer extends DefaultTableCellRenderer
{
  public void setValue(Object value) {
    if (value instanceof ColorData) {
      ColorData cvalue = (ColorData)value;
      setForeground(cvalue.m_color);
      setText(cvalue.m_data.toString());
    }
    else if (value instanceof IconData) {
      IconData ivalue = (IconData)value;
      setIcon(ivalue.m_icon);
      setText(ivalue.m_data.toString());
    }
    else
      super.setValue(value);
  }
}

class ColorData
{
  public Color  m_color;
  public Object m_data;
  public static Color GREEN = new Color(0, 128, 0);
  public static Color RED = Color.red;

  public ColorData(Color color, Object data) {
    m_color = color;
    m_data  = data;
  }

  public ColorData(Double data) {
    m_color = data.doubleValue() >= 0 ? GREEN : RED;
    m_data  = data;
  }

  public String toString() {
    return m_data.toString();
```

```
  }
}

class IconData
{
  public ImageIcon  m_icon;
  public Object m_data;

  public IconData(ImageIcon icon, Object data) {
    m_icon = icon;
    m_data = data;
  }

  public String toString() {
    return m_data.toString();
  }
}

class StockData
{
  public static ImageIcon ICON_UP = new ImageIcon("ArrUp.gif");
  public static ImageIcon ICON_DOWN = new ImageIcon("ArrDown.gif");
  public static ImageIcon ICON_BLANK = new ImageIcon("blank.gif");

  public IconData  m_symbol;
  public String  m_name;
  public Double  m_last;
  public Double  m_open;
  public ColorData  m_change;
  public ColorData  m_changePr;
  public Long    m_volume;

  public StockData(String symbol, String name, double last,
   double open, double change, double changePr, long volume) {
    m_symbol = new IconData(getIcon(change), symbol);
    m_name = name;
    m_last = new Double(last);
    m_open = new Double(open);
    m_change = new ColorData(new Double(change));
    m_changePr = new ColorData(new Double(changePr));
    m_volume = new Long(volume);
  }

  public static ImageIcon getIcon(double change) {
    return (change>0 ? ICON_UP : (change<0 ? ICON_DOWN :
      ICON_BLANK));
  }
}

// Class StockTableData unchanged from section 18.2
```

Understanding the Code

Class StocksTable

The only change we need to make in the base frame class is to change the column renderer to a new class, `ColoredTableCellRenderer`. `ColoredTableCellRenderer` should be able to draw icons and colored text (but not both at the same time -- although this could be done using this same approach).

Class ColoredTableCellRenderer

This class extends `DefaultTableCellRenderer` and overrides only one method: `setValue()`. This

method will be called prior to the rendering of a cell to retrieve its corresponding data (of any nature) as an `Object`. Our overridden `setValue()` method is able to recognize two specific kinds of cell data: `ColorData`, which adds color to a data object, and `IconData`, which adds an icon (both are described below). If a `ColorData` instance is detected, its encapsulated color is set as the foreground for the renderer. If an `IconData` instance is detected, its encapsulated icon is assigned to the renderer with the `setIcon()` method (which is inherited from `JLabel`). If the value is neither a `ColorData` or an `IconData` instance we call the super-class `setValue()` method.

Class ColorData

This class is used to bind a specific color, `m_color`, to a data object of any nature, `m_data`. Two static `Colors`, `RED` and `GREEN`, are declared to avoid creation of numerous temporary objects. Two constructors are provided for this class. The first constructor takes `Color` and `Object` parameters and assigns them to instance variables `m_color` and `m_data` respectively. The second constructor takes a `Double` parameter which gets assigned to `m_data`, and `m_color` is assigned the green color if the parameter is positive, and red if negative. The `toString()` method simply calls the `toString()` method of the data object.

Class IconData

This class is used to bind `ImageIcon m_icon` to a data object of any nature, `m_data`. Its only constructor takes `ImageIcon` and `Object` parameters. The `toString()` method simply calls the `toString()` method of the data object.

Class StockData

This class has been enhanced from its previous version to to provide images and new variable data types. We've prepared three static `ImageIcon` instances holding images: arrow up, arrow down, and a blank (all transparent) image. The static `getIcon()` method returns one of these images depending on the sign of the given `double` parameter. We've also changed three instance variables to bind data with the color and image attributes according to the following table:

Field	New type	Data object	Description
m_symbol	IconData	String	Stock's symbol (NYSE or NASDAQ)
m_change	ColorData	Double	Absolute change in price
m_changePr	ColorData	Double	Percent change in price

The corresponding changes are also required in the `StockData` constructor.

Running the Code

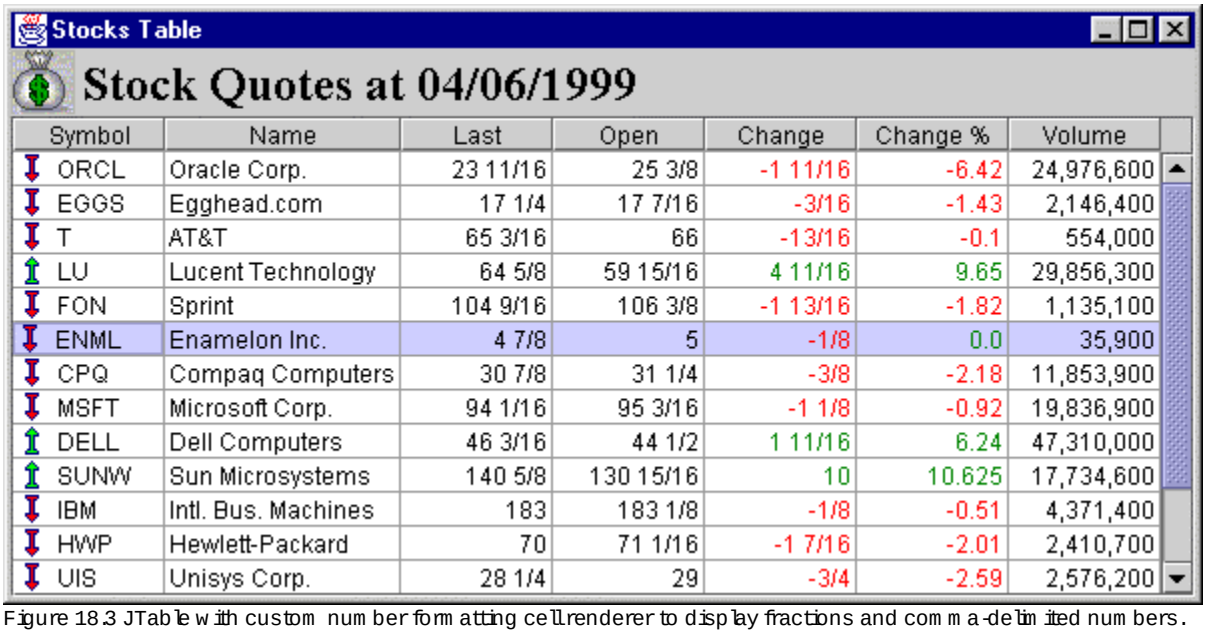
Figure 18.2 shows `StocksTable` with custom rendering in action. Note the correct usage of color and icons, which considerably enhances the visualization of our data.

UI Guideline: Improving visual communication Tables can be data intensive and consequently it can be very difficult for the viewer to quickly pick out the important information. The table in fig 18.1 highlighted this. In fig 18.2, we are improving the visual communication with the introduction of visual layers. The icons in the first column quickly tell the viewer whether a price is rising or falling. This is visually re-inforced with the red and green introduced on the change columns.

Red particularly is a very strong color. By introducing red and green only on the change columns and not across the entire row, we avoid the danger of the red becoming overpowering. If we had introduced red and green across the full width of the table, the colors may have become intrusive and impaired the visual communication.

18.4 StocksTable: part III - Data formatting

To further enhance the presentation of our stock data, in this section we will take into account that actual stock prices (at least on NYSE and NASDAQ) are expressed in fractions of 32, not in decimals. Another issue that we will deal with is the volume of trade, which can reach hundreds of millions of dollars. Volume is not immediately legible without separating thousands and millions places with commas.



Symbol	Name	Last	Open	Change	Change %	Volume
ORCL	Oracle Corp.	23 11/16	25 3/8	-1 11/16	-6.42	24,976,600
EGGS	Egghead.com	17 1/4	17 7/16	-3/16	-1.43	2,146,400
T	AT&T	65 3/16	66	-13/16	-0.1	554,000
LU	Lucent Technology	64 5/8	59 15/16	4 11/16	9.65	29,856,300
FON	Sprint	104 9/16	106 3/8	-1 13/16	-1.82	1,135,100
ENML	Enamelon Inc.	4 7/8	5	-1/8	0.0	35,900
CPQ	Compaq Computers	30 7/8	31 1/4	-3/8	-2.18	11,853,900
MSFT	Microsoft Corp.	94 1/16	95 3/16	-1 1/8	-0.92	19,836,900
DELL	Dell Computers	46 3/16	44 1/2	1 11/16	6.24	47,310,000
SUNW	Sun Microsystems	140 5/8	130 15/16	10	10.625	17,734,600
IBM	Intl. Bus. Machines	183	183 1/8	-1/8	-0.51	4,371,400
HWP	Hewlett-Packard	70	71 1/16	-1 7/16	-2.01	2,410,700
UIS	Unisys Corp.	28 1/4	29	-3/4	-2.59	2,576,200

Figure 18.3 JTable with custom number formatting cell renderer to display fractions and comma-delimited numbers.
<<file figure18-3.gif>>

The Code: StocksTable.java
see \Chapter18\3

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

// Unchanged code from section 18.3

class Fraction
{
  public int m_whole;
  public int m_nom;
  public int m_den;

  public Fraction(double value) {
    int sign = value <0 ? -1 : 1;
    value = Math.abs(value);
    m_whole = (int)value;
    m_den = 32;
    m_nom = (int)((value-m_whole)*m_den);
    while (m_nom!=0 && m_nom%2==0) {
```

```
      m_nom /= 2;
      m_den /= 2;
    }
    if (m_whole==0)
      m_nom *= sign;
    else
      m_whole *= sign;
  }

  public double doubleValue() {
    return (double)m_whole + (double)m_nom/m_den;
  }

  public String toString() {
    if (m_nom==0)
      return ""+m_whole;
    else if (m_whole==0)
      return ""+m_nom+"/"+m_den;
    else
      return ""+m_whole+" "+m_nom+"/"+m_den;
  }
}

class SmartLong
{
  protected static NumberFormat FORMAT;

  static {
    FORMAT = NumberFormat.getInstance();
    FORMAT.setGroupingUsed(true);
  }

  public long m_value;

  public SmartLong(long value) { m_value = value; }

  public long longValue() { return m_value; }

  public String toString() { return FORMAT.format(m_value); }
}

class ColorData
{
  public ColorData(Fraction data) {
    m_color = data.doubleValue() >= 0 ? GREEN : RED;
    m_data  = data;
  }

  // Unchanged code from section 18.3
}

class StockData
{
  public static ImageIcon ICON_UP = new ImageIcon("ArrUp.gif");
  public static ImageIcon ICON_DOWN = new ImageIcon("ArrDown.gif");
  public static ImageIcon ICON_BLANK = new ImageIcon("blank.gif");

  public IconData  m_symbol;
  public String  m_name;
  public Fraction  m_last;
  public Fraction  m_open;
  public ColorData  m_change;
  public ColorData  m_changePr;
```

```
  public SmartLong  m_volume;

  public StockData(String symbol, String name, double last,
   double open, double change, double changePr, long volume) {
    m_symbol = new IconData(getIcon(change), symbol);
    m_name = name;
    m_last = new Fraction(last);
    m_open = new Fraction(open);
    m_change = new ColorData(new Fraction(change));
    m_changePr = new ColorData(new Double(changePr));
    m_volume = new SmartLong(volume);
  }

  // Unchanged code from section 18.3
}
```

Understanding the Code

Class Fraction

This new data class encapsulates fractions with denominator 32 (or in a reduced form). Three instance variables represent the whole number, numerator, and denominator of the fraction. The only constructor takes a double parameter and carefully extracts these values, performing numerator and denominator reduction, if possible. Note that negative absolute values are taken into account.

Method `doubleValue()` performs the opposite task: it converts the fraction into a `double` value. Method `toString()` forms a `String` representation of the fraction.

Note that a zero whole number or a zero nominator are omitted to avoid unseemly output like "0 1/2" or "12 0/32".

Class SmartLong

This class encapsulates `long` values. The only constructor takes a `long` as parameter and stores it in the `m_value` instance variable. The real purpose of creating this class is the overridden `toString()` method, which inserts commas separating thousands, millions, etc. places. For this purpose we use the `java.text.NumberFormat` class. An instance of this class is created as a static variable, and formatting values using `NumberFormat`'s `format()` method couldn't be easier.

Note: SmartLong cannot extend `Long` (although it would be natural), because `java.lang.Long` is a `final` class.

Class ColorData

This class requires a new constructor to cooperate with the new `Fraction` data type. This third constructor takes a `Fraction` instance as parameter and uses its color attribute in the same way it did previously: green for positive values, and red for negative values.

Class StockData

This class uses new data types for its instance variables. The list of instance variables now looks like the following:

Field	Type	Data object	Description
m_symbol	IconData	String	Stock symbol (NYSE or NASDAQ)
m_name	String	N/A	Company name
m_last	Fraction	N/A	The price of the last trade
m_open	Fraction	N/A	The price at the beginning of the trade day

m_change	ColorData	Fraction	Absolute change in price
m_changePr	ColorData	Double	Percent change in price
m_volume	SmartLong	N/A	Day's volume of trade (in $) for this stock

The StockData constructor is modified accordingly. Meanwhile the parameters in the StockData constructor have not changed, so there is no need to make any changes to the data model class using StockData.

Note. We could use the different approach of just formatting the initial contents of our table's cells into text strings and operating on them without introducing new data classes. This approach, however, is not desirable from a design point of view, as it strips the data of its true nature: numbers. Using our method we are still able to retrieve number values, if necessary, through the doubleValue() or longValue() methods.

Running the Code

Figure 18.3 shows StocksTable in action with number formatting. Presented this way, our data looks much more familiar to most of those who follow the stock trade on a regular basis.

UI Guideline: Talking the User's language In this example, we have changed the rendering of the stock prices into fractions rather than decimals. This is a good example of providing better visual communication by speaking the same language as the viewers. In the North American stock markets, prices are quoted in fractional amounts of dollars rather than dollars and cents. Switching to this type of display helps the application to communicate more quickly and more influentially to the viewer, improving usability.

18.5 StocksTable: part IV - sorting columns

Note: This and the following StocksTable examples require Java 2 as they make use of the new java.util.Collections functionality.

In this section we add the ability to sort any column in ascending or descending order. The most suitable graphical element for selection of sort order are the column headers. We adopt the following model for our sorting functionality:

1. A single click on the header of a certain column causes the table to re-sort based on this column.
2. A repeated click on the same column changes the sort direction from ascending to descending and vice versa.
3. The header of the column which provides the current sorting should be marked.

To do this we add a mouse listener to the table header to capture mouse clicks and trigger a table sort. Fortunately sorting can be accomplished fairly easily using the new Collections functionality in Java 2.

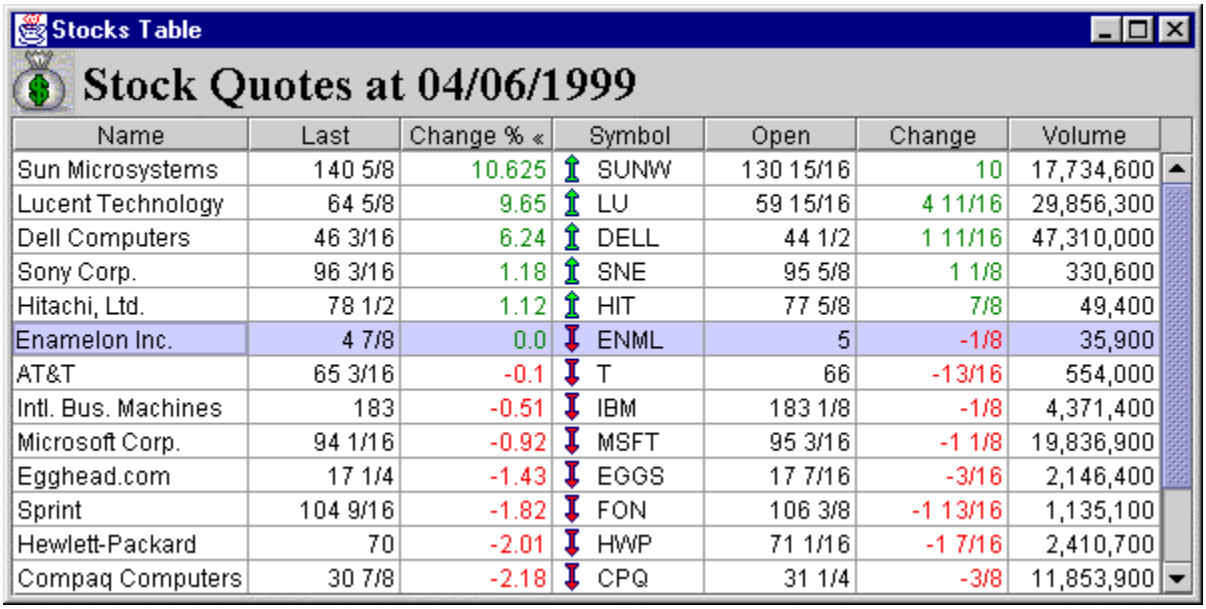
Note: Class java.util.Collections contains a set of static methods used to manipulate Java collections, including java.util.Vector which is used in this example.

We use the java.util.Collections.sort(List lst, Comparator c) method to sort any collection implementing the java.util.List interface based on a given Comparator. A Comparator implementation requires two methods:

int compare(Object o1, Object o2): Compares two objects and returns the result as an int (zero if equal, negative value if the first is less than the second, positive value if the first is more than the

second).

`boolean equals(Object obj)`: Returns `true` if the given object is equal to this `Comparator`.



Name	Last	Change %	Symbol	Open	Change	Volume
Sun Microsystems	140 5/8	10.625	SUNW	130 15/16	10	17,734,600
Lucent Technology	64 5/8	9.65	LU	59 15/16	4 11/16	29,856,300
Dell Computers	46 3/16	6.24	DELL	44 1/2	1 11/16	47,310,000
Sony Corp.	96 3/16	1.18	SNE	95 5/8	1 1/8	330,600
Hitachi, Ltd.	78 1/2	1.12	HIT	77 5/8	7/8	49,400
Enamelon Inc.	4 7/8	0.0	ENML	5	-1/8	35,900
AT&T	65 3/16	-0.1	T	66	-13/16	554,000
Intl. Bus. Machines	183	-0.51	IBM	183 1/8	-1/8	4,371,400
Microsoft Corp.	94 1/16	-0.92	MSFT	95 3/16	-1 1/8	19,836,900
Egghead.com	17 1/4	-1.43	EGGS	17 7/16	-3/16	2,146,400
Sprint	104 9/16	-1.82	FON	106 3/8	-1 13/16	1,135,100
Hewlett-Packard	70	-2.01	HWP	71 1/16	-1 7/16	2,410,700
Compaq Computers	30 7/8	-2.18	CPQ	31 1/4	-3/8	11,853,900

Figure 18.4 JTable with ascending and descending sorting of all columns.
<<file figure18-4.gif>>

The Code: StocksTable.java
see \Chapter18\4

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class StocksTable extends JFrame
{
  // Unchanged code from section 18.4

  public StocksTable() {
    // Unchanged code from section 18.4

    JTableHeader header = m_table.getTableHeader();
    header.setUpdateTableInRealTime(true);
    header.addMouseListener(m_data.new ColumnListener(m_table));
    header.setReorderingAllowed(true);

    // Unchanged code from section 18.4
  }

  public static void main(String argv[]) {
    new StocksTable();
  }
}
```

```
// Unchanged code from section 18.4

class StockTableData extends AbstractTableModel
{
  // Unchanged code from section 18.2

  protected SimpleDateFormat m_frm;
  protected Vector m_vector;
  protected Date   m_date;

  protected int m_sortCol = 0;
  protected boolean m_sortAsc = true;

  public StockTableData() {
    m_frm = new SimpleDateFormat("MM/dd/yyyy");
    m_vector = new Vector();
    setDefaultData();
  }

  public void setDefaultData() {
    // Unchanged code from section 18.4

    Collections.sort(m_vector, new
      StockComparator(m_sortCol, m_sortAsc));
  }

  // Unchanged code from section 18.4

  public String getColumnName(int column) {
    String str = m_columns[column].m_title;
    if (column==m_sortCol)
      str += m_sortAsc ? " »" : " «";
    return str;
  }

  // Unchanged code from section 18.4

  class ColumnListener extends MouseAdapter
  {
    protected JTable m_table;

    public ColumnListener(JTable table) {
      m_table = table;
    }

    public void mouseClicked(MouseEvent e) {
      TableColumnModel colModel = m_table.getColumnModel();
      int columnModelIndex = colModel.getColumnIndexAtX(e.getX());
      int modelIndex = colModel.getColumn(columnModelIndex).getModelIndex();

      if (modelIndex < 0)
        return;
      if (m_sortCol==modelIndex)
        m_sortAsc = !m_sortAsc;
      else
        m_sortCol = modelIndex;

      for (int i=0; i < m_columns.length; i++) {
        TableColumn column = colModel.getColumn(i);
        column.setHeaderValue(getColumnName(column.getModelIndex()));
      }
```

```
      m_table.getTableHeader().repaint();

      Collections.sort(m_vector, new
        StockComparator(modelIndex, m_sortAsc));
      m_table.tableChanged(
        new TableModelEvent(StockTableData.this));
      m_table.repaint();
    }
  }
}

class StockComparator implements Comparator
{
  protected int     m_sortCol;
  protected boolean m_sortAsc;

  public StockComparator(int sortCol, boolean sortAsc) {
    m_sortCol = sortCol;
    m_sortAsc = sortAsc;
  }

  public int compare(Object o1, Object o2) {
    if(!(o1 instanceof StockData) || !(o2 instanceof StockData))
      return 0;
    StockData s1 = (StockData)o1;
    StockData s2 = (StockData)o2;
    int result = 0;
    double d1, d2;
    switch (m_sortCol) {
      case 0:    // symbol
        String str1 = (String)s1.m_symbol.m_data;
        String str2 = (String)s2.m_symbol.m_data;
        result = str1.compareTo(str2);
        break;
      case 1:    // name
        result = s1.m_name.compareTo(s2.m_name);
        break;
      case 2:    // last
        d1 = s1.m_last.doubleValue();
        d2 = s2.m_last.doubleValue();
        result = d1<d2 ? -1 : (d1>d2 ? 1 : 0);
        break;
      case 3:    // open
        d1 = s1.m_open.doubleValue();
        d2 = s2.m_open.doubleValue();
        result = d1<d2 ? -1 : (d1>d2 ? 1 : 0);
        break;
      case 4:    // change
        d1 = ((Fraction)s1.m_change.m_data).doubleValue();
        d2 = ((Fraction)s2.m_change.m_data).doubleValue();
        result = d1<d2 ? -1 : (d1>d2 ? 1 : 0);
        break;
      case 5:    // change %
        d1 = ((Double)s1.m_changePr.m_data).doubleValue();
        d2 = ((Double)s2.m_changePr.m_data).doubleValue();
        result = d1<d2 ? -1 : (d1>d2 ? 1 : 0);
        break;
      case 6:    // volume
        long l1 = s1.m_volume.longValue();
        long l2 = s2.m_volume.longValue();
        result = l1<l2 ? -1 : (l1>l2 ? 1 : 0);
        break;
    }
```

```
    if (!m_sortAsc)
      result = -result;
    return result;
  }

  public boolean equals(Object obj) {
    if (obj instanceof StockComparator) {
      StockComparator compObj = (StockComparator)obj;
      return (compObj.m_sortCol==m_sortCol) &&
        (compObj.m_sortAsc==m_sortAsc);
    }
    return false;
  }
}
```

Understanding the Code

Class StocksTable

In the `StocksTable` constructor we set the `updateTableInRealTime` property to show column contents while columns are dragged, and we add an instance of the `ColumnListener` class (see below) as a mouse listener to the table's header.

Class StockTableData

Here we declare two new instance variables: `int m_sortCol` to hold the index of the current column chosen for sorting, and `boolean m_sortAsc`, which is `true` when sorting in ascending order, and `false` when sorting in descending order. These variables determine the initial sorting order. To be consistent we sort our table initially by calling the `Collections.sort()` method in our `setDefaultData()` method (which is called from the `StockTableData` constructor).

We also add a special marker for the sorting column's header: '»' for ascending and '«' for descending sorting. This changes the way we retrieve a column's name, which no longer is a constant value:

```
  public String getColumnName(int column) {
    String str = m_columns[column].m_title;
    if (column==m_sortCol)
      str += m_sortAsc ? " »" : " «";
    return str;
  }
```

Class StockTableData.ColumnListener

Because this class interacts heavily with our table data, it is implemented as an inner class in `StockTableData`. `ColumnListener` takes a reference to a `JTable` in its constructor and stores that reference in its `m_table` instance variable.

The `mouseClicked()` method is invoked when the user clicks on a header. First it determines the index of the `TableColumn` clicked based on the coordinate of the click. If for any reason the returned index is negative (i.e. the column cannot be determined) the method cannot continue and we return. Otherwise, we check whether this index corresponds to the column which already has been selected for sorting. If so, we invert the `m_sortCol` flag to reverse the sorting order. If the index corresponds to newly selected column we store the new sorting index in the `m_sortCol` variable.

Then we refresh the header names by iterating through the `TableColumns` and assinging them a name corresponding to the column they represent in the `TableModel`. To do this we pass each `TableColumn`'s `modelIndex` property to our `getColumnName()` method (see above). Finally our table data is re-sorted by

calling the `Collections.sort()` method and passing in a new `StockComparator` object. We then refresh the table by calling `tableChanged()` and `repaint()`.

Class StockComparator

This class implements the rule of comparison for two objects, which in our case are `StockDatas`. Instances of the `StockComparator` class are passed to the `Collections.sort()` method to perform data sorting.

Two instance variables are defined:

`int m_sortCol` represents the index of the column which performs the comparison

`boolean m_sortAsc` is `true` for ascending sorting and `false` for descending.

The `StockComparator` constructor takes two parameters and stores them in these instance variables.

The `compare()` method takes two objects to be compared and returns an integer value according to the rules determined in the `Comparator` interface:

`0` if object1 equals 2

A positive number if object1 is greater than object2,

A negative number if object1 is less than object2.

Since we are dealing only with `StockData` objects, first we cast both objects and return `0` if this cast isn't possible. The next issue is to define what it means when one `StockData` objects is greater, equal, or less than another. This is done in a `switch-case` structure, which, depending on the index of the comparison column, extracts two fields and forms an integer result of the comparison. When the `switch-case` structure finishes, we know the result of an ascending comparison. For descending comparison we simply need to invert the sign of the result.

The `equals()` method takes another `Comparator` instance as parameter and returns `true` if that parameter represents the same `Comparator`. We determine this by comparing Comparator instance variables: `m_sortCol` and `m_sortAsc`.

Running the Code

Figure 18.4 shows `StocksTable` sorted by decreasing "Change % :" Click different column headers and note that resorting occurs as expected. Click the same column header twice and note that sorting order flips from ascending to descending and vice versa. Also note that the currently selected sorting column header is marked by the '»' or '«' symbol. This sorting functionality is very useful. Particularly, for stock market data we can instantly determine which stocks have the highest price fluctuations or the most heavy trade.

UI Guideline : Sort by header selection idiom Introducing table sorting using the column headers is introducing another design idiom to the User Interface. This design idiom is becoming widely accepted and widely used in many applications. It is a useful and powerful technique which you can introduce when sorting table data is a requirement. The technique is not intuitive and there is little visual affordance to suggest that clicking a column header will have any effect. So consider that the introduction of this technique may require additional User training.

18.6 StocksTable: part V - JDBC

Despite all of our sorting functionality and enhanced data display, our application is still quite boring because it displays only data for a pre-defined day! Of course, in the real world we need to connect such an application

to the source of fresh information such as a database. Very often tables are used to display data retrieved from databases, or to edit data to be stored in databases. In this section we show how to feed our StocksTable data extracted from a database using the Java Database Connectivity (JDBC) API.

First, we need to create the database. We chose to use two SQL tables (do not confuse SQL table with JTable) whose structure precisely corresponds to the market data structure described in section 18.2:

Table SYMBOLS
Field Name Type
symbol Text
name Text

Table DATA
Field Name Type
symbol Text
date1 Date/Time
last Number
change Number
changeproc Number
open Number
volume Number

For this example we use the JDBC-ODBC bridge which is a standard part of JDK since the 1.1 release and links Java programs to Microsoft Access databases. If you are using another database engine, you can work with this example as well, but you must make sure that the structure of your tables is the same. Before running the example in a Windows environment we need to register a database in an ODBC Data Source Administrator which is accessible through the ControlPanel (this is not a JDBC tutorial, so we'll skip the details).

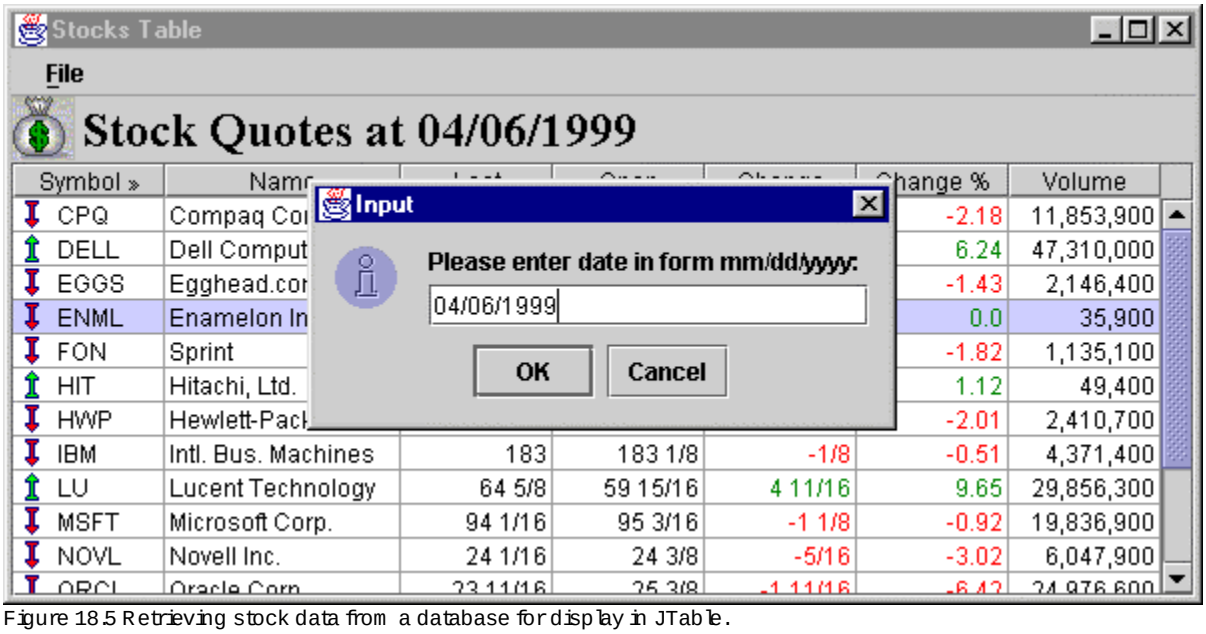


Figure 18.5 Retrieving stock data from a database for display in JTable.
<<file figure18-5.gif>>

The Code: StocksTable.java
see \Chapter18\5

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

```
import java.util.*;
import java.io.*;
import java.text.*;
import java.sql.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class StocksTable extends JFrame
{
  protected JTable m_table;
  protected StockTableData m_data;
  protected JLabel m_title;

  public StocksTable() {
    // Unchanged code from section 18.4

    JMenuBar menuBar = createMenuBar();
      setJMenuBar(menuBar);

    // Unchanged code
  }

  protected JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();

    JMenu mFile = new JMenu("File");
    mFile.setMnemonic('f');

    JMenuItem mData = new JMenuItem("Retrieve Data...");
    mData.setMnemonic('r');
    ActionListener lstData = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        retrieveData();
      }
    };
    mData.addActionListener(lstData);
    mFile.add(mData);
    mFile.addSeparator();

    JMenuItem mExit = new JMenuItem("Exit");
    mExit.setMnemonic('x');
    ActionListener lstExit = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    };
    mExit.addActionListener(lstExit);
    mFile.add(mExit);
    menuBar.add(mFile);

    return menuBar;
  }

  public void retrieveData() {
    SimpleDateFormat frm = new SimpleDateFormat("MM/dd/yyyy");
    String currentDate = frm.format(m_data.m_date);
    String result = (String)JOptionPane.showInputDialog(this,
      "Please enter date in form mm/dd/yyyy:", "Input",
      JOptionPane.INFORMATION_MESSAGE, null, null,
      currentDate);
```

```
    if (result==null)
      return;

    java.util.Date date = null;
    try {
      date = frm.parse(result);
    }
    catch (java.text.ParseException ex) {
      date = null;
    }

    if (date == null) {
      JOptionPane.showMessageDialog(this,
        result+" is not a valid date",
        "Warning", JOptionPane.WARNING_MESSAGE);
      return;
    }

    setCursor( Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR) );
    switch (m_data.retrieveData(date)) {
      case 0:    // Ok with data
        m_title.setText(m_data.getTitle());
        m_table.tableChanged(new TableModelEvent(m_data));
        m_table.repaint();
        break;
      case 1: // No data
        JOptionPane.showMessageDialog(this,
          "No data found for "+result,
          "Warning", JOptionPane.WARNING_MESSAGE);
        break;
      case -1: // Error
        JOptionPane.showMessageDialog(this,
          "Error retrieving data",
          "Warning", JOptionPane.WARNING_MESSAGE);
        break;
    }
    setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
  }

  public static void main(String argv[]) {
    new StocksTable();
  }
}

// Unchanged code from section 18.4

class StockTableData extends AbstractTableModel
{
  static final public ColumnData m_columns[] = {
    // Unchanged code from section 18.2
  };

  protected SimpleDateFormat m_frm;
  protected Vector m_vector;
  protected java.util.Date m_date; // conflict with

  protected int m_sortCol = 0;
  protected boolean m_sortAsc = true;

  protected int m_result = 0;

  public StockTableData() {
```

```
    m_frm = new SimpleDateFormat("MM/dd/yyyy");
    m_vector = new Vector();
    setDefaultData();
  }

  // Unchanged code from section 18.4

  public int retrieveData(final java.util.Date date) {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    int month = calendar.get(Calendar.MONTH)+1;
    int day = calendar.get(Calendar.DAY_OF_MONTH);
    int year = calendar.get(Calendar.YEAR);

    final String query = "SELECT data.symbol, symbols.name, "+
      "data.last, data.open, data.change, data.changeproc, "+
      "data.volume FROM DATA INNER JOIN SYMBOLS "+
      "ON DATA.symbol = SYMBOLS.symbol WHERE "+
      "month(data.date1)="+month+" AND day(data.date1)="+day+
      " AND year(data.date1)="+year;

    Thread runner = new Thread() {
      public void run() {
        try {
          // Load the JDBC-ODBC bridge driver
          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
          Connection conn = DriverManager.getConnection(
            "jdbc:odbc:Market", "admin", "");

          Statement stmt = conn.createStatement();
          ResultSet results = stmt.executeQuery(query);

          boolean hasData = false;
          while (results.next()) {
            if (!hasData) {
              m_vector.removeAllElements();
              hasData = true;
            }
            String  symbol = results.getString(1);
            String  name = results.getString(2);
            double  last = results.getDouble(3);
            double  open = results.getDouble(4);
            double  change = results.getDouble(5);
            double  changePr = results.getDouble(6);
            long volume = results.getLong(7);
            m_vector.addElement(new StockData(symbol, name, last,
              open, change, changePr, volume));
          }
          results.close();
          stmt.close();
          conn.close();

          if (!hasData)    // We've got nothing
            m_result = 1;
        }
        catch (Exception e) {
          e.printStackTrace();
          System.err.println("Load data error: "+e.toString());
          m_result = -1;
        }
        m_date = date;
        Collections.sort(m_vector,
          new StockComparator(m_sortCol, m_sortAsc));
```

```
        m_result = 0;
      }
    };
    runner.start();

    return m_result;
  }

  // Unchanged code from section 18.4
}

// Class StockComparator unchanged from section 18.4
```

Understanding the Code

Class StocksTable

A `JMenuBar` instance is created with our custom `createMenuBar()` method and added to our frame.

The `createMenuBar()` method creates a menu bar containing a single menu titled "File." Two menu items are added: "Retrieve Data..." and "Exit" with a separator in between. Anonymous `ActionListeners` are added to each. The first calls our custom `retrieveData()` method, and the second simply kills the application using `System.exit(0)`.

The `retrieveData()` method is called in response to a "Retrieve Data..." menu item activation. First it prompts the user to enter the date by displaying a `JOptionPane` dialog. Once the date has been entered, this method parses it using a `SimpleDateFormat` object. If the entered string cannot be parsed into a valid date, the method shows a warning message and returns. Otherwise, we connect to JDBC and retrieve new data. To indicate that the program will be busy for some time the wait mouse cursor is displayed. The main job is performed by our new `StockTableData retrieveData()` method (see below), which is invoked on the `m_data` object. `StockTableData`'s `retrieveData()` method returns an error code that the rest of this method depends on:

- `0`: Normal finish, some data retrieved. The table model is updated and repainted.
- `1`: Normal finish, no data retrieved. A warning message is displayed, and no changes in the table model are made.
- `-1`: An error has occurred. An error message is displayed, and no changes in the table model are made.

Class StockTableData

First, a minor change is required in the declaration of the `m_date` variable. Since now we've imported the `java.sql` package, which also includes the `Date` class, we have to do provide the fully qualified calls name `java.util.Date`.

A new instance variable is added to store the result of a data retrieval request in the `retrieveData()` method. As mentioned above, `retrieveData()` retrieves a table's data for a given date of trade. Our implementation uses the JDBC bridge driver and should be familiar to JDBC-aware readers. The first thing we do is construct an SQL statement. Since we cannot compare a `java.util.Date` object and an SQL date stored in the database, we have to extract the date's components (year, month, and day) and compare them separately. An instance of `GregorianCalendar` is used to manipulate the date object.

We load the JDBC-ODBC bridge driver to Microsoft Access by using the `Class.forName` method, and then connect to a database with the `DriverManager.getConnection()` method. If no exception is thrown, we can create a `Statement` instance for the newly created `Connection` object and retrieve a `ResultSet` by executing the previously constructed query.

While new data is available (checked with the `ResultSet.next()` method), we retrieve new data using basic `getXX()` methods, create a new `StockData` instance to encapsulate the new data, and add it to `m_vector`.

Note how the `hasData` local variable is used to distinguish the case in which we do not have any data in our `RecordSet`. The first time we receive some valid data from our `RecordSet` in the `while` loop, we set this variable to `true` and clean up our `m_vector` collection. If no data is found, we have an unchanged initial vector and the `hasData` flag is set to `false`. Finally we close our `ResultSet`, `Statement`, and `Connection` instances. If any exception occurs, the method prints the exception trace and returns a `-1` to indicate an error. Otherwise our newly retrieved data is sorted with the `Collections.sort()` method and a `0` is returned to indicate success.

Running the Code

Figure 18.6 shows `StocksTable` with data retrieved from a database. Try loading data for different dates in your database. A sample Microsoft Access database, `marketmdb`, containing some real market data, can be found in the `\swing\Chapter18` directory.

18.7 StocksTable: part VI - column addition and removal

`JTable` allows us to dynamically add and remove `TableColumns` on the fly. Recall that the `TableColumnModel` interface provides the methods `addColumn()` and `removeColumn()` to programmatically add or remove a `TableColumn` respectively. In this section we add dynamic column addition and removal to our `StocksTable` application.

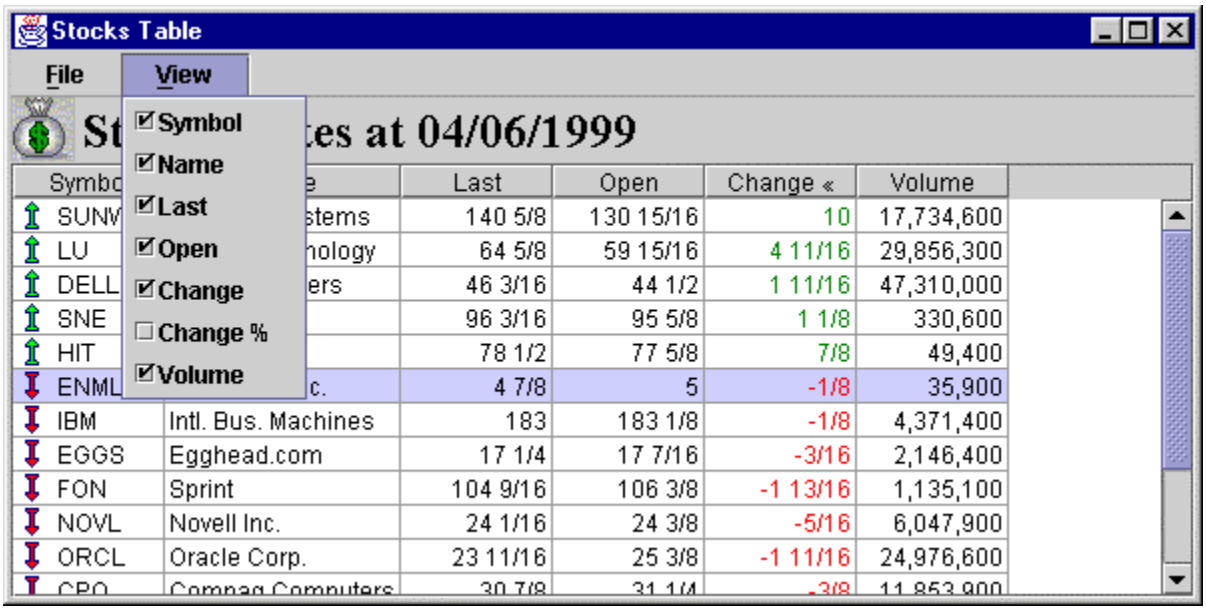


Figure 18.6 JTable with dynamic column addition and removal.

<<file figure18-6.gif>>

The Code: StocksTable.java

see `\Chapter18\6`

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;
```

```
import java.sql.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class StocksTable extends JFrame
{
  // Unchanged code from section 18.5

  public StocksTable() {
    // Unchanged code from section 18.5
    header.setReorderingAllowed(true);

    m_table.getColumnModel().addColumnModelListener(
      m_data.new ColumnMovementListener());

    // Unchanged code from section 18.5
  }

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 18.5

    JMenu mView = new JMenu("View");
    mView.setMnemonic('v');
    TableColumnModel model = m_table.getColumnModel();
    for (int k = 0; k < StockTableData.m_columns.length; k++) {
      JCheckBoxMenuItem item = new JCheckBoxMenuItem(
        StockTableData.m_columns[k].m_title);
      item.setSelected(true);
      TableColumn column = model.getColumn(k);
      item.addActionListener(new ColumnKeeper(column,
        StockTableData.m_columns[k]));
      mView.add(item);
    }
    menuBar.add(mView);

    return menuBar;
  }

  // Unchanged code from section 18.5

  class ColumnKeeper implements ActionListener
  {
    protected TableColumn m_column;
    protected ColumnData  m_colData;

    public ColumnKeeper(TableColumn column, ColumnData  colData) {
      m_column = column;
      m_colData = colData;
    }

    public void actionPerformed(ActionEvent e) {
      JCheckBoxMenuItem item = (JCheckBoxMenuItem)e.getSource();
      TableColumnModel model = m_table.getColumnModel();
      if (item.isSelected()) {
        model.addColumn(m_column);
      }
      else {
        model.removeColumn(m_column);
      }
```

```
      m_table.tableChanged(new TableModelEvent(m_data));
      m_table.repaint();
    }
  }

  public static void main(String argv[]) {
    new StocksTable();
  }
}

// Unchanged code from section 18.5

class StockTableData extends AbstractTableModel
{
  // Unchanged code from section 18.5

  protected SimpleDateFormat m_frm;
  protected Vector m_vector;
  protected java.util.Date m_date;
  protected int m_columnsCount = m_columns.length;

  // Unchanged code from section 18.5

  public int getColumnCount() {
    return m_columnsCount;
  }

  // Unchanged code from section 18.5

  class ColumnListener extends MouseAdapter
  {
    // Unchanged code from section 18.4

    public void mouseClicked(MouseEvent e) {
      // Unchanged code from section 18.4

      for (int i=0; i < m_columnsCount; i++) {
        TableColumn column = colModel.getColumn(i);
        column.setHeaderValue(getColumnName(column.getModelIndex()));
      }
      m_table.getTableHeader().repaint();

      // Unchanged code from section 18.4
    }
  }

  class ColumnMovementListener implements TableColumnModelListener
  {
    public void columnAdded(TableColumnModelEvent e) {
      m_columnsCount++;
    }

    public void columnRemoved(TableColumnModelEvent e) {
      m_columnsCount--;
    }

    public void columnMarginChanged(ChangeEvent e) {}
    public void columnMoved(TableColumnModelEvent e) {}
    public void columnSelectionChanged(ListSelectionEvent e) {}
  }
  // Unchanged code from section 18.5
}
```

```
// Class StockComparator unchanged from section 18.4
```

Understanding the Code

Class StocksTable

The `StocksTable` constructor now adds an instance of `StockTableData.ColumnMovementListener` (see below) to our table's `TableColumnModel` to listen for column additions and removals.

Our `createMenuBar()` method now adds several checkbox menu items to a new "View" menu -- one for each column. Each of these check box menu items recieves a `ColumnKeeper` instance (see below) as `ActionListener`.

Class StocksTable.ColumnKeeper

This inner class implements the `ActionListener` interface and serves to keep track of when the user removes and adds columns to the table. The constructor receives a `TableColumn` instance and a `ColumnData` object. The `actionPerformed()` method adds this column to the model with the `addColumn()` method if the corresponding menu item is checked, and removes this column from the model with `removeColumn()` if it is unchecked. To update the table to properly reflect these changes, we call its `tableChanged()` method followed by a `repaint()` request.

Class StockTableData

`StockTableData` now contains instance variable `m_columnsCount` to keep track of the current column count. This variable is decremented and incremented in the `columnRemoved()` and `columnAdded()` methods of inner class `ColumnMovementListener`. It is also used in the `StockTableData.ColumnListener` class's `mouseClicked()` method for properly setting header values for the visible columns only.

Class StockTableData.ColumnMovementListener

This class implements `TableColumnModelListener` to increment and decrement `StockTableData`'s `m_columnsCount` variable when a column addition or removal occurs, respectively. An instance of this inner class is added to our table's `TableColumnModel` in the `StocksTable` constructor.

Running the Code

Figure 18.7 shows the new "View" menu with an unchecked "Change %" menu item, and the corresponding column hidden. Reselecting this menu item will place the column back in the table at the end position. Verify that each menu item functions similarly.

18.8 Custom models, editors, and renderers

In constructing our `StocksTable` application we talked mostly about displaying and retrieving data in `JTable`. In this section we will construct a basic expense report application, and in doing so we will concentrate on table cell editing. We will also see how to implement dynamic addition and removal of table rows.

The editing of data generally follows this scheme:

- Create an instance of the `TableCellEditor` interface. We can use the `DefaultCellEditor` class or implement our own. The `DefaultCellEditor` class takes a GUI component as a parameter to its constructor: `JTextField`, `JCheckBox` or `JComboBox`. This component will be used for editing.
- If we are developing a custom editor, we need to implement the `getTableCellEditorComponent()`

method which will be called each time a cell is about to be edited.

In our table model we need to implement the `setValueAt(Object value, int nRow, int nCol)` method which will be called to change a value in the table when an edit ends. This is where we can perform any necessary data processing and validation.

The data model for this example is designed as follows (where each row represents a column in our `JTable`):

Name	Type	Description
Date	`String`	Date of expense
Amount	`Double`	Amount of expense
Category	`Integer`	Category from pre-defined list
Approved	`Boolean`	Sign of approval for this expense.
Description	`String`	Brief description

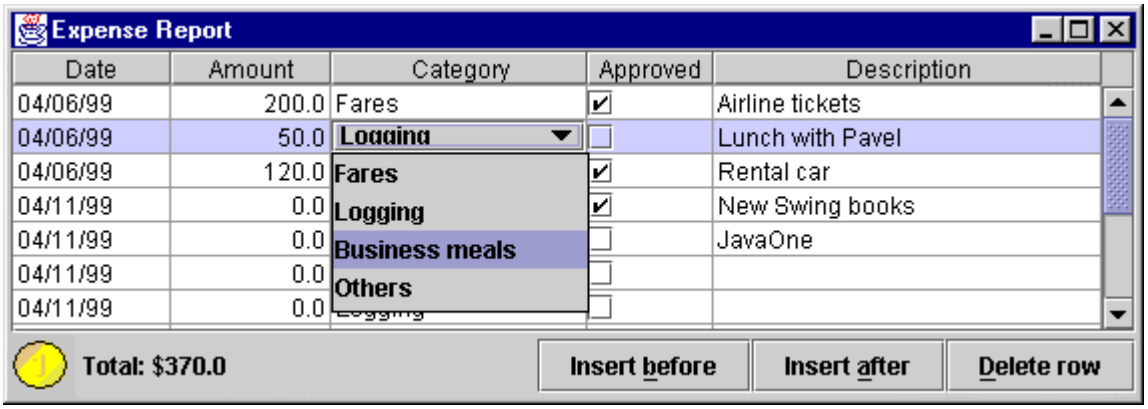


Figure 18.7 An expense report app illustrating custom cell editing, rendering, and row addition/removal.
<<file figure18-7.gif>>

Note: Since the only math that is done with our "Amount" values is addition, using `Doubles` is fine. However, in more professional implementations we may need to use rounding techniques or a custom renderer to remove unneccessary fractional amounts.

The Code: ExpenseReport.java
see \Chapter18\7

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.SimpleDateFormat;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class ExpenseReport extends JFrame
{
  protected JTable m_table;
  protected ExpenseReportData m_data;
  protected JLabel m_title;

  public ExpenseReport() {
    super("Expense Report");
```

```
    setSize(570, 200);

    m_data = new ExpenseReportData(this);

    m_table = new JTable();
    m_table.setAutoCreateColumnsFromModel(false);
    m_table.setModel(m_data);
    m_table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    for (int k = 0; k < ExpenseReportData.m_columns.length; k++) {
      TableCellRenderer renderer;
      if (k==ExpenseReportData.COL_APPROVED)
        renderer = new CheckCellRenderer();
      else {
        DefaultTableCellRenderer textRenderer =
          new DefaultTableCellRenderer();
        textRenderer.setHorizontalAlignment(
          ExpenseReportData.m_columns[k].m_alignment);
        renderer = textRenderer;
      }

      TableCellEditor editor;

      if (k==ExpenseReportData.COL_CATEGORY)
        editor = new DefaultCellEditor(new JComboBox(
          ExpenseReportData.CATEGORIES));
      else if (k==ExpenseReportData.COL_APPROVED)
        editor = new DefaultCellEditor(new JCheckBox());
      else
        editor = new DefaultCellEditor(new JTextField());

      TableColumn column = new TableColumn(k,
        ExpenseReportData.m_columns[k].m_width,
          renderer, editor);
        m_table.addColumn(column);
    }

    JTableHeader header = m_table.getTableHeader();
    header.setUpdateTableInRealTime(false);

    JScrollPane ps = new JScrollPane();
    ps.setSize(550, 150);
    ps.getViewport().add(m_table);
    getContentPane().add(ps, BorderLayout.CENTER);

    JPanel p = new JPanel();
    p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));

    ImageIcon penny = new ImageIcon("penny.gif");
    m_title = new JLabel("Total: $",
      penny, JButton.LEFT);
    m_title.setForeground(Color.black);
    m_title.setAlignmentY(0.5f);
    p.add(m_title);
    p.add(Box.createHorizontalGlue());

    JButton bt = new JButton("Insert before");
    bt.setMnemonic('b');
    bt.setAlignmentY(0.5f);
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int row = m_table.getSelectedRow();
        m_data.insert(row);
```

```
      m_table.tableChanged(new TableModelEvent(
        m_data, row, row, TableModelEvent.ALL_COLUMNS,
        TableModelEvent.INSERT));
      m_table.repaint();
    }
  };
  bt.addActionListener(lst);
  p.add(bt);

  bt = new JButton("Insert after");
  bt.setMnemonic('a');
  bt.setAlignmentY(0.5f);
  lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      int row = m_table.getSelectedRow();
      m_data.insert(row+1);
      m_table.tableChanged(new TableModelEvent(
        m_data, row+1, row+1, TableModelEvent.ALL_COLUMNS,
        TableModelEvent.INSERT));
      m_table.repaint();
    }
  };
  bt.addActionListener(lst);
  p.add(bt);

  bt = new JButton("Delete row");
  bt.setMnemonic('d');
  bt.setAlignmentY(0.5f);
  lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      int row = m_table.getSelectedRow();
      if (m_data.delete(row)) {
        m_table.tableChanged(new TableModelEvent(
          m_data, row, row, TableModelEvent.ALL_COLUMNS,
          TableModelEvent.INSERT));
        m_table.repaint();
        calcTotal();
      }
    }
  };
  bt.addActionListener(lst);
  p.add(bt);

  getContentPane().add(p, BorderLayout.SOUTH);

  calcTotal();

  WindowListener wndCloser = new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  };
  addWindowListener(wndCloser);

  setVisible(true);
}

public void calcTotal() {
  double total = 0;
  for (int k=0; k<m_data.getRowCount(); k++) {
    Double amount = (Double)m_data.getValueAt(k,
      ExpenseReportData.COL_AMOUNT);
    total += amount.doubleValue();
```

```
    }
    m_title.setText("Total: $"+total);
  }

  public static void main(String argv[]) {
    new ExpenseReport();
  }
}

class CheckCellRenderer extends JCheckBox implements TableCellRenderer
{
  protected static Border m_noFocusBorder;

  public CheckCellRenderer() {
    super();
    m_noFocusBorder = new EmptyBorder(1, 2, 1, 2);
    setOpaque(true);
    setBorder(m_noFocusBorder);
  }

  public Component getTableCellRendererComponent(JTable table,
   Object value, boolean isSelected, boolean hasFocus,
   int row, int column)
  {
    if (value instanceof Boolean) {
      Boolean b = (Boolean)value;
      setSelected(b.booleanValue());
    }

    setBackground(isSelected && !hasFocus ?
      table.getSelectionBackground() : table.getBackground());
    setForeground(isSelected && !hasFocus ?
      table.getSelectionForeground() : table.getForeground());

    setFont(table.getFont());
    setBorder(hasFocus ? UIManager.getBorder(
      "Table.focusCellHighlightBorder") : m_noFocusBorder);

    return this;
  }
}

class ExpenseData
{
  public Date    m_date;
  public Double  m_amount;
  public Integer m_category;
  public Boolean m_approved;
  public String  m_description;

  public ExpenseData() {
    m_date = new Date();
    m_amount = new Double(0);
    m_category = new Integer(1);
    m_approved = new Boolean(false);
    m_description = "";
  }

  public ExpenseData(Date date, double amount, int category,
   boolean approved, String description)
  {
    m_date = date;
    m_amount = new Double(amount);
```

```
    m_category = new Integer(category);
    m_approved = new Boolean(approved);
    m_description = description;
  }
}

class ColumnData
{
  public String  m_title;
  int m_width;
  int m_alignment;

  public ColumnData(String title, int width, int alignment) {
    m_title = title;
    m_width = width;
    m_alignment = alignment;
  }
}

class ExpenseReportData extends AbstractTableModel
{
  public static final ColumnData m_columns[] = {
    new ColumnData( "Date", 80, JLabel.LEFT ),
    new ColumnData( "Amount", 80, JLabel.RIGHT ),
    new ColumnData( "Category", 130, JLabel.LEFT ),
    new ColumnData( "Approved", 80, JLabel.LEFT ),
    new ColumnData( "Description", 180, JLabel.LEFT )
  };

  public static final int COL_DATE = 0;
  public static final int COL_AMOUNT = 1;
  public static final int COL_CATEGORY = 2;
  public static final int COL_APPROVED = 3;
  public static final int COL_DESCR = 4;

  public static final String[] CATEGORIES = {
    "Fares", "Logging", "Business meals", "Others"
  };

  protected ExpenseReport m_parent;
  protected SimpleDateFormat m_frm;
  protected Vector m_vector;

  public ExpenseReportData(ExpenseReport parent) {
    m_parent = parent;
    m_frm = new SimpleDateFormat("MM/dd/yy");
    m_vector = new Vector();
    setDefaultData();
  }

  public void setDefaultData() {
    m_vector.removeAllElements();
    try {
      m_vector.addElement(new ExpenseData(
      m_frm.parse("04/06/99"), 200, 0, true,
        "Airline tickets"));
      m_vector.addElement(new ExpenseData(
        m_frm.parse("04/06/99"), 50,  2, false,
        "Lunch with client"));
      m_vector.addElement(new ExpenseData(
        m_frm.parse("04/06/99"), 120, 1, true,
        "Hotel"));
    }
```

```
    catch (java.text.ParseException ex) {}
  }

  public int getRowCount() {
    return m_vector==null ? 0 : m_vector.size();
  }

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

  public String getColumnName(int column) {
    return m_columns[column].m_title;
  }

  public boolean isCellEditable(int nRow, int nCol) {
    return true;
  }

  public Object getValueAt(int nRow, int nCol) {
    if (nRow < 0 || nRow>=getRowCount())
      return "";
    ExpenseData row = (ExpenseData)m_vector.elementAt(nRow);
    switch (nCol) {
      case COL_DATE: return m_frm.format(row.m_date);
      case COL_AMOUNT: return row.m_amount;
      case COL_CATEGORY: return CATEGORIES[row.m_category.intValue()];
      case COL_APPROVED: return row.m_approved;
      case COL_DESCR: return row.m_description;
    }
    return "";
  }

  public void setValueAt(Object value, int nRow, int nCol) {
    if (nRow < 0 || nRow>=getRowCount())
      return;
    ExpenseData row = (ExpenseData)m_vector.elementAt(nRow);
    String svalue = value.toString();

    switch (nCol) {
      case COL_DATE:
        Date  date = null;
        try {
          date = m_frm.parse(svalue);
        }
        catch (java.text.ParseException ex) {
          date = null;
        }
        if (date == null) {
          JOptionPane.showMessageDialog(null,
            svalue+" is not a valid date",
            "Warning", JOptionPane.WARNING_MESSAGE);
          return;
        }
        row.m_date = date;
        break;
      case COL_AMOUNT:
        try {
          row.m_amount = new Double(svalue);
        }
        catch (NumberFormatException e) { break; }
        m_parent.calcTotal();
        break;
```

```
      case COL_CATEGORY:
        for (int k=0; k<CATEGORIES.length; k++)
          if (svalue.equals(CATEGORIES[k])) {
            row.m_category = new Integer(k);
            break;
          }
        break;
      case COL_APPROVED:
        row.m_approved = (Boolean)value;
        break;
      case COL_DESCR:
        row.m_description = svalue;
        break;
    }
  }

  public void insert(int row) {
    if (row < 0)
      row = 0;
    if (row > m_vector.size())
      row = m_vector.size();
    m_vector.insertElementAt(new ExpenseData(), row);
  }

  public boolean delete(int row) {
    if (row < 0 || row >= m_vector.size())
      return false;
    m_vector.remove(row);
      return true;
  }
}
```

Understanding the Code

Class ExpenseReport

Class `ExpenseReport` extends `JFrame` and defines three instance variables:

`JTable m_table`: table to edit data.

`ExpenseReportData m_data`: data model for this table.

`JLabel m_total`: label to dynamically display total amount of expenses.

The `ExpenseReport` constructor first instantiates our table model, `m_data`, and then instantiates our table, `m_table`. The selection mode is set to single selection and we iterate through the number of columns creating cell renderers and editors based on each specific column. The "Approved" column uses an instance of our custom `CheckCellRenderer` class as renderer. All other columns use a `DefaultTableCellRenderer`. All columns also use a `DefaultCellEditor`. However, the component used for editing varies: the "Category" column uses a `JComboBox`, the "Approved" column uses a `JCheckBox`, and all other columns use a `JTextField`. These components are passed to the `DefaultTableCellRenderer` constructor.

Several components are added to the bottom of our frame: `JLabel m_total`, used to display the total amount of expenses, and three `JButtons` used to manipulate tables rows. (Note that the horizontal glue component added between the label and the button pushes buttons to the right side of the panel, so they remain glued to the right when our frame is resized.)

These three buttons, titled "Insert before," "Insert after," and "Delete row," behave as their titles imply. The first two use the `insert()` method from the `ExpenseReportData` model to insert a new row before or after the currently selected row. The last one deletes the currently selected row by calling the `delete()`

method. In all cases the modified table is updated and repainted.

Method `calcTotal()` calculates the total amount of expenses in column `COL_AMOUNT` using our table's data model, `m_data`.

Class CheckCellRenderer

Since we use check boxes to edit our table's "Approved" column, to be consistent we also need to use check boxes for that column's cell renderer (recall that cell renderers just act as "rubber stamps" and are not at all interactive). The only GUI component which can be used in the existing `DefaultTableCellRenderer` is `JLabel`, so we have to provide our own implementation of the `TableCellRenderer` interface. This class, `CheckCellRenderer`, uses `JCheckBox` as a super-class. Its constructor sets the border to indicate whether the component has the focus and sets its opaque property to `true` to indicate that the component's background will be filled with the background color.

The only method which must be implemented in the `TableCellRenderer` interface is `getTableCellRendererComponent()`. This method will be called each time the cell is about to be rendered to deliver new data to the renderer. It takes six parameters:

- `JTable table`: reference to table instance.
- `Object value`: data object to be sent to the renderer.
- `boolean isSelected`: `true` if the cell is currently selected.
- `boolean hasFocus`: `true` if the cell currently has the focus.
- `int row`: cell's row.
- `int column`: cell's column.

Our implementation sets whether the `JCheckBox` is checked depending on the `value` passed as `Boolean`. Then it sets the background, foreground, font, and border to ensure that each cell in the table has a similar appearance.

Class ExpenseData

Class `ExpenseData` represents a single row in the table. It holds five variables corresponding to our data structure described in the beginning of this section.

Class ColumnData

Class `ColumnData` holds each column's title, width, and header alignment.

Class ExpenseReportData

`ExpenseReportData` extends `AbstractTableModel` and should look somewhat familiar from previous examples in this chapter (e.g. `StockTableData`), so we will not discuss this class in complete detail. However, we need to take a closer look at the `setValueAt()` method, which is new for this example (all previous examples did not accept new data). This method is called each time an edit is made to a table cell. First we determine which `ExpenseData` instance (table's row) is affected, and if it is invalid we simply return. Otherwise, depending on the column of the changed cell, we define several cases in a `switch` structure to accept and store a new value, or to reject it:

- For the "Date" column the input string is parsed using our `SimpleDateFormat` instance. If parsing is successful, a new date is saved as a `Date` object, otherwise an error message is displayed.
- For the "Amount" column the input string is parsed as a `Double` and stored in the table if parsing is successful. Also a new total amount is recalculated and displayed in the "Total" `JLabel`.

For the "Category" column the input string is placed in the `CATEGORIES` array at the corresponding index and is stored in the table model.

For the "Approved" column the input object is cast to a `Boolean` and stored in the table model.

For the "Description" column the input string is directly saved in our table model.

Running the Code

Try editing different columns and note how the corresponding cell editors work. Experiment with adding and removing table rows and note how the total amount is updated each time the "Amount" column is updated. Figure 18.8 shows `ExpenseReport` with a combo box opened to change a cell's value.

18.9 A JavaBeans property editor

Now that we're familiar with the table API we can complete the JavaBeans container introduced in the chapter 4 and give it the capability to edit the properties of JavaBeans. This dramatically increases the possible uses of our simple container and makes it a quite powerful tool for studying JavaBeans.

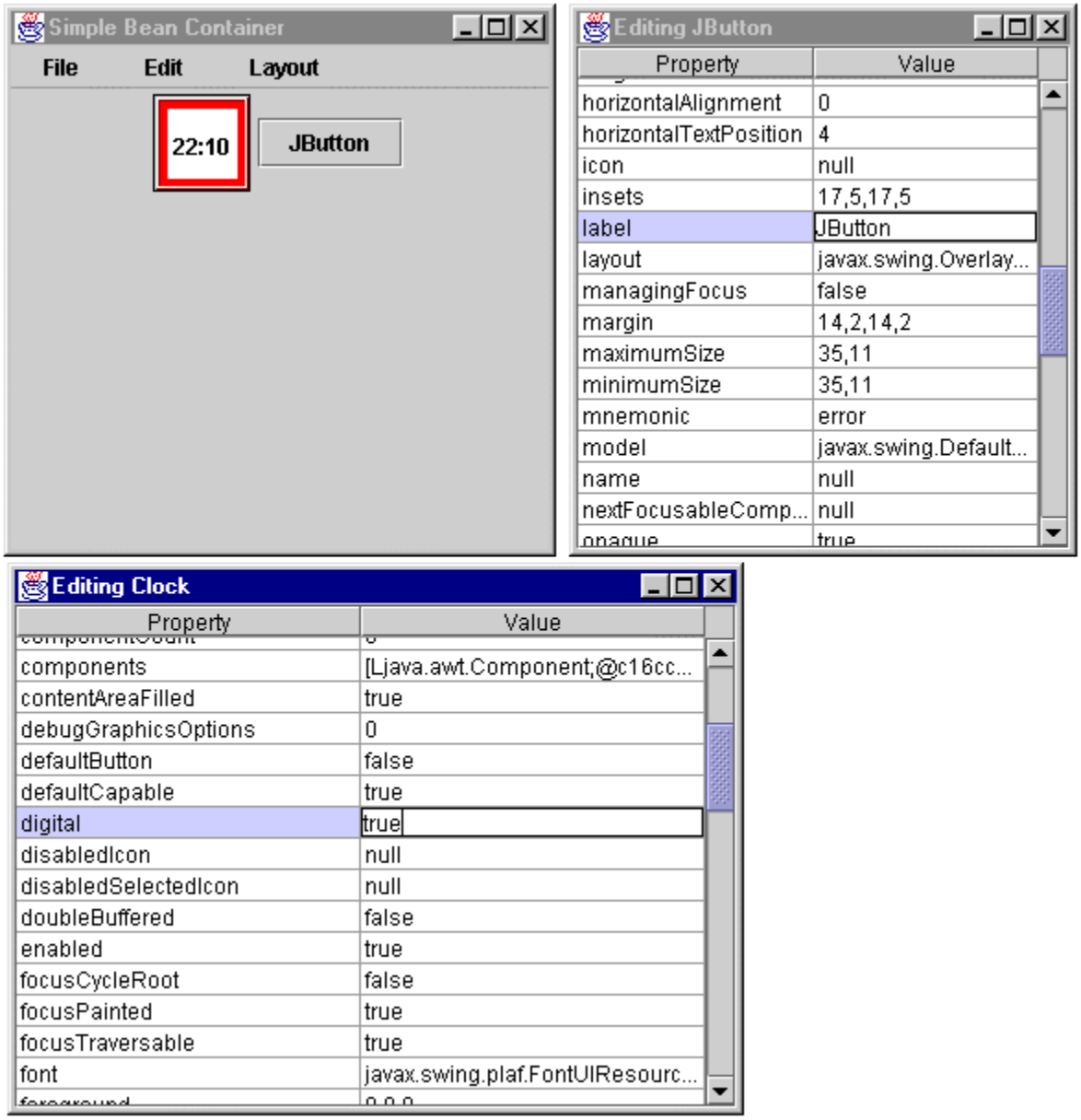


Figure 18.8 BeanContainer JavaBeans property editor using JTables as editing forms.
<<file figure18-8.gif>>

The Code: BeanContainer.java
see \Chapter18\8

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.beans.*;
import java.lang.reflect.*;
import java.util.*;

import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;

import dl.*;
```

```
public class BeanContainer extends JFrame implements FocusListener
{
  protected Hashtable m_editors = new Hashtable();

  // Unchanged code from section 4.7

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 4.7

    JMenu mEdit = new JMenu("Edit");
    mItem = new JMenuItem("Delete");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if (m_activeBean == null)
          return;
        Object obj = m_editors.get(m_activeBean);
        if (obj != null) {
          BeanEditor editor = (BeanEditor)obj;
          editor.dispose();
          m_editors.remove(m_activeBean);
        }
        getContentPane().remove(m_activeBean);
        m_activeBean = null;
        validate();
        repaint();
      }
    };
    mItem.addActionListener(lst);
    mEdit.add(mItem);

    mItem = new JMenuItem("Properties...");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if (m_activeBean == null)
          return;
        Object obj = m_editors.get(m_activeBean);
        if (obj != null) {
          BeanEditor editor = (BeanEditor)obj;
          editor.setVisible(true);
          editor.toFront();
        }
        else {
          BeanEditor editor = new BeanEditor(m_activeBean);
          m_editors.put(m_activeBean, editor);
        }
      }
    };
    mItem.addActionListener(lst);
    mEdit.add(mItem);
    menuBar.add(mEdit);

    // Unchanged code from section 4.7

    return menuBar;
  }

  // Unchanged code from section 4.7
}

class BeanEditor extends JFrame implements PropertyChangeListener
{
  protected Component m_bean;
```

```
  protected JTable m_table;
  protected PropertyTableData m_data;

  public BeanEditor(Component bean) {
    m_bean = bean;
    m_bean.addPropertyChangeListener(this);

    Point pt = m_bean.getLocationOnScreen();
    setBounds(pt.x+50, pt.y+10, 400, 300);
    getContentPane().setLayout(new BorderLayout());

    m_data = new PropertyTableData(m_bean);
    m_table = new JTable(m_data);

    JScrollPane ps = new JScrollPane();
    ps.getViewport().add(m_table);
    getContentPane().add(ps, BorderLayout.CENTER);

    setDefaultCloseOperation(HIDE_ON_CLOSE);
    setVisible(true);
  }

  public void propertyChange(PropertyChangeEvent evt) {
    m_data.setProperty(evt.getPropertyName(), evt.getNewValue());
  }

  class PropertyTableData extends AbstractTableModel
  {
    protected String[][] m_properties;
    protected int m_numProps = 0;
    protected Vector m_v;

    public PropertyTableData(Component bean) {
      try {
        BeanInfo info = Introspector.getBeanInfo(
          m_bean.getClass());
        BeanDescriptor descr = info.getBeanDescriptor();
        setTitle("Editing "+descr.getName());
        PropertyDescriptor[] props = info.getPropertyDescriptors();
        m_numProps = props.length;

        m_v = new Vector(m_numProps);
        for (int k=0; k<m_numProps; k++) {
          String name = props[k].getDisplayName();
          boolean added = false;
          for (int i=0; i<m_v.size(); i++) {
            String str = ((PropertyDescriptor)m_v.elementAt(i)).
              getDisplayName();
            if (name.compareToIgnoreCase(str) < 0) {
              m_v.insertElementAt(props[k], i);
              added = true;
              break;
            }
          }
          if (!added)
            m_v.addElement(props[k]);
        }

        m_properties = new String[m_numProps][2];
        for (int k=0; k<m_numProps; k++) {
          PropertyDescriptor prop =
            (PropertyDescriptor)m_v.elementAt(k);
          m_properties[k][0] = prop.getDisplayName();
```

```
      Method mRead = prop.getReadMethod();
      if (mRead != null &&
       mRead.getParameterTypes().length == 0) {
        Object value = mRead.invoke(m_bean, null);
        m_properties[k][1] = objToString(value);
      }
      else
        m_properties[k][1] = "error";
    }
  }
  catch (Exception ex) {
    ex.printStackTrace();
    JOptionPane.showMessageDialog(
      BeanEditor.this, "Error: "+ex.toString(),
      "Warning", JOptionPane.WARNING_MESSAGE);
  }
}

public void setProperty(String name, Object value) {
  for (int k=0; k<m_numProps; k++)
    if (name.equals(m_properties[k][0])) {
      m_properties[k][1] = objToString(value);
      m_table.tableChanged(new TableModelEvent(this, k));
      m_table.repaint();
      break;
    }
}

public int getRowCount() { return m_numProps; }

public int getColumnCount() { return 2; }

public String getColumnName(int nCol) {
  return nCol==0 ? "Property" : "Value";
}

public boolean isCellEditable(int nRow, int nCol) {
   return (nCol==1);
}

public Object getValueAt(int nRow, int nCol) {
  if (nRow < 0 || nRow>=getRowCount())
    return "";
  switch (nCol) {
    case 0: return m_properties[nRow][0];
    case 1: return m_properties[nRow][1];
  }
  return "";
}

public void setValueAt(Object value, int nRow, int nCol) {
  if (nRow < 0 || nRow>=getRowCount())
    return;
  String str = value.toString();
  PropertyDescriptor prop = (PropertyDescriptor)m_v.
    elementAt(nRow);
  Class cls = prop.getPropertyType();
  Object obj = stringToObj(str, cls);
  if (obj==null)
    return;        // can't process

  Method mWrite = prop.getWriteMethod();
  if (mWrite == null || mWrite.getParameterTypes().length != 1)
```

```
      return;
    try {
      mWrite.invoke(m_bean, new Object[]{ obj });
      m_bean.getParent().doLayout();
      m_bean.getParent().repaint();
      m_bean.repaint();
    }
    catch (Exception ex) {
      ex.printStackTrace();
      JOptionPane.showMessageDialog(
        BeanEditor.this, "Error: "+ex.toString(),
        "Warning", JOptionPane.WARNING_MESSAGE);
    }
    m_properties[nRow][1] = str;
  }

  public String objToString(Object value) {
    if (value==null)
      return "null";
    if (value instanceof Dimension) {
      Dimension dim = (Dimension)value;
      return ""+dim.width+","+dim.height;
    }
    else if (value instanceof Insets) {
      Insets ins = (Insets)value;
      return ""+ins.left+","+ins.top+","+ins.right+","+ins.bottom;
    }
    else if (value instanceof Rectangle) {
      Rectangle rc = (Rectangle)value;
      return ""+rc.x+","+rc.y+","+rc.width+","+rc.height;
    }
    else if (value instanceof Color) {
      Color col = (Color)value;
      return ""+col.getRed()+","+col.getGreen()+","+col.getBlue();
    }
    return value.toString();
  }

  public Object stringToObj(String str, Class cls) {
    try {
      if (str==null)
        return null;
      String name = cls.getName();
      if (name.equals("java.lang.String"))
        return str;
      else if (name.equals("int"))
        return new Integer(str);
      else if (name.equals("long"))
        return new Long(str);
      else if (name.equals("float"))
        return new Float(str);
      else if (name.equals("double"))
        return new Double(str);
      else if (name.equals("boolean"))
        return new Boolean(str);
      else if (name.equals("java.awt.Dimension")) {
        int[] i = strToInts(str);
        return new Dimension(i[0], i[1]);
      }
      else if (name.equals("java.awt.Insets")) {
        int[] i = strToInts(str);
        return new Insets(i[0], i[1], i[2], i[3]);
      }
```

```
        else if (name.equals("java.awt.Rectangle")) {
          int[] i = strToInts(str);
          return new Rectangle(i[0], i[1], i[2], i[3]);
        }
        else if (name.equals("java.awt.Color")) {
          int[] i = strToInts(str);
          return new Color(i[0], i[1], i[2]);
        }
        return null;    // not supported
      }
      catch(Exception ex) { return null; }
    }

    public int[] strToInts(String str) throws Exception {
      int[] i = new int[4];
      StringTokenizer tokenizer = new StringTokenizer(str, ",");
      for (int k=0; k<i.length &&
       tokenizer.hasMoreTokens(); k++)
        i[k] = Integer.parseInt(tokenizer.nextToken());
      return i;
    }
  }
}
```

Understanding the Code

Class BeanContainer

This class (formerly `BeanContainer` from section 4.7) has received a new collection, `Hashtable m_editors`, added as an instance variable. This `Hashtable` holds references to `BeanEditor` frames (used to edit beans, see below) as values and the corresponding `Components` being edited as keys.

A new menu item titled "Properties..." is added to the "Edit" menu. This item is used to create a new editor for the selected bean or activate an existing one (if any). The attached `ActionListener` looks for an existing `BeanEditor` corresponding to the currently selected `m_activeBean` component in the `m_editors` collection. If such an editor is found it is made visible and brought to the front. Otherwise, a new instance of `BeanEditor` is created to edit the currently active `m_activeBean` component, and is added to the `m_editors` collection.

The `ActionListener` attached to menu item "Delete," which removes the currently active component, receives additional functionality. The added code looks for an existing `BeanEditor` corresponding to the currently selected `m_activeBean` component in the `m_editors` collection. If such an editor is found it is disposed and its' reference is removed from the hashtable.

Class BeanEditor

This class extends `JFrame` and implements the `PropertyChangeListener` interface. `BeanEditor` is used to display and edit the properties exposed by a given JavaBean. Three instance variables are declared:

`Component m_bean`: JavaBean component to be edited.

`JTable m_table`: table component to display a bean's properties.

`PropertyTableData m_data`: table model for `m_table`.

The `BeanEditor` constructor takes a reference to the JavaBean component to be edited, and stores it in instance variable `m_bean`. The initial location of the editor frame is selected depending on the location of the component being edited.

The table component, `m_table`, is created and added to a `JScrollPane` to provide scrolling capabilities. Note that we do not add a `WindowListener` to this frame. Instead we use the `HIDE_ON_CLOSE` default close operation (see chapter 3):

```
setDefaultCloseOperation(HIDE_ON_CLOSE);
setVisible(true);
```

Upon closing, this frame will be hidden but not disposed. Its' reference will still be present in the `m_editors` collection, and this frame will be re-activated if the user chooses to see the properties of the associated bean again.

Note that an instance of the `BeanEditor` class is added as a `PropertyChangeListener` to the corresponding bean being edited. The `propertyChange()` method is invoked if the bean has changed it's state during editing and a `PropertyChangeEvent` has been fired. This method simply triggers a call to the `setProperty()` method of the table model.

Class BeanEditor.PropertyTableData

`PropertyTableData` extends `AbstractTableModel` and provides the table model for each bean editor. Three instance variables are declared:

`String[][] m_properties`: an array of data displayed in the table.

`int m_numProps`: number of a bean properties (corresponds to the number of rows in the table).

`Vector m_v`: collection of `PropertyDescriptor` objects sorted in alphabetical order.

The constructor of the `PropertyTableData` class takes a given bean instance and retrieves it's properties. First it uses the `Introspector.getBeanInfo()` method to get a `BeanInfo` instance:

```
BeanInfo info = Introspector.getBeanInfo(
  m_bean.getClass());
BeanDescriptor descr = info.getBeanDescriptor();
setTitle("Editing "+descr.getName());
PropertyDescriptor[] props = info.getPropertyDescriptors();
m_numProps = props.length;
```

This provides us with all available information about a bean (see chapter 2). We determine the bean's name and use it as the editor frame's title (note that this is an inner class, so `setTitle()` refers to the parent `BeanEditor` instance). We then extract an array of `PropertyDescriptors` which will provide us with the actual information about a bean's properties.

Bean properties are sorted by name in alphabetical order. The name of each property is determined with the `getDisplayName()` method. The sorted `PropertyDescriptors` are stored in our `m_v Vector` collection. Then we can create the 2-dimensional array, `m_properties`, which holds data to be displayed in the table. This array has `m_numProps` rows and 2 columns (for property name and value). To determine a property's value we need to obtain a reference to its `getXX()` method with `getReadMethod()` and make a call using the reflection API. We can call only `getXX()` methods without parameters (since we don't know anything about these parameters). Note that our `objToString()` helper method is invoked to translate a property's value into a display string (see below).

The `setProperty()` method searches for the given name in the 0-th column of the `m_properties` array. If such a property is found this method sets it's new value and updates the table component.

Several other simple methods included in this class have already been presented in previous examples and need not be explained again here. However, note that the `isCellEditable()` method returns `true` only for cells in the second column (property names, obviously, cannot be changed).

The `setValueAt()` method deserves additional explanation because it not only saves the modified data in the table model, but it also sends these modifications to the bean component itself. To do this we obtain a `PropertyDescriptor` instance stored in the `m_v Vector` collection. The modified property value is always a `String`, so first we need to convert it into its proper object type using our `stringToObj()` helper method (if we can do this, see below). If the conversion succeeds (i.e. the result is not `null`), we can continue.

To modify a bean value we determine the reference to it's `setXX()` method (corresponding to a certain property) and invoke it. Note that an anonymous array containing one element is used as parameter (these constructions are typical when dealing with the reflection API). Then the bean component and it's container (which can also be affected by changes in such properties as size and color) are refreshed to reflect the bean's new property value. Finally, if the above procedures were successful, we store the new value in the `m_properties` data array.

The `objToString()` helper method converts a given `Object` into a `String` suitable for editing. In many cases the `toString()` method returns a long string starting with the class name. This is not very appropriate for editable data values. So for several classes we provide our own conversion into a string of comma-delimited numbers. For instance a `Dimension` object is converted into a "width, height" form, `Color` is converted into "red, green, blue" form, etc. If no special implementation is provided, an object's `toString()` string is returned.

The `stringToObj()` helper method converts a given `String` into an `Object` of the given `Class`. The class's name is analyzed and a conversion method is chosen to build the correct type of object based on this name. The simplest case is the `String` class: we don't need to do any conversion at all in this case. For the primitive data types such as `int` or `boolean` we return the corresponding encapsulating (wrapper class) objects. For the several classes which receive special treatment in the `objToString()` method (such as a `Dimension` or `Color` object), we parse the comma-delimited string of numbers and construct the proper object. For all other classes (or if a parsing exception occurs) we return `null` to indicate that we cannot perform the required conversion.

Running the Code

Figure 18.9 shows the `BeanContainer` container and two editing frames displaying the properties of `Clock` and `JButton` components. This application provides a simple but powerful tool for investigating Swing and AWT components as well as custom JavaBeans. We can see all exposed properties and modify many of them. If a component's properties change as a result of user interaction, our component properly notifies its' listeners and we see an automatic editor table update. Try serializing a modified component and restoring it from its' file. Note how the previously modified properties are saved as expected.

It is natural to imagine using this example as a base for constructing a custom Swing IDE (Interface DevelopmentEnvironment). `BeanContainer`, combined with the custom resize edge components developed in chapters 15 and 16, provides a fairly powerful base to work from.

Chapter 19. Inside Text Components

In this chapter:

- Text package overview
- Date and time editor by David M. Karr

19.1 Text package overview

A truly exhaustive discussion of the text package is beyond the scope of this book.[1] However, in this chapter we hope to provide enough information about text components, and their underlying constituents, to leave you with a solid understanding of their inner workings. Picking up where chapter 11 left off, we continue our discussion of the most significant aspects of the text package classes and interfaces.[2] This chapter concludes with an example of a custom text field used for several variations of date and time selection. In the next chapter, we continue our study of text components with the development of a full-featured word processor application. The examples in chapter 20 demonstrate practical applications of many of the complex topics covered in this chapter.

19.1.1 More about JTextComponent

abstract class javax.swing.text.JTextComponent

Associated with each `JTextComponent` is a set of `Actions` which are normally bound to specific `KeyStrokes` (see 2.13) and managed in a hierarchically resolving set of `Keymaps` (see 19.1.23). We can retrieve a text component's `Actions` as an array with the `getActions()` method. We can retrieve and assign a new `Keymap` with `getKeymap()` and `setKeymap()` respectively.

All text components share a set of default `Actions`. Each of these `Actions` are instances of `TextAction` by default (see 19.1.24). `JTextComponent` provides a private static `EditorKit` (see 19.1.25) which consists of a set of four pre-built `TextActions` shared by all text components through the use of a default `Keymap` instance (see 19.1.26).

`JTextComponent` maintains a private reference to the text component that most recently had the keyboard focus. `TextActions` are designed to take advantage of this, and each `TextAction` will operate on this component when invoked in the event that the source of the invoking event is not a text component.

`Document` content is structured hierarchically by `Element` implementations (see 19.1.9). Each `Element` maintains a set of attributes encapsulated in implementations of the `AttributeSet` interface (see 19.1.12). Many `Elements` also contain one or more child `Elements`. Attributes that apply to one element also apply to all child `Elements`, but not vice versa. Each `Element` has an associated start and end `Position` (see 19.1.6).

---

[1] The `html`, `html.parser`, and `rtf` packages were still under construction at the time of this writing. Due to their complexity, as well as space and time constraints placed on this book, we decided that detailed coverage of these packages would best be left for a future edition.

[2] If, after reading this chapter, you require a more thorough treatment of the text package, we recommend Java Swing, by Robert Eckstein, Marc Loy, and Dave Wood, O'Reilly & Associates, 1998. This book includes roughly 300 pages of detailed class and interface descriptions. In particular, the discussion of Views and EditorKits provides indispensable knowledge for any developer working on support for a custom content type.

AttributeSets can be applied manually to a region of text. However, it is often more convenient to use Styles (see 19.1.14). Styles are AttributeSet implementations that we do not instantiate directly. Rather, Styles are created and maintained by instances of StyleContext (see 19.1.16), and each Style has an associated name allowing easy reference. StyleContext also provides a means for sharing AttributeSets across a document or possibly multiple documents, and is particularly useful in large documents.

The cursor of a text component is defined by implementations of the Caret interface (see 19.1.19). We can retrieve the current Caret with getCaret(), and assign a new one with setCaret(). A text component's Caret is instantiated (but not maintained) by its UI delegate. So when the L&F of a particular text component changes, the Caret in use will also change. JTextComponent supports the addition of CaretListeners that will receive CaretEvents whenever the position of the Caret changes.

Text components also support an arbitrary number of highlights through implementations of the Highlighter interface (see 19.1.17). Highlighters are most often used to indicate a specific selection. They can also be used for many other things such as marking new text additions. Highlighter maintains each highlighted region as an implementation of Highligher.Highlight, and each Highlight can be rendered using a Highlighter.HighlightPainter implementation. As with Carets, a text area's Highlighter is instantiated by its UI delegate. We can assign/retrieve a text component's Highlighter with setHighlighter() and getHighlighter() respectively.

JTextComponent also maintains a bound focusAccelerator property, as we discussed in chapter 11, which is a char that is used to transfer focus to a text component when the corresponding key is pressed simultaneously with the ALT key. JTextComponent defines a private Action called focusAction whose actionPerformed() method calls requestFocus(). Initially focusAction is not attached to the text component (that is, it is turned off). To activate it we use the setFocusAccelerator() method. Sending '\0' to the setFocusAccelerator() method turns it off. Internally, this method searches through all registered KeyStrokes and checks whether any are associated with focusAction, using the getActionForKeyStroke() method. If any are found they are unregistered, using the unregsiterKeyboardAction() method of JComponent. Finally the character passed in is used to construct a KeyStroke to register and associate with focusAction. This action is registered such that it will be invoked whenever the top-level window containing the given text component has the focus:

```
// from JTextComponent.java
registerKeyboardAction(
  focusAction,KeyStroke.getKeyStroke(aKey,ActionEvent.ALT_MASK),
  JComponent.WHEN_IN_FOCUSED_WINDOW);
```

Each text component uses a sub-class of BasicTextUI as its UI delegate. As we mentioned above, each text component also has an EditorKit for storing Actions. This EditorKit is referenced by the UI delegate. JTextField and JTextArea have default editor kits assigned by the UI delegate, whereas JEditorPane and JTextPane maintain their own editor kits independent of their UI delegate.

Unlike most Swing components, a text component's UI delegate does not directly define how that text component is rendered and laid out. Rather, it implements the ViewFactory interface (see 19.1.29) which requires the implementation of one method: create(Element e). This method returns a View instance (see 19.1.28) responsible for rendering the given Element. Each Element has an associated View that is used to render it. There are many different views provided in the text package, and it is rare that we will need to implement our own (although this is certainly possible). JTextArea, JTextField, and JPasswordField have specific Views returned by their UI delegate's create() method. JEditorPane and JTextPane Views are created by the current EditorKit.

We can retrieve a Point location in the coordinate system of a text component corresponding to a character

offset with `JTextComponent`'s `viewToModel()` method. Similarly, we can retrieve a `Rectangle` instance describing the size and location of the `View` responsible for rendering an `Element` occupying a given character offset with `modelToView()`.

`JTextComponent`'s `margin` property specifies the space to use between its border and its document content. Also note that standard clipboard operations can be programmatically performed with the `cut()`, `copy()`, and `paste()` methods.

19.1.2 The Document interface

abstract interface javax.swing.text.Document

In MVC terms, the model of a text component contains the text itself, and the `Document` interface describes this model. A hierarchical set of `Elements` (see 19.1.9) define the structure of a `Document`. Each `Document` contains one or more root `Elements`, potentially allowing more than one way of structuring the same content. Most documents only have one structure, and hence one root element. This element can be accessed with `getDefaultRootElement()`. All root elements, including the default root element, are accessible with `getRootElements()`, which returns an `Element` array.

Note: We will not discuss the details of maintaining multiple structures, as this is very rarely desired. See the API docs for examples of situations in which multiple structures might be useful.

`Documents` maintain two `Positions` which keep track of the beginning and end positions of the content. These can be accessed with `getStartPosition()` and `getEndPosition()` respectively. `Documents` also maintain a `length` property, accessible with `getLength()`, that maintains the number of contained characters.

The `Document` interface declares methods for adding and removing `DocumentListeners` (see 19.1.8), for notification of any content changes, and `UndoableEditListeners` (allowing easy access to built-in undo/redo support -- refer back to chapter 11 for an example of adding undo/redo support to a textarea).

Methods for retrieving, inserting, and removing content are also declared: `getText()`, `insertString()`, and `remove()`. Each of these throws a `BadLocationException` if an illegal (i.e. nonexistent) location in the document is specified. The `insertString()` method requires an `AttributeSet` instance describing the attributes to apply to the given text (`null` can be used for this parameter). Plain text components will not pay any attention to this attribute set. Text components using a `StyledDocument` instance most likely will pay attention to these attributes.

The `createPosition()` method is intended for inserting a `Position` instance at a given index, and the `putProperty()` and `getProperty()` methods are meant to insert and retrieve various properties stored in an internal collection.

The `render()` method is unique. It takes a `Runnable` as parameter, and is intended to ensure thread safety by not allowing document content to change while that `Runnable` is running. This method is used by each text component's UI delegate during painting.

19.1.3 The StyledDocument interface

abstract interface javax.swing.StyledDocument

This interface extends the `Document` interface to add functionality for working with `Styles` and other `AttributeSets`. Implementations are expected to maintain a collection of `Style` implementations. This interface also declares the notion of character and paragraph attributes, and logical styles. What these mean is

specific to each StyledDocument implementation (we will discuss these more when we talk about DefaultStyledDocument in 19.1.11).

The setCharacterAttributes() method is intended to assign a given set of attributes to a given range of document content. A boolean parameter is also required which is meant to specify whether or not pre-existing attributes of the affected content should be overwritten (true) or merged (false -- only new attributes are assigned). The setParagraphAttributes() method should work the same as setCharacterAttributes(), but is meant to apply to the number of paragraphs spanned by a given range of content. The getFont(), getBackground(), and getForeground() methods take an AttributeSet parameter, and are intended to be used for convenient access to the corresponding attribute in the given set (if it exists).

StyledDocuments are meant to allow addition, removal, and retrieval of Styles from an internal collection of Styles. The addStyle() method is intended to take a String and parent Style as parameters and return a new Style with the given name and given Style as its resolving parent. The getLogicalStyle() method should return a Style corresponding to the paragraph containing the given character offset. The setLogicalStyle() method is intended to assign a Style to the paragraph containing the given character offset. The getStyle() and removeStyle() methods should retrieve and remove a Style with the given name, respectively, in the internal collection.

The getCharacterElement() and getParagraphElement() methods are intended to allow retrieval of Elements corresponding to a given character offset. The definition of these methods will vary based on the definition of paragraph and character Elements in a StyledDocument implementation. Typically, a character Element represents a range of text containing a given offset, and a paragraph Element represents a paragraph containing the given offset.

19.1.4 AbstractDocument

abstract class javax.swing.text.AbstractDocument

AbstractDocument implements the Document interface and provides a base implementation for text component models. Two provided classes that extend AbstractDocument are used by the Swing text components as their default models: PlainDocument and DefaultStyledDocument. PlainDocument is used by all the plain text components, such as JTextArea, JTextField and its subclass, JPasswordField. It provides support for character data content only and does not support markup (i.e. multiple fonts, colors, etc.) of this content. DefaultStyledDocument is used by more sophisticated text components such as JEditorPane and its subclass, JTextPane. It provides support for markup of text by implementing the StyleDocument interface.

AbstractDocument specifies a mechanism that separates the storage of character data from the structuring of that data. Thus, we have the capability to store our text however we like without concern for how the document is structured and marked up. Similarly, we can structure a document with little concern for how its data is stored. The significance of this structure-storage separation will make more sense after we have discussed Elements and attributes below. Character data is stored in an instance of the inner Content interface which we will also discuss below.

This class defines functionality for a basic read/write locking scheme. This scheme enforces the rule that no write can occur while a read is occurring. However, multiple reads can occur simultaneously. To obtain a read lock we use the render() method. This method releases the read lock when it finishes executing the Runnable passed to it. No other access methods acquire such a lock (making them non-thread safe). The getText() method, for example, does not acquire a read lock. In a multithreaded environment, any text retrieved with this method may be corrupted if a write was occurring at the time the text was retrieved.

The read lock is basically just an increment in an internal variable that keeps track of the number of readers.

The `readLock()` method does this for us, and will force the current thread to wait until no write locks exist. When the `Runnable` finishes executing, the internal reader-count variable is decremented. The `readUnlock()` method is responsible for this. Note that both of these methods will simply do nothing and return if the current thread is the writer thread. Also note that a `StateInvariantError` exception will be thrown if a read unlock is requested when there are no readers.

The write lock is a reference to the writing thread. The `writeLock()` and `writeUnlock()` methods take care of this for us. Whenever a modification is requested the write lock must first be obtained. If the writer thread is not `null`, and is not the same as the invoking thread, `writeLock()` blocks the current thread until the current writer releases the lock by calling `writeUnlock()`.

If we intend to use the protected reader and writer locking methods ourselves in a sub-class, we are encouraged to ensure that a `readUnlock()` call will be made no matter what happens in the try block, using the following semantics:

```
// From AbstractDocument.java
try {
  readLock();
  // do something
} finally {
  readUnlock();
}
```

All methods that modify document content must obtain a write lock before any modification can take place. These methods include `insertString()` and `remove()`.

`AbstractDocument`'s `dump()` method prints the document's `Element` hierarchy to the given `PrintStream` for debugging purposes. For example, the following class will dump a `JTextArea`'s `Element` hierarchy to standard output.

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

public class DumpDemo extends JFrame
{
  JTextArea m_editor;

  public DumpDemo() {
    m_editor = new JTextArea();

    JScrollPane js1 = new JScrollPane(m_editor);
    getContentPane().add(js1, BorderLayout.CENTER);

    JButton dumpButton = new JButton("Dump");
    dumpButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        ((PlainDocument) m_editor.getDocument()).dump(System.out);
      }
    });

    JPanel buttonPanel = new JPanel();
    buttonPanel.add(dumpButton);

    getContentPane().add(buttonPanel, BorderLayout.SOUTH);

    setSize(300,300);
```

```
    setVisible(true);
  }

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

Typing this text in the `JTextArea`:

```
Swing is
powerful!!
```

...produces the following output when the `dump()` method is invoked (this will make more sense after we discuss `Elements` in 19.1.9).

```
<paragraph>
  <content>
    [0,9][Swing is
]
  <content>
    [9,20][powerful!!
]
  <content>
    [20,21][
]
<bidi root>
  <bidi level
    bidiLevel=0
  >
    [0,21][Swing is
powerful!!
]
```

`AbstractDocument` also includes several significant inner classes and interfaces. Most we will discuss in appropriate places later in this chapter. A brief overview is appropriate here:

- `abstract class AbstractDocument.AbstractElement`: This class implements the `Element` and `MutableAttributeSet` interfaces, allowing instances to act both as `Elements` and the mutable `AttributeSets` that describe them. This class also implements the `TreeNode` interface, providing an easy means of displaying document structure with a `JTree`.
- `class AbstractDocument.BranchElement`: A concrete sub-class of `AbstractDocument.AbstractElement` that represents an `Element` which can contain multiple child `Elements` (see 19.1.9).
- `class AbstractDocument.LeafElement`: A concrete sub-class of `AbstractDocument.AbstractElement` that represents an `Element` which cannot contain child `Elements` (see 19.1.9).
- `static abstract interface AbstractDocument.Content`: Defines the data storage mechanism used by `AbstractDocument` sub-classes (see 19.1.9).
- `static abstract interface AbstractDocument.AttributeContext`: Used for efficient `AttributeSet` management (see 19.1.16).
- `static class AbstractDocument.ElementEdit`: Extends `AbstractUndoableEdit`, implements `DocumentEvent.ElementChange` (see 19.1.7), and allows document changes to be undone and redone.
- `class AbstractDocument.DefaultDocumentEvent`: Extends `CompoundEdit` and implements

DocumentEvent (see 19.1.7). Instances of this class are used by documents to create UndoableEdits, which can be used to create UndoableEditEvents for dispatching to UndoableEditListeners. Instances of this class are also fired to any registered DocumentListeners (see 19.1.8) for change notification.

19.1.5 The Content interface

abstract static interface javax.swing.text.AbstractDocument.Content

In order to implement a data storage mechanism for text, AbstractDocument provides the static Content interface. Every Document character storage mechanism must implement this interface. (Images and other embedded objects are not considered part of a document's content.) Each Content instance represents a sequence of character data, and provides the ability to insert, remove, and retrieve character data with the insertString(), remove(), getString(), and getChars() methods.

Note: A special convenience class called Segment allows access to fragments of actual document text without the need to copy characters into a new array for processing. This is used internally by text components to speed up searching and rendering large documents.

Implementations of Content must also provide the ability to create position markers that keep track of a certain location between characters in storage with the createPosition() method. These markers are implementations of the Position interface.

Content implementations provide UndoableEdit objects that represent the state of storage before and after any change is made. The insertString() and remove() methods are meant to return such an object each time they are invoked, allowing insertions and removals to be undone and redone.

Two Content implementations are included in the javax.swing.text package: StringContent and GapContent. StringContent stores character data in a normal char array. GapContent also stores data in a char array but it purposefully leaves an empty space, a gap, in this array. "The gap is moved to the location of changes to take advantage of common behavior where most changes are in the same location. Changes that occur at a gap boundary are generally cheap and moving the gap is generally cheaper than moving the array contents directly to accommodate the change"[API]. This gap is strictly used for internal efficiency purposes and is not accessible outside of this class.

Note: StringContent was used in earlier implementations of PlainDocument and DefaultStyledDocument, but has been replaced by GapContent (which extends a package private class called GapVector). The gap buffer algorithm used in GapContent is very efficient for keeping track of large numbers of Positions and, interestingly, is used in the popular emacs editor.

19.1.6 The Position interface

abstract interface javax.swing.text.Position

This interface consists of one method, getOffset(), which returns an int value representing the location, or offset, from the beginning of the document's content. Figure 19.1 illustrates what happens to a Position marker when text is inserted and removed from storage. This figure starts by showing a document containing "Swing text" as its content. There are initially Position markers at offsets 0, 4, and 7. When we remove the characters from offset 4 through 9 the Position at offset 7 is moved to offset 4. At this point there are two Positions at offset 4 and the document content is "Swin". When we insert "g is" at offset 4 both Positions at offset 4 are moved to offset 8 and the document content becomes "Swing is".

Note: The term range refers to a sequence of characters between two Position markers as shown in figure 19.1.

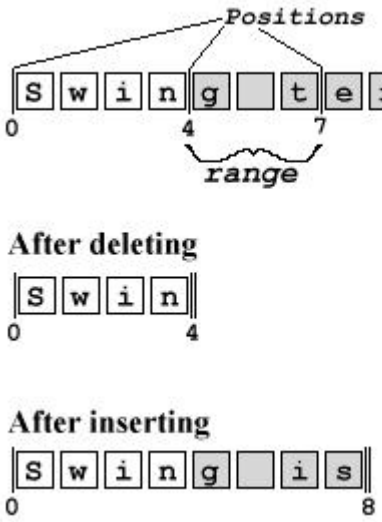


Figure 19.1 Position movement

19.1.7 The DocumentEvent interface

abstract interface javax.swing.event.DocumentEvent

Changes to a Document's content are encapsulated in implementations of the DocumentEvent interface, the default implementation of which is AbstractDocument.DefaultDocumentEvent. There are three types of changes that can occur to document content: CHANGE, INSERT, and REMOVE (fields defined within the DocumentEvent.EventType inner class). DocumentEvent also defines an interface within it called ElementChange. Implementations of this interface, the default of which is AbstractDocument.ElementEdit, are responsible for storing information about changes to the structure of a document for use in, among other things, undo and redo operations. AbstractDocument handles the firing of DefaultDocumentEvents appropriately with its fireXXUpdate() methods.

The getChange() method is meant to take an Element instance as parameter, and return an instance of DocumentEvent.ElementChange describing the elements that were added and/or removed, as well as the location of a change. The getDocument() method should return a reference to the Document instance that generated this event. The getLength() method is intended to return the length of a change, and the getOffset() method should return the offset at which a change began. The getType() method is meant to return an instance of Document.EventType specifying the type of change that occurred to the document.

19.1.8 The DocumentListener interface

abstract interface javax.swing.event.DocumentListener

Instances of this interface can be attached to Document implementations to be notified of changes in that document's content. It is important to note that this notification will always occur after any content has been updated. Knowing this, it is even more important to realize that we should not perform any changes to the content of a document from within a DocumentListener. This can potentially result in an infinite loop in situations where a document event causes another to be fired.

Note: Never modify the contents of a document from within a DocumentListener.

The `insertUpdate()` and `removeUpdate()` methods are meant to give notification of content insertions and removals respectively. The `changedUpdate()` method is intended to provide notification of attribute changes.

19.1.9 The Element interface

abstract interface javax.swing.text.Element

`Elements` provide a hierarchical means of structuring a `Document`'s content. Associated with each `Element` is a set of attributes encapsulated in an `AttributeSet` implementation. These attributes provide a means of specifying the markup of content associated with each `Element`. `AttributeSets` most often take the form of `Style` implementations and are grouped together inside a `StyleContext` object. `StyleContext` objects are used by `StyledDocument` implementations such as `DefaultStyledDocument`. The objects that are responsible for actually rendering text components are implementations of the abstract `View` class. Each `Element` has a separate `View` object associated with it, and each `View` recognizes a predefined set of attributes used in the actual rendering and layout of that `Element`.

Note: Elements are objects that impose structure on a text component's content. They are actually part of the document model, but they are also used by views for text component rendering.

The `getAttributes()` method is meant to retrieve an `AttributeSet` collection of attributes describing an `Element`. The `getElement()` method is intended to fetch a child `Element` at the given index, where the index is given in terms of the number of child `Elements`. The `getElementCount()` method should return the index of the `Element` closest to the provided document content offset. The `getElementCount()` method is meant to return the number of child `Elements` an `Element` contains (`0` if the parent `Element` itself is a leaf). The `isLeaf()` method is intended to tell us whether or not an `Element` is a leaf element, and `getParentElement()` should return an `Element`'s parent `Element`.

The `getDocument()` method is meant to retrieve the `Document` instance an `Element` belongs to. The `getStartOffset()` and `getEndOffset()` methods should return the offset of the beginning and end of an `Element`, respectively, from the beginning of the document. The `getName()` method is intended to return a short `String` description of an `Element`.

`AbstractDocument` defines the inner class `AbstractElement`, which implements the `Element` interface. As we mentioned earlier, there are two subclasses of `AbstractElement` defined within `AbstractDocument`: `LeafElement` and `BranchElement`. Each `LeafElement` has a specific range of content text associated with it (this range can change when content is inserted, removed, or replaced -- figures 19.2 and 19.3 illustrate). `LeafElements` cannot have any child `Elements`. `BranchElements` can have any number of child `Elements`. The range of content text associated with `BranchElements` is the union of all content text associated with their child `LeafElements`. (Thus the start offset of a `BranchElement` is the lowest start offset of all its child `LeafElements`, and its end offset is the highest end offset of all its child `LeafElements`.) `DefaultStyledDocument` provides a third type of element called `SectionElement` which extends `BranchElement`. The meaning of each type of element differs depending on the type of document.

The text package also includes an `ElementIterator` class, which is designed to traverse an `Element` hierarchy in a depth first fashion (i.e. postorder -- see 17.1.2). Methods `first()`, `current()`, `depth()`, `next()`, and `previous()` can be used to obtain information about, and programmatically traverse, an `Element` hierarchy. We can construct an `ElementIterator` object by providing either a `Document` or an `Element` to the `ElementIterator` constructor. If a `Document` is provided, the default root `Element` of that document is used as the root of the `Element` hierarchy traversed by `ElementIterator`.

Note: `ElementIterator` does not provide any thread safety by default, so it is our responsibility to ensure that

no Element changes occur during traversal.

19.1.10 PlainDocument

class javax.swing.text.PlainDocument

This class extends AbstractDocument and is used by the basic text components: JTextField, JPasswordField, and JTextArea. When enforcing certain input, usually in a JTextField, we normally override AbstractDocument's insertString() method in a PlainDocument sub-class (see the discussion of JTextField in chapter 11 for an example).

PlainDocument uses a BranchElement as its root and has only LeafElements as children. In this case each LeafElement represents a line of text and the root BranchElement represents the whole document text. PlainDocument identifies a BranchElement as "paragraph" and a LeafElement as "content". Note that the notion of a paragraph in PlainDocument is much different than our normal notion of a paragraph. Usually we think of paragraphs as sections of text separated by line breaks. However, PlainDocument considers each section of text ending with a line break as a line of "content" in its never-ending "paragraph". Figure 19.2 and 19.3 show the structure of a sample PlainDocument and illustrate how Elements and their associated Positions can change when document content changes.

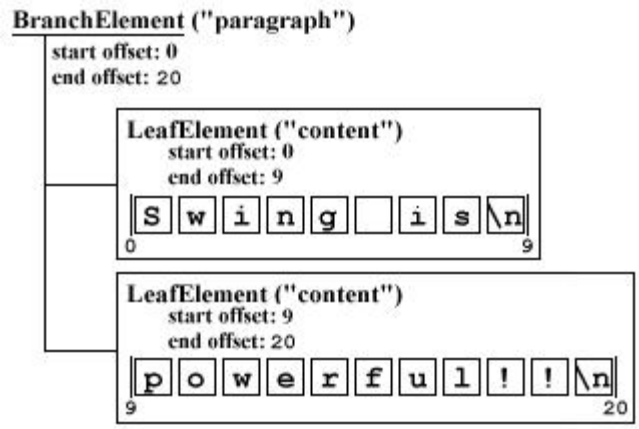


Figure 19.2 Sample PlainDocument structure

In figure 19.2 we see a PlainDocument containing three elements. Two LeafElements represent two lines of text and are children of the root BranchElement. Note that the this root element begins at offset 0, the start offset of the first LeafElement, and ends at 19, the end offset of the last LeafElement. This document would be displayed in a JTextArea as:

```
Swing is
powerful!!
```

Note: The line break at the end of the second LeafElement is always present at the end of the last Element in any PlainDocument. It does not represent a line break that was actually inserted into the document and is not counted when the document length is queried using the getLength() method. Thus the length of the document shown in figure 19.2 would be returned as 19.

Now suppose we insert two line breaks at offset 5. Figure 19.3 shows the structure that would result from this addition.

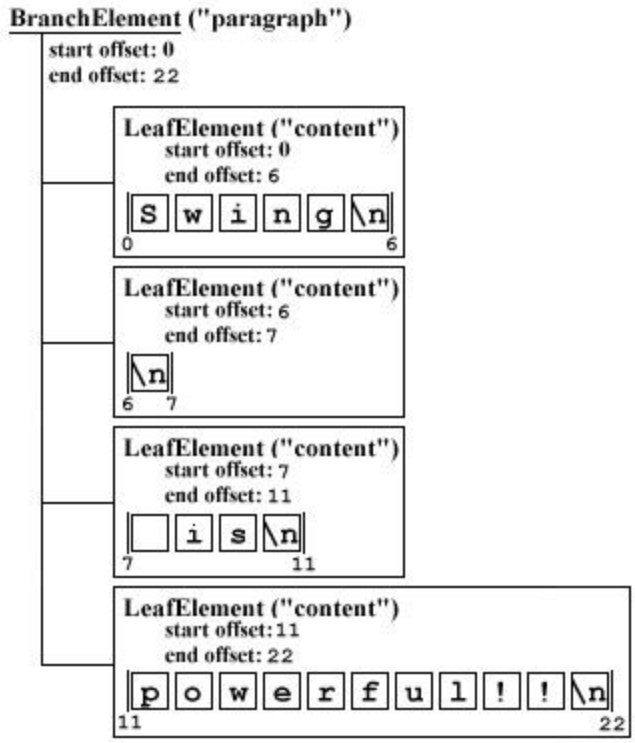


Figure 19.3 Sample PlainDocument structure after inserting two line breaks at offset 19

This document would now be displayed in a `JTextArea` as:

```
Swing

 is
powerful!!
```

`JTextArea`, `JTextField`, and `JPasswordField` use `PlainDocument` as their model. Only `JTextArea` allows its document to contain multiple `LeafElements`. `JTextField` and its `JPasswordField` subclass allow only one `LeafElement`.

19.1.11 DefaultStyledDocument

class javax.swing.text.DefaultStyledDocument

`DefaultStyledDocument` provides significantly more power over the `PlainDocument` structure described above. This `StyledDocument` implementation (see 19.1.3) is used for marked up (styled) text. `JTextPane` uses an instance of `DefaultStyledDocument` by default (although this may change based on its content type).

`DefaultStyledDocument` uses and instance of its inner `SectionElement` class as its root `Element`, which has only instances of `AbstractDocument.BranchElement` as children. These `BranchElements` represent paragraphs, referred to as paragraph `Elements`, and they contain instances of `AbstractDocument.LeafElement` as children. These `LeafElements` represent what are referred to as character `Elements`. Character `Elements` represent regions of text (possibly multiple lines within a paragraph) that share the same attributes.

We can retrieve the character `Element` occupying a given offset with the `getCharacterElement()` method, and we can retrieve the paragraph `Element` occupying a given offset with the `getParagraphElement()` method.

We will discuss attributes, AttributeSets, and their usage details soon enough. However, it is important to understand here that AttributeSets assigned to DefaultStyledDocument Elements resolve hierarchically. For instance, a character Element will inherit all attributes assigned to itself, as well as those assigned to the parent paragraph Element. Character Element attributes override those of the same type defined in the parent paragraph Element's AttributeSet.

Note: The Elements used by DefaultStyledDocument are derived from AbstractDocument.AbstractElement, which implements both the Element and MutableAttributeSet interfaces. This allows these Elements to act as their own AttributeSets, and use each other as resolving parents.

Figure 19.4 shows a simple DefaultStyledDocument in a JTextPane with two paragraphs.

Figure 19.4 A two-paragraph DefaultStyledDocument, with several different attributes, in a JTextPane.

Using AbstractDocument's dump() method to display this document's Element structure to standard output (see 19.1.4), we get the following:

```
<section>
  <paragraph
    RightIndent=0.0
    LeftIndent=0.0
    resolver=NamedStyle:default {name=default,nrefs=2}
    FirstLineIndent=0.0
  >
    <content
      underline=false
      bold=true
      foreground=java.awt.Color[r=0,g=128,b=0]
      size=22
      italic=false
      family=SansSerif
    >
      [0,6][Swing
]
  <paragraph
    RightIndent=0.0
    LeftIndent=0.0
    resolver=NamedStyle:default {name=default,nrefs=2}
    FirstLineIndent=0.0
  >
    <content
      underline=false
      bold=false
      foreground=java.awt.Color[r=0,g=0,b=0]
      size=12
      italic=false
      family=SansSerif
```

```
    >
      [6,9][is ]
    <content
      underline=false
      bold=false
      foreground=java.awt.Color[r=0,g=0,b=0]
      size=12
      italic=false
      family=SansSerif
    >
      [9,19][extremely ]
    <content
      underline=false
      bold=false
      foreground=java.awt.Color[r=0,g=0,b=192]
      size=18
      italic=true
      family=SansSerif
    >
      [19,27][powerful]
    <content
      underline=false
      bold=true
      foreground=java.awt.Color[r=255,g=0,b=0]
      size=20
      italic=false
      family=SansSerif
    >
      [27,28][!]
    <content>
      [28,29][
]
<bidi root>
  <bidi level
    bidiLevel=0
  >
    [0,29][Swing
is extremely powerful!
]
```

Note the use of <section>, <paragraph>, and <content> to denote SectionElement, BranchElement, and LeafElement respectively. Also note that the <paragraph> and <content> tags each contain several attributes. The <paragraph> attributes represent paragraph Element attributes and the <content> attributes represent character Element attributes. We will discuss specific attributes in more detail below. Note that the <bidi root> tag specifies a second root Element allowing bidirectional text (this functionality is incomplete as of Java 2 FCS).

We can assign paragraph and character attributes to a region of text with the setParagraphAttributes() and setCharacterAttributes() methods respectively. These methods require a start and end offset, specifying the region to apply the attributes to, as well as an AttributeSet containing the attributes, and a boolean flag specifying whether or not to replace pre-existing attributes with the new attributes.

Regarding the range of text, paragraph attributes will be applied to paragraph Elements that contain at least some portion of the specified range. Character attributes will be applied to all character Elements that intersect that range. If the specified range only partially extends into a character Element, that Element will be split into two, so that only the specified range of text will receive the new attributes (this splitting is handled by an instance of the ElementBuffer inner class).

Regarding the boolean flag, if the flag is true, all pre-existing paragraph Element attributes are removed

before the new set is applied. Otherwise, the new set is merged with the old set, and any new attributes overwrite pre-existing attributes. Character attributes work in a similar way, but they do not change paragraph attributes at all--they simply override them.

`DefaultStyledDocument` also defines the notion of logical paragraph `Styles`. A logical paragraph `Style` acts as the resolving parent of a paragraph `Element`'s `AttributeSet`. So attributes defined in a paragraph `Element`'s `AttributeSet` override those defined in that paragraph's logical `Style`. We can change a specific paragraph `Element`'s logical style with the `setLogicalStyle()` method. The logical style of each paragraph defaults to `StyleContext.DEFAULT_STYLE` (which is empty by default).

`JTextPane` implements `getParagraphAttributes()`, `setParagraphAttributes()`, `getLogicalStyle()`, and `setLogicalStyle()` methods which communicate directly with its `StyledDocument`. `JTextPane`'s paragraph attributes and logical style `setXX()` methods apply to the paragraph the caret currently resides in if there is no selection. If there is a selection, these methods apply to all paragraphs spanned by the selected region. `JTextPane`'s paragraph attributes and logical style `getXX()` methods apply to the paragraph currently containing the caret.

`JTextPane` also implements `getCharacterAttributes()` and `setCharacterAttributes()` methods. If there is a selection, the `setCharacterAttributes()` method will act as described above, splitting `Elements` as needed. If there is no selection, this method will modify `JTextPane`'s input attributes.

Note: `JTextPane`'s input attributes is a reference to an `AttributeSet` which changes with the location of the caret. This reference always points to the attributes of the character `Element` at the current caret location. We can retrieve it at any time with `JTextPane`'s `getInputAttributes()` method. Whenever text is inserted in a `JTextPane`, the current input attributes will be applied to that text by default. However, any attributes explicitly assigned to newly inserted text will override those defined in the current input attributes.

A `StyleContext` instance (see 19.1.16) is associated with each `DefaultStyledDocument`. As we mentioned in the beginning of this chapter, the `Style` interface describes a named mutable `AttributeSet`, and the `StyledDocument` interface describes a `Document` which manages a set of `Styles`. A `DefaultStyledDocument`'s `StyleContext` instance is what performs the actual management, creation, and assignment of that document's `Styles`. If a `StyleContext` is not provided to the `DefaultStyledDocument` constructor, a default version is created.

`JTextPane` defines several methods for adding, removing, and retrieving `Styles`, as well as specific attributes within a given `AttributeSet` (such as the `getFont()` and `getForeground()` methods). Calls to these methods are forwarded to methods of the same signature in `JTextPane`'s `StyledDocument`, and, in the case of `DefaultStyledDocument`, these calls are forwarded to the `StyleContext` in charge of all the `Styles`.

`DefaultStyledDocument` also includes several significant inner classes:

- `static class DefaultStyledDocument.AttributeUndoableEdit`: This class extends `AbstractUndoableEdit` to allow `AttributeSet` undo/redo functionality with `Elements`.
- `class DefaultStyledDocument.ElementBuffer`: Instances of this class are used to manage structural changes in a `DefaultStyledDocument`, such as the splitting of `Elements`, or the insertion and removal of text, resulting in the modification of, and the insertion and/or removal of, various `Elements`. This class also plays a critical role in constructing `AbstractDocument.DefaultDocumentEvents` (see 19.1.4).
- `static class DefaultStyledDocument.ElementSpec`: This class describes an `Element` that can be created and inserted into a document in the future with an `ElementBuffer`.
- `protected class DefaultStyledDocument.SectionElement`: This class extends

`AbstractDocument.BranchElement` and acts as a `DefaultStyledDocument`'s default root `Element`. It contains only `BranchElement` children (representing paragraphs).

19.1.12 The AttributeSet interface

abstract interface javax.swing.text.AttributeSet

An attribute is simply a key/value pair (as in a `Hashtable`) that should be recognized by some `View` implementation available to the text component being used. As we know from our discussion above, each `Element` in a `DefaultStyledDocument` has an associated set of attributes which resolves hierarchically. The attributes play a critical role in how that piece of the document will be rendered by a `View`. For example, one commonly used attribute is `FontFamily`. The `FontFamily` attribute key is an `Object` consisting of the `String` "family". The `FontFamily` attribute value is a `String` representing the name of a font (i.e. "monospaced"). Other examples of attribute keys include "Icon" and "Component," whose values are instances of `Icon` and `Component` respectively.

If an attribute is not recognized by a `View`, the `Element` associated with that view will not be rendered correctly. Thus, there is a predefined set of attributes that is recognized by the Swing `View` classes, and these attribute keys should be considered reserved -- in other words, all new attributes should use new keys. These predefined attribute keys are all accessible as static `Objects` in the `StyleConstants` class (see 19.1.15).

Sets of attributes are encapsulated in implementations of either the `AttributeSet` interface, the `MutableAttributeSet` interface (see 19.1.13), or the `Style` interface (see 19.1.14). `Style` extends `MutableAttributeSet` which, in turn, extends `AttributeSet`. The `AttributeSet` interface describes a read-only set of attributes because it does not provide methods for changing, adding, or removing attributes from that set.

The `containsAttribute()` and `containsAttributes()` methods are intended for checking whether an `AttributeSet` contains a given attribute key/value pair, or any number of such pairs. The `copyAttributes()` method is meant to return a fresh, immutable copy of the `AttributeSet` it is invoked on. The `getAttributeCount()` method should return the number of attributes contained in a set, and `getAttributeNames()` should retrieve an `Enumeration` of the keys describing each attribute. The `isDefined()` method is intended for checking whether a given attribute key corresponds to an attribute directly stored in the `AttributeSet` the method is invoked on (resolving parents are not searched). The `isEqual()` method is meant to compare two `AttributeSets` and return whether or not they contain identical attribute key/value pairs. The `getResolveParent()` method should return a reference to an `AttributeSet`'s resolving parent, if any, and the `getAttribute()` method is intended to return the value of an attribute corresponding to a given key.

The `AttributeSet` interface also provides four empty static interfaces: `CharacterAttribute`, `ColorAttribute`, `FontAttribute`, `ParagraphAttribute`. The only reason these interfaces exist is to provide a signature (i.e. information about the class in which it is defined) which is expected of each attribute key. This signature can be used to verify whether an attribute belongs to a certain category (see 19.1.15).

Only one direct implementation of the `AttributeSet` interface exists within the text package: `StyleContext.SmallAttributeSet`. A `SmallAttributeSet` is an array of attribute key/value pairs stored in the alternating pattern: key1, value1, key2, value2, etc. (thus the number of attributes contained in a `SmallAttributeSet` is actually half the size of its array). An array is used for storage because `AttributeSet` describes a read-only set of attributes, and using an array is more memory-efficient than dynamically resizable storage such as that provided by a `Hashtable`. However, it is less time-efficient to search through an array than a `Hashtable`. For this reason, `SmallAttributeSet` is intended to be used only for small sets of attributes. These sets are usually shared between several `Elements`. Because of the way sharing works (see 19.1.16), the smaller the set of attributes the better candidate that set is for being shared.

19.1.13 The MutableAttributeSet interface

abstract interface javax.swing.text.MutableAttributeSet

The MutableAttributeSet interface extends the AttributeSet interface and declares additional methods intended to allow attribute addition, removal, and resolving parent assignment: addAttribute(), addAttributes(), setResolveParent(), removeAttribute(), and two variations of removeAttributes().

MutableAttributeSet also has two direct implementations within the text package: AbstractDocument.AbstractElement and SimpleAttributeSet. The fact that AbstractElement implements MutableAttributeSet allows such Elements to act as resolving parents to one another. It also reduces object overhead by combining structural information about a region of text with that region's stylistic attributes.

SimpleAttributeSet uses a Hashtable to store attribute key/value pairs because it must be dynamically resizable. By nature, a Hashtable is less efficient than an array in memory usage, but more efficient in look-up speed. For this reason, SimpleAttributeSets are used for large sets of attributes that are not shared.

Note: In the past few sections we have alluded to the importance of efficiency in attribute storage. Efficiency here refers to both memory usage and speed of attribute location. To summarize the issues: A View uses attributes to determine how to render its associated Element. These attribute values must be located, by key, within that Element's attribute set hierarchy. The faster this location occurs the quicker the view is rendered and the more responsive the user interface becomes. So look-up speed is a large factor in deciding how to store attribute key/value pairs. Memory usage is also a large issue. Obtaining efficient look-up speed involves sacrificing efficient memory usage, and vice-versa. This necessary trade-off is taken into account through the implementation of the different attribute storage mechanisms described above, and the intelligent management of when each mechanism is used. We will soon see that the StyleContext class acts as, among other things, this intelligent manager.

19.1.14 The Style interface

abstract interface javax.swing.text.Style

The Style interface extends MutableAttributeSet and provides the ability to attach listeners for notification of changes to its set of attributes. Style also adds a String used for name identification. The only direct implementation of the Style interface is provided by StyleContext.NamedStyle. Internally, NamedStyle maintains its own private AttributeSet implementation that contains all its attributes. This AttributeSet can be an instance of StyleContext.SmallAttributeSet or SimpleAttributeSet, and may switch back and forth between these types over the course of its lifetime (this will become clear after our discussion of StyleContext).

19.1.15 StyleConstants

class javax.swing.text.StyleConstants

The StyleConstants class categorizes predefined attribute keys into members of four static inner classes: CharacterConstants, ColorConstants, FontConstants, and ParagraphConstants. These Objects are all aliased from their outer class, StyleConstants, so they are more easily accessible (aliasing here means providing a reference to an object of an inner class). Also, both ColorConstants and FontConstants keys are aliased by CharacterConstants to provide a sensible hierarchy of attribute key organization.

Note: Not all aliased keys use the same name in each class. For instance, FontFamily in StyledConstants is an alias of Family in StyledConstants.CharacterConstants. However, Family in StyledConstants.CharacterConstants is an alias of Family (the actual key) in StyledConstants.FontConstants. Each is a reference to the same key object and it makes no difference which one we use.

Most keys are self-explanatory in meaning. The StyleConstants API documentation page contains a helpful diagram illustrating the meaning of some of the less self-explanitory attribute keys that apply to paragraphs of styled text. (Each of the keys illustrated in this diagram is an alias of the actual key defined in StyleConstants.ParagraphConstants.)

StyleConstants also defines static methods for assigning and retrieving many predefined attributes in an AttributeSet. For example, to assign a specific font family attribute to an AttributeSet (assuming it is mutable), we can use StyleConstants' setFontFamily() method.

19.1.16 StyleContext

class javax.swing.text.StyleContext

StyleContext implements the AbstractDocument.AttributeContext interface, and declares a set of methods used to modify or fetch new instances of AttributeSet implementations. AbstractContext was designed with the understanding that the implementor may use more than one type of AttributeSet implementation to store sets of attributes. The decision to use one type over another may be based on any number of factors, and StyleContext takes full advantage of this design.

StyleContext's main role is to act as a container for Styles that may be used by one or more DefaultStyledDocuments. It maintains a private NamedStyle instance used to store its Styles and allow access by name. Each of these contained Styles is also an instance of NamedStyle. So, to clarify, StyleContext maintains a NamedStyle instance whose key/value pairs are of the form String/NamedStyle.

StyleContext also maintains a subset of these NamedStyle values in a Hashtable. Only those NamedStyle's whose AttributeSet contains 9 or less attributes are stored in this Hashtable and their AttributeSets are maintained as instances of SmallAttributeSet. Those NamedStyles with an AttributeSet containing 10 or more attributes are not stored in the Hashtable, and their AttributeSets are maintained as instances of SimpleAttributeSet.

This partitioning is managed dynamically by StyleContext, and is the result of combining the AbstractContext design with the use of a compression threshold (a hard-coded int value of 9). Whenever an attribute is added or removed, StyleContext checks the number of attributes in the target AttributeSet. If the resulting set will contain 9 or less attributes it remains, or is converted to, a SmallAttributeSet and is added to the Hashtable if it wasn't already there. If the resulting set will contain 10 or more attributes it remains, or is converted to, a SimpleAttributeSet and is removed from the Hashtable if it was already there.

The reason for this partitioning is to support efficient AttributeSet sharing. Most styled documents contain many distinct regions of identically styled text. These regions normally have a small number of attributes associated with them. It is clear that the best thing to do in this situation is to assign the same AttributeSet to each of these regions. And the best AttributeSet implementation to use for this is SmallAttributeSet due to its superior memory efficiency (since look-up speed is a minor issue with a very small number of attributes). Larger sets of attributes are, in general, rare. The best AttributeSet implementation to use for this is SimpleAttributeSet due to its superior look-up capabilities (since memory usage will most likely be a minor issue with a relatively small number of SimpleAttributeSets).

19.1.17 The Highlighter interface

abstract interface javax.swing.text.Highlighter

This interface describes how specific regions of text can be marked up with instances of the inner `Highlighter.Highlight` interface. A `Highlight` maintains a beginning and end offset, and a reference to an instance of the inner `Highlighter.HighlightPainter` interface. A `HighlightPainter`'s only responsibility is to render the background of a specific region of text.

A text component's UI delegate is responsible for maintaining its `Highlighter`. For this reason the `Highlighter` can change when a text component's look and feel changes. `JTextComponent` provides methods for working with a text component's `Highlighter` so we generally ignore the fact that such methods really get forwarded to the UI delegate.

A `Highlighter` is intended to maintain an array of `Highlighter.Highlight` instances, and we are meant to be able to add to this array using the `addHighlight()` method. This method takes two `int`s defining the range of text to highlight, as well as a `Highlighter.HighlightPainter` instance specifying how that `Highlight` should be rendered. Thus, by defining various `HighlightPainters`, we can add an arbitrary number of highlighted regions with distinct visual effects.

The range a `Highlight` encompasses is meant to be modified with the `changeHighlight()` method, and `Highlights` can be removed from a `Highlighter`'s array with the `removeAllHighlights()` or `removeHighlight()` methods. The `paint()` method is meant to manage the rendering of all a `Highlighter`'s `Highlights`.

We can assign a new `Highlighter` with `JTextComponent`'s `setHighlighter()` method. Similarly, we can retrieve a reference to the existing one with `JTextComponent`'s `getHighlighter()` method. Each `JTextComponent` also maintains a `selectionColor` property which specifies the color to use in rendering default highlights.

19.1.18 DefaultHighlighter

class javax.swing.text.DefaultHighlighter

`DefaultHighlighter` extends the abstract `LayeredHighlighter` class. `LayeredHighlighter` implements the `Highlighter` interface and defines a `paintLayeredHighlights()` method, which is responsible for managing potentially multiple overlapping `Highlights`. `LayeredHighlighter` also declares an inner abstract static class called `LayerPainter` from which the static `DefaultHighlighter.DefaultHighlightPainter` extends. This implementation paints a solid background, behind the specified region of text, in the current text component selection color.

19.1.19 The Caret interface

abstract interface javax.swing.text.Caret

This interface describes a text component's cursor. The `paint()` method is responsible for rendering the caret, and the `setBlinkRate()`/`getBlinkRate()` methods are meant to assign/retrieve a specific caret blink interval (normally in milliseconds). The `setVisible()` and `isVisible()` methods are intended to hide/show the caret and check for caret visibility, respectively.

The `setDot()`/`getDot()` methods are meant to assign/retrieve the offset of the caret within the current document. The `getMark()` method should return a location in the document where the caret's mark has been assigned. The `moveDot()` method is intended to assign a mark position, and move the caret to a new location while highlighting the text between the dot and the mark. The

setSelectionVisible()/isSelectionVisible() methods are meant to assign/query the visible state of the highlight specifying the currently selected text.

The setMagicCaretPosition()/getMagicCaretPosition() methods manage a dynamic caret position used when moving the caret up and down between lines with the arrow keys. When moving up and down between lines with an unequal number of characters, the magic position should place the caret as close to the same location within each line as possible. If the magic position is greater than the length of the current line, the caret should be placed at the end of the line. Note that this feature is common in almost all modern text applications, and is implemented for us in the DefaultCaret class.

The Caret interface also declares methods for the registration of ChangeListeners for notification of changes in the caret's position: addChangeListener(), removeChangeListener().

19.1.20 DefaultCaret

class javax.swing.text.DefaultCaret

This class extends java.awt.Rectangle, and represents a concrete implementation of the Caret interface used by all text components by default. It is rendered as a blinking vertical line in the color specified by its associated text component's caretColor property. DefaultCaret also implements the FocusListener, MouseListener, and MouseMotionListener interfaces.

The only MouseListener methods without empty implementations are mouseClicked() and mousePressed(). If a mouse click occurs with the left mouse button, and the click count is two (i.e. a double-click), mouseClicked() will invoke the Action returned by DefaultEditorKit.selectWordAction() to select the word containing the caret. If the click count is three, mouseClicked() will invoke the Action returned by DefaultEditorKit.selectLineAction() to select the line of text containing the caret. The mousePressed() method sends its MouseEvent parameter to DefaultCaret's positionCaret() method, which sets the dot property to the document offset corresponding to the mouse press, and clears the magicCaretPosition property. The mousePressed() method also checks to see if the text component is enabled, and if it is, its requestFocus() method is invoked.

The only MouseMotionListener method without an empty implementation is mouseDragged(). This method simply passes its MouseEvent parameter to DefaultCaret's moveCaret() method. The moveCaret() method determines the offset of the caret destination by passing the MouseEvent's coordinates to the text component's viewToModel() method. The moveDot() method is then invoked to actually move the caret to the determined position (recall that the moveDot() method sets the mark property and selects the text between the mark position and the new dot position).

Both FocusListener methods are nonempty. The focusGained() method checks whether the text component is editable, and if it is, the caret is made visible. The focusLost() method simply hides the caret. These methods are invoked when the text component gains or loses the focus.

We can customize the way a selection's highlight appears by overriding DefaultCaret's getSelectionPainter() method to return our own Highlighter.HighlightPainter implementation. We can also customize the appearance of a caret by overriding the paint() method. If we do reimplement the paint() method, however, we must also override the damage() method. The damage() method is passed a Rectangle representing the region of the text component to repaint when the caret is moved.

For instance, the following is a simple `DefaultCaret` sub-class that renders a wide black caret.[3]

```
class WideCaret extends DefaultCaret
{
  protected int caretWidth = 6;

  protected void setWidth(int w) {
    caretWidth = w;
  }

  // Since DefaultCaret extends Rectangle, it inherits
  // the x, y, width, and height variables which are
  // used here to allow proper repainting.
  protected synchronized void damage(Rectangle r) {
    if (r != null) {
      x = r.x - width;
      y = r.y;
      width = width;
      height = r.height;
      repaint();
    }
  }

  public void paint(Graphics g) {
    if(isVisible()) {
      try {
        TextUI mapper = getComponent().getUI();
        Rectangle r = mapper.modelToView(
          getComponent(), getComponent().getCaretPosition());
        g.setColor(getComponent().getCaretColor());
        g.fillRect(r.x, r.y, caretWidth, r.height - 1);
      }
      catch (Exception e) {
        System.err.println("Problem painting cursor");
      }
    }
  }
}
```

19.1.21 The CaretListener interface

abstract interface javax.swing.event.CaretListener

This interface describes a listener that is notified whenever a change occurs in a text component's caret position. It declares one method, `caretUpdate()`, which takes a `CaretEvent` as parameter. We can attach and remove `CaretListeners` to any `JTextComponent` with the `addCaretListener()` and `removeCaretListener()` methods respectively.

19.1.22 CaretEvent

class javax.swing.event.CaretEvent

This event simply encapsulates a reference to its source object (normally a text component). `CaretEvents` are passed to all attached `CaretListeners` whenever the associated text component's caret position changes.

[3] We have implemented a short example in a Swing Connection 'Tips and Tricks' article showing how to use a similar custom caret for designating an overwrite mode. In the same article we also show how to customize a `PlainDocument` model to allow 'insert' and 'overwrite' modes, and how to track caret position with a `CaretListener`. See http://java.sun.com/products/jfc/tsc/friends/tips_4/tips_4.html.

19.1.23 The Keymap interface

abstract interface javax.swing.text.Keymap

This interface describes a collection of bindings between `KeyStrokes` (see 2.13.2) and `Actions` (see 12.1.23). We are meant to add new `KeyStroke`/`Action` bindings to a `Keymap` with the `addActionForKeyStroke()` method. Like `AttributeSets`, `Keymaps` resolve hierarchically. Like `Styles`, `Keymaps` have a name used to reference them by.

We are meant to query the `Action` that corresponds to a specific `KeyStroke` with the `getAction()` method. If no corresponding `Action` is located in the `Keymap`, its resolving parents should be searched until either no more resolving parents exist, or a match is found. Similarly, we are intended to retrieve an array of `KeyStrokes` mapped to a given `Action` with the `getKeyStrokesForAction()` method.[4] The `isLocallyDefined()` method is meant to check whether or not a given `KeyStroke` is bound to an `Action` in the `Keymap` under investigation. The `removeBindings()` method should remove all bindings in a `Keymap`, and the `removeKeyStrokeBinding()` method is intended to remove only those bindings corresponding to a given `KeyStroke`.

By default, all `JTextComponents` share the same `Keymap` instance. This is what enables the default functionality of the Backspace, Delete, and left and right arrow keys on any text component. For this reason, it is not a good idea to retrieve a text component's `Keymap` and modify it directly. Rather, we are encouraged to create our own `Keymap` instance, and assign the default `Keymap` as its resolving parent. Also note that by assigning a resolving parent of `null`, we can effectively disable all bindings on a text component, other than those in the given component's `Keymap` itself (the underlying role `Keymaps` play in text components will become clear after we discuss `DefaultEditorKit` below).

We can obtain a text component's `Keymap` with either of `JTextComponent`'s `getKeymap()` methods. We can assign a text component a new `Keymap` with the `setKeymap()` method, and we can add a new `Keymap` anywhere within the `Keymap` hierarchy with the `addKeymap()` method. We can also remove a `Keymap` from the hierarchy with the `removeKeymap()` method.

For example, to create and add a new `Keymap` to a `JTextField` and use the default text component `Keymap` as resolving parent, we might do something like the following:

```
Keymap keymap = myJTextField.getKeymap();
Keymap myKeymap = myJTextField.addKeymap("MyKeymap", keymap);
```

We can then add `KeyStroke`/`Action` pairs to `myKeymap` with the `addActionForKeyStroke()` method (we will see an example of this in the next section).

Note: Recall from section 2.13.4 that `KeyListeners` will receive key events before a text component's `Keymap`. Although the use of `Keymaps` is encouraged, handling keyboard events with `KeyListeners` is still allowed.

19.1.24 TextAction

abstract class javax.swing.text.TextAction

`EditorKits` are, among other things, responsible for making a set of `Actions` available for performing common text editor functions based on a given content type. `EditorKits` normally use inner sub-classes of `TextAction` for this, as it extends `AbstractAction` (see 12.1.24), and provides a relatively powerful means of determining the target component to invoke the action on (by taking advantage of the fact that

[4] As of Java 2 FCS, this method simply returns `null`, and is marked as "`TBD`" (assumably meaning "to be done").

JTextComponent keeps track of the most recent text component with the focus, retrievable with its static getFocusedComponent() method). The TextAction constructor takes the String to be used as that action's name, and passes it to its super-class constructor. When sub-classing TextAction, we normally define an actionPerformed() method, which is responsible for performing the desired action when passed an ActionEvent. Within this method, we can use TextAction's getTextComponent() method to determine which text component the action should be invoked on.

19.1.25 EditorKit

abstract class javax.swing.text.EditorKit

EditorKits are responsible for the following functionality:

- Support for an appropriate Document model. An EditorKit is meant to specifically support one type of content, a String description of which should be retrievable with the getContentType() method. A corresponding Document instance should be returned by the createDefaultDocument() method, and the EditorKit should be able to read() and write() that Document to InputStreams/OutputStreams and Readers/Writers, respectively
- Support for View production through a ViewFactory implementation. This behavior is actually optional, as View production will default to a text component's UI delegate if its EditorKit's getViewFactory() method returns null (see 19.1.28 and 19.1.29 for more about Views and the ViewFactory interface).
- Support for a set of Actions that can be invoked on a text component using the appropriate Document. Normally these Actions are instances of TextAction and are defined as inner classes. An EditorKit's Actions are meant to be retrievable in an array with its getActions() method.

19.1.26 DefaultEditorKit

class javax.swing.text.DefaultEditorKit

DefaultEditorKit extends EditorKit, and defines a series of TextAction sub-classes and corresponding name Strings (see API docs). Eight of these forty-six inner action classes are public, and can be instantiated with a default constructor: BeepAction, CopyAction, CutAction, DefaultKeyTypedAction, InsertBreakAction, InsertContentAction, InsertTabAction, and PasteAction. DefaultEditorKit maintains instances of all its inner Action classes in an array retrievable with its getActions() method. We can access any of these Actions easily by defining a Hashtable with Action.NAME keys and Action values:[5]

```
Hashtable actionTable = new Hashtable
Action[] actions = myEditorKit.getActions();
for (int i=0; i < actions.length; i++) {
  String actionName = (String) actions[i].getValue(Action.NAME);
  actionTable.put(actionName, actions[i]);
}
```

We can then retrieve any of these Actions with DefaultEditorKit's static String fields. For example, the following retrieves the action responsible for selecting all text in a document:

```
Action selectAll = (Action) actionTable.get(
  DefaultEditorKit.selectAllAction);
```

These Actions can be used in menus and toolbars, or with other controls, for convenient control of plain text components.

[5] See p.918, Java Swing, by Robert Eckstein, Marc Loy, and Dave Wood, O'Reilly & Associates, 1998.

DefaultEditorKit's getViewFactory() method returns null, which means the UI delegate is responsible for creating the hierarchy of Views necessary for rendering a text component correctly. As we mentioned in the beginning of this chapter, JTextField, JPasswordField, and JTextArea all use a DefaultEditorKit.

Although EditorKits are responsible for managing a set of Actions and their corresponding names, they are not actually directly responsible for making these Actions accessible to specific text components. This is where Keymaps fit in. For instance, take a look at the following code showing how the default JTextComponent Keymap is created (from JTextComponent.java):

```
/**
 * This is the name of the default keymap that will be shared by all
 * JTextComponent instances unless they have had a different
 * keymap set.
 */
public static final String DEFAULT_KEYMAP = "default";

/**
 * Default bindings for the default keymap if no other bindings
 * are given.
 */
static final KeyBinding[] defaultBindings = {
  new KeyBinding(KeyStroke.getKeyStroke(KeyEvent.VK_BACK_SPACE, 0),
    DefaultEditorKit.deletePrevCharAction),
  new KeyBinding(KeyStroke.getKeyStroke(KeyEvent.VK_DELETE, 0),
    DefaultEditorKit.deleteNextCharAction),
  new KeyBinding(KeyStroke.getKeyStroke(KeyEvent.VK_RIGHT, 0),
    DefaultEditorKit.forwardAction),
  new KeyBinding(KeyStroke.getKeyStroke(KeyEvent.VK_LEFT, 0),
    DefaultEditorKit.backwardAction)
};

static {
  try {
    keymapTable = new Hashtable(17);
    Keymap binding = addKeymap(DEFAULT_KEYMAP, null);
    binding.setDefaultAction(new
      DefaultEditorKit.DefaultKeyTypedAction());
    EditorKit kit = new DefaultEditorKit();
    loadKeymap(binding, defaultBindings, kit.getActions());
  } catch (Throwable e) {
    e.printStackTrace();
    keymapTable = new Hashtable(17);
  }
}
```

19.1.27 StyledEditorKit

class javax.swing.text.StyledEditorKit

This class extends DefaultEditorKit and defines seven additional inner Action classes, each of which is publicly accessible: AlignmentAction, BoldAction, FontFamilyAction, FontSizeAction, ForegroundAction, ItalicAction, and UnderlineAction. All seven Actions are sub-classes of the inner StyledTextAction convenience class which extends TextAction.

Each of StyledEditorKit's Actions apply to styled text documents, and they are used by JEditorPane and JTextPane. StyledEditorKit does not define its own capabilities for reading and writing styled text.

Instead this functionality is inherited from `DefaultEditorKit` which only provides support for saving and loading plain text. The two `StyledEditorKit` sub-classes included with Swing, `javax.swing.text.html.HTMLEditorKit` and `javax.swing.text.rtf.RTFEditorKit`, do support styled text saving and loading for HTML and RTF content types respectively.

`StyledEditorKit`'s `getViewFactory()` method returns an instance of a private static inner class called `StyledViewFactory` which implements the `ViewFactory` interface as follows (from StyledEditorKit.java):

```
static class StyledViewFactory implements ViewFactory {
  public View create(Element elem) {
    String kind = elem.getName();
    if (kind != null) {
      if (kind.equals(AbstractDocument.ContentElementName)) {
        return new LabelView(elem);
      } else if (kind.equals(AbstractDocument.ParagraphElementName)) {
          return new ParagraphView(elem);
      } else if (kind.equals(AbstractDocument.SectionElementName)) {
          return new BoxView(elem, View.Y_AXIS);
      } else if (kind.equals(StyleConstants.ComponentElementName)) {
          return new ComponentView(elem);
      } else if (kind.equals(StyleConstants.IconElementName)) {
          return new IconView(elem);
      }
    }
    // default to text display
    return new LabelView(elem);
  }
}
```

The `Views` returned by this factory's `create()` method are based on the name property of the `Element` passed as parameter. If an `Element` is not recognized, a `LabelView` is returned. In sum, because `StyledEditorKit`'s `getViewFactory()` method doesn't return `null`, styled text components depend on their `EditorKits` rather than their UI delegates for providing `Views`. The opposite is true with plain text components, which rely on their UI delegate for `View` creation.

19.1.28 View

abstract class javax.swing.text.View

This class describes an object responsible for graphically representing a portion of a text component's document model. The text package includes several extensions of this class meant for use by various types of `Elements`. We will not discuss these classes in detail, but a brief overview will be enough to provide a high level understanding of how text components are actually rendered.[6]

- `abstract interface TabableView`: Used by `Views` whose size depends on the size of tabs.
- `abstract interface TabExpander`: This interface extends `TabableView` and is used by `Views` that support `TabStops` and `TabSets` (a set of `TabStops`). A `TabStop` describes the positioning of a tab character and the text appearing immediately after it.
- `class ComponentView`: Used as a gateway view to a fully interactive embedded `Component`.
- `class IconView`: Used as a gateway `View` to an embedded `Icon`.
- `class PlainView`: Used for rendering one line of non-wrapped text with one font and one color.

[6] We have only included the most commonly used set of text component `Views` in this list. There are several others responsible for significant text rendering functionality. See the O'Reilly book and API docs for details.

`class FieldView`: Extends `PlainView` and adds specialized functionality for representing a single-line editor view (i.e. the ability to center text in a `JTextField`).

`class PasswordView`: Extends `FieldView` and adds the ability to render its content using the echo character of the associated component if it is a `JPasswordField`.

`class LabelView`: Used to render a range of styled text.

`abstract class CompositeView`: A `View` containing multiple child `View`s. All `View`s can contain child `View`s, but only instances of `CompositeView` and `BasicTextUI`'s `RootView` (discussed below) actually contain child `View`s by default.

`class BoxView`: Extends `CompositeView` and arranges a group of child `View`s in a rectangular box.

`class ParagraphView`: Extends `BoxView` and is responsible for rendering a paragraph of styled text. `ParagraphView` is made up of a number of child `Element`s organized as, or within, `View`s representing single rows of styled text. This `View` supports line wrapping, and if an `Element` within the content paragraph spans multiple lines, more than one `View` will be used to represent it.

`class WrappedPlainView`: Extends `BoxView` and is responsible for rendering multi-line, plain text with line wrapping.

All text components in Swing use UI delegates derived from `BasicTextUI` by default. This class defines an inner class called `RootView` which acts as a gateway between a text component and the actual `View` hierarchy used to render it.

Note: In chapter 22 we will take advantage of `BasicTextUI`'s root view while implementing a solution for printing styled text. The solution also requires us to implement a custom `BoxView` sub-class responsible for rendering each of its child `View`s to a `Graphics` instance used in the printing process (see 22.4).

19.1.29 The ViewFactory interface

abstract interface javax.swing.text.ViewFactory

This interface declares one method: `create(Element elem)`. This method is responsible for returning a `View`, possibly containing a hierarchy of `View`s, used to render a given `Element`. `BasicTextUI` implements this interface, and unless a text component's `EditorKit` provides its own `ViewFactory`, `BasicTextUI`'s `create()` method will be responsible for providing all `View`s. This is the case with plain text components: `JTextField`, `JPasswordField`, and `JTextArea`. However, styled text components, `JEditorPane` and `JTextPane`, vary greatly depending on their current content type. For this reason their `View`s are provided by the currently installed `EditorKit`. In this way, custom `View`s can be provided to render different types of styled content.

19.2 Date and time editor

...by David M. Karr of BestConsulting and TCSI Corporation

The `DateTimeEditor` class is a panel containing a text field that allows display and editing of a date, time, or date/time value. It doesn't use direct entry of text, but uses the `Up` and `Down` arrow keys or mouse clicks on "spinner" buttons to increment and decrement field values (e.g. day, month, year, or hour, minute, second). The mouse can also be used to select particular subfields. The `Left` and `Right` arrow keys move the caret between fields.

This class is designed to be internationalized, although it assumes some conventions, such as a left-to-right reading direction. It doesn't have any locale-specific code, it just uses the locale framework integrated into Java 2. If the VM used doesn't support a particular locale, neither will this component. The `Locale` class

encapsulates a "language", "country", and optional "variant". Each of these are strings. The possible values of "language" and "country" are defined in the ISO 639 and ISO 3166 standards, respectively. The variants are not standardized. For instance, the language codes for English, French, Chinese, and Japanese are "en", "fr", "zh", and "ja". The country codes for the USA, France, and Canada are "US", "FR", and "CA".

The current `Locale` setting is used to qualify the variety of resource class or properties file to obtain. For instance, a class name with a suffix of "_fr_FR" indicates resources for french in France. The suffix of "_fr_CA" indicates resources for french in Canada.

Java 2 has specific resource settings for most of the known locales, including currency formats, date formats, number formats, common text strings, etcetera. This is the information that the `DateTimeEditor` class uses indirectly, without having to manually encode locale-specific information.

`DateTimeEditor` uses several Swing classes including: `JTextField`, `Keymap`, `AbstractAction`, `TextAction`, and `Caret`. It also uses several non-Swing classes including: `Collections`, `Calendar` (both in package java.util), `FieldPosition`, and `DateFormat` (both in package java.text). The `Collections` class is used to sort a list of `FieldPosition` objects by the `beginIndex` of each `FieldPosition`. A custom `Spinner` class, described below, is used to allow incrementing or decrementing of values with the mouse.

`DateTimeEditor`'s textfield is an ordinary `JTextField`, and it uses the methods of `JTextComponent` to communicate with and manipulate its `Caret`.

The inner classes `UpDownAction`, `BackwardAction`, `ForwardAction`, `BeginAction`, and `EndAction` are subclasses of `TextAction` and `AbstractAction`, and are used to handle the arrow keys, and the Home and End keys. All of these inner classes are used in concert with the `Keymap` class to combine key definitions with action definitions.

The `DateTimeEditor` textfield listens for caret state changes. It does this so it knows exactly which field the caret is in, and also to constrain the caret position to always be at the beginning of the current field.

The most interesting interactions are with `DateFormat`'s fields and its `format()` method. What `DateTimeEditor` gains from this is the ability to know what field the caret is in, so it knows how to interpret the "increment" and "decrement" actions.

One of `DateFormat`'s `format()` methods takes a `Date` value, a `StringBuffer` to write the stringified result into, and a single `FieldPosition` object. This last parameter is the key to the entire `DateTimeEditor` component. The `format()` method will update the given `FieldPosition` object with the `begin` and `end` offset for that field in the given date/time string. The `DateTimeEditor` has a hardcoded list of all fields from `DateFormat`. In a loop, it plugs in each of those constants into the `format` function, and then stores the resulting `FieldPosition` object. It then uses the `Collections.sort()` method to sort the list of `FieldPositions` by the beginning index of each. Using this sorted list and a given caret position, we can easily determine what field the caret resides in.

The `Calendar` class is used to fill in some functionality that the `Date.setTime()` method doesn't provide. In particular, in the code which increments or decrements the current field value, there are four `DateFormat` fields which cannot be set using `Date.setTime()`. Those fields are `MONTH`, `WEEK_OF_MONTH`, `WEEK_OF_YEAR`, and `YEAR`. For these fields, `DateTimeEditor` manipulates a `Calendar` instance and then calls `Calendar.getTime()` to get a new `Date` value.

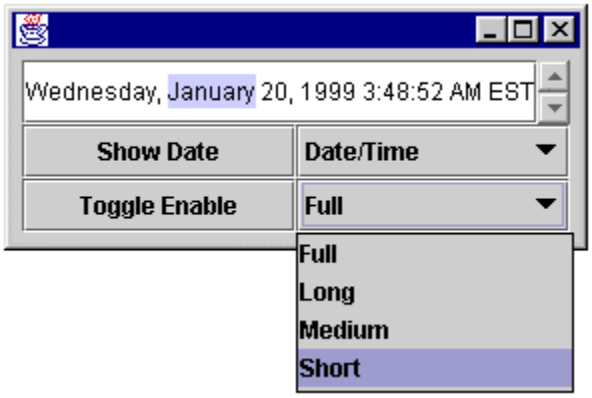


Figure 19.5 DateTimeEditor in the en_US locale
<<file figure19-5.gif>>

Figure 19.6 Spinner
<<file figure19-6.gif>>

The Code: DateTimeEditor.java
see \Chapter19\Karr

```
import java.awt.event.*;
import java.text.*;
import java.util.*;
import java.awt.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.text.*;
import javax.swing.event.*;

public class DateTimeEditor extends JPanel {
  public static final long ONE_SECOND = 1000;
  public static final long ONE_MINUTE = 60*ONE_SECOND;
  public static final long ONE_HOUR = 60*ONE_MINUTE;
  public static final long ONE_DAY = 24*ONE_HOUR;
  public static final long ONE_WEEK = 7*ONE_DAY;
  public final static int TIME = 0;
  public final static int DATE = 1;
  public final static int DATETIME = 2;
  private int m_timeOrDateType;
  private int m_lengthStyle;
  private DateFormat m_format;
  private Calendar m_calendar = Calendar.getInstance();
  private ArrayList m_fieldPositions = new ArrayList();
  private Date m_lastDate = new Date();
  private Caret m_caret;
  private int m_curField = -1;
  private JTextField m_textField;
  private Spinner m_spinner;
  private AbstractAction m_upAction =
    new UpDownAction(1, "up");
  private AbstractAction m_downAction =
    new UpDownAction(-1, "down");
  private int[] m_fieldTypes = {
```

```
    DateFormat.ERA_FIELD,
    DateFormat.YEAR_FIELD,
    DateFormat.MONTH_FIELD,
    DateFormat.DATE_FIELD,
    DateFormat.HOUR_OF_DAY1_FIELD,
    DateFormat.HOUR_OF_DAY0_FIELD,
    DateFormat.MINUTE_FIELD,
    DateFormat.SECOND_FIELD,
    DateFormat.MILLISECOND_FIELD,
    DateFormat.DAY_OF_WEEK_FIELD,
    DateFormat.DAY_OF_YEAR_FIELD,
    DateFormat.DAY_OF_WEEK_IN_MONTH_FIELD,
    DateFormat.WEEK_OF_YEAR_FIELD,
    DateFormat.WEEK_OF_MONTH_FIELD,
    DateFormat.AM_PM_FIELD,
    DateFormat.HOUR1_FIELD,
    DateFormat.HOUR0_FIELD
  };

  public DateTimeEditor() {
    m_timeOrDateType = DATETIME;
    m_lengthStyle = DateFormat.SHORT;
    init();
  }

  public DateTimeEditor(int timeOrDateType) {
    m_timeOrDateType = timeOrDateType;
    m_lengthStyle = DateFormat.FULL;
    init();
  }

  public DateTimeEditor(int timeOrDateType, int lengthStyle) {
    m_timeOrDateType = timeOrDateType;
    m_lengthStyle = lengthStyle;
    init();
  }

  private void init() {
    setLayout(new BorderLayout());
    m_textField = new JTextField();
    m_spinner = new Spinner();
    m_spinner.getIncrementButton().addActionListener(m_upAction);
    m_spinner.getDecrementButton().addActionListener(m_downAction);
    add(m_textField, "Center");
    add(m_spinner, "East");
    m_caret = m_textField.getCaret();
    m_caret.addChangeListener(new ChangeListener() {
      public void stateChanged(ChangeEvent evt) {
        setCurField();
      }
    });
    setupKeymap();
    reinit();
  }

  public int getTimeOrDateType() { return m_timeOrDateType; }
  public void setTimeOrDateType(int timeOrDateType) {
    m_timeOrDateType = timeOrDateType;
    reinit();
  }

  public int getLengthStyle() { return m_lengthStyle; }
```

```
public void setLengthStyle(int lengthStyle) {
  m_lengthStyle = lengthStyle;
  reinit();
}

public Date getDate() { return (m_lastDate); }
public void setDate(Date date) {
  m_lastDate = date;
  m_calendar.setTime(m_lastDate);
  m_textField.setText(m_format.format(m_lastDate));
  getFieldPositions();
}

private int getFieldBeginIndex(int fieldNum) {
  int beginIndex = -1;
  for (Iterator iter = m_fieldPositions.iterator();
   iter.hasNext(); )
  {
    FieldPosition fieldPos = (FieldPosition) iter.next();
    if (fieldPos.getField() == fieldNum) {
      beginIndex = fieldPos.getBeginIndex();
      break;
    }
  }
  return (beginIndex);
}

private FieldPosition getFieldPosition(int fieldNum) {
  FieldPosition result = null;
  for (Iterator iter = m_fieldPositions.iterator();
   iter.hasNext(); )
  {
    FieldPosition fieldPosition = (FieldPosition) iter.next();
    if (fieldPosition.getField() == fieldNum) {
      result = fieldPosition;
      break;
    }
  }
  return (result);
}

private void reinit() {
  setupFormat();
  setDate(m_lastDate);
  m_caret.setDot(0);
  setCurField();
  repaint();
}

protected void setupFormat() {
  switch (m_timeOrDateType) {
    case TIME:
      m_format = DateFormat.getTimeInstance(m_lengthStyle);
      break;
    case DATE:
      m_format = DateFormat.getDateInstance(m_lengthStyle);
      break;
    case DATETIME:
      m_format = DateFormat.getDateTimeInstance(m_lengthStyle,
      m_lengthStyle);
      break;
  }
```

```
}

protected class UpDownAction extends AbstractAction
{
  int m_direction; // +1 = up; -1 = down

  public UpDownAction(int direction, String name) {
    super(name);
    m_direction = direction;
  }

  public void actionPerformed(ActionEvent evt) {
    if (!this.isEnabled())
      return;
    boolean dateSet  = true;
    switch (m_curField) {
      case DateFormat.AM_PM_FIELD:
        m_lastDate.setTime(m_lastDate.getTime() +
          (m_direction * 12*ONE_HOUR));
        break;
      case DateFormat.DATE_FIELD:
      case DateFormat.DAY_OF_WEEK_FIELD:
      case DateFormat.DAY_OF_WEEK_IN_MONTH_FIELD:
      case DateFormat.DAY_OF_YEAR_FIELD:
        m_lastDate.setTime(m_lastDate.getTime() +
          (m_direction * ONE_DAY));
        break;
      case DateFormat.ERA_FIELD:
        dateSet = false;
        break;
      case DateFormat.HOUR0_FIELD:
      case DateFormat.HOUR1_FIELD:
      case DateFormat.HOUR_OF_DAY0_FIELD:
      case DateFormat.HOUR_OF_DAY1_FIELD:
        m_lastDate.setTime(m_lastDate.getTime() +
          (m_direction * ONE_HOUR));
        break;
      case DateFormat.MILLISECOND_FIELD:
        m_lastDate.setTime(m_lastDate.getTime() +
          (m_direction * 1));
        break;
      case DateFormat.MINUTE_FIELD:
        m_lastDate.setTime(m_lastDate.getTime() +
          (m_direction * ONE_MINUTE));
        break;
      case DateFormat.MONTH_FIELD:
        m_calendar.set(Calendar.MONTH,
          m_calendar.get(Calendar.MONTH) + m_direction);
        m_lastDate = m_calendar.getTime();
        break;
      case DateFormat.SECOND_FIELD:
        m_lastDate.setTime(m_lastDate.getTime() +
          (m_direction * ONE_SECOND));
        break;
      case DateFormat.WEEK_OF_MONTH_FIELD:
        m_calendar.set(Calendar.WEEK_OF_MONTH,
          m_calendar.get(Calendar.WEEK_OF_MONTH) +
          m_direction);
        m_lastDate = m_calendar.getTime();
        break;
      case DateFormat.WEEK_OF_YEAR_FIELD:
        m_calendar.set(Calendar.WEEK_OF_MONTH,
```

```
          m_calendar.get(Calendar.WEEK_OF_MONTH) +
          m_direction);
        m_lastDate = m_calendar.getTime();
        break;
      case DateFormat.YEAR_FIELD:
        m_calendar.set(Calendar.YEAR,
          m_calendar.get(Calendar.YEAR) + m_direction);
        m_lastDate = m_calendar.getTime();
        break;
      default:
        dateSet = false;
    }

    if (dateSet) {
      int fieldId = m_curField;
      setDate(m_lastDate);
      FieldPosition fieldPosition = getFieldPosition(fieldId);
      m_caret.setDot(fieldPosition.getBeginIndex());

      m_textField.requestFocus();
      repaint();
    }
  }
}

protected class BackwardAction extends TextAction
{
  BackwardAction(String name) { super(name); }

  public void actionPerformed(ActionEvent e) {
    JTextComponent target = getTextComponent(e);
    if (target != null) {
      int dot = target.getCaretPosition();
      if (dot > 0) {
        FieldPosition position = getPrevField(dot);
        if (position != null)
          target.setCaretPosition(
            position.getBeginIndex());
        else {
          position = getFirstField();
          if (position != null)
            target.setCaretPosition(
              position.getBeginIndex());
        }
      }
      else
        target.getToolkit().beep();
      target.getCaret().setMagicCaretPosition(null);
    }
  }
}

protected class ForwardAction extends TextAction
{
  ForwardAction(String name) { super(name); }

  public void actionPerformed(ActionEvent e) {
    JTextComponent target = getTextComponent(e);
    if (target != null) {
      FieldPosition position = getNextField(
        target.getCaretPosition());
      if (position != null)
```

```
        target.setCaretPosition(position.getBeginIndex());
      else {
        position = getLastField();
        if (position != null)
          target.setCaretPosition(
            position.getBeginIndex());
      }
      target.getCaret().setMagicCaretPosition(null);
    }
  }
}

protected class BeginAction extends TextAction
{
  BeginAction(String name) { super(name); }

  public void actionPerformed(ActionEvent e) {
    JTextComponent target = getTextComponent(e);
    if (target != null) {
      FieldPosition position = getFirstField();
      if (position != null)
        target.setCaretPosition(position.getBeginIndex());
    }
  }
}

protected class EndAction extends TextAction
{
  EndAction(String name) { super(name); }

  public void actionPerformed(ActionEvent e) {
    JTextComponent target = getTextComponent(e);
    if (target != null) {
      FieldPosition position = getLastField();
      if (position != null)
        target.setCaretPosition(position.getBeginIndex());
    }
  }
}

protected void setupKeymap() {
  Keymap keymap = m_textField.addKeymap("DateTimeKeymap", null);
  keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(
    KeyEvent.VK_UP, 0), m_upAction);
  keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(
    KeyEvent.VK_DOWN, 0), m_downAction);
  keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(
    KeyEvent.VK_LEFT, 0), new BackwardAction(DefaultEditorKit.
    backwardAction));
  keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(
    KeyEvent.VK_RIGHT, 0), new ForwardAction(DefaultEditorKit.
    forwardAction));
  keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(
    KeyEvent.VK_HOME, 0), new BeginAction(DefaultEditorKit.
    beginAction));
  keymap.addActionForKeyStroke(KeyStroke.getKeyStroke(
    KeyEvent.VK_END, 0), new EndAction(DefaultEditorKit.
    endAction));
  m_textField.setKeymap(keymap);
}

private void getFieldPositions() {
```

```
  m_fieldPositions.clear();
  for (int ctr = 0; ctr < m_fieldTypes.length; ++ ctr) {
    int fieldId = m_fieldTypes[ctr];
    FieldPosition fieldPosition = new FieldPosition(fieldId);
    StringBuffer formattedField = new StringBuffer();
    m_format.format(m_lastDate, formattedField, fieldPosition);
    if (fieldPosition.getEndIndex() > 0)
      m_fieldPositions.add(fieldPosition);
  }
  m_fieldPositions.trimToSize();
  Collections.sort(m_fieldPositions,
    new Comparator() {
      public int compare(Object o1, Object o2) {
        return (((FieldPosition) o1).getBeginIndex() -
         ((FieldPosition) o2).getBeginIndex());
      }
    }
  );
}

private FieldPosition getField(int caretLoc) {
  FieldPosition fieldPosition = null;
  for (Iterator iter = m_fieldPositions.iterator();
   iter.hasNext(); )
  {
    FieldPosition chkFieldPosition =
      (FieldPosition) iter.next();
    if ((chkFieldPosition.getBeginIndex() <= caretLoc) &&
     (chkFieldPosition.getEndIndex() > caretLoc))
    {
      fieldPosition = chkFieldPosition;
      break;
    }
  }
  return (fieldPosition);
}

private FieldPosition getPrevField(int caretLoc) {
  FieldPosition fieldPosition = null;
  for (int ctr = m_fieldPositions.size() - 1; ctr > -1; -- ctr) {
    FieldPosition chkFieldPosition =
      (FieldPosition) m_fieldPositions.get(ctr);
    if (chkFieldPosition.getEndIndex() <= caretLoc) {
      fieldPosition = chkFieldPosition;
      break;
    }
  }
  return (fieldPosition);
}

private FieldPosition getNextField(int caretLoc) {
  FieldPosition  fieldPosition = null;
  for (Iterator iter = m_fieldPositions.iterator();
   iter.hasNext(); )
  {
    FieldPosition chkFieldPosition =
      (FieldPosition) iter.next();
    if (chkFieldPosition.getBeginIndex() > caretLoc) {
      fieldPosition = chkFieldPosition;
      break;
    }
  }
```

```
    return (fieldPosition);
  }

  private FieldPosition getFirstField() {
    FieldPosition result = null;
    try { result = ((FieldPosition) m_fieldPositions.get(0)); }
    catch (NoSuchElementException ex) {}
    return (result);
  }

  private FieldPosition getLastField() {
    FieldPosition result = null;
    try {
      result =
        ((FieldPosition) m_fieldPositions.get(
          m_fieldPositions.size() - 1));
    }
    catch (NoSuchElementException ex) {}
    return (result);
  }

  private void setCurField() {
    FieldPosition fieldPosition = getField(m_caret.getDot());
    if (fieldPosition != null) {
      if (m_caret.getDot() != fieldPosition.getBeginIndex())
        m_caret.setDot(fieldPosition.getBeginIndex());
    }
    else {
      fieldPosition = getPrevField(m_caret.getDot());
      if (fieldPosition != null)
        m_caret.setDot(fieldPosition.getBeginIndex());
      else {
        fieldPosition = getFirstField();
        if (fieldPosition != null)
          m_caret.setDot(fieldPosition.getBeginIndex());
      }
    }

    if (fieldPosition != null)
      m_curField = fieldPosition.getField();
    else
      m_curField = -1;
  }

  public void setEnabled(boolean enable) {
    m_textField.setEnabled(enable);
    m_spinner.setEnabled(enable);
  }

  public boolean isEnabled() {
    return (m_textField.isEnabled() && m_spinner.isEnabled());
  }

  public static void main (String[] args) {
    JFrame frame = new JFrame();
    frame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent evt)
      { System.exit(0); }
    });

    JPanel panel = new JPanel(new BorderLayout());
    panel.setBorder(new EmptyBorder(5, 5, 5, 5));
```

```
    frame.setContentPane(panel);
    final DateTimeEditor field =
      new DateTimeEditor(DateTimeEditor.DATETIME,
      DateFormat.FULL);
    panel.add(field, "North");

    JPanel buttonBox = new JPanel(new GridLayout(2, 2));
    JButton showDateButton = new JButton("Show Date");
    buttonBox.add(showDateButton);

    final JComboBox timeDateChoice = new JComboBox();
    timeDateChoice.addItem("Time");
    timeDateChoice.addItem("Date");
    timeDateChoice.addItem("Date/Time");
    timeDateChoice.setSelectedIndex(2);
    timeDateChoice.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        field.setTimeOrDateType(timeDateChoice.
        getSelectedIndex());
      }
    });
    buttonBox.add(timeDateChoice);

    JButton toggleButton = new JButton("Toggle Enable");
    buttonBox.add(toggleButton);
    showDateButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt)
      { System.out.println(field.getDate()); }
    });
    toggleButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt)
      { field.setEnabled(!field.isEnabled());}
    });
    panel.add(buttonBox, "South");

    final JComboBox lengthStyleChoice = new JComboBox();
    lengthStyleChoice.addItem("Full");
    lengthStyleChoice.addItem("Long");
    lengthStyleChoice.addItem("Medium");
    lengthStyleChoice.addItem("Short");
    lengthStyleChoice.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent evt) {
        field.setLengthStyle(lengthStyleChoice.
          getSelectedIndex());
      }
    });
    buttonBox.add(lengthStyleChoice);

    frame.pack();
    Dimension dim = frame.getToolkit().getScreenSize();
    frame.setLocation(dim.width/2 - frame.getWidth()/2,
      dim.height/2 - frame.getHeight()/2);
    frame.show();
  }
}
```

The Code: Spinner.java
see \Chapter19\Karr

```
import java.util.*;
import java.lang.reflect.*;
```

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

public class Spinner extends JPanel
{
  private int m_orientation = SwingConstants.VERTICAL;
  private BasicArrowButton m_incrementButton;
  private BasicArrowButton m_decrementButton;

  public Spinner() { createComponents(); }

  public Spinner(int orientation) {
    m_orientation = orientation;
    createComponents();
  }

  public void setEnabled(boolean enable) {
    m_incrementButton.setEnabled(enable);
    m_decrementButton.setEnabled(enable);
  }

  public boolean isEnabled() {
    return (m_incrementButton.isEnabled() &&
      m_decrementButton.isEnabled());
  }

  protected void createComponents() {
    if (m_orientation == SwingConstants.VERTICAL) {
      setLayout(new GridLayout(2, 1));
      m_incrementButton = new BasicArrowButton(
        SwingConstants.NORTH);
      m_decrementButton = new BasicArrowButton(
        SwingConstants.SOUTH);
      add(m_incrementButton);
      add(m_decrementButton);
    }
    else if (m_orientation == SwingConstants.HORIZONTAL) {
      setLayout(new GridLayout(1, 2));
      m_incrementButton = new BasicArrowButton(
        SwingConstants.EAST);
      m_decrementButton = new BasicArrowButton(
        SwingConstants.WEST);
      add(m_decrementButton);
      add(m_incrementButton);
    }
  }

  public JButton getIncrementButton() {
    return (m_incrementButton); }
  public JButton getDecrementButton() {
    return (m_decrementButton); }

  public static void main(String[] args) {
    JFrame frame = new JFrame();
    JPanel panel = (JPanel) frame.getContentPane();
    panel.setLayout(new BorderLayout());
    JTextField  field = new JTextField(20);
    Spinner spinner = new Spinner();

    panel.add(field, "Center");
```

```
    panel.add(spinner, "East");

    Dimension dim = frame.getToolkit().getScreenSize();
    frame.setLocation(dim.width/2 - frame.getWidth()/2,
      dim.height/2 - frame.getHeight()/2);
    frame.pack();
    frame.show();
  }
}
```

Understanding the Code:

Class DateTimeEditor

The `m_fieldTypes` array contains all of the field alignment constants defined in the `DateFormat` class. These are all of the pieces of a time or date value that we should expect to see. The order in this list is not important. Each value is plugged into `DateFormat.format()` to determine where each field is in the stringified date/time value.

The default constructor makes the field display date and time, in a `SHORT` format, which the `DateFormat` class describes as "completely numeric", such as "12.13.52" or "3:30pm". The second constructor can specify whether the field will display time, date, or date and time. In addition, it sets it into the `FULL` format, which the `DateFormat` class describes as "pretty completely specified", such as "Tuesday, April 12, 1952 AD" or "3:30:42pm PST". The third constructor can specify the time/date type, and the length style, being `SHORT`, `MEDIUM`, `LONG`, or `FULL` (fields in `DateFormat`).

Each of the constructors calls a common `init()` method, which initializes the caret, registers a `ChangeListener` on the caret (to update which field the caret is in), sets up the keymap (up, down, left, and right arrow keys), and calls the `reinit()` method which does some additional initialization (this method can be called any time, not just during initial construction).

The `setupKeymap()` method defines the keymap for the Up, Down, Left, and Right arrow keys. It first adds a new keymap with a `null` parent, so that no other keymaps will be used. It associates `Actions` with the key strokes we want to allow. Then the `setKeymap()` method is called to assign this keymap to our text field.

Each time a new date is set, either at initialization or by changing one of the field values, the `getFieldPositions()` method is called. This method uses the `DateFormat.format()` method, plugging in the `Date` value, and each one of the `DateFormat` fields. A new `FieldPosition` object is set which specifies the beginning and end indices for each field of the given date. All of the resulting `FieldPosition` objects are stored into the `m_fieldPositions` list, and sorted using the beginning index (using the `Collections` class). It is sorted in this fashion to make it easy to determine the field associated with a particular caret location. The `BackwardAction` and `ForwardAction` classes (see below) use this sorted list to quickly move to the previous or next date/time value.

After the `m_fieldPositions` list is set, several methods search that list, either directly or indirectly, to move to a particular field, or find out what the current field is. The `getField()`, `getPrevField()`, and `getNextField()` methods all take a caret location and return the current, previous, or next field, respectively. The `getFirstField()` and `getLastField()` methods return the first and last fields, respectively. And finally, the `setCurField()` method gets the field the caret is in and adjusts the caret to lie at the beginning of the field. This method is used when a new date is set, or the user uses the mouse to set the caret location.

The `setEnabled()` and `isEnabled()` methods allow the component to be disabled or enabled, and to check on the enabled status of the component (which includes both the text field and the custom spinner).

The `main()` method of this class is used as a demonstration of its capabilities. It presents a `DateTimeEditor`, a "Show Date" button, and a "Toggle Enable" button. When the "Show Date" button is pressed, it prints the current date value shown in the field to standard output. (The string printed is always in the "english US" locale, irrespective of the current locale being used to display the `DateTimeEditor`.) When the "Toggle Enable" button is pressed, it will toggle the enabled status of the component which grays out the text field and the spinner buttons when disabled.

As of the first official Java 2 public release there is a bug in the area of distribution of key events. In the method `setupKeymap()`, we specifically limit the keymap so that only six keystrokes should be recognized in the component, the four arrow keys and the Home and End keys. However, as a result of this bug, some platforms will allow normal characters to be inserted into the field, violating the integrity of the `Date` value.

To work around this, a small amount of code can be added to this example to avoid the problem. The solution requires two pieces:

1. In the `setDate()` method, which is the only place where the text of the field should be modified, we toggle a flag just before and after setting the text, indicating that we are trying to set the text of the field.

2. We create a new class, `DateTimeDocument`, extending `PlainDocument`, and send an instance of this class to the `setDocument` method of `JTextField`. The `insertString()` method of `DateTimeDocument` only calls `super.insertString()` if the flag (from item 1) is true.

The exact changes are the following:

1. Add the declaration of `m_settingDateText` to the variables section:

```
private boolean m_settingDateText = false;
```

2. Change the `setDate` method to the following:

```
public void setDate(Date date) {
  m_lastDate = date;
  m_calendar.setTime(m_lastDate);
  m_settingDateText = true;
  m_textField.setText(m_format.format(m_lastDate));
  m_settingDateText = false;
  getFieldPositions();
}
```

3. In the `init` method, send an instance of `DateTimeDocument` to the `setDocument` method of the `JTextField` instance to set the `Document`:

```
m_textField.setDocument(new DateTimeDocument());
```

3. Add the `DateTimeDocument` class:

```
protected class DateTimeDocument extends PlainDocument
{
  public void insertString(int offset,
   String str, AttributeSet a) throws BadLocationException
  {
    if (m_settingDateText)
      super.insertString(offset, str, a);
  }
}
```

Class DateTimeEditor.UpDownAction

The `UpDownAction` class is used as the action for the "up" and "down" arrow keys. When executed, this will increment or decrement the value of the field the caret is in. When values "roll over" (or "roll down"), like incrementing the day from "31" to "1", then this will change other fields, like the month field, in this example. One instance of this class is used to move in the "up" direction, and one instance is used to move in the "down" direction. for each field, it calculates the new time or date value, and uses `Date.setTime()` or `Calendar.set()` to set the new date or time. It will check for all of the field types specified in the `DateFormat` class (also listed in the `m_fieldTypes` array), although several would never be seen in certain locales. If the component is presently disabled, no modifications will be performed on the data.

Class DateTimeEditor.BackwardAction

The `BackwardAction` class is used as the action for the left arrow key. When executed, it will move the text caret from the beginning of one field to the beginning of the previous field. It uses the `getPrevField()` method to get the field previous to the current one.

Class DateTimeEditor.ForwardAction

The `ForwardAction` class is used as the action for the right arrow key. When executed, it will move the text caret from the beginning of the current field to the beginning of the next field. It uses the `getNextField()` method to get the field following the current one.

Class DateTimeEditor.BeginAction & DateTimeEditor.EndAction

The `BeginAction` and `EndAction` classes move the text caret to the beginning of the first and last fields, respectively.

Class Spinner

The `Spinner` class just uses two `BasicArrowButtons`, in either a vertical or horizontal orientation. It provides an API to get the increment or decrement buttons so you can attach listeners to them.

Running the Code

`DateTimeEditor` can be compiled and executed as is. By default, it will present a date/time value in the current locale. You can experiment with this by setting the "`LANG`" environment variable to a legal locale string. It's possible that not all legal locale strings will show any difference in the presentation, or even be correctly recognized. I found only major locales like "es" (spanish), "fr" (french), and "it" (italian) would work.

When you push the "Show Date" button, it will print the english value of the `Date` to standard output. When you push the "Toggle Enable" button, it will toggle the enabled state of the text field. When it is disabled, the text is slightly grayed out, the up and down arrow keys do nothing, and the spinner buttons are insensitive. Figure 19.5 shows `DateTimeEditor` in action.

In addition, the `Spinner` class can be compiled and run as a standalone demonstration. When run, it will present an empty text field with the spinner buttons to the right of it. As presented, it doesn't do much, not showing any behavioral connection between the component (the text field) and the `Spinner`, but this does show what the `Spinner` looks like when connected to a component. Figure 19.6 shows what the `Spinner` class looks like when run.

Chapter 20. Constructing a Word Processor

In this chapter:

- Word Processor: part I - Introducing RTF
- Word Processor: part II - Managing fonts
- Word Processor: part III - Colors and images
- Word Processor: part IV - Working with styles
- Word Processor: part V - Clipboard and undo/redo
- Word Processor: part VI - Advanced font mangement
- Word Processor: part VII - Paragraph formatting
- Word Processor: part VIII - Find and replace
- Word Processor: part IX - Spellchecker [using JDBC and SQL]

This chapter is devoted to the construction of a fully-functional RTF word processor application. Though Swing's HTML and RTF capabilities are very powerful, they are not yet complete. RTF support is further along than HTML, and this is why we chose to design our word processor for use with RTF documents.[7] The examples in this chapter demonstrate practical applications of many of the topics covered in chapter 19. The main focus throughout is working with styled text documents, and the techniques discussed here can be applied to any styled text editor.

Note: When running the examples in this chapter, do not be surprised when you see a series of 'unknown keyword' warnings or exception problems with various Views. You will also see the following message displayed to emphasize the fact that RTF support is still in the works: "Problems encountered: Note that RTF support is still under development."

20.1 Word Processor: part I - Introducing RTF

This basic example uses the capabilities of `JTextPane` and `RTFEditorKit` to display and edit RTF documents. It demonstrates very basic word processor functionality, opening and saving an RTF file, and serves as the foundation for our word processor application to be expanded upon throughout this chapter.

Note: In this series of examples our goal is to demonstrate the most significant available features of advanced text editing in Swing (even if they do not all currently work properly). To avoid losing focus of this goal we intentionally omit several typical word processor features such as an MDI interface, status bar, and prompts to save the current file before closing.

---

[7] Expect to see complete HTML coverage and examples in a future edition of this book. Also keep an eye on the Swing Connection site for updates. As we go to print there are rumors of several HTML editor examples in the works.

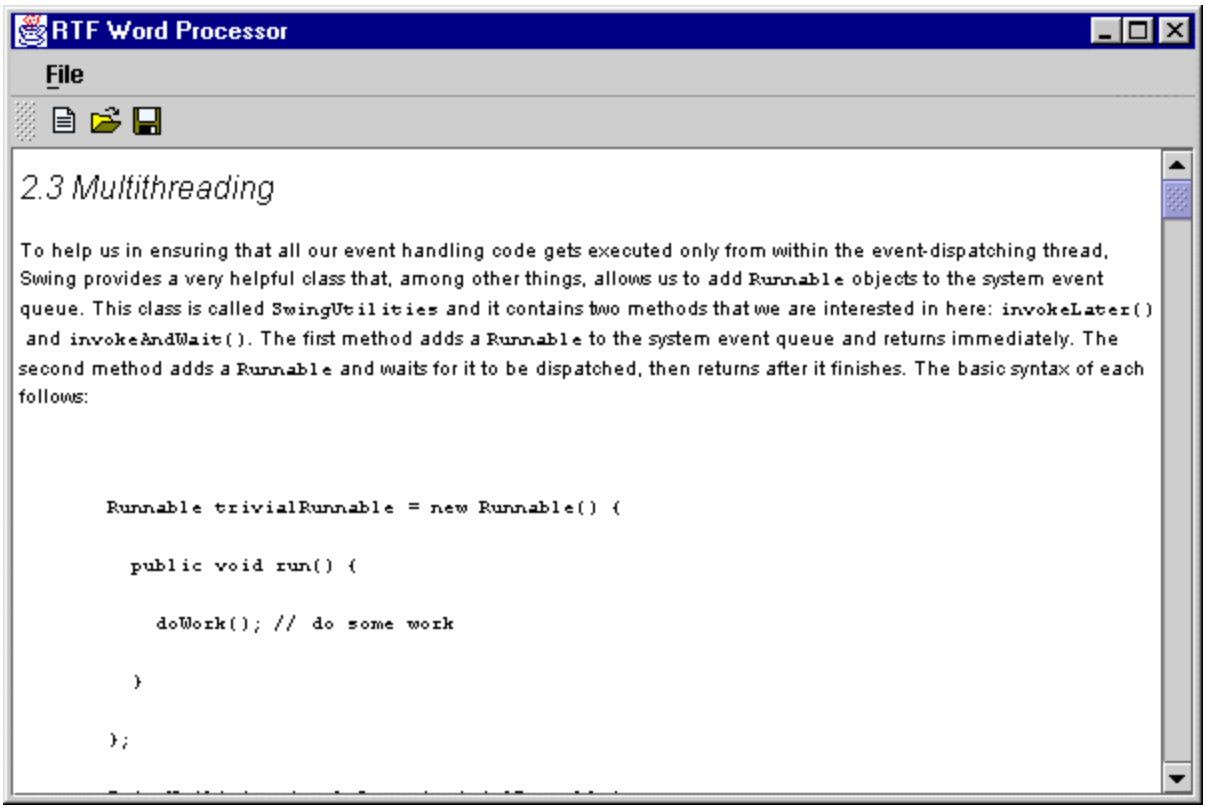


Figure 20.1 JTextPane displaying an RTF document.
<<file figure20-1.gif>>

The Code: WordProcessor.java
see \Chapter20\1

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

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;

public class WordProcessor extends JFrame
{
  protected JTextPane m_monitor;
  protected StyleContext m_context;
  protected DefaultStyledDocument m_doc;
  protected RTFEditorKit m_kit;
  protected JFileChooser m_chooser;
  protected SimpleFilter m_rtfFilter;
  protected JToolBar m_toolBar;

  public WordProcessor() {
    super("RTF Word Processor");
    setSize(600, 400);

    // Make sure we install the editor kit before creating
    // the initial document.
```

```
    m_monitor = new JTextPane();
    m_kit = new RTFEditorKit();
    m_monitor.setEditorKit(m_kit);
    m_context = new StyleContext();
    m_doc = new DefaultStyledDocument(m_context);
    m_monitor.setDocument(m_doc);

    JScrollPane ps = new JScrollPane(m_monitor);
    getContentPane().add(ps, BorderLayout.CENTER);

    JMenuBar menuBar = createMenuBar();
    setJMenuBar(menuBar);

    m_chooser = new JFileChooser();
    m_chooser.setCurrentDirectory(new File("."));
    m_rtfFilter = new SimpleFilter("rtf", "RTF Documents");
    m_chooser.setFileFilter(m_rtfFilter);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  protected JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();

    JMenu mFile = new JMenu("File");
    mFile.setMnemonic('f');

    ImageIcon iconNew = new ImageIcon("file_new.gif");
    Action actionNew = new AbstractAction("New", iconNew) {
      public void actionPerformed(ActionEvent e) {
        m_doc = new DefaultStyledDocument(m_context);
        m_monitor.setDocument(m_doc);
      }
    };
    JMenuItem item =  mFile.add(actionNew);
    item.setMnemonic('n');

    ImageIcon iconOpen = new ImageIcon("file_open.gif");
    Action actionOpen = new AbstractAction("Open...", iconOpen) {
      public void actionPerformed(ActionEvent e) {
        WordProcessor.this.setCursor(
          Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
        Thread runner = new Thread() {
          public void run() {
            if (m_chooser.showOpenDialog(WordProcessor.this) !=
             JFileChooser.APPROVE_OPTION)
              return;
            WordProcessor.this.repaint();
            File fChoosen = m_chooser.getSelectedFile();

            // Recall that text component read/write operations are
            // thread safe. Its ok to do this in a separate thread.
            try {
              InputStream in = new FileInputStream(fChoosen);
              m_doc = new DefaultStyledDocument(m_context);
```

```
          m_kit.read(in, m_doc, 0);
          m_monitor.setDocument(m_doc);
          in.close();
        }
        catch (Exception ex) {
          ex.printStackTrace();
        }
        WordProcessor.this.setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
      }
    };
    runner.start();
  }
};
item =  mFile.add(actionOpen);
item.setMnemonic('o');

ImageIcon iconSave = new ImageIcon("file_save.gif");
Action actionSave = new AbstractAction("Save...", iconSave) {
  public void actionPerformed(ActionEvent e) {
    WordProcessor.this.setCursor(
      Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    Thread runner = new Thread() {
      public void run() {
        if (m_chooser.showSaveDialog(WordProcessor.this) !=
         JFileChooser.APPROVE_OPTION)
          return;
        WordProcessor.this.repaint();
        File fChoosen = m_chooser.getSelectedFile();

        // Recall that text component read/write operations are
        // thread safe. Its ok to do this in a separate thread.
        try {
          OutputStream out = new FileOutputStream(fChoosen);
          m_kit.write(out, m_doc, 0, m_doc.getLength());
          out.close();
        }
        catch (Exception ex) {
          ex.printStackTrace();
        }

        // Make sure chooser is updated to reflect new file
        m_chooser.rescanCurrentDirectory();
        WordProcessor.this.setCursor(Cursor.getPredefinedCursor(
          Cursor.DEFAULT_CURSOR));
      }
    };
    runner.start();
  }
};
item = mFile.add(actionSave);
item.setMnemonic('s');

mFile.addSeparator();

Action actionExit = new AbstractAction("Exit") {
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
};

item =  mFile.add(actionExit);
```

```
    item.setMnemonic('x');
    menuBar.add(mFile);

    m_toolBar = new JToolBar();
    JButton bNew = new SmallButton(actionNew, "New document");
    m_toolBar.add(bNew);

    JButton bOpen = new SmallButton(actionOpen, "Open RTF document");
    m_toolBar.add(bOpen);

    JButton bSave = new SmallButton(actionSave, "Save RTF document");
    m_toolBar.add(bSave);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    return menuBar;
  }

  public static void main(String argv[]) {
    new WordProcessor();
  }
}

// Class SmallButton unchanged from section 12.4

// Class SimpleFilter unchanged from section 14.1.9
```

Understanding the Code

Class WordProcessor

This class extends `JFrame` to provide the supporting frame for this example. Several instance variables are declared:

- `JTextPane m_monitor`: text component.
- `StyleContext m_context`: a group of styles and their associated resources for the documents in this example.
- `DefaultStyledDocument m_doc`: current document model.
- `RTFEditorKit m_kit`: editor kit that knows how to read/write RTF documents.
- `JFileChooser m_chooser`: file chooser used to load and save RTF files.
- `SimpleFilter m_rtfFilter`: file filter for "*.rtf" files.
- `JToolBar m_toolBar`: toolbar containing open, save, and new document buttons.

The `WordProcessor` constructor first instantiates our `JTextPane` and `RTFEditorKit`, and assigns the editor kit to the text pane (it is important that this is done before any documents are created). Next our `StyleContext` is instantiated and we build our `DefaultStyledDocument` with it. The `DefaultStyledDocument` is then set as our text pane's current document.

The `createMenuBar()` method creates a menu bar with a single menu titled "File." Menu items "New," "Open," "Save," and "Exit" are added to the menu. The first three items are duplicated in the toolbar. This code is very similar to the code used in the examples of chapter 12. The important difference is that we use `InputStreams` and `OutputStreams` rather than `Readers` and `Writers`. The reason for this is that RTF

uses 1-byte encoding which is incompatible with the 2-byte encoding used by readers and writers.

Warning: An attempt to invoke `read()` will throw an exception when `JTextPane` is using an `RTFEditorKit`.

Running the Code

Use menu or toolbar buttons to open an `RTF` file (a sample `RTF` file is provided in the `\swing\Chapter20` directory). Save the `RTF` file and open it in another `RTF`-aware application (such as Microsoft Word) to verify compatibility.

20.2 Word Processor: part II - Managing fonts

The following example adds the ability to select any font available on the system. This functionality is similar to the "Font" menu used in the examples of chapter 12. The important difference here is that the selected font applies not to the whole text component (the only possible thing with plain text documents), but to the selected region of our `RTF` styled document text.

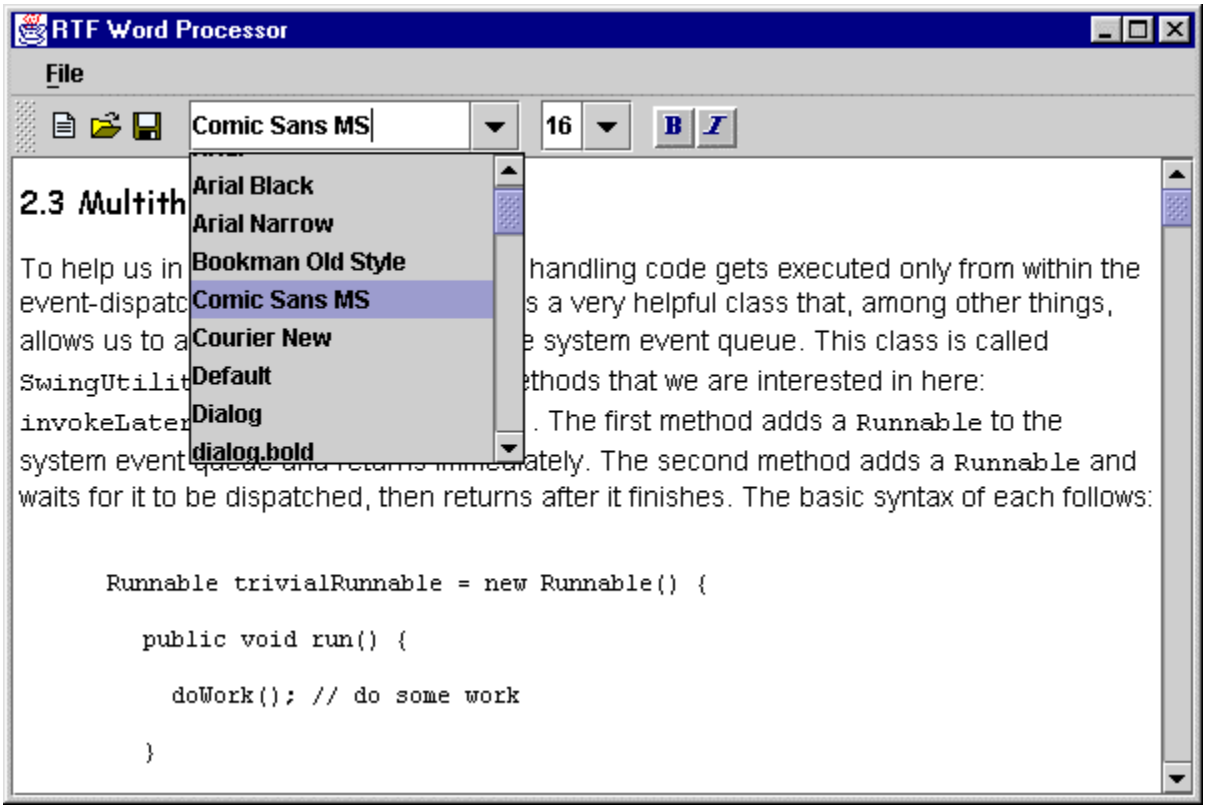


Figure 20.2 JTextPane word processor allowing font attribute assignments to selected text.

<<file figure20-1.gif>>

The Code: WordProcessor.java

see \Chapter20\2

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

```
import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;

public class WordProcessor extends JFrame
{
  protected JTextPane m_monitor;
  protected StyleContext m_context;
  protected DefaultStyledDocument m_doc;
  protected RTFEditorKit m_kit;
  protected JFileChooser m_chooser;
  protected SimpleFilter m_rtfFilter;
  protected JToolBar m_toolBar;

  protected JComboBox m_cbFonts;
  protected JComboBox m_cbSizes;
  protected SmallToggleButton m_bBold;
  protected SmallToggleButton m_bItalic;

  protected String m_fontName = "";
  protected int m_fontSize = 0;
  protected boolean m_skipUpdate;

  protected int m_xStart = -1;
  protected int m_xFinish = -1;

  public WordProcessor() {
    // Unchanged code from section 20.1

    CaretListener lst = new CaretListener() {
      public void caretUpdate(CaretEvent e) {
        showAttributes(e.getDot());
      }
    };
    m_monitor.addCaretListener(lst);

    FocusListener flst = new FocusListener() {
      public void focusGained(FocusEvent e) {
        if (m_xStart>=0 && m_xFinish>=0)
          if (m_monitor.getCaretPosition()==m_xStart) {
            m_monitor.setCaretPosition(m_xFinish);
            m_monitor.moveCaretPosition(m_xStart);
          }
          else
            m_monitor.select(m_xStart, m_xFinish);
      }

      public void focusLost(FocusEvent e) {
        m_xStart = m_monitor.getSelectionStart();
        m_xFinish = m_monitor.getSelectionEnd();
      }
    };
    m_monitor.addFocusListener(flst);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);
```

```
    showAttributes(0);
    setVisible(true);
  }

  protected JMenuBar createMenuBar() {
    // Unchaged code from section 20.1

    // The following line is added to the end of the
    // actionNew and actionOpen actionPerformed() methods:
    //
    //   showAttributes(0);
    //
    // (see source code; these methods are not shown here
    //  to conserve space)

    // Unchaged code from section 20.1

    GraphicsEnvironment ge = GraphicsEnvironment.
      getLocalGraphicsEnvironment();
    String[] fontNames = ge.getAvailableFontFamilyNames();

    m_toolBar.addSeparator();
    m_cbFonts = new JComboBox(fontNames);
    m_cbFonts.setMaximumSize(m_cbFonts.getPreferredSize());
    m_cbFonts.setEditable(true);

    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_fontName = m_cbFonts.getSelectedItem().toString();
        MutableAttributeSet attr = new SimpleAttributeSet();
        StyleConstants.setFontFamily(attr, m_fontName);
        setAttributeSet(attr);
        m_monitor.grabFocus();
      }
    };
    m_cbFonts.addActionListener(lst);
    m_toolBar.add(m_cbFonts);

    m_toolBar.addSeparator();
    m_cbSizes = new JComboBox(new String[] {"8", "9", "10",
      "11", "12", "14", "16", "18", "20", "22", "24", "26",
      "28", "36", "48", "72"});
    m_cbSizes.setMaximumSize(m_cbSizes.getPreferredSize());
    m_cbSizes.setEditable(true);

    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int fontSize = 0;
        try {
          fontSize = Integer.parseInt(m_cbSizes.
            getSelectedItem().toString());
        }
        catch (NumberFormatException ex) { return; }

        m_fontSize = fontSize;
        MutableAttributeSet attr = new SimpleAttributeSet();
        StyleConstants.setFontSize(attr, fontSize);
        setAttributeSet(attr);
        m_monitor.grabFocus();
      }
    };
```

```
    m_cbSizes.addActionListener(lst);
    m_toolBar.add(m_cbSizes);

    m_toolBar.addSeparator();
    ImageIcon img1 = new ImageIcon("font_bold1.gif");
    ImageIcon img2 = new ImageIcon("font_bold2.gif");
    m_bBold = new SmallToggleButton(false, img1, img2,
      "Bold font");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        MutableAttributeSet attr = new SimpleAttributeSet();
        StyleConstants.setBold(attr, m_bBold.isSelected());
        setAttributeSet(attr);
        m_monitor.grabFocus();
      }
    };
    m_bBold.addActionListener(lst);
    m_toolBar.add(m_bBold);

    img1 = new ImageIcon("font_italic1.gif");
    img2 = new ImageIcon("font_italic2.gif");
    m_bItalic = new SmallToggleButton(false, img1, img2,
      "Italic font");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        MutableAttributeSet attr = new SimpleAttributeSet();
        StyleConstants.setItalic(attr, m_bItalic.isSelected());
        setAttributeSet(attr);
        m_monitor.grabFocus();
      }
    };
    m_bItalic.addActionListener(lst);
    m_toolBar.add(m_bItalic);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    return menuBar;
  }

  protected void showAttributes(int p) {
    m_skipUpdate = true;
    AttributeSet a = m_doc.getCharacterElement(p).
      getAttributes();
    String name = StyleConstants.getFontFamily(a);
    if (!m_fontName.equals(name)) {
      m_fontName = name;
      m_cbFonts.setSelectedItem(name);
    }
    int size = StyleConstants.getFontSize(a);
    if (m_fontSize != size) {
      m_fontSize = size;
      m_cbSizes.setSelectedItem(Integer.toString(m_fontSize));
    }
    boolean bold = StyleConstants.isBold(a);
    if (bold != m_bBold.isSelected())
      m_bBold.setSelected(bold);
    boolean italic = StyleConstants.isItalic(a);
    if (italic != m_bItalic.isSelected())
      m_bItalic.setSelected(italic);
    m_skipUpdate = false;
  }
```

```
  protected void setAttributeSet(AttributeSet attr) {
    if (m_skipUpdate)
      return;
    int xStart = m_monitor.getSelectionStart();
    int xFinish = m_monitor.getSelectionEnd();
    if (!m_monitor.hasFocus()) {
      xStart = m_xStart;
      xFinish = m_xFinish;
    }
    if (xStart != xFinish) {
      m_doc.setCharacterAttributes(xStart, xFinish - xStart,
        attr, false);
    }
    else {
      MutableAttributeSet inputAttributes =
        m_kit.getInputAttributes();
      inputAttributes.addAttributes(attr);
    }
  }

  public static void main(String argv[]) {
        new WordProcessor();
  }
}

// Unchanged code from section 20.1

// Class SmallToggleButton unchanged from section 12.4
```

Understanding the Code

Class WordProcessor

Several new instance variables have been added:

- `JComboBox m_cbFonts`: toolbar component to select the font name.
- `JComboBox m_cbSizes`: toolbar component to select the font size.
- `SmallToggleButton m_bBold`: toolbar component to select the bold font style.
- `SmallToggleButton m_bItalic`: toolbar component to select the italic font style.
- `String m_fontName`: current font name.
- `int m_fontSize`: current font size.
- `boolean m_skipUpdate`: flag used to skip word processor update (see below).
- `int m_xStart`: used to store the selection start position.
- `int m_xFinish`: used to store the selection end position.

The constructor of the `WordProcessor` class adds a `CaretListener` to our `m_monitor` text pane. The `caretUpdate()` method of this listener is invoked whenever the caret position is changed. The `showAttributes()` (see below) will be called in response to update the toolbar components and display the currently selected font attributes.

A `FocusListener` is also added to our `m_monitor` component. The two methods of this listener, `focusGained()` and `focusLost()`, will be invoked when the editor gains and loses the focus respectively.

The purpose of this implementation is to save and restore the starting and end positions of the text selection. The reason we do this is because Swing supports only one text selection at any time throughout an app. This means that if the user selects some text in the editor component to modify it's attributes, and then goes off and makes a text selection in some other component, the original text selection will disappear. This can potentially be very annoying to the user. To fix this problem we save the selection before the editor component loses the focus. When the focus is gained we restore the previously saved selection. We distinguish between two possible situations: when the caret is located at the beginning of the selection and when it is located at the end of the selection. In the first case we position the caret at the end of the stored interval with the `setCaretPosition()` method, and then move the caret backward to the beginning of the stored interval with the `moveCaretPosition()` method. The second situation is easily handled using the `select()` method.

The `showAttributes()` method is now called prior to the display of a new document or a newly loaded document.

The `createMenuBar()` method creates new components to manage font properties for the selected text interval. First, the `m_cbFonts` combo box is used to select the font family name. Unlike the example in chapter 12, which used several pre-defined font names, this example uses all fonts available to the user's system. A complete list of the available font names can be obtained through the `getAvailableFontFamilyNames()` method of `GraphicsEnvironment` (see 2.8). Also note that the `editable` property of this combobox component is set to `true`, so the font name can be both selected from the drop-down list and enetered in by hand.

Once a new font name is selected, it is applied to the selected text through the use of an attached `ActionListener`. The selected font family name is assigned to a `SimpleAttributeSet` instance with the `StyleConstants.setFontFamily()` method. Then our custom `setAttributeSet()` (see below) is called to modify the attributes of the selected text according to this `SimpleAttributeSet`.

The `m_cbSizes` combo box is used to select the font size. It is initiated with a set of pre-defined sizes. The `editable` property is set to `true` so the font size can be both selected from the drop-down list and entered by hand. Once a new font size is selected, it is applied to the selected text through the use of an attached `ActionListener`. The setup is similar to that used for the `m_cbFonts` component. The `StyleConstants.setFontSize()` method is used to set the font size. Our custom `setAttributeSet()` method is then used to apply this attribute set to the selected text.

The bold and italic properties are managed by two `SmallToggleButtons` (a custom button class we developed in chapter 12): `m_bBold` and `m_bItalic` respectively. These buttons receive `ActionListeners` which create a `SimpleAttributeSet` instance with the bold or italic property with `StyleConstants.setBold()` or `StyleConstants.setItalic()`. Then our custom `setAttributeSet()` method is called to apply this attribute set.

The `showAttributes()` method is called to set the state of the toolbar components described above according to the font properties of the text at the given caret position. This method sets the `m_skipUpdate` flag to `true` at the beginning and `false` at the end of it's execution (the purpose of this will be explained soon below). Then an `AttributeSet` instance corresponding to the character element at the current caret position in the editor's document is retrieved with the `getAttributes()` method. The `StyleConstants.getFontFamily()` method is used to retrieve the current font name from this attribute

set. If it is not equal to the previously selected font name (stored in the `m_fontName` instance variable) it is selected in the `m_cbFonts` combobox. The other toolbar controls are handled in a similar way.

Our `setAttributeSet()` method is used to assign a given set of attributes to the currently selected text. Note that this method does nothing (simply returns) if the `m_skipUpdate` flag is set to `true`. This is done to prevent the backward link with the `showAttributes()` method. As soon as we assign some value to a combo boc in the `showAttributes()` method (e.g. font size) this internally triggers a call to the `setAttributeSet()` method (because `ActionListeners` attached to combo boxes are invoked even when selection changes occur programmatically). The purpose of `showAttributes()` is to simply make sure that the attributes corresponding to the character element at the current text position are accurately reflected in the toolbar components. To prevent the combo box `ActionListeners` from invoking unnecessary operations we prohibit any text property updates from occuring in `setAttributeSet()` while the `showAttributes()` method is being executed (this is the whole purpose of the `m_skipUpdate` flag).

The `setAttributeSet()` method first determines the start and end positions of the selected text. If `m_monitor` currently does not have the focus, the stored bounds, `m_xStart` and `m_xFinish`, are used instead. If the selection is not empty (`xStart != xFinish`), the `setCharacterAttributes()` method is called to assign the given set of attributes to the selection. Note that this new attribute set does not have to contain a complete set of attributes. It simply replaces only the existing attributes for which it has new values, leaving the remainder unchanged. If the selection is empty, the new attributes are added to the input attributes of the editor kit (recall that `StyledEditorKit`'s input attributes are those attributes that will be applied to newly inserted text).

Running the Code

Open an existing RTF file and move the cursor to various positions in the text. Note that the text attributes displayed in the toolbar components are updated correctly. Select a portion of text and use the toolbar components to modify the selection's font attributes. Type a new font name or font size in the editable combobox and press "Enter." This has the same effect as selecting a choice from the drop-down list. Save the RTF file and open it in another RTF-aware application to verify that your changes were saved correctly.

Bug Alert! Bold and italic font properties are often not updated on the screen properly, even though they are assigned and saved correctly. We hope that this problem will be fixed in future Swing releases.

20.3 Word Processor: part III - Colors and images

Important RTF features we will exploit in this section include the ability to use foreground and background colors and insert images into the text. In this example we show how to add these capabilities to our growing RTF word processor application.

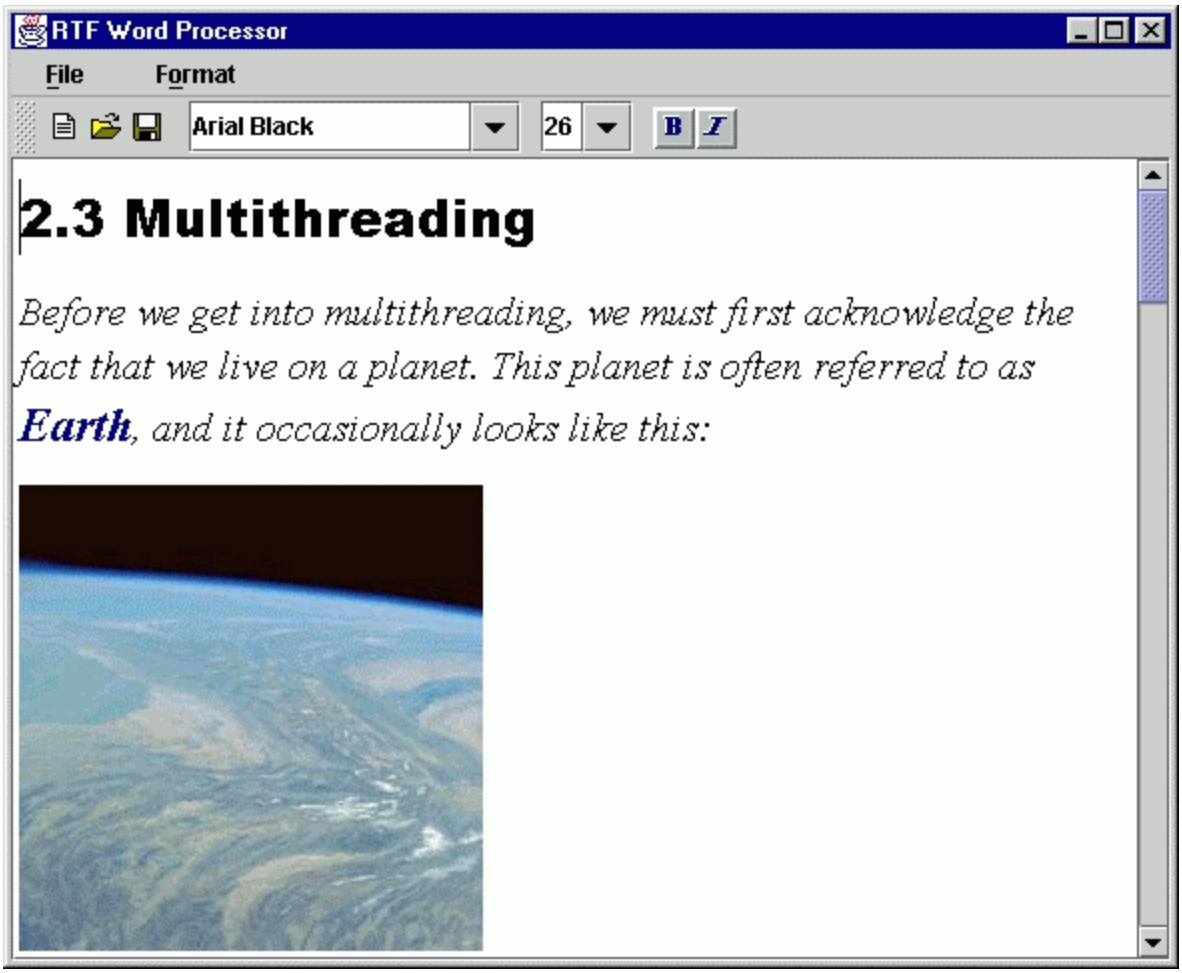


Figure 20.3 JTextPane with diverse font styles, foreground colors, and an embedded image.
<<file figure20-3.gif>>

The Code: WordProcessor.java
see \Chapter20\3

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

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;

public class WordProcessor extends JFrame
{
  // Unchanged code from section 20.2

  protected SimpleFilter m_jpgFilter;
  protected SimpleFilter m_gifFilter;

  protected ColorMenu m_foreground;
  protected ColorMenu m_background;
```

```
  public WordProcessor() {
    // Unchanged code from section 20.2

    m_chooser = new JFileChooser();
    m_chooser.setCurrentDirectory(new File("."));
    m_rtfFilter = new SimpleFilter("rtf", "RTF Documents");
    m_chooser.setFileFilter(m_rtfFilter);

    m_gifFilter = new SimpleFilter("gif", "GIF images");
    m_jpgFilter = new SimpleFilter("jpg", "JPG images");

    // Unchanged code from section 20.2
  }

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 20.2

    JMenu mFormat = new JMenu("Format");
    mFormat.setMnemonic('o');

    m_foreground = new ColorMenu("Foreground");
    m_foreground.setColor(m_monitor.getForeground());
    m_foreground.setMnemonic('f');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        MutableAttributeSet attr = new SimpleAttributeSet();
        StyleConstants.setForeground(attr, m_foreground.getColor());
        setAttributeSet(attr);
      }
    };
    m_foreground.addActionListener(lst);
    mFormat.add(m_foreground);

    MenuListener ml = new MenuListener() {
      public void menuSelected(MenuEvent e) {
        int p = m_monitor.getCaretPosition();
        AttributeSet a = m_doc.getCharacterElement(p).
          getAttributes();
        Color c = StyleConstants.getForeground(a);
        m_foreground.setColor(c);
      }

      public void menuDeselected(MenuEvent e) {}

      public void menuCanceled(MenuEvent e) {}
    };
    m_foreground.addMenuListener(ml);

    // Bug Alert! JEditorPane background color
    // doesn't work as of Java 2 FCS.
    m_background = new ColorMenu("Background");
    m_background.setColor(m_monitor.getBackground());
    m_background.setMnemonic('b');
    lst = new ActionListener()  {
      public void actionPerformed(ActionEvent e) {
        MutableAttributeSet attr = new SimpleAttributeSet();
        StyleConstants.setBackground(attr, m_background.getColor());
        setAttributeSet(attr);
      }
    };
    m_background.addActionListener(lst);
    mFormat.add(m_background);
```

```
    ml = new MenuListener() {
      public void menuSelected(MenuEvent e) {
        int p = m_monitor.getCaretPosition();
        AttributeSet a = m_doc.getCharacterElement(p).
          getAttributes();
        Color c = StyleConstants.getBackground(a);
        m_background.setColor(c);
      }

      public void menuDeselected(MenuEvent e) {}

      public void menuCanceled(MenuEvent e) {}
    };
    m_background.addMenuListener(ml);

    // Bug Alert! Images do not get saved.
    mFormat.addSeparator();
    item = new JMenuItem("Insert Image");
    item.setMnemonic('i');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_chooser.addChoosableFileFilter(m_gifFilter);
        m_chooser.addChoosableFileFilter(m_jpgFilter);
        m_chooser.setFileFilter(m_gifFilter);
        m_chooser.removeChoosableFileFilter(m_rtfFilter);
        Thread runner = new Thread() {
          public void run() {
            if (m_chooser.showOpenDialog(WordProcessor.this) !=
            JFileChooser.APPROVE_OPTION)
              return;
            WordProcessor.this.repaint();
            File fChoosen = m_chooser.getSelectedFile();
            ImageIcon icon = new ImageIcon(fChoosen.getPath());
            int w = icon.getIconWidth();
            int h = icon.getIconHeight();
            if (w<=0 || h<=0) {
              JOptionPane.showMessageDialog(WordProcessor.this,
                "Error reading image file\n"+
                fChoosen.getPath(), "Warning",
                JOptionPane.WARNING_MESSAGE);
                return;
            }
            MutableAttributeSet attr = new SimpleAttributeSet();
            StyleConstants.setIcon(attr, icon);
            int p = m_monitor.getCaretPosition();
            try {
              m_doc.insertString(p, " ", attr);
            }
            catch (BadLocationException ex) {}

            // Its ok to do this outside of the event-dispatching
            // thread because the chooser is not visible here.
            m_chooser.addChoosableFileFilter(m_rtfFilter);
            m_chooser.setFileFilter(m_rtfFilter);
            m_chooser.removeChoosableFileFilter(m_gifFilter);
            m_chooser.removeChoosableFileFilter(m_jpgFilter);
          }
        };
        runner.start();
      }
    };
```

```
    item.addActionListener(lst);
    mFormat.add(item);

    menuBar.add(mFormat);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    return menuBar;
  }

  // Unchanged code from section 20.2
}

// Unchanged code from section 20.2

// Class ColorMenu unchanged from section 12.5
```

Understanding the Code

Class WordProcessor

Four new instance variables have been added:

- `SimpleFilter m_jpgFilter`: Used for JPEG image selection with `JFileChooser` (see chapter 14 for more about `SimpleFilter`).
- `SimpleFilter m_gifFilter`: Used for GIF image selection with `JFileChooser`.
- `ColorMenu m_foreground`: used to choose the selected text foreground color.
- `ColorMenu m_background`: used to choose the selected text background color.

The `ColorMenu` class was constructed and discussed in chapter 12, and is used here without modification. This class represents a custom menu component used to select a color from a collection of 64 pre-defined colors. To deploy this component in our application we add a menu titled "Format" with menu items titled "Foreground" and "Background." We also add, after a menu separator, the menu item "Insert Image" (see below).

The `ColorMenu m_foreground` receives an `ActionListener` which retrieves the selected color with the `getColor()` method and then applies it to the selected text by making calls to `StyleConstants.setForeground()` and `setAttributeSet()` methods, similar to how we assigned new font attributes in the previous example. In order to maintain consistency we need to update the `ColorMenu` component prior to displaying it to make the initial color selection consistent with the text foreground at the cursor position. For this purpose we add a `MenuListener` to the `m_foreground` component. The `menuSelected()` method will be called prior to menu selection. Similar to the `showAttributes()` method discussed above, this code retrieves an `AttributeSet` instance corresponding to the current caret location and determines the selected foreground color with `getForeground()`. Then this color is passed to the `ColorMenu` component for use as the selection.

The `m_background ColorMenu` works similar to `m_foreground`, but manages selected text background color. Note that this feature doesn't work with the current RTF API release: it is neither displayed nor saved in the file. The background color menu is added to our word processor for the sake of completeness.

The "Insert Image" menu item receives an `ActionListener` which uses our `JFileChooser` to select an

image file. If an image file is successfully selected and read, we create a `MutableAttributeSet` instance and pass our image to it with `StyleConstants.setIcon()`. Then we insert a dummy single space character with that icon attribute using `insertString()`. Note that all this occurs in a separate thread to avoid the possibility of clogging the event-dispathing thread.

Running the Code

Open an existing RTF file, select a portion of text and use the custom color menu component to modify it's foreground. Save the RTF file and open it in another RTF-aware application to verify that your changes have been saved correctly. Try using the "Insert Image" menu to bring up a file chooser and select an image for insertion.

Bug Alert! Unfortunately embedded images are neither saved to file or read from an existing RTF document. We hope that this problem/limitation will be fixed soon in a future release.

20.4 Word Processor: part IV - Working with styles

Using Styles to manage a set of attributes as a single named entity can greatly simplify text editing. The user only has to apply a known style to a selected region of text rather than selecting all appropriate text attributes from the provided toolbar components. By adding a combo box allowing the choice of styles, we can not only save the user time and effort, but we can also provide more uniform text formatting throughout the resulting document (or potentially set of documents). In this section we'll add style management to our word processor. We'll also show how it is possible to create a new style, modify an existing style, or reapply a style to modified text.

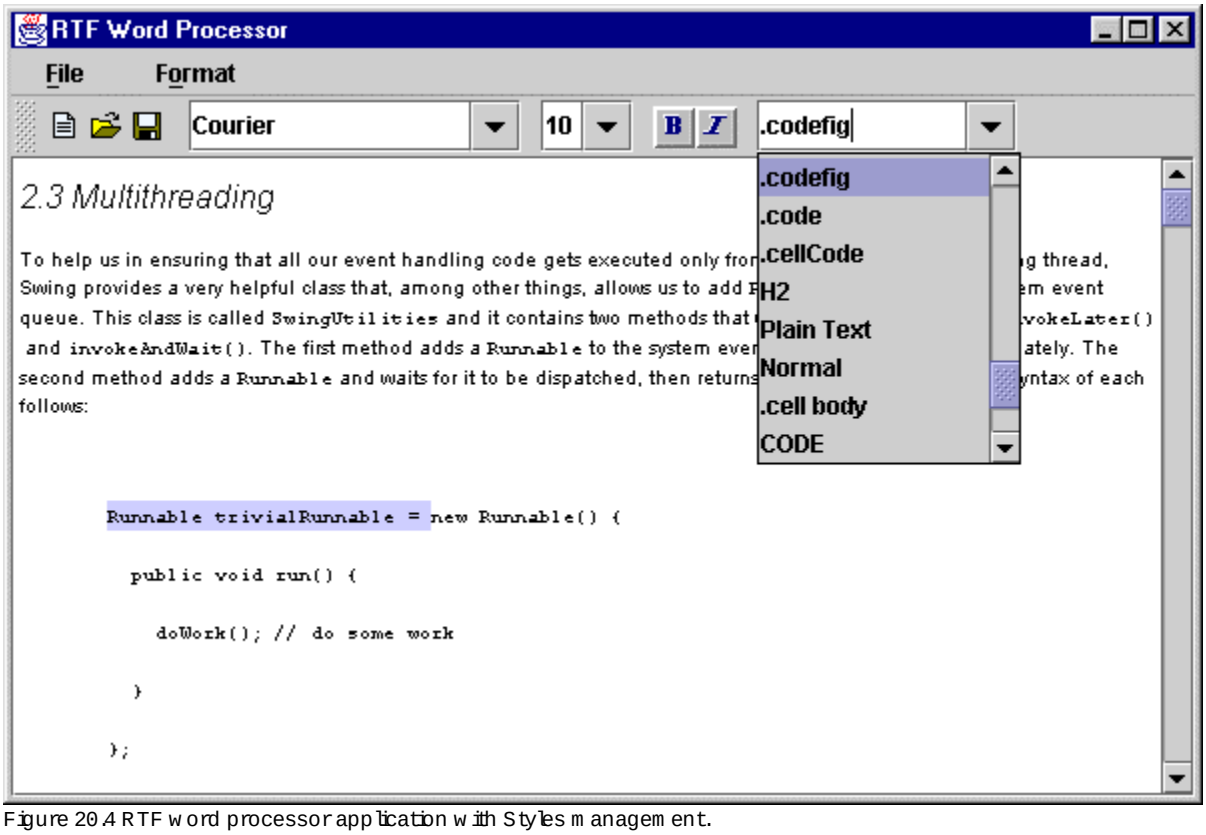


Figure 20.4 RTF word processor application with Styles management.

<<file figure20-4.gif>>

The Code: WordProcessor.java
see \Chapter20\4

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

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;

public class WordProcessor extends JFrame
{
  // Unchanged code from section 20.3

  protected JComboBox m_cbStyles;

  public WordProcessor() {
    // Unchanged code from section 20.3

    showAttributes(0);
    showStyles();
    setVisible(true);
  }

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 20.3

    // The following line is added to the end of the
    // actionNew and actionOpen actionPerformed() methods:
    //
    //   showStyles();
    //
    // (see source code; these methods are not shown here
    //  to conserve space)

    // Unchanged code from section 20.3

    JMenu mFormat = new JMenu("Format");
    mFormat.setMnemonic('o');

    JMenu mStyle = new JMenu("Style");
    mStyle.setMnemonic('s');
    mFormat.add(mStyle);

    item = new JMenuItem("Update");
    item.setMnemonic('u');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String name = (String)m_cbStyles.getSelectedItem();
        Style style = m_doc.getStyle(name);
        int p = m_monitor.getCaretPosition();
        AttributeSet a = m_doc.getCharacterElement(p).
          getAttributes();
        style.addAttributes(a);
        m_monitor.repaint();
```

```
      }
    };
    item.addActionListener(lst);
    mStyle.add(item);

    item = new JMenuItem("Reapply");
    item.setMnemonic('r');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String name = (String)m_cbStyles.getSelectedItem();
        Style style = m_doc.getStyle(name);
        setAttributeSet(style);
      }
    };
    item.addActionListener(lst);
    mStyle.add(item);

    mFormat.addSeparator();

    // Unchanged code from section 20.3

    menuBar.add(mFormat);

    m_toolBar.addSeparator();
    m_cbStyles = new JComboBox();
    m_cbStyles.setMaximumSize(m_cbStyles.getPreferredSize());
    m_cbStyles.setEditable(true);
    m_toolBar.add(m_cbStyles);

    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        if (m_skipUpdate || m_cbStyles.getItemCount()==0)
          return;
        String name = (String)m_cbStyles.getSelectedItem();
        int index = m_cbStyles.getSelectedIndex();
        int p = m_monitor.getCaretPosition();

        // New name entered
        if (index == -1) {
          m_cbStyles.addItem(name);
          Style style = m_doc.addStyle(name, null);
          AttributeSet a = m_doc.getCharacterElement(p).
            getAttributes();
          style.addAttributes(a);
          return;
        }

        // Apply the selected style
        Style currStyle = m_doc.getLogicalStyle(p);
        if (!currStyle.getName().equals(name)) {
          Style style = m_doc.getStyle(name);
          setAttributeSet(style);
        }
      }
    };
    m_cbStyles.addActionListener(lst);

    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    return menuBar;
  }
```

```
  protected void showAttributes(int p) {
    // Unchanged code from section 20.2

    Style style = m_doc.getLogicalStyle(p);
    name = style.getName();
    m_cbStyles.setSelectedItem(name);

    m_skipUpdate = false;
  }

  // Unchanged code from section 20.3

  protected void showStyles() {
    m_skipUpdate = true;
    if (m_cbStyles.getItemCount() > 0)
      m_cbStyles.removeAllItems();
    Enumeration en = m_doc.getStyleNames();
    while (en.hasMoreElements()) {
      String str = en.nextElement().toString();
      m_cbStyles.addItem(str);
    }
    m_skipUpdate = false;
  }

  public static void main(String argv[]) {
    new WordProcessor();
  }
}

// Unchanged code from section 20.3
```

Understanding the Code

Class WordProcessor

One new instance variable has been added:

`JComboBox m_cbStyles`: toolbar component to manage styles.

Note that a new custom method `showStyles()` (see below) is now called after creating a new document or after loading an existing one.

The `createMenuBar()` method creates a new menu with two new menu items for updating and reapplying styles, and a new combo box for style selection. The editable styles combobox, `m_cbStyles`, will hold a list of styles declared in the current document (we will see how this component is populated below). It receives an `ActionListener` which checks whether the currently selected style name is present among the existing styles. If not, we add it to the drop-down list and retrieve a new `Style` instance for the selected name using `StyledDocument`'s `addStyle()` method. This new `Style` instance is associated with the text attributes of the character element at the current caret position. Otherwise, if the given style name is known already, we retrieve the selected style using `StyledDocument`'s `getStyle()` method and apply it to the selected text by passing it to our custom `setAttributeSet()` method (as we did in previous examples when assigning text attributes).

An ambiguous situation occurs when the user selects a style for text which already has the same style, but whose attributes have been modified. The user may either want to update the selected style using the selected text as a base, or reapply the existing style to the selected text. To resolve this situation we need to ask the user

what to do. We chose to add two menu items which allow the user to either update or reapply the current selection.

Note: In ambiguous situations such as this, making the decision to allow users to choose between two options, or enforcing a single behavior, can be a tough one to make. In general, the less experienced the target audience is, the less choices that audience should need to become familiar with. In this case we would suggest that a selected style override all attributes of the selected text.

The menu items to perform these tasks are titled "Update" and "Reapply," and are grouped into the "Style" menu. The "Style" menu is added to the "Format" menu. The "Update" menu item receives an `ActionListener` which retrieves the text attributes of the character element at the current caret position, and assigns them to the selected style. The "Reapply" menu item receives an `ActionListener` which applies the selected style to the selected text (one might argue that this menu item would be more appropriately titled "Apply" -- the implications are ambiguous either way).

Our `showAttributes()` method receives additional code to manage the new styles combobox, `m_cbStyles`, when the caret moves through the document. It retrieves the style corresponding to the current caret position with `StyledDocument`'s `getLogicalStyle()` method, and selects the appropriate entry in the combobox.

The new `showStyles()` method is called to populate the `m_cbStyles` combobox with the style names from a newly created or loaded document. First it removes the current content of the combobox if it is not empty (another work around due to the fact that if you call `removeAllItems()` on an empty `JComboBox`, an exception will be thrown). An `Enumeration` of style names is then retrieved with `StyledDocument`'s `getStyleNames()` method, and these names are added to the combobox.

Running the Code

Open an existing RTF file, and note how the styles combobox is populated by the style names defined in this document. Verify that the selected style is automatically updated while the caret moves through the document. Select a portion of text and select a different style from the styles combobox. Note how all text properties are updated according to the new style.

Try selecting a portion of text and modifying its attributes (for instance, foreground color). Type a new name in the styles combobox and press Enter. This will create a new style which can be applied to any other document text.

Note: New styles will not be saved along with an RTF document under the current version of `RTFEditorKit`.

Try modifying an attribute of a selected region of text (e.g. the font size) and select the "Style|Update" menu item. This will update the style to incorporate the newly selected attributes. Apply the modified style to another portion of text and verify that it applies according to the updated style.

Note: When a style is updated, any regions of text that had been applied with this style do not automatically get updated accordingly. This is another ambiguity that must be considered, depending on what the user expects and what level or experience the target audience has. In this case we assume that the user only wants selected text to be affected by a style update.

Now try modifying some attributes of a portion of selected text and then select the "Style|Reapply" menu item. This will restore the original text attributes associated with the appropriate style.

Note: Recall that we are using one `StyleContext` instance, `m_context`, for all documents. This object collects all document styles. These styles are always available when a new document is created or loaded. We might develop a document template mechanism by serializing this `StyleContext` instance into a file and restoring it with the appropriate document.

20.5 Word Processor: part V - Clipboard and Undo/Redo

Clipboard and undo/redo operations have become common and necessary components of all modern text editing environments. We have discussed these features in chapters 11 and 19, and in this section we show how to integrate them into our RTF word processor.

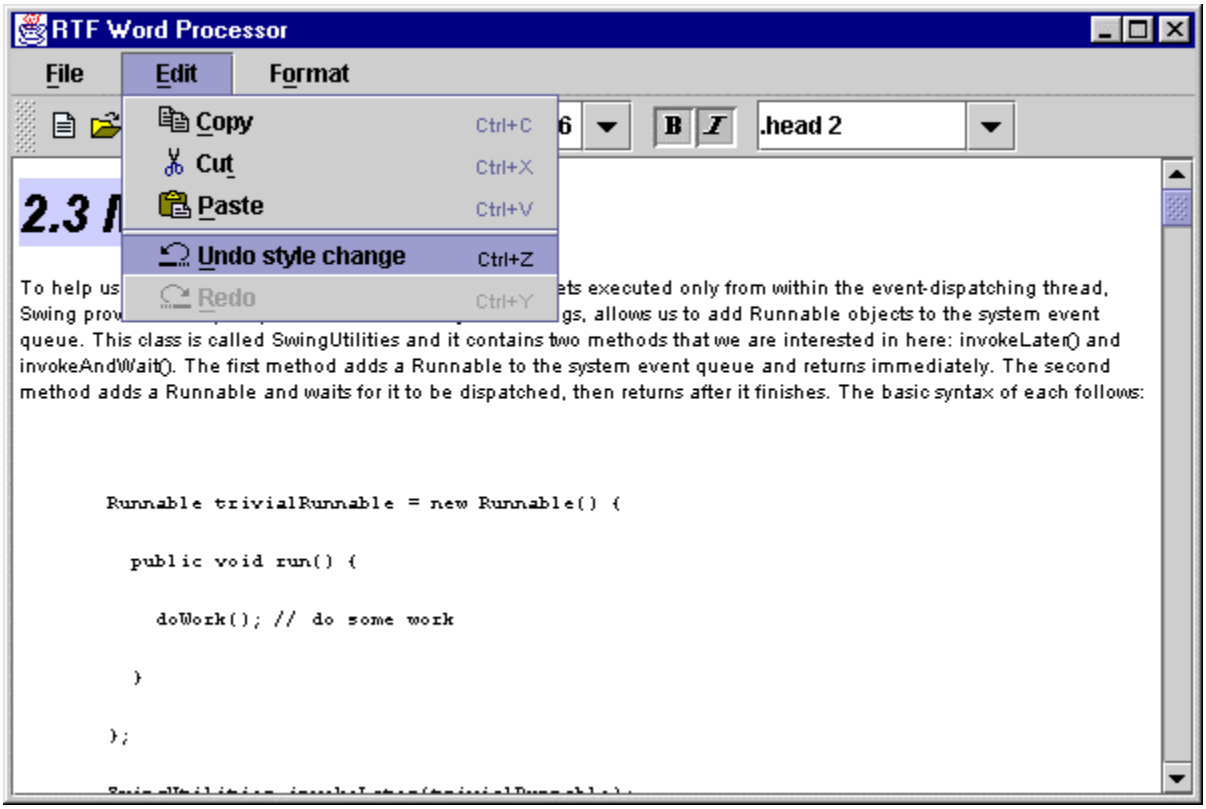


Figure 20.5 RTF word processor with undo/redo and clipboard functionality.
<<file figure20-5.gif>>

The Code: WordProcessor.java
see \Chapter20\5

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

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

```
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;
import javax.swing.undo.*;

public class WordProcessor extends JFrame
{
  // Unchanged code from section 20.4

  protected UndoManager m_undo = new UndoManager();
  protected Action m_undoAction;
  protected Action m_redoAction;

  public WordProcessor() {
    // Unchanged code from section 20.4

    showAttributes(0);
    showStyles();
    m_doc.addUndoableEditListener(new Undoer());
    setVisible(true);
  }

  protected JMenuBar createMenuBar() {
    // The following line is added to the end of the
    // actionNew and actionOpen actionPerformed() methods:
    //
    //   m_doc.addUndoableEditListener(new Undoer());
    //
    // (see source code; these methods are not shown here
    //  to conserve space)

    // Unchanged code from section 20.4

    JButton bSave = new SmallButton(actionSave, "Save RTF document");
    m_toolBar.add(bSave);

    JMenu mEdit = new JMenu("Edit");
    mEdit.setMnemonic('e');

    Action action = new AbstractAction("Copy",
     new ImageIcon("edit_copy.gif"))
    {
      public void actionPerformed(ActionEvent e) {
        m_monitor.copy();
      }
    };
    item = mEdit.add(action);
    item.setMnemonic('c');
    item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_C,
      KeyEvent.CTRL_MASK));

    action = new AbstractAction("Cut",
     new ImageIcon("edit_cut.gif"))
    {
      public void actionPerformed(ActionEvent e) {
        m_monitor.cut();
      }
    };
    item = mEdit.add(action);
    item.setMnemonic('t');
    item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_X,
      KeyEvent.CTRL_MASK));
```

```
    action = new AbstractAction("Paste",
     new ImageIcon("edit_paste.gif"))
    {
      public void actionPerformed(ActionEvent e) {
        m_monitor.paste();
      }
    };
    item = mEdit.add(action);
    item.setMnemonic('p');
    item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_V,
      KeyEvent.CTRL_MASK));

    mEdit.addSeparator();

    m_undoAction = new AbstractAction("Undo",
     new ImageIcon("edit_undo.gif"))
    {
      public void actionPerformed(ActionEvent e) {
        try {
          m_undo.undo();
        }
        catch (CannotUndoException ex) {
          System.err.println("Unable to undo: " + ex);
        }
        updateUndo();
      }
    };
    item = mEdit.add(m_undoAction);
    item.setMnemonic('u');
    item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Z,
      KeyEvent.CTRL_MASK));

    m_redoAction = new AbstractAction("Redo",
     new ImageIcon("edit_redo.gif"))
    {
      public void actionPerformed(ActionEvent e) {
        try {
          m_undo.redo();
        }
        catch (CannotRedoException ex) {
          System.err.println("Unable to redo: " + ex);
        }
        updateUndo();
      }
    };
    item =  mEdit.add(m_redoAction);
    item.setMnemonic('r');
    item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_Y,
      KeyEvent.CTRL_MASK));

    menuBar.add(mEdit);

    GraphicsEnvironment ge = GraphicsEnvironment.
      getLocalGraphicsEnvironment();
    String[] fontNames = ge.getAvailableFontFamilyNames();

    // Unchanged code from section 20.4

    return menuBar;
  }
```

```
  // Unchanged code from section 20.4

  protected void updateUndo() {
    if(m_undo.canUndo()) {
      m_undoAction.setEnabled(true);
      m_undoAction.putValue(Action.NAME,
      m_undo.getUndoPresentationName());
    }
    else {
      m_undoAction.setEnabled(false);
      m_undoAction.putValue(Action.NAME, "Undo");
    }
    if(m_undo.canRedo()) {
      m_redoAction.setEnabled(true);
      m_redoAction.putValue(Action.NAME,
      m_undo.getRedoPresentationName());
    }
    else {
      m_redoAction.setEnabled(false);
      m_redoAction.putValue(Action.NAME, "Redo");
    }
  }

  public static void main(String argv[]) {
    new WordProcessor();
  }

  class Undoer implements UndoableEditListener
  {
    public Undoer() {
      m_undo.die();
      updateUndo();
    }

    public void undoableEditHappened(UndoableEditEvent e) {
      UndoableEdit edit = e.getEdit();
      m_undo.addEdit(e.getEdit());
      updateUndo();
    }
  }
}

// Unchanged code from section 20.4
```

Understanding the Code

Class WordProcessor

We now import the `javax.swing.undo` package and add three new instance variables:

`UndoManager m_undo`: used to manage undo/redo operations.

`Action m_undoAction`: used for a menu item/action to perform undo operations.

`Action m_redoAction`: used for a menu item/action to perform redo operations.

Note that a new `Undoer` instance (see below) is now added as an `UndoableEditListener` to all newly created or loaded documents.

The `createMenuBar()` method now creates a menu titled "Edit" (which traditionally follows the "File"

menu) containing menu items titled "Copy," "Cut," "Paste," "Undo," and "Redo." The first three items merely trigger calls to the `copy()`, `cut()`, and `paste()` methods of our `m_monitor` text pane. These methods perform clipboard operations using plain text without any attributes. They are available when the editor has the current focus and the appropriate keyboard accelerator is pressed. These items are added to our "Edit" menu to provide a convenient and informative interface.

The "Undo" menu item is created from an `AbstractAction` whose `actionPerformed()` method first invokes `undo()` on the `UndoManager`, and then invokes our custom `updateUndo()` method to update our undo/redo menu items appropriately. Similarly, the "Redo" menu item is created from an `AbstractAction` which invokes `redo()` on the `UndoManager`, and then calls our `updateUndo()` method.

The `updateUndo()` method enables or disables the undo and redo menu items, and updates their names according to the operation which can be undone/redone (if any). If the `UndoManager`'s `canUndo()` method returns `true`, the `m_undoAction` is enabled and its name is set to the string returned by `getUndoPresentationName()`. Otherwise it is disabled and its name is set to "Undo." The "Redo" menu item is handled similarly.

Class WordProcessor.Undoer

This inner class implements the `UndoableEditListener` interface to receive notifications about undoable operations. The `undoableEditHappened()` method receives `UndoableEditEvents`, retrieves their encapsulated `UndoableEdit` instances, and passes them to the `UndoManager`. Our `updateUndo()` method is also invoked to update the undo/redo menu items appropriately.

Running the Code

Open an existing RTF file and verify that copy, cut, and paste clipboard operations transfer plain text successfully. Make some changes to the textual content or styles and note that the title of the "Undo" menu item is updated. Select this menu item, or press its keyboard accelerator (Ctrl-Z) to undo a series of changes. This will enable the "Redo" menu item. Use this menu item or press its keyboard accelerator (Ctrl-Y) to redo a series of changes.

20.6 Word Processor: part VI - Advanced font management

In section 20.2 we used toolbar components to change manipulate font properties. This is useful for making a quick modification without leaving the main application frame, and is typical for word processor applications. However, all serious editor applications also provide a dialog for the editing of all available font properties from one location. In the following example we'll show how to create such a dialog, which includes components to select various font properties and preview the result.

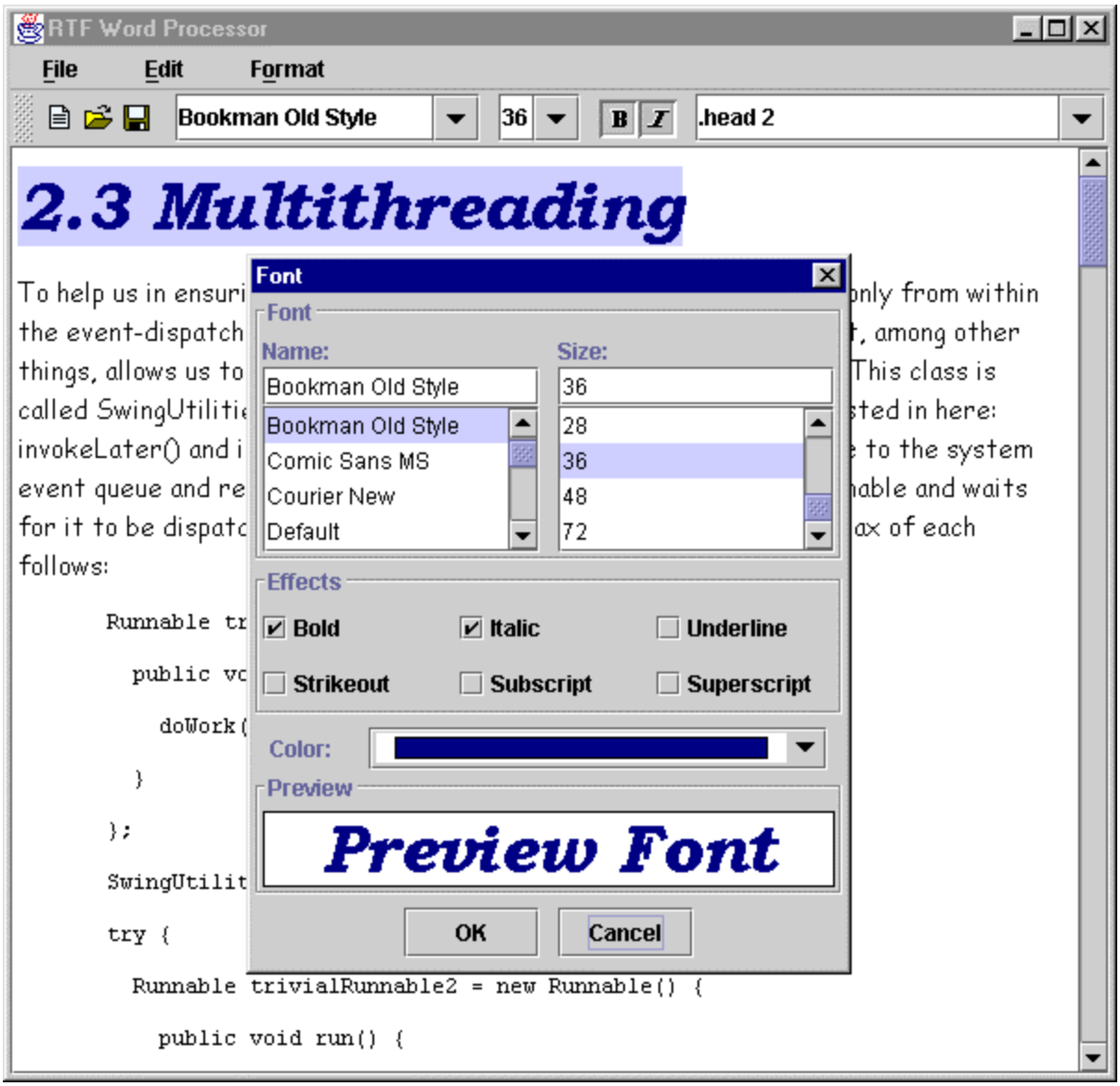


Figure 20.6 RTF word processor with custom font properties and preview dialog.

<<file figure20-6.gif>>

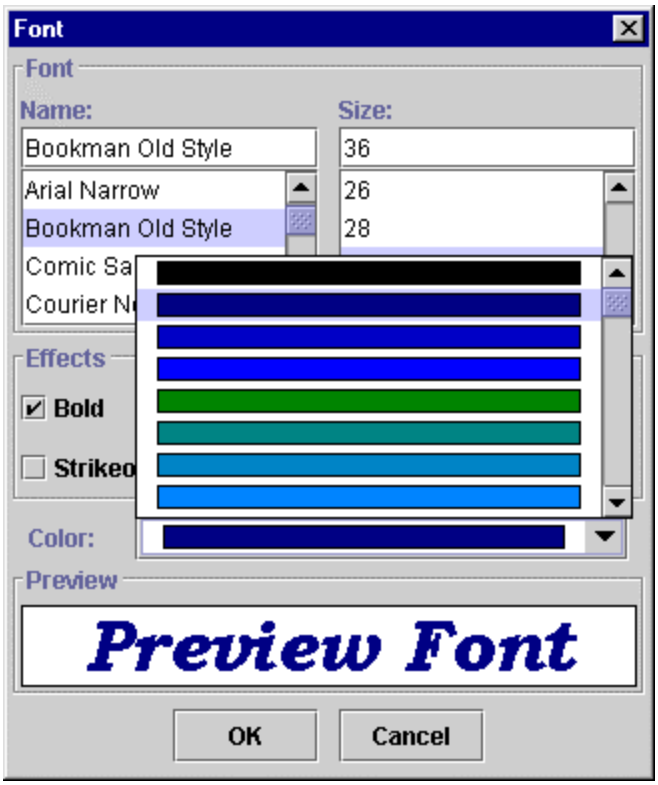


Figure 20.7 FontDialog with custom JList and ListCellRenderer for foreground color selection.
<<file figure20-7.gif>>

The Code: WordProcessor.java
see \Chapter20\6

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

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;
import javax.swing.undo.*;

public class WordProcessor extends JFrame
{
  // Unchanged code from section 20.5

  protected String[] m_fontNames;
  protected String[] m_fontSizes;

  protected FontDialog m_fontDialog;

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 20.5

    GraphicsEnvironment ge = GraphicsEnvironment.
      getLocalGraphicsEnvironment();
    m_fontNames = ge.getAvailableFontFamilyNames();
```

```
    m_toolBar.addSeparator();
    m_cbFonts = new JComboBox(m_fontNames);
    m_cbFonts.setMaximumSize(m_cbFonts.getPreferredSize());
    m_cbFonts.setEditable(true);

    // Unchanged code from section 20.5

    m_toolBar.addSeparator();
    m_fontSizes = new String[] {"8", "9", "10", "11", "12", "14",
      "16", "18", "20", "22", "24", "26", "28", "36", "48", "72"};
    m_cbSizes = new JComboBox(m_fontSizes);
    m_cbSizes.setMaximumSize(m_cbSizes.getPreferredSize());
    m_cbSizes.setEditable(true);

    m_fontDialog = new FontDialog(this, m_fontNames, m_fontSizes);

    // Unchanged code from section 20.5

    JMenu mFormat = new JMenu("Format");
    mFormat.setMnemonic('o');

    item = new JMenuItem("Font...");
    item.setMnemonic('o');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        WordProcessor.this.repaint();
        AttributeSet a = m_doc.getCharacterElement(
          m_monitor.getCaretPosition()).getAttributes();
        m_fontDialog.setAttributes(a);

        Dimension d1 = m_fontDialog.getSize();
        Dimension d2 = WordProcessor.this.getSize();
        int x = Math.max((d2.width-d1.width)/2, 0);
        int y = Math.max((d2.height-d1.height)/2, 0);
        m_fontDialog.setBounds(x + WordProcessor.this.getX(),
          y + WordProcessor.this.getY(), d1.width, d1.height);

        m_fontDialog.show();
        if (m_fontDialog.getOption()==JOptionPane.OK_OPTION) {
          setAttributeSet(m_fontDialog.getAttributes());
          showAttributes(m_monitor.getCaretPosition());
        }
      }
    };
    item.addActionListener(lst);
    mFormat.add(item);

    mFormat.addSeparator();

    // Unchanged code from section 20.5

    return menuBar;
  }

  // Unchanged code from section 20.5
}

// Unchanged code from section 20.5

class FontDialog extends JDialog
{
  protected int m_option = JOptionPane.CLOSED_OPTION;
```

```
protected OpenList m_lstFontName;
protected OpenList m_lstFontSize;
protected MutableAttributeSet m_attributes;
protected JCheckBox m_chkBold;
protected JCheckBox m_chkItalic;
protected JCheckBox m_chkUnderline;

protected JCheckBox m_chkStrikethrough;
protected JCheckBox m_chkSubscript;
protected JCheckBox m_chkSuperscript;

protected JComboBox m_cbColor;
protected JLabel m_preview;

public FontDialog(JFrame parent,
  String[] names, String[] sizes)
{
  super(parent, "Font", true);
  getContentPane().setLayout(new BoxLayout(getContentPane(),
    BoxLayout.Y_AXIS));

  JPanel p = new JPanel(new GridLayout(1, 2, 10, 2));
  p.setBorder(new TitledBorder(new EtchedBorder(), "Font"));
  m_lstFontName = new OpenList(names, "Name:");
  p.add(m_lstFontName);

  m_lstFontSize = new OpenList(sizes, "Size:");
  p.add(m_lstFontSize);
  getContentPane().add(p);

  p = new JPanel(new GridLayout(2, 3, 10, 5));
  p.setBorder(new TitledBorder(new EtchedBorder(), "Effects"));
  m_chkBold = new JCheckBox("Bold");
  p.add(m_chkBold);
  m_chkItalic = new JCheckBox("Italic");
  p.add(m_chkItalic);
  m_chkUnderline = new JCheckBox("Underline");
  p.add(m_chkUnderline);
  m_chkStrikethrough = new JCheckBox("Strikeout");
  p.add(m_chkStrikethrough);
  m_chkSubscript = new JCheckBox("Subscript");
  p.add(m_chkSubscript);
  m_chkSuperscript = new JCheckBox("Superscript");
  p.add(m_chkSuperscript);
  getContentPane().add(p);

  getContentPane().add(Box.createVerticalStrut(5));
  p = new JPanel();
  p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
  p.add(Box.createHorizontalStrut(10));
  p.add(new JLabel("Color:"));
  p.add(Box.createHorizontalStrut(20));
  m_cbColor = new JComboBox();

  int[] values = new int[] { 0, 128, 192, 255 };
  for (int r=0; r<values.length; r++) {
    for (int g=0; g<values.length; g++) {
      for (int b=0; b<values.length; b++) {
        Color c = new Color(values[r], values[g], values[b]);
        m_cbColor.addItem(c);
      }
    }
```

```
    }

    m_cbColor.setRenderer(new ColorComboRenderer());
    p.add(m_cbColor);
    p.add(Box.createHorizontalStrut(10));
    getContentPane().add(p);

    p = new JPanel(new BorderLayout());
    p.setBorder(new TitledBorder(new EtchedBorder(), "Preview"));
    m_preview = new JLabel("Preview Font", JLabel.CENTER);
    m_preview.setBackground(Color.white);
    m_preview.setForeground(Color.black);
    m_preview.setOpaque(true);
    m_preview.setBorder(new LineBorder(Color.black));
    m_preview.setPreferredSize(new Dimension(120, 40));
    p.add(m_preview, BorderLayout.CENTER);
    getContentPane().add(p);

    p = new JPanel(new FlowLayout());
    JPanel p1 = new JPanel(new GridLayout(1, 2, 10, 2));
    JButton btOK = new JButton("OK");
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_option = JOptionPane.OK_OPTION;
        setVisible(false);
      }
    };
    btOK.addActionListener(lst);
    p1.add(btOK);

    JButton btCancel = new JButton("Cancel");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_option = JOptionPane.CANCEL_OPTION;
        setVisible(false);
      }
    };
    btCancel.addActionListener(lst);
    p1.add(btCancel);
    p.add(p1);
    getContentPane().add(p);

    pack();
    setResizable(false);
    Dimension d1 = getSize();
    Dimension d2 = parent.getSize();
    int x = Math.max((d2.width-d1.width)/2, 0);
    int y = Math.max((d2.height-d1.height)/2, 0);
    setBounds(x, y, d1.width, d1.height);

    ListSelectionListener lsel = new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent e) {
        updatePreview();
      }
    };
    m_lstFontName.addListSelectionListener(lsel);
    m_lstFontSize.addListSelectionListener(lsel);

    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        updatePreview();
      }
```

```
    };
    m_chkBold.addActionListener(lst);
    m_chkItalic.addActionListener(lst);
    m_cbColor.addActionListener(lst);
  }

  public void setAttributes(AttributeSet a) {
    m_attributes = new SimpleAttributeSet(a);
    String name = StyleConstants.getFontFamily(a);
    m_lstFontName.setSelected(name);
    int size = StyleConstants.getFontSize(a);
    m_lstFontSize.setSelectedInt(size);
    m_chkBold.setSelected(StyleConstants.isBold(a));
    m_chkItalic.setSelected(StyleConstants.isItalic(a));
    m_chkUnderline.setSelected(StyleConstants.isUnderline(a));
    m_chkStrikethrough.setSelected(
      StyleConstants.isStrikeThrough(a));
    m_chkSubscript.setSelected(StyleConstants.isSubscript(a));
    m_chkSuperscript.setSelected(StyleConstants.isSuperscript(a));
    m_cbColor.setSelectedItem(StyleConstants.getForeground(a));
    updatePreview();
  }

  public AttributeSet getAttributes() {
    if (m_attributes == null)
      return null;
    StyleConstants.setFontFamily(m_attributes,
      m_lstFontName.getSelected());
    StyleConstants.setFontSize(m_attributes,
      m_lstFontSize.getSelectedInt());
    StyleConstants.setBold(m_attributes,
      m_chkBold.isSelected());
    StyleConstants.setItalic(m_attributes,
      m_chkItalic.isSelected());
    StyleConstants.setUnderline(m_attributes,
      m_chkUnderline.isSelected());
    StyleConstants.setStrikeThrough(m_attributes,
      m_chkStrikethrough.isSelected());
    StyleConstants.setSubscript(m_attributes,
      m_chkSubscript.isSelected());
    StyleConstants.setSuperscript(m_attributes,
      m_chkSuperscript.isSelected());
    StyleConstants.setForeground(m_attributes,
      (Color)m_cbColor.getSelectedItem());
    return m_attributes;
  }

  public int getOption() { return m_option; }

  protected void updatePreview() {
    String name = m_lstFontName.getSelected();
    int size = m_lstFontSize.getSelectedInt();
    if (size <= 0)
      return;
    int style = Font.PLAIN;
    if (m_chkBold.isSelected())
      style |= Font.BOLD;
    if (m_chkItalic.isSelected())
      style |= Font.ITALIC;

    // Bug Alert! This doesn't work if only style is changed.
    Font fn = new Font(name, style, size);
```

```
    m_preview.setFont(fn);

    Color c = (Color)m_cbColor.getSelectedItem();
    m_preview.setForeground(c);
    m_preview.repaint();
  }
}

class OpenList extends JPanel
  implements ListSelectionListener, ActionListener
{
  protected JLabel m_title;
  protected JTextField m_text;
  protected JList m_list;
  protected JScrollPane m_scroll;

  public OpenList(String[] data, String title) {
    setLayout(null);
    m_title = new JLabel(title, JLabel.LEFT);
    add(m_title);
    m_text = new JTextField();
    m_text.addActionListener(this);
    add(m_text);
    m_list = new JList(data);
    m_list.setVisibleRowCount(4);
    m_list.addListSelectionListener(this);
    m_scroll = new JScrollPane(m_list);
    add(m_scroll);
  }

  public void setSelected(String sel) {
    m_list.setSelectedValue(sel, true);
    m_text.setText(sel);
  }

  public String getSelected() { return m_text.getText(); }

  public void setSelectedInt(int value) {
    setSelected(Integer.toString(value));
  }

  public int getSelectedInt() {
    try {
      return Integer.parseInt(getSelected());
    }
    catch (NumberFormatException ex) { return -1; }
  }

  public void valueChanged(ListSelectionEvent e) {
    Object obj = m_list.getSelectedValue();
    if (obj != null)
      m_text.setText(obj.toString());
  }

  public void actionPerformed(ActionEvent e) {
    ListModel model = m_list.getModel();
    String key = m_text.getText().toLowerCase();
    for (int k=0; k<model.getSize(); k++) {
      String data = (String)model.getElementAt(k);
      if (data.toLowerCase().startsWith(key)) {
        m_list.setSelectedValue(data, true);
        break;
```

```
      }
    }
  }

  public void addListSelectionListener(ListSelectionListener lst) {
    m_list.addListSelectionListener(lst);
  }

  public Dimension getPreferredSize() {
    Insets ins = getInsets();
    Dimension d1 = m_title.getPreferredSize();
    Dimension d2 = m_text.getPreferredSize();
    Dimension d3 = m_scroll.getPreferredSize();
    int w = Math.max(Math.max(d1.width, d2.width), d3.width);
    int h = d1.height + d2.height + d3.height;
    return new Dimension(w+ins.left+ins.right,
      h+ins.top+ins.bottom);
  }

  public Dimension getMaximumSize() {
    Insets ins = getInsets();
    Dimension d1 = m_title.getMaximumSize();
    Dimension d2 = m_text.getMaximumSize();
    Dimension d3 = m_scroll.getMaximumSize();
    int w = Math.max(Math.max(d1.width, d2.width), d3.width);
    int h = d1.height + d2.height + d3.height;
    return new Dimension(w+ins.left+ins.right,
      h+ins.top+ins.bottom);
  }

  public Dimension getMinimumSize() {
    Insets ins = getInsets();
    Dimension d1 = m_title.getMinimumSize();
    Dimension d2 = m_text.getMinimumSize();
    Dimension d3 = m_scroll.getMinimumSize();
    int w = Math.max(Math.max(d1.width, d2.width), d3.width);
    int h = d1.height + d2.height + d3.height;
    return new Dimension(w+ins.left+ins.right,
      h+ins.top+ins.bottom);
  }

  public void doLayout() {
    Insets ins = getInsets();
    Dimension d = getSize();
    int x = ins.left;
    int y = ins.top;
    int w = d.width-ins.left-ins.right;
    int h = d.height-ins.top-ins.bottom;

    Dimension d1 = m_title.getPreferredSize();
    m_title.setBounds(x, y, w, d1.height);
    y += d1.height;
    Dimension d2 = m_text.getPreferredSize();
    m_text.setBounds(x, y, w, d2.height);
    y += d2.height;
    m_scroll.setBounds(x, y, w, h-y);
  }
}

class ColorComboRenderer extends JPanel implements ListCellRenderer
{
  protected Color m_color = Color.black;
```

```
  protected Color m_focusColor =
    (Color) UIManager.get("List.selectionBackground");
  protected Color m_nonFocusColor = Color.white;

  public Component getListCellRendererComponent(JList list,
   Object obj, int row, boolean sel, boolean hasFocus)
  {
    if (hasFocus || sel)
      setBorder(new CompoundBorder(
        new MatteBorder(2, 10, 2, 10, m_focusColor),
        new LineBorder(Color.black)));
    else
      setBorder(new CompoundBorder(
        new MatteBorder(2, 10, 2, 10, m_nonFocusColor),
        new LineBorder(Color.black)));

    if (obj instanceof Color)
      m_color = (Color) obj;
    return this;
  }

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

Understanding the Code

Class WordProcessor

Three new instance variables are added:

`String[] m_fontNames`: array of available font family names.

`String[] m_fontSizes`: array of font sizes.

`FontDialog m_fontDialog`: custom font properties and preview dialog.

These arrays were used earlier as local variables to create the toolbar combobox components. Since we need to use them in our font dialog as well, we decided to make them public instance variables (this requires minimal changes to the `createMenuBar()` method).

Note: Reading the list of available fonts takes a significant amount of time. For performance reasons it is best to do this once in a program.

A new menu item titled "Font..." is now added to the "Format" menu. When the corresponding `ActionListener` is invoked, the application is repainted, the attributes of the character element corresponding to the current caret position are retrieved as an `AttributeSet` instance and passed to the dialog for selection (using its `setAttributes()`), and the dialog is centered relative to the parent frame and displayed. If the dialog is closed with the "OK" button (determined by checking a value returned by `FontDialog`'s `getOption()` method), we retrieve the new font attributes with `FontDialog.getAttributes()`, and assign these attributes to the selected text with our `setAttributeSet()` method. Finally, our toolbar components are updated with our `showAttributes()` method.

Class FontDialog

This class extends JDialog and acts as a font properties editor and previewer for our word processor application. Several instance variables are declared:

- int m_option: indicates how the dialog is closed: by pressing the "OK" button, by pressing the "Cancel" button, or by closing the dialog window directly from the title bar. The constants defined in JOptionPane are reused for this variable.
- MutableAttributeSet m_attributes: a collection of font attributes used to preserve the user's selection.
- OpenList m_lstFontName: custom JList sub-class for selecting the font family name.
- OpenList m_lstFontSize: custom JList sub-class for selecting the font size.
- JCheckBox m_chkBold: checkbox to select the bold attribute.
- JCheckBox m_chkItalic: checkbox to select the italic attribute.
- JCheckBox m_chkUnderline: checkbox to select the font underline attribute.
- JCheckBox m_chkStrikethrough: checkbox to select the font strikethrough attribute.
- JCheckBox m_chkSubscript: checkbox to select the font subscript attribute.
- JCheckBox m_chkSuperscript: checkbox to select the font superscript attribute.
- JComboBox m_cbColor: combobox to select the font foreground color.
- JLabel m_preview: label to preview the selections.

The FontDialog constructor first creates a super-class modal dialog titled "Font." The constructor creates and initializes all GUI components used in this dialog. A y-oriented BoxLayout is used to place component groups from top to bottom.

Two OpenList components (see below) are placed at the top to select an available font family name and font size. These components encapsulate a label, text box and list components which work together. They are comparable to editable comboboxes that always keep their drop-down list open. Below the OpenLists, a group of six checkboxes are placed for selecting bold, italic, underline, strikethrough, subscript, and superscript font attributes. JComboBox m_cbColor is placed below this group, and is used to select the font foreground color. 64 Colors are added, and an instance of our custom ColorComboRenderer class (see below) is used as its list cell renderer. JLabel m_preview is used to preview the selected font before applying it to the editing text, and is placed below the foreground color combo box.

Two buttons labeled "OK" and "Cancel" are placed at the bottom of the dialog. They are placed in a panel managed by a 1x2 GridLayout, which is in turn placed in a panel managed by a FlowLayout. This is to ensure the equal sizing and central placement of the buttons. Both receive ActionListeners which hide the dialog and set the m_option instance variable to JOptionPane.OK_OPTION and JOptionPane.CANCEL_OPTION respectively. An application (WordProcessor in our case) will normally check this value once the modal dialog is dismissed by calling its getOption() method. This tells the application whether or not the changes should be ignored (CANCEL_OPTION) or applied (OK_OPTION).

The dialog window is packed to give it a natural size, and is then centered with respect to the parent frame.

The `m_lstFontName` and `m_lstFontSize OpenList` components each receive the same `ListSelectionListener` instance which calls our custom `updatePreview()` method (see below) whenever the list selection is changed. Similarly, two checkboxes and the foreground color combobox receive an `ActionListener` which does the same thing. This provides dynamic preview of the selected font attributes as soon as any is changed.

Bug Alert! Underline, strikethrough, subscript, and superscript font properties are not supported by the `AWT Font` class, so they cannot be shown in the `JLabel` component. This is why the corresponding checkbox components do not receive an `ActionListener`. As we will see, these properties also do not work properly in `RTF` documents. They are included in this dialog for completeness, in the hopes that they will work properly in a future Swing release.

The `setAttributes()` method takes an `AttributeSet` instance as parameter. It copies this attribute set into a `SimpleAttributeSet` stored as our `m_attributes` instance variable. Appropriate font attributes are extracted using `StyleConstants` methods, and used to assign values to the dialog's controls. Finally the preview label is updated according to these new settings by calling our `updatePreview()` method. Note that the `setAttributes()` method is `public` and is used for data exchange between this dialog and it's owner (in our case `WordProcessor`).

The `getAttributes()` method plays an opposite role with respect to `setAttributes()`. It retrieves data from dialog's controls, packs them into an `AttributeSet` instance using `StyleConstants` methods, and returns this set to the caller

The `getOption()` method returns a code indicating how the dialog was closed by the user. This value should be checked prior to retrieving data from the dialog to determine whether or not the user canceled (`JOptionPane.CANCEL_OPTION`) or ok'd (`JOptionPane.OK_OPTION`) the changes.

The `updatePreview()` method is called to update the font preview label when a font attribute is changed. It retrieves the selected font attributes (family name, size, bold and italic properties) and creates a new `Font` instance to render the label. The selected color is retrieved from the `m_cbColor` combobox and set as the label's foreground.

Class OpenList

This component consists of a title label, a text field, and a list in a scroll pane. The user can either select a value from the list, or enter it in the text box manually. `OpenList` extends `JPanel` and maintains the following four instance variables:

- `JLabel m_title`: title label used to identify the purpose of this component.
- `JTextField m_text`: editable text field.
- `JList m_list`: list component.
- `JScrollPane m_scroll`: scroll pane containing the list component.

The `OpenList` constructir assigns a `null` layout manager because this container manages its child components on its own. The four components listed above are instantiated and simply added to this container.

The `setSelected()` method sets the text field text to that of the given `String`, and selects the

corresponding item in the list (which is scrolled to show display the newly selected value). The `getSelected()` method retrieves and returns the selected item as a `String`.

Methods `setSelectedInt()`/`getSelectedInt()` do the same but with `int` values. These methods are implemented to simplify working with a list of `int`s.

The `valueChanged()` and `actionPerformed()` methods provide coordination between the list component and the text field. The `valueChanged()` method is called whenever the list selection changes, and will assign the result of a `toString()` call on the selected item as the text field's text. The `actionPerformed()` method will be called when the user presses Enter while the text field has the current focus. This implementation performs a case-insensitive search through the list items in an effort to find an item which begins with the entered text. If such an item is found, it is selected.

The public `addListSelectionListener()` method adds a `ListSelectionListener` to our list component (which is protected). In this way, external objects can dynamically receive notifications about changes in that list's selection.

The `getPreferredSize()`, `getMaximumSize()`, and `getMinimumSize()` methods calculate and return a preferred, maximum, and minimum dimension of this container respectively. They assume that the three child components (label, text field, and scroll pane containing the list) will be laid out one under another from top to bottom, receiving an equal width and their preferable heights. The `doLayout()` method actually lays out the components according to this scheme. Note that the insets (resulting from an assigned border, for instance) must always be taken into account (see chapter 4 for more about custom layout management).

Class ColorComboRenderer

This class implements the `ListCellRenderer` interface (discussed in chapters 9 and 10) and is used to represent various `Colors`. Three instance variables are defined:

- `Color m_color`: used for the main background color to represent a `Color`.
- `Color m_focusColor`: used for the thick border color of a selected item.
- `Color m_nonFocusColor`: used for the thick border color of an unselected item.

The `getListCellRendererComponent()` method is called prior to the rendering of each list item (in our `WordProcessor` example this list is contained within our foreground colors combo box). The `Color` instance is retrieved and stored in the `m_color` instance variable. This color is used as the renderer's background, while a white matte border is used to surround unselected cells, and a light blue matte border is used to surround a selected cell. The `paintComponent()` method simply sets the background to `m_color` and calls the super-class `paintComponent()` method.

Running the Code

Open an existing RTF file, select a portion of text, and bring up the font dialog. Verify that the initial values correspond to the font attributes of the character element at the current caret position. Try selecting different font attributes and note that the preview component is updated dynamically. Press the "OK" button to apply the selected attributes to the selected text. Also verify that pressing the "Cancel" button does not apply any changes. Figures 20.6 and 20.7 illustrate.

20.7 Word Processor: part VII - Paragraph formatting

Control over paragraph formatting attributes (e.g. line spacing, text alignment, left and right margins, etc.) is just as necessary in word processor applications as font attribute control. Swing supports a number of paragraph settings discussed below which we discussed briefly in chapter 19. In this section we'll add a dialog specifically for editing these settings. The most interesting aspect of this dialog is a special component we've designed to allow a preview of formatted text. In this way the user can get a feeling for how a setting change, or group of changes, will affect the actual document.

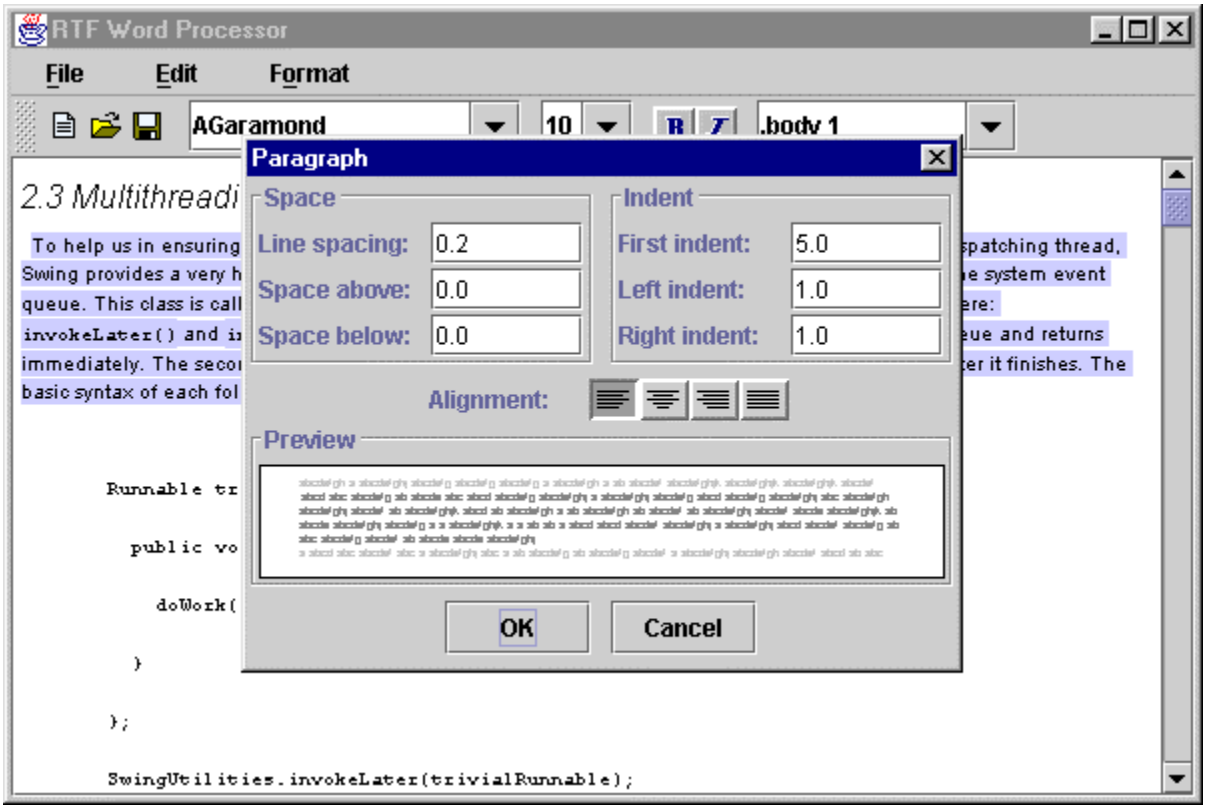


Figure 20.8 RTF word processor displaying a custom paragraph attributes dialog.
<<file figure20-8.gif>>

The Code: WordProcessor.java
see \Chapter20\

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

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;
import javax.swing.undo.*;

public class WordProcessor extends JFrame
{
  // Unchanged code from section 20.6
```

```
  protected FontDialog m_fontDialog;
  protected ParagraphDialog m_paragraphDialog;

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 20.6

    m_fontDialog = new FontDialog(this, m_fontNames, m_fontSizes);

    m_paragraphDialog = new ParagraphDialog(this);

    // Unchanged code from section 20.6

    item = new JMenuItem("Paragraph...");
    item.setMnemonic('p');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        WordProcessor.this.repaint();
        AttributeSet a = m_doc.getCharacterElement(
          m_monitor.getCaretPosition()).getAttributes();
        m_paragraphDialog.setAttributes(a);

        Dimension d1 = m_paragraphDialog.getSize();
        Dimension d2 = WordProcessor.this.getSize();
        int x = Math.max((d2.width-d1.width)/2, 0);
        int y = Math.max((d2.height-d1.height)/2, 0);
        m_paragraphDialog.setBounds(x + WordProcessor.this.getX(),
          y + WordProcessor.this.getY(), d1.width, d1.height);

        m_paragraphDialog.show();
        if (m_paragraphDialog.getOption()==JOptionPane.OK_OPTION) {
          setAttributeSet(dlg.getAttributes(), true);
          showAttributes(m_monitor.getCaretPosition());
        }
      }
    };
    item.addActionListener(lst);
    mFormat.add(item);

    mFormat.addSeparator();

    // Unchanged code from section 20.6

    return menuBar;
  }

  // Unchanged code from section 20.6

  protected void setAttributeSet(AttributeSet attr) {
    setAttributeSet(attr, false);
  }

  protected void setAttributeSet(AttributeSet attr,
   boolean setParagraphAttributes)
  {
    if (m_skipUpdate)
      return;
    int xStart = m_monitor.getSelectionStart();
    int xFinish = m_monitor.getSelectionEnd();
    if (!m_monitor.hasFocus()) {
      xStart = m_xStart;
      xFinish = m_xFinish;
    }
```

```
    if (setParagraphAttributes)
      m_doc.setParagraphAttributes(xStart,
      xFinish - xStart, attr, false);
    else if (xStart != xFinish)
      m_doc.setCharacterAttributes(xStart,
        xFinish - xStart, attr, false);
    else {
      MutableAttributeSet inputAttributes =
        m_kit.getInputAttributes();
      inputAttributes.addAttributes(attr);
    }
  }

  // Unchanged code from section 20.6
}

// Unchanged code from section 20.6

class ParagraphDialog extends JDialog
{
  protected int m_option = JOptionPane.CLOSED_OPTION;
  protected MutableAttributeSet m_attributes;
  protected JTextField m_lineSpacing;
  protected JTextField m_spaceAbove;
  protected JTextField m_spaceBelow;
  protected JTextField m_firstIndent;
  protected JTextField m_leftIndent;
  protected JTextField m_rightIndent;
  protected SmallToggleButton m_btLeft;
  protected SmallToggleButton m_btCenter;
  protected SmallToggleButton m_btRight;
  protected SmallToggleButton m_btJustified;

  protected ParagraphPreview m_preview;

  public ParagraphDialog(JFrame parent) {
    super(parent, "Paragraph", true);
    getContentPane().setLayout(new BoxLayout(getContentPane(),
      BoxLayout.Y_AXIS));

    JPanel p = new JPanel(new GridLayout(1, 2, 5, 2));

    JPanel ps = new JPanel(new GridLayout(3, 2, 10, 2));
    ps.setBorder(new TitledBorder(new EtchedBorder(), "Space"));
    ps.add(new JLabel("Line spacing:"));
    m_lineSpacing = new JTextField();
    ps.add(m_lineSpacing);
    ps.add(new JLabel("Space above:"));
    m_spaceAbove = new JTextField();
    ps.add(m_spaceAbove);
    ps.add(new JLabel("Space below:"));
    m_spaceBelow = new JTextField();
    ps.add(m_spaceBelow);
    p.add(ps);

    JPanel pi = new JPanel(new GridLayout(3, 2, 10, 2));
    pi.setBorder(new TitledBorder(new EtchedBorder(), "Indent"));
    pi.add(new JLabel("First indent:"));
    m_firstIndent = new JTextField();
    pi.add(m_firstIndent);
    pi.add(new JLabel("Left indent:"));
    m_leftIndent = new JTextField();
```

```
pi.add(m_leftIndent);
pi.add(new JLabel("Right indent:"));
m_rightIndent = new JTextField();
pi.add(m_rightIndent);
p.add(pi);
getContentPane().add(p);

getContentPane().add(Box.createVerticalStrut(5));
p = new JPanel();
p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
p.add(Box.createHorizontalStrut(10));
p.add(new JLabel("Alignment:"));
p.add(Box.createHorizontalStrut(20));

ButtonGroup bg = new ButtonGroup();
ImageIcon img = new ImageIcon("al_left.gif");
m_btLeft = new SmallToggleButton(false, img, img, "Left");
bg.add(m_btLeft);
p.add(m_btLeft);
img = new ImageIcon("al_center.gif");
m_btCenter = new SmallToggleButton(false, img, img, "Center");
bg.add(m_btCenter);
p.add(m_btCenter);
img = new ImageIcon("al_right.gif");
m_btRight = new SmallToggleButton(false, img, img, "Right");
bg.add(m_btRight);
p.add(m_btRight);
img = new ImageIcon("al_justify.gif");
m_btJustified = new SmallToggleButton(false, img, img,
  "Justify");
bg.add(m_btJustified);
p.add(m_btJustified);
getContentPane().add(p);

p = new JPanel(new BorderLayout());
p.setBorder(new TitledBorder(new EtchedBorder(), "Preview"));
m_preview = new ParagraphPreview();
p.add(m_preview, BorderLayout.CENTER);
getContentPane().add(p);

p = new JPanel(new FlowLayout());
JPanel p1 = new JPanel(new GridLayout(1, 2, 10, 2));
JButton btOK = new JButton("OK");
ActionListener lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_option = JOptionPane.OK_OPTION;
    setVisible(false);
  }
};
btOK.addActionListener(lst);
p1.add(btOK);

JButton btCancel = new JButton("Cancel");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    m_option = JOptionPane.CANCEL_OPTION;
    setVisible(false);
  }
};
btCancel.addActionListener(lst);
p1.add(btCancel);
p.add(p1);
```

```
    getContentPane().add(p);

    pack();
    setResizable(false);

    FocusListener flst = new FocusListener() {
      public void focusGained(FocusEvent e) {}

      public void focusLost(FocusEvent e) { updatePreview(); }
    };
    m_lineSpacing.addFocusListener(flst);
    m_spaceAbove.addFocusListener(flst);
    m_spaceBelow.addFocusListener(flst);
    m_firstIndent.addFocusListener(flst);
    m_leftIndent.addFocusListener(flst);
    m_rightIndent.addFocusListener(flst);

    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        updatePreview();
      }
    };
    m_btLeft.addActionListener(lst);
    m_btCenter.addActionListener(lst);
    m_btRight.addActionListener(lst);
    m_btJustified.addActionListener(lst);
  }

  public void setAttributes(AttributeSet a) {
    m_attributes = new SimpleAttributeSet(a);
    m_lineSpacing.setText(Float.toString(
      StyleConstants.getLineSpacing(a)));
    m_spaceAbove.setText(Float.toString(
      StyleConstants.getSpaceAbove(a)));
    m_spaceBelow.setText(Float.toString(
      StyleConstants.getSpaceBelow(a)));
    m_firstIndent.setText(Float.toString(
      StyleConstants.getFirstLineIndent(a)));
    m_leftIndent.setText(Float.toString(
      StyleConstants.getLeftIndent(a)));
    m_rightIndent.setText(Float.toString(
      StyleConstants.getRightIndent(a)));

    int alignment = StyleConstants.getAlignment(a);
    if (alignment == StyleConstants.ALIGN_LEFT)
      m_btLeft.setSelected(true);
    else if (alignment == StyleConstants.ALIGN_CENTER)
      m_btCenter.setSelected(true);
    else if (alignment == StyleConstants.ALIGN_RIGHT)
      m_btRight.setSelected(true);
    else if (alignment == StyleConstants.ALIGN_JUSTIFIED)
      m_btJustified.setSelected(true);

    updatePreview();
  }

  public AttributeSet getAttributes() {
    if (m_attributes == null)
      return null;
    float value;
    try {
      value = Float.parseFloat(m_lineSpacing.getText());
```

```
      StyleConstants.setLineSpacing(m_attributes, value);
    } catch (NumberFormatException ex) {}
    try {
      value = Float.parseFloat(m_spaceAbove.getText());
      StyleConstants.setSpaceAbove(m_attributes, value);
    } catch (NumberFormatException ex) {}
    try {
      value = Float.parseFloat(m_spaceBelow.getText());
      StyleConstants.setSpaceBelow(m_attributes, value);
    } catch (NumberFormatException ex) {}
    try {
      value = Float.parseFloat(m_firstIndent.getText());
      StyleConstants.setFirstLineIndent(m_attributes, value);
    } catch (NumberFormatException ex) {}
    try {
      value = Float.parseFloat(m_leftIndent.getText());
      StyleConstants.setLeftIndent(m_attributes, value);
    } catch (NumberFormatException ex) {}
    try {
      value = Float.parseFloat(m_rightIndent.getText());
      StyleConstants.setRightIndent(m_attributes, value);
    } catch (NumberFormatException ex) {}

    StyleConstants.setAlignment(m_attributes, getAlignment());

    return m_attributes;
  }

  public int getOption() {
    return m_option;
  }

  protected void updatePreview() {
    m_preview.repaint();
  }

  protected int getAlignment() {
    if (m_btLeft.isSelected())
      return StyleConstants.ALIGN_LEFT;
    if (m_btCenter.isSelected())
      return StyleConstants.ALIGN_CENTER;
    else if (m_btRight.isSelected())
      return StyleConstants.ALIGN_RIGHT;
    else
      return StyleConstants.ALIGN_JUSTIFIED;
  }

  class ParagraphPreview extends JPanel
  {
    protected Font m_fn = new Font("Monospace", Font.PLAIN, 6);
    protected String m_dummy = "abcdefghjklm";
    protected float  m_scaleX = 0.25f;
    protected float  m_scaleY = 0.25f;
    protected Random m_random = new Random();

    public ParagraphPreview() {
      setBackground(Color.white);
      setForeground(Color.black);
      setOpaque(true);
      setBorder(new LineBorder(Color.black));
      setPreferredSize(new Dimension(120, 56));
    }
```

```
public void paintComponent(Graphics g) {
  super.paintComponent(g);
  float lineSpacing = 0;
  float spaceAbove  = 0;
  float spaceBelow  = 0;
  float firstIndent = 0;
  float leftIndent  = 0;
  float rightIndent = 0;

  try {
    lineSpacing = Float.parseFloat(m_lineSpacing.getText());
  } catch (NumberFormatException ex) {}
  try {
    spaceAbove = Float.parseFloat(m_spaceAbove.getText());
  } catch (NumberFormatException ex) {}
  try {
    spaceBelow = Float.parseFloat(m_spaceBelow.getText());
  } catch (NumberFormatException ex) {}
  try {
    firstIndent = Float.parseFloat(m_firstIndent.getText());
  } catch (NumberFormatException ex) {}
  try {
    leftIndent = Float.parseFloat(m_leftIndent.getText());
  } catch (NumberFormatException ex) {}
  try {
    rightIndent = Float.parseFloat(m_rightIndent.getText());
  } catch (NumberFormatException ex) {}

  m_random.setSeed(1959);    // Use same seed every time

  g.setFont(m_fn);
  FontMetrics fm = g.getFontMetrics();
  int h = fm.getAscent();
  int s  = Math.max((int)(lineSpacing*m_scaleY), 1);
  int s1 = Math.max((int)(spaceAbove*m_scaleY), 0) + s;
  int s2 = Math.max((int)(spaceBelow*m_scaleY), 0) + s;
  int y = 5+h;

  int xMarg = 20;
  int x0 = Math.max((int)(firstIndent*m_scaleX)+xMarg, 3);
  int x1 = Math.max((int)(leftIndent*m_scaleX)+xMarg, 3);
  int x2 = Math.max((int)(rightIndent*m_scaleX)+xMarg, 3);
  int xm0 = getWidth()-xMarg;
  int xm1 = getWidth()-x2;

  int n = (int)((getHeight()-(2*h+s1+s2-s+10))/(h+s));
  n = Math.max(n, 1);

  g.setColor(Color.lightGray);
  int x = xMarg;
  drawLine(g, x, y, xm0, xm0, fm, StyleConstants.ALIGN_LEFT);
  y += h+s1;

  g.setColor(Color.gray);
  int alignment = getAlignment();
  for (int k=0; k<n; k++) {
    x = (k==0 ? x0 : x1);
    int xLen = (k==n-1 ? xm1/2 : xm1);
    if (k==n-1 && alignment==StyleConstants.ALIGN_JUSTIFIED)
      alignment = StyleConstants.ALIGN_LEFT;
    drawLine(g, x, y, xm1, xLen, fm, alignment);
```

```
        y += h+s;
      }

      y += s2-s;
      x = xMarg;
      g.setColor(Color.lightGray);
      drawLine(g, x, y, xm0, xm0, fm, StyleConstants.ALIGN_LEFT);
    }

    protected void drawLine(Graphics g, int x, int y, int xMax,
     int xLen, FontMetrics fm, int alignment)
    {
      if (y > getHeight()-3)
        return;
      StringBuffer s = new StringBuffer();
      String str1;
      int xx = x;
      while (true) {
        int m = m_random.nextInt(10)+1;
        str1 = m_dummy.substring(0, m)+" ";
        int len = fm.stringWidth(str1);
        if (xx+len >= xLen)
          break;
        xx += len;
        s.append(str1);
      }
      String str = s.toString();

      switch (alignment) {
        case StyleConstants.ALIGN_LEFT:
          g.drawString(str, x, y);
          break;
        case StyleConstants.ALIGN_CENTER:
          xx = (xMax+x-fm.stringWidth(str))/2;
          g.drawString(str, xx, y);
          break;
        case StyleConstants.ALIGN_RIGHT:
          xx = xMax-fm.stringWidth(str);
          g.drawString(str, xx, y);
          break;
        case StyleConstants.ALIGN_JUSTIFIED:
          while (x+fm.stringWidth(str) < xMax)
            str += "a";
          g.drawString(str, x, y);
          break;
      }
    }
  }
}
```

Understanding the Code

Class WordProcessor

One new instance variable has been added:

`ParagraphDialog m_paragraphDialog`: custom dialog used to manage paragraph attributes.

A new menu item titled "Paragraph..." is now added to the "Format" menu. A corresponding `ActionListener` acts similarly to the listener of "Font..." menu. It repaints the entire application, retrieves

a set of attributes corresponding to the character element at the current caret position, and passes this set to `m_paragraphDialog`. The dialog is then centered with respect to its parent (`WordProcessor` in our case) and shows itself. When the "OK" or "Cancel" button is pressed, the result returned by the `getOption()` method is normally checked. If the returned value is equal to `JOptionPane.OK_OPTION` (i.e. "OK" was pressed) we retrieve the selected attributes with `ParagraphDialog`'s `getAttributes()` method and assign them to the selected text with our `setAttributeSet()` method. Otherwise we make no changes.

An additional parameter, a boolean, is added to our `setAttributeSet()` method. This is used to distinguish between setting character attributes and setting paragraph attributes. A value of `true` indicates that the given attribute set corresponds to paragraph attributes. A value of `false` indicates character attributes. To preserve the existing code without requiring extensive modification, we keep the old `setAttributeSet()` method with one parameter, and redirect it to the new method by having it call `setAttributeSet(attr, false)`.

Class ParagraphDialog

This class extends `JDialog` and acts as a paragraph attributes editor for our word processor application. Several instance variables are declared:

- `int m_option`: indicates how the dialog is closed: by pressing the "OK" button, by pressing the "Cancel" button, or by closing the dialog window directly from the title bar. The constants defined in `JOptionPane` are reused for this variable.
- `MutableAttributeSet m_attributes`: a collection of paragraph attributes used to preserve the user's selection.
- `JTextField m_lineSpacing`: used to specify paragraph line spacing.
- `JTextField m_spaceAbove`: used to specify above line spacing.
- `JTextField m_spaceBelow`: used to specify below line spacing.
- `JTextField m_firstIndent`: used to specify left indent of the first paragraph line.
- `JTextField m_leftIndent`: used to specify left indent of all other paragraph lines (other than the first).
- `JTextField m_rightIndent`: used to specify right indent of all paragraph lines.
- `SmallToggleButton m_btLeft`: used to toggle left text alignment.
- `SmallToggleButton m_btCenter`: used to toggle center text alignment.
- `SmallToggleButton m_btRight`: used to toggle right text alignment.
- `SmallToggleButton m_btJustified`: used to toggle justified text alignment.
- `ParagraphPreview m_preview`: custom component for previewing paragraph attribute effects.

The `ParagraphDialog` constructor first creates a super-class modal dialog titled "Paragraph." This dialog uses a y-oriented `BoxLayout` to place component groups from top to bottom. Six text fields listed above are placed in two side-by-side panels titled "Space" and "Indent." These controls allow the user to specify spacing attributes. Below these fields is a group of `SmallToggleButtons` (introduced in chapter 12) used to control text alignment: left, center, right, or justified.

An instance of our custom `ParagraphPreview` component, `m_preview`, is used to preview the selected

paragraph attributes before applying them to the selected document text. We will discuss how this component works below.

Two buttons labeled "OK" and "Cancel" are placed at the bottom of the dialog and act identically to those at the bottom of our font dialog (see previous example). The six text boxes mentioned above receive an identical `FocusListener` which invokes our `updatePreview()` method (see below) when a text field loses focus. Similarly, the four toggle buttons receive an identical `ActionListener` which does the same thing. This provides a dynamic preview of the selected paragraph attributes whenever any attribute is changed.

The `setAttributes()` method takes an `AttributeSet` instance as parameter. It copies this attribute set into a `SimpleAttributeSet` stored as our `m_attributes` instance variable. Appropriate paragraph attributes are extracted using `StyleConstants` methods, and used to assign values to the dialog's controls. Finally the preview component is updated according to these new settings by calling our `updatePreview()` method. Note that the `setAttributes()` method is `public` and is used for data exchange between this dialog and it's owner (in our case `WordProcessor`).

The method `getAttributes()` plays an opposite role with respect to `setAttributes()`. It retrieves data from the dialog's controls, packs them into an `AttributeSet` instance using `StyleConstants` methods, and return this set to the caller.

Note: All spacing variables are of type `float`, even though they are actually measured in discrete screen pixels.

The `getOption()` method returns a code indicating how the dialog was closed by the user. This value should be checked prior to retrieving data from the dialog to determine whether or not the user canceled (`JOptionPane.CANCEL_OPTION`) or ok'd (`JOptionPane.OK_OPTION`) the changes.

The `updatePreview()` method is called to update the paragraph preview component whenever a paragraph attribute is changed. It simply forces our `m_preview` component to repaint itself.

The `getAlignment()` method checks for the selected toggle button and returns the corresponding alignment attribute.

Class ParagraphDialog.ParagraphPreview

This inner class represents our custom component used to display an imitation paragraph used to preview a set of paragraph attributes. The actual rendering consists of three parts:

1. A light gray text line representing the end of a preceding paragraph. The indent and spacing of this line is not affected by the current paragraph attribute settings.
2. Several gray text lines representing a paragraph being modified. Indentations and line spacing depend on the current paragraph attribute settings. The number of these lines is calculated so as to fill the component's height naturally, and depends on the current line spacing attribute settings.
3. A light gray text line representing the beginning of a following paragraph. The space between this line and the last paragraph line depends on the current above line spacing attribute. The indentation of this line is not affected by the current paragraph attributes.

Several instance variables are declared in this inner class:

`Font m_fn`: the font used for preview paragraph rendering. This font is intentionally made small and barely recognizable, because the displayed text itself does not have any meaning; only the paragraph formatting does.

`String m_dummy`: a dummy string used in random generation of a paragraph.

`float m_scaleX`: the scaling factor used to re-calculate sizes in the vertical direction.

`float m_scaleY`: the scaling factor used to re-calculate sizes in the horizontal direction.

`Random m_random`: random number generator used in piecing together a paragraph of characters.

Reminder: `java.util.Random` provides random number generation capabilities, including seed selection, generation of integers in a given range, etc. In the simplest cases we can use the `Math.random()` static method instead.

The `ParagraphPreview` constructor initializes the colors, border, and preferred size for this component.

The most interesting aspect of the preview component's work is done in the `paintComponent()` method. First, this method retrieves the paragraph attributes specified by the user. Then we set a hard-coded seed value for our random number generator, `m_random`, which we use to generate a paragraph of gibberish. The following local variables are used to control the placement of this paragraph's lines:

`int h`: the height of the text string determined by the font selected for preview.

`int s`: spacing between lines in screen pixels.

`int s1`: actual spacing between the previous paragraph and the first line of this paragraph.

`int s2`: actual spacing between the following paragraph and the last line of this paragraph.

`int y`: vertical position of the text being drawn (to be updated during the rendering process).

`int xMarg`: left and right fixed margins.

`int x0`: actual left indent for the first line of this paragraph.

`int x1`: actual left indent for the second and remaining following lines of this paragraph.

`int x2`: actual right indent for the lines of this paragraph.

`int xm0`: maximum x-coordinate for the text lines without regard to the specified right indent.

`int xm1`: maximum x-coordinate for the text lines with regard to the specified right indent.

`int n`: number of paragraph lines which can fit vertically within this component, taking into account the specified line spacing and both the preceding and following paragraph lines.

Once all these variables are calculated, the rendering can be performed relatively easily. The `drawLine()` method is used to draw a single line of text. First we draw a line denoting the preceding paragraph, then we draw each line of the current paragraph, and finally, a line denoting the following paragraph. Note that the last line of the current paragraph is intentionally half the length of a normal line to produce a more realistic impression of the text. Also, when justified alignment is specified, it is suppressed for the last line of text, since the last line should not be stretched.

The `drawLine()` method takes seven parameters:

`Graphics g`: used for all rendering.

`int x, int y`: coordinates of the beginning of a text line.

`int xMax`: maximum x-coordinate a line can occupy.

`int xLen`: line length plus left margin size.

`FontMetrics fm`: retrieved from the current `Graphics` instance.

`int alignment`: current text alignment.

First this method prepares a line to be drawn by concatenating random pieces of the `m_dummy` string until the resulting length, plus the left margin size, is greater than or equal to the `xLen` parameter. (Note that a `StringBuffer` instance is used here to improve performance.) Then we draw this line depending on the selected alignment. For left alignment we simply start drawing at the left margin. For center and right alignments we calculate the start position by working with the maximum x-coordinate, and the width of that line. For justified alignment we should recalculate space between words so the resulting line will occupy all available width (however, in our case, since the preview text is totally meaningless, we just add some more text at the right end).

Running the Code

Open an existing RTF file, select some text and bring up the "Paragraph" dialog. Verify that the initial values of all components correspond to the paragraph attributes at the current caret position. Specify new paragraph attributes and note how the preview component is updated dynamically. Press the "OK" button to apply the specified attributes to the selected paragraphs of document text.

Bug Alert! Justified text alignment has not been implemented as of Java 2 FCS.

20.8 Word Processor: part VIII - Find and replace

Along with font and paragraph dialogs, find and replace functionality has also become a fairly common tool in GUI-based text editing environments. It is safe to assume that most users would be sadly disappointed if this functionality was not included in a new word processor application. In this section we will show how to add this functionality. Traditionally such tools are represented in an "Edit" menu and can be activated by keyboard accelerators. We will use a dialog containing a single tabbed pane with tabs for finding and replacing a specific region of text. We will also provide several options for searching: match case, search whole words only, and search up or down.

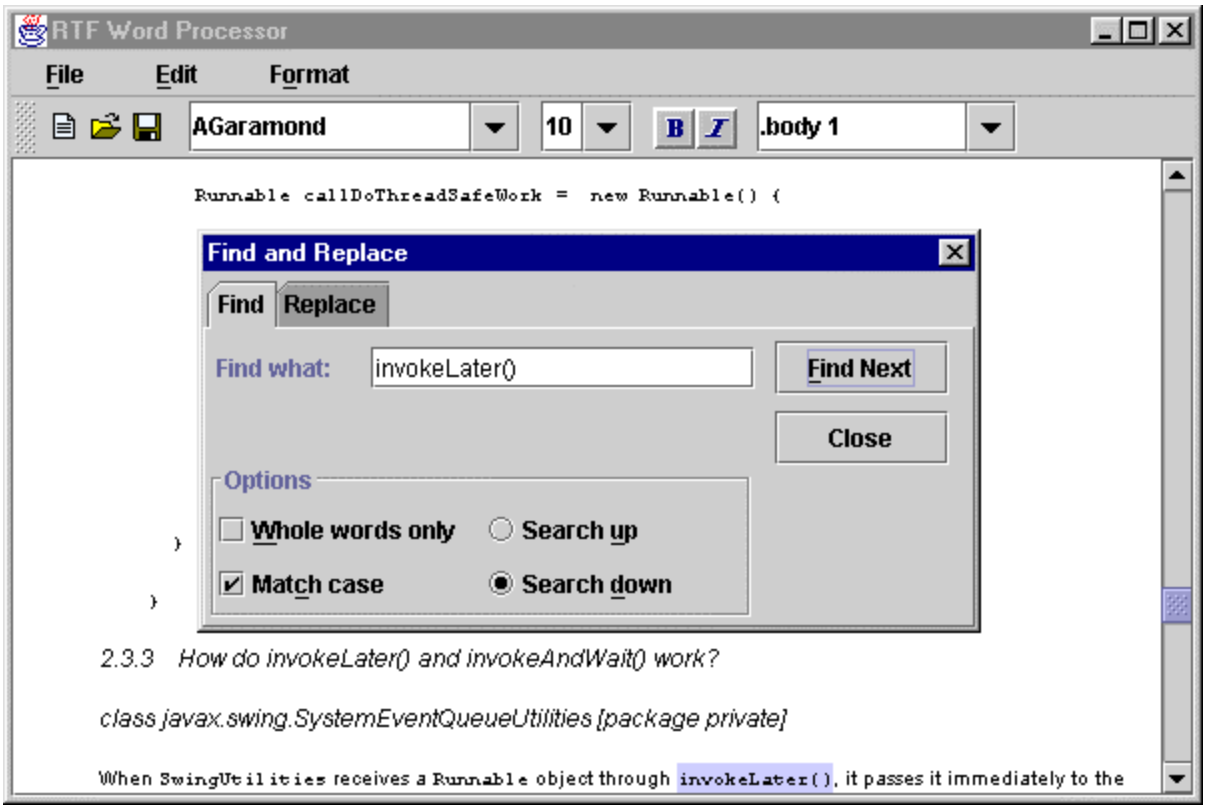


Figure 20.9 WordProcessor with complete find and replace functionality; "Find" tab shown here
<<file figure20-9.gif>>

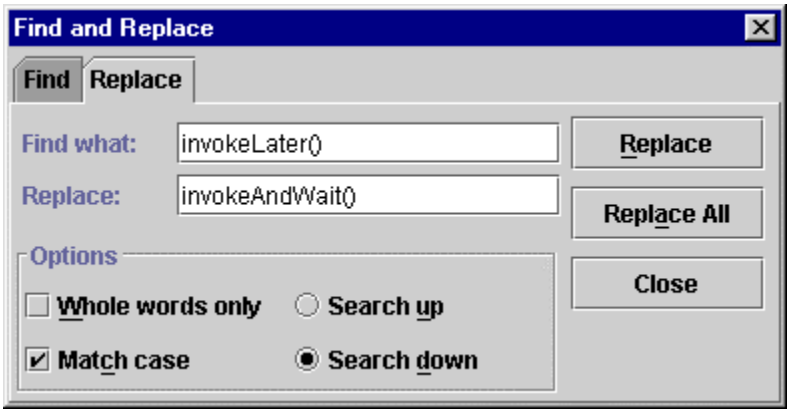


Figure 20.10 "Replace" tab of our custom find and replace dialog used in a word processor application.
<<file figure20-10.gif>>

The Code: WordProcessor.java
see \Chapter20\8

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

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;
```

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

import dl.*;

public class WordProcessor extends JFrame
{
  // Unchanged code from section 20.7

  protected FontDialog m_fontDialog;
  protected ParagraphDialog m_paragraphDialog;
  protected FindDialog m_findDialog;

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 20.7

    mEdit.addSeparator();

    Action findAction = new AbstractAction("Find...",
     new ImageIcon("edit_find.gif"))
    {
      public void actionPerformed(ActionEvent e) {
        WordProcessor.this.repaint();
        if (m_findDialog==null)
          m_findDialog = new FindDialog(WordProcessor.this, 0);
        else
          m_findDialog.setSelectedIndex(0);

        Dimension d1 = m_findDialog.getSize();
        Dimension d2 = WordProcessor.this.getSize();
        int x = Math.max((d2.width-d1.width)/2, 0);
        int y = Math.max((d2.height-d1.height)/2, 0);
        m_findDialog.setBounds(x + WordProcessor.this.getX(),
          y + WordProcessor.this.getY(), d1.width, d1.height);

        m_findDialog.show();
      }
    };
    item = mEdit.add(findAction);
    item.setMnemonic('f');
    item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_F,
      KeyEvent.CTRL_MASK));

    Action replaceAction = new AbstractAction("Replace...") {
      public void actionPerformed(ActionEvent e) {
        WordProcessor.this.repaint();
        if (m_findDialog==null)
          m_findDialog = new FindDialog(WordProcessor.this, 1);
        else
          m_findDialog.setSelectedIndex(1);

        Dimension d1 = m_findDialog.getSize();
        Dimension d2 = WordProcessor.this.getSize();
        int x = Math.max((d2.width-d1.width)/2, 0);
        int y = Math.max((d2.height-d1.height)/2, 0);
        m_findDialog.setBounds(x + WordProcessor.this.getX(),
          y + WordProcessor.this.getY(), d1.width, d1.height);

        m_findDialog.show();
      }
    };
    item =  mEdit.add(replaceAction);
    item.setMnemonic('r');
```

```
    item.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_H,
      KeyEvent.CTRL_MASK));

    menuBar.add(mEdit);

    // Unchanged code from section 20.7

    return menuBar;
  }

  // Unchanged code from section 20.7

  public Document getDocument() { return m_doc; }

  public JTextPane getTextPane() { return m_monitor; }

  public void setSelection(int xStart, int xFinish, boolean moveUp) {
    if (moveUp) {
      m_monitor.setCaretPosition(xFinish);
      m_monitor.moveCaretPosition(xStart);
    }
    else
      m_monitor.select(xStart, xFinish);
    m_xStart = m_monitor.getSelectionStart();
    m_xFinish = m_monitor.getSelectionEnd();
  }

  public static void main(String argv[]) {
    new WordProcessor();
  }

  // Unchanged code from section 20.7
}

// Unchanged code from section 20.7

class FindDialog extends JDialog
{
  protected WordProcessor m_owner;
  protected JTabbedPane m_tb;
  protected JTextField m_txtFind1;
  protected JTextField m_txtFind2;
  protected Document m_docFind;
  protected Document m_docReplace;
  protected ButtonModel m_modelWord;
  protected ButtonModel m_modelCase;
  protected ButtonModel m_modelUp;
  protected ButtonModel m_modelDown;

  protected int m_searchIndex = -1;
  protected boolean m_searchUp = false;
  protected String  m_searchData;

  public FindDialog(WordProcessor owner, int index) {
    super(owner, "Find and Replace", false);
    m_owner = owner;

    m_tb = new JTabbedPane();

    // "Find" panel
    JPanel p1 = new JPanel(new BorderLayout());
```

```
JPanel pc1 = new JPanel(new BorderLayout());

JPanel pf = new JPanel();
pf.setLayout(new DialogLayout(20, 5));
pf.setBorder(new EmptyBorder(8, 5, 8, 0));
pf.add(new JLabel("Find what:"));

m_txtFind1 = new JTextField();
m_docFind = m_txtFind1.getDocument();
pf.add(m_txtFind1);
pc1.add(pf, BorderLayout.CENTER);

JPanel po = new JPanel(new GridLayout(2, 2, 8, 2));
po.setBorder(new TitledBorder(new EtchedBorder(),
  "Options"));

JCheckBox chkWord = new JCheckBox("Whole words only");
chkWord.setMnemonic('w');
m_modelWord = chkWord.getModel();
po.add(chkWord);

ButtonGroup bg = new ButtonGroup();
JRadioButton rdUp = new JRadioButton("Search up");
rdUp.setMnemonic('u');
m_modelUp = rdUp.getModel();
bg.add(rdUp);
po.add(rdUp);

JCheckBox chkCase = new JCheckBox("Match case");
chkCase.setMnemonic('c');
m_modelCase = chkCase.getModel();
po.add(chkCase);

JRadioButton rdDown = new JRadioButton("Search down", true);
rdDown.setMnemonic('d');
m_modelDown = rdDown.getModel();
bg.add(rdDown);
po.add(rdDown);
pc1.add(po, BorderLayout.SOUTH);

p1.add(pc1, BorderLayout.CENTER);

JPanel p01 = new JPanel(new FlowLayout());
JPanel p = new JPanel(new GridLayout(2, 1, 2, 8));

ActionListener findAction = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    findNext(false, true);
  }
};
JButton btFind = new JButton("Find Next");
btFind.addActionListener(findAction);
btFind.setMnemonic('f');
p.add(btFind);

ActionListener closeAction = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    setVisible(false);
  }
};
JButton btClose = new JButton("Close");
btClose.addActionListener(closeAction);
```

```
    btClose.setDefaultCapable(true);
    p.add(btClose);
    p01.add(p);
    p1.add(p01, BorderLayout.EAST);

    m_tb.addTab("Find", p1);

    // "Replace" panel
    JPanel p2 = new JPanel(new BorderLayout());

    JPanel pc2 = new JPanel(new BorderLayout());

    JPanel pc = new JPanel();
    pc.setLayout(new DialogLayout(20, 5));
    pc.setBorder(new EmptyBorder(8, 5, 8, 0));

    pc.add(new JLabel("Find what:"));
    m_txtFind2 = new JTextField();
    m_txtFind2.setDocument(m_docFind);
    pc.add(m_txtFind2);

    pc.add(new JLabel("Replace:"));
    JTextField txtReplace = new JTextField();
    m_docReplace = txtReplace.getDocument();
    pc.add(txtReplace);
    pc2.add(pc, BorderLayout.CENTER);

    po = new JPanel(new GridLayout(2, 2, 8, 2));
    po.setBorder(new TitledBorder(new EtchedBorder(),
      "Options"));

    chkWord = new JCheckBox("Whole words only");
    chkWord.setMnemonic('w');
    chkWord.setModel(m_modelWord);
    po.add(chkWord);

    bg = new ButtonGroup();
    rdUp = new JRadioButton("Search up");
    rdUp.setMnemonic('u');
    rdUp.setModel(m_modelUp);
    bg.add(rdUp);
    po.add(rdUp);

    chkCase = new JCheckBox("Match case");
    chkCase.setMnemonic('c');
    chkCase.setModel(m_modelCase);
    po.add(chkCase);

    rdDown = new JRadioButton("Search down", true);
    rdDown.setMnemonic('d');
    rdDown.setModel(m_modelDown);
    bg.add(rdDown);
    po.add(rdDown);
    pc2.add(po, BorderLayout.SOUTH);

    p2.add(pc2, BorderLayout.CENTER);

    JPanel p02 = new JPanel(new FlowLayout());
    p = new JPanel(new GridLayout(3, 1, 2, 8));

    ActionListener replaceAction = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
```

```
    findNext(true, true);
  }
};
JButton btReplace = new JButton("Replace");
btReplace.addActionListener(replaceAction);
btReplace.setMnemonic('r');
p.add(btReplace);

ActionListener replaceAllAction = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    int counter = 0;
    while (true) {
      int result = findNext(true, false);
      if (result < 0)    // error
        return;
      else if (result == 0)    // no more
        break;
      counter++;
    }
    JOptionPane.showMessageDialog(m_owner,
      counter+" replacement(s) have been done", "Info",
      JOptionPane.INFORMATION_MESSAGE);
  }
};
JButton btReplaceAll = new JButton("Replace All");
btReplaceAll.addActionListener(replaceAllAction);
btReplaceAll.setMnemonic('a');
p.add(btReplaceAll);

btClose = new JButton("Close");
btClose.addActionListener(closeAction);
btClose.setDefaultCapable(true);
p.add(btClose);
p02.add(p);
p2.add(p02, BorderLayout.EAST);

// Make button columns the same size
p01.setPreferredSize(p02.getPreferredSize());

m_tb.addTab("Replace", p2);

m_tb.setSelectedIndex(index);

getContentPane().add(m_tb, BorderLayout.CENTER);

WindowListener flst = new WindowAdapter() {
  public void windowActivated(WindowEvent e) {
    m_searchIndex = -1;
    if (m_tb.getSelectedIndex()==0)
      m_txtFind1.grabFocus();
    else
      m_txtFind2.grabFocus();
  }

  public void windowDeactivated(WindowEvent e) {
    m_searchData = null;
  }
};
addWindowListener(flst);

pack();
setResizable(false);
```

```
  }

  public void setSelectedIndex(int index) {
    m_tb.setSelectedIndex(index);
    setVisible(true);
    m_searchIndex = -1;
  }

  public int findNext(boolean doReplace, boolean showWarnings) {
    JTextPane monitor = m_owner.getTextPane();
    int pos = monitor.getCaretPosition();
    if (m_modelUp.isSelected() != m_searchUp) {
      m_searchUp = m_modelUp.isSelected();
      m_searchIndex = -1;
    }

    if (m_searchIndex == -1) {
      try {
        Document doc = m_owner.getDocument();
        if (m_searchUp)
          m_searchData = doc.getText(0, pos);
        else
          m_searchData = doc.getText(pos, doc.getLength()-pos);
        m_searchIndex = pos;
      }
      catch (BadLocationException ex) {
        warning(ex.toString());
        return -1;
      }
    }

    String key = "";
    try { key = m_docFind.getText(0, m_docFind.getLength()); }
    catch (BadLocationException ex) {}
    if (key.length()==0) {
      warning("Please enter the target to search");
      return -1;
    }
    if (!m_modelCase.isSelected()) {
      m_searchData = m_searchData.toLowerCase();
      key = key.toLowerCase();
    }
    if (m_modelWord.isSelected()) {
      for (int k=0; k<Utils.WORD_SEPARATORS.length; k++) {
        if (key.indexOf(Utils.WORD_SEPARATORS[k]) >= 0) {
          warning("The text target contains an illegal "+
            "character \'"+Utils.WORD_SEPARATORS[k]+"\'");
          return -1;
        }
      }
    }

    String replacement = "";
    if (doReplace) {
      try {
        replacement = m_docReplace.getText(0,
          m_docReplace.getLength());
      } catch (BadLocationException ex) {}
    }

    int xStart = -1;
    int xFinish = -1;
```

```
  while (true)
  {
    if (m_searchUp)
      xStart = m_searchData.lastIndexOf(key, pos-1);
    else
      xStart = m_searchData.indexOf(key, pos-m_searchIndex);
    if (xStart < 0) {
      if (showWarnings)
        warning("Text not found");
      return 0;
    }

    xFinish = xStart+key.length();

    if (m_modelWord.isSelected()) {
      boolean s1 = xStart>0;
      boolean b1 = s1 && !Utils.isSeparator(m_searchData.charAt(
        xStart-1));
      boolean s2 = xFinish<m_searchData.length();
      boolean b2 = s2 && !Utils.isSeparator(m_searchData.charAt(
        xFinish));

      if (b1 || b2)    // Not a whole word
      {
        if (m_searchUp && s1)    // Can continue up
        {
          pos = xStart;
          continue;
        }
        if (!m_searchUp && s2)    // Can continue down
        {
          pos = xFinish;
          continue;
        }
        // Found, but not a whole word, and we cannot continue
        if (showWarnings)
          warning("Text not found");
        return 0;
      }
    }
    break;
  }

  if (!m_searchUp) {
    xStart += m_searchIndex;
    xFinish += m_searchIndex;
  }
  if (doReplace) {
    m_owner.setSelection(xStart, xFinish, m_searchUp);
    monitor.replaceSelection(replacement);
    m_owner.setSelection(xStart, xStart+replacement.length(),
      m_searchUp);
    m_searchIndex = -1;
  }
  else
    m_owner.setSelection(xStart, xFinish, m_searchUp);
  return 1;
}

protected void warning(String message) {
  JOptionPane.showMessageDialog(m_owner,
    message, "Warning", JOptionPane.INFORMATION_MESSAGE);
```

```
  }
}

class Utils
{
  public static final char[] WORD_SEPARATORS = {' ', '\t', '\n',
    '\r', '\f', '.', ',', ':', '-', '(', ')', '[', ']', '{',
    '}', '<', '>', '/', '|', '\\', '\'', '\"'};

  public static boolean isSeparator(char ch) {
    for (int k=0; k<WORD_SEPARATORS.length; k++)
      if (ch == WORD_SEPARATORS[k])
        return true;
    return false;
  }
}
```

Understanding the Code

Class WordProcessor

This class now imports the `dl` package (constructed and discussed in chapter 4) to use our `DialogLayout` manager. `WordProcessor` also declares one new instance variable:

`FindDialog m_findDialog`: custom dialog for finding and replacing a selected region of text.

Two new menu items titled "`Find...`" and "`Replace...`," are added to the "`Edit`" menu. These items are activated with keyboard accelerators Ctrl+F and Ctrl+H respectively. When pressed, both items create an instance of `FindDialog` (if `m_findDialog` is `null`) or activate the existing instance, and the dialog is then displayed. The only difference between the two is that the "`Find...`" menu item activates the 0-indexed tabbed pane tab, and the "`Replace...`" menu item activates the tab at index 1.

Three new `public` methods have been added to this class to make access to our text pane component, and related objects, easier from external classes. The `getDocument()` method retrieves the text pane's current document instance, and the `getTextPane()` method retrieves the text pane itself. The `setSelection()` method selects a portion of text between given start and end positions, and positions the caret at the beginning or end of the selection, depending on the value of the `moveUp` boolean parameter. The coordinates of such a selection are then stored in the `m_xStart` and `m_xFinish` instance variables (recall that these variables always hold the coordinates of the current text selection and are used to restore this selection when our text pane regains the focus).

Class FindDialog

This class is a modal, `JDialog` sub-class encapsulating our find and replace functionality. It contains a tabbed pane with two tabs, "`Find`" and "`Replace`." Among unique controls, both also contain several common controls: checkboxs for "`Whole words only`" and "`Match case`," radio button for "`Search up`" and "`Search down`," and a text field for the text to "`Find`." What we need to is a set of controls which are common across both pages. To simplify this task we create two components and use the same model for each. This guarantees that consistency will be maintained for us, without the need for any maintenance or state checks.

`FindDialog` maintains the following instance variables:

`WordProcessor m_owner`: an explicit reference to our `WordProcessor` parent application frame.

`JTabbedPane m_tb`: the tabbed pane containing the find and replace pages.

`JTextField m_txtFind1`: used to enter the string to find.

`JTextField m_txtFind2`: used to enter the string to replace.

`Document m_docFind`: a shared data model for the "Find" text fields.

`Document m_docReplace`: a data model for the "Replace" text field.

`ButtonModel m_modelWord`: a shared data model for the "Whole words only" checkboxes.

`ButtonModel m_modelCase`: a shared data model for the "Match case" checkboxes.

`ButtonModel m_modelUp`: a shared data model for the "Search up" radio buttons.

`ButtonModel m_modelDown`: a shared data model for the "Search down" radio buttons.

`int m_searchIndex`: position in the document to start searching from.

`boolean m_searchUp`: a search direction flag.

`String m_searchData`: string to search for.

The `FindDialog` constructor creates a super-class non-modal dialog instance titled "Find and Replace." The main tabbed pane, `m_tb`, is created, and `JPanel p1` (the main container of the "Find" tab) receives the `m_txtFind1` text field along with a "Find what:" label. This text field is used to enter the target string to be searched for. Note that the `Document` instance associated with this textbox is stored in the `m_docFind` instance variable (which will be used to facilitate sharing between another text field).

Note: In a more sophisticated implementation one might use editable comboboxes with memory in place of text fields, similar to those discussed in the final examples of chapter 9.

Two checkboxes titled "Whole words only" and "Match case," and two radio buttons titled "Search up" and "Search down" (initially selected) are placed at the bottom of the `p1` panel. These components are surrounded by a titled "Options" border. Two `JButtons` titled "Find Next" and "Close" are placed at the right side of the panel. The first button calls our `findNext()` method (see below) when pressed. The second button hides the dialog. Finally the `p1` panel is added to `m_tb` with a tab title of "Find."

`JPanel p2` (the main container of the "Replace" tab) receives the `m_txtFind2` text field along with a "Find what:" label. It also receives another pair labeled "Replace:". An instance of our custom layout manager, `DialogLayout` (discussed in chapter 4), is used to lay out these text fields and corresponding labels without involving any intermediate containers. The same layout is used in the "Find" panel. We also synchronize the preferred size of the two panels to avoid movement of the mimicked components when a new page is activated.

Note that the `m_docFind` data object is set as the document for the `m_txtFind2` text field. This ensures consistency between the two different "Find" text fields in the two tabbed panels.

Two checkboxes and two radio buttons are placed at the bottom of the panel to control the replacement options. They have identical meaning and representation as the corresponding four controls in the "Find" panel, and to ensure consistency between them, the data models are shared between each 'identical' component.

Three `JButtons` titled "Replace," "Replace All," and "Close" are placed at the right side of the panel. The

"Replace" button makes a single call to our `findNext()` method (discussed below) when pressed. The "Replace All" button is associated with an `actionPerformed()` method which repeatedly invokes `findNext()` to perform replacement until it returns -1 to signal an error, or 0 to signal that no more replacements can be made. If an error occurs this method returns, the `actionPerformed()` method simply returns (since an error will be properly reported to the user by the `findNext()` method). Otherwise the number of replacements made is reported to the user in a `JOptionPane` message dialog. The "Close" button hides the dialog. Finally the `p2` panel is added to the `m_tb` tabbed pane with a tab title of "Replace."

Since this is a non-modal dialog, the user can freely switch to the main application frame and return back to the dialog while each remains visible (a typical find-and-replace feature). Once the user leaves the dialog he/she can modify the document's content, or move the caret position. To account for this, we add a `WindowListener` to the dialog whose `windowActivated()` method sets `m_searchIndex` to `-1`. This way, the next time `findNext()` is called (see below) the search data will be re-initialized, allowing the search to continue as expected, corresponding to the new caret position and document content.

The `setSelectedIndex()` method activates a page with the given index and makes this dialog visible. This method is intended mostly for use externally by our app when it wants to display this dialog with a specific tab selected.

The `findNext()` method is responsible for performing the actual find and replace operations. It takes two arguments:

`boolean doReplace`: if `true`, find and replace, otherwise just find.

`boolean showWarnings`: if `true`, display a message dialog if target text cannot be found, otherwise do not display a message.

`findNext()` returns an `int` result with the following meaning:

`-1`: an error has occurred.

`0`: the target text cannot be found.

`1`: a find or find and replace was completed successfully.

The `m_searchIndex == -1` condition specified that the text to be searched through must be re-calculated. In this case we store the portion of text from the beginning of the document to the current caret position if we are searching up, or between the current caret position and the end of the document if we are searching down. This text is stored in the `m_searchData` instance variable. The current caret position is stored in the `m_searchIndex` variable.

Note: This solution may not be adequate for large documents. However, a more sophisticated solution would take us too far from the primary goal of this example.

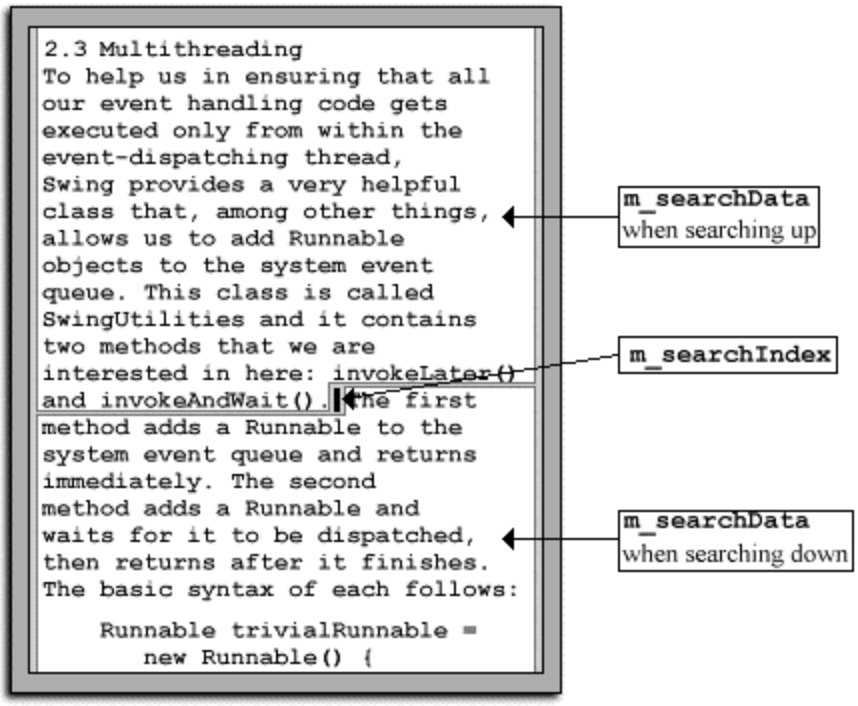


Figure 20.11 Usage of instance variables for searching up and down through document text.
<<file figure20-11.gif>>

The text to search for is retrieved from the `m_docFind` shared `Document`. If the case-insensitive option is selected, both the `m_searchData` text and the text to search for are converted into lower case. If the "Whole words only" option is selected, we check whether the text to search for contains any separator characters defined in our `Util` utilities class (see below).

Note: If a given `String` is already completely in lower or upper case, the `toLowerCase()` (or `toUpperCase()`) method returns the original `String` without creating a new object.

After this, if the `doReplace` parameter is `true`, we retrieve the replacement text from our `m_docReplace` `Document`. At this point we're ready to actually perform a search. We take advantage of existing `String` functionality to accomplish this:

```
if (m_searchUp)
  xStart = m_searchData.lastIndexOf(key, pos-1);
else
  xStart = m_searchData.indexOf(key, pos-m_searchIndex);
```

If we are seaching up, we search for the last occurrence of the target string from the current caret position. Otherwise we search for the first occurrence of the target string from the current caret position. If the target string is not found, we cannot continue the search, and a warning is displayed if the `showWarnings` parameter is `true`.

This simple scheme is complicated considerably if the "Whole words only" option is selected. In this case we need to verify whether symbols on the left and on the right of a matching region pf text are either word separators defined in our `Utils` class, or the string lies at the end of the data being searched. If these conditions are not satisfied, we attempt to continue searching, unless the end of the search data is reached.

In any case, if we locate an acceptable match, we select the located text. If the replace option is selected, we replace this selected region with the specified replacement text and then select the new replacement text. In this latter case we also set `m_searchIndex` to `-1` to force the `m_searchData` variable to be updated. This is necessary for continued searching because the data being searched most likely changes after each replace. The location of the caret also usually changes.

Class Utils

This class provides a simple static utility method and an array of chars representing word separator characters. The `isSeparator()` method simply checks whether a given character belongs to the static `WORD_SEPARATORS` char array.

Running the Code

Open an existing RTF file and use the "Edit" menu, or the appropriate keyboard accelerator, to bring up the "Find and Replace" dialog with the "Find" tab selected. Enter some text to search for, select some search options, and press the "Find Next" button. If your target text is found, the matching region will be highlighted in the base document. Press this button again to find subsequent entries (if any). Verify that the "Whole words only" and "Match case" options function as discussed above. Change focus to the main application window and modify the document and/or change the caret position. Return to the "Find and Replace" dialog and note that the search continues as expected.

Select the "Replace" tab and verify that the state of all search options, including the search target string, are preserved from the "Find" tab (and vice versa when switching back). Enter a replacement string and verify that the "Replace" and "Replace All" buttons work as expected.

20.9 Word Processor: part IX - Spellchecker [using JDBC and SQL]

All modern word processor applications worth mentioning offer tools and utilities which help the user in finding grammatical and spelling mistakes in a document. In this section we will add spell-checking to our word processor application. To do this we will need to perform some of our own multithreading, and communicate with JDBC. We will use a simple database with one table, `Data`, which has the following structure:

Name	Type	Description
word	String	A single English word
soundex	String	A 4-letter SOUNDEX code

An example of this database, populated with words from several Shakespeare comedies and tragedies, is provided in this example's directory: `Shakespeare.mdb`. (This database must be a registered database in your database manager prior to using it. This is not a JDBC tutorial, so we'll skip the details.)

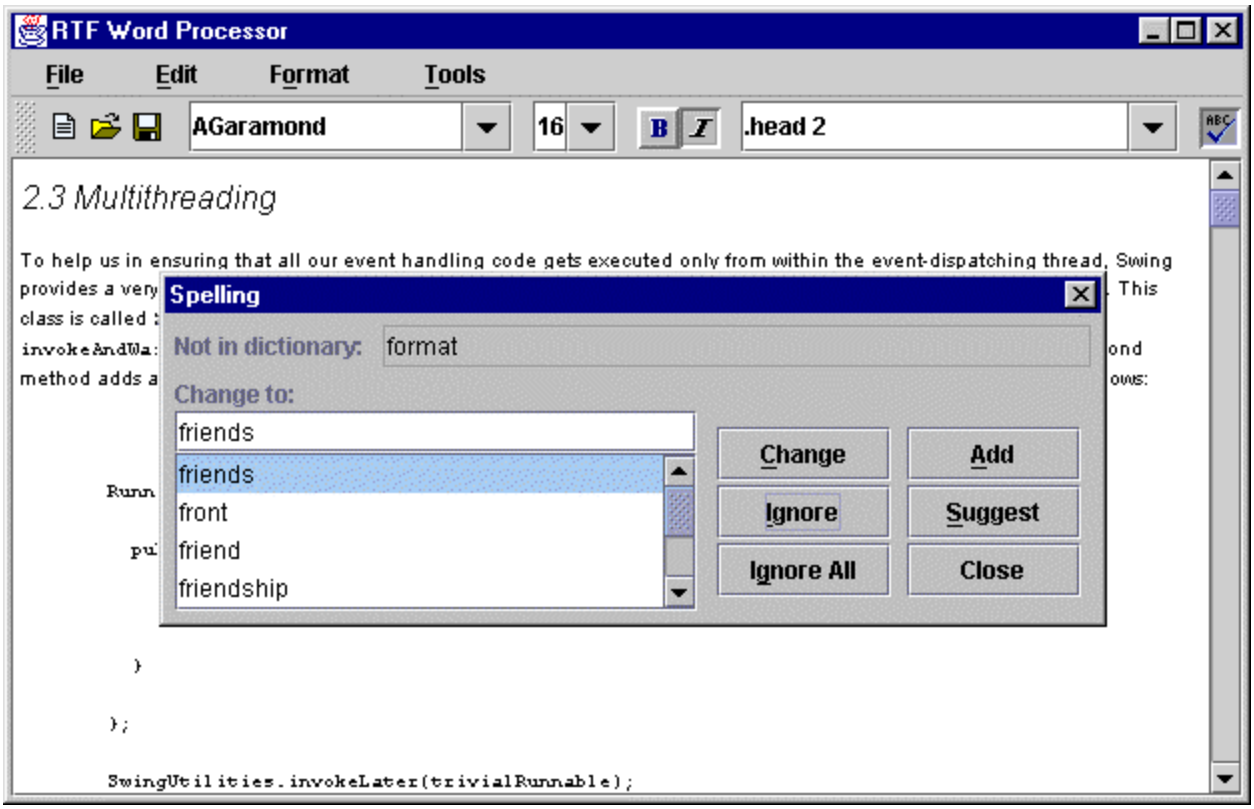


Figure 20.12 WordProcessor with spellchecking functionality.

<<file figure20-12.gif>>

Note: The custom SOUNDEX algorithm used in this example hashes words for efficiency by using a simple model which approximates the sound of the word when spoken. Each word is reduced to a four character string, the first character being an upper case letter and the remaining three being digits.

The Code: WordProcessor.java

see \Chapter20\9

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.sql.*;

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;
import javax.swing.undo.*;

import dl.*;

public class WordProcessor extends JFrame
{
  // Unchanged code from section 20.8

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 20.8
```

```
    JMenu mTools = new JMenu("Tools");
    mTools.setMnemonic('t');

    Action spellAction = new AbstractAction("Spelling...",
     new ImageIcon("tools_abc.gif"))
    {
      public void actionPerformed(ActionEvent e) {
        SpellChecker checker = new SpellChecker(WordProcessor.this);
        WordProcessor.this.setCursor(Cursor.getPredefinedCursor(
          Cursor.WAIT_CURSOR));
        checker.start();
      }
    };
    item =  mTools.add(spellAction);
    item.setMnemonic('s');
    item.setAccelerator(KeyStroke.getKeyStroke(
      KeyEvent.VK_F7, 0));
    menuBar.add(mTools);

    m_toolBar.addSeparator();
    m_toolBar.add(new SmallButton(spellAction,
      "Spell checker"));

    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    return menuBar;
  }

  // Unchanged code from section 20.8
}

class OpenList extends JPanel
 implements ListSelectionListener, ActionListener
{
  // Unchanged code from section 20.6

  public OpenList(String title, int numCols) {
    setLayout(null);
    m_title = new JLabel(title, JLabel.LEFT);
    add(m_title);
    m_text = new JTextField(numCols);
    m_text.addActionListener(this);
    add(m_text);
    m_list = new JList();
    m_list.setVisibleRowCount(4);
    m_list.addListSelectionListener(this);
    m_scroll = new JScrollPane(m_list);
    add(m_scroll);
  }

  public void appendResultSet(ResultSet results, int index,
   boolean toTitleCase)
  {
    m_text.setText("");
    DefaultListModel model = new DefaultListModel();
    try {
      while (results.next()) {
        String str = results.getString(index);
        if (toTitleCase)
          str = Utils.titleCase(str);
        model.addElement(str);
```

```
      }
    }
    catch (SQLException ex) {
      System.err.println("appendResultSet: "+ex.toString());
    }
    m_list.setModel(model);
    if (model.getSize() > 0)
      m_list.setSelectedIndex(0);
  }

  // Unchanged code from section 20.6
}

// Unchanged code from section 20.6

class SpellChecker extends Thread
{
  protected static String SELECT_QUERY =
    "SELECT Data.word FROM Data WHERE Data.word = ";
  protected static String SOUNDEX_QUERY =
    "SELECT Data.word FROM Data WHERE Data.soundex = ";

  protected WordProcessor m_owner;
  protected Connection m_conn;
  protected DocumentTokenizer m_tokenizer;
  protected Hashtable  m_ignoreAll;
  protected SpellingDialog m_dlg;

  public SpellChecker(WordProcessor owner) {
    m_owner = owner;
  }

  public void run() {
    JTextPane monitor = m_owner.getTextPane();
    m_owner.setEnabled(false);
    monitor.setEnabled(false);

    m_dlg = new SpellingDialog(m_owner);
    m_ignoreAll = new Hashtable();

    try {
      // Load the JDBC-ODBC bridge driver
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      m_conn = DriverManager.getConnection(
        "jdbc:odbc:Shakespeare", "admin", "");
      Statement selStmt = m_conn.createStatement();

      Document doc = m_owner.getDocument();
      int pos = monitor.getCaretPosition();
      m_tokenizer = new DocumentTokenizer(doc, pos);
      String word, wordLowCase;

      while (m_tokenizer.hasMoreTokens()) {
        word = m_tokenizer.nextToken();
        if (word.equals(word.toUpperCase()))
          continue;
        if (word.length()<=1)
          continue;
        if (Utils.hasDigits(word))
          continue;
        wordLowCase = word.toLowerCase();
        if (m_ignoreAll.get(wordLowCase) != null)
```

```
        continue;

      ResultSet results = selStmt.executeQuery(
        SELECT_QUERY+"'"+wordLowCase+"'");
      if (results.next())
        continue;

      results = selStmt.executeQuery(SOUNDEX_QUERY+
        "'"+Utils.soundex(wordLowCase)+"'");
      m_owner.setSelection(m_tokenizer.getStartPos(),
        m_tokenizer.getEndPos(), false);
      if (!m_dlg.suggest(word, results))
        break;
    }

    m_conn.close();
    System.gc();
    monitor.setCaretPosition(pos);
  }
  catch (Exception ex) {
    ex.printStackTrace();
    System.err.println("SpellChecker error: "+ex.toString());
  }

  monitor.setEnabled(true);
  m_owner.setEnabled(true);
  m_owner.setCursor(Cursor.getPredefinedCursor(
    Cursor.DEFAULT_CURSOR));
}

protected void replaceSelection(String replacement) {
  int xStart = m_tokenizer.getStartPos();
  int xFinish = m_tokenizer.getEndPos();
  m_owner.setSelection(xStart, xFinish, false);
  m_owner.getTextPane().replaceSelection(replacement);
  xFinish = xStart+replacement.length();
  m_owner.setSelection(xStart, xFinish, false);
  m_tokenizer.setPosition(xFinish);
}

protected void addToDB(String word) {
  String sdx = Utils.soundex(word);
  try {
    Statement stmt = m_conn.createStatement();
    stmt.executeUpdate(
      "INSERT INTO DATA (Word, Soundex) VALUES ('"+
      word+"', '"+sdx+"')");
  }
  catch (Exception ex) {
    ex.printStackTrace();
    System.err.println("SpellChecker error: "+ex.toString());
  }
}

class SpellingDialog extends JDialog
{
  protected JTextField  m_txtNotFound;
  protected OpenList    m_suggestions;

  protected String   m_word;
  protected boolean  m_continue;
```

```
public SpellingDialog(WordProcessor owner) {
  super(owner, "Spelling", true);

  JPanel p = new JPanel();
  p.setBorder(new EmptyBorder(5, 5, 5, 5));
  p.setLayout(new BoxLayout(p, BoxLayout.X_AXIS));
  p.add(new JLabel("Not in dictionary:"));
  p.add(Box.createHorizontalStrut(10));
  m_txtNotFound = new JTextField();
  m_txtNotFound.setEditable(false);
  p.add(m_txtNotFound);
  getContentPane().add(p, BorderLayout.NORTH);

  m_suggestions = new OpenList("Change to:", 12);
  m_suggestions.setBorder(new EmptyBorder(0, 5, 5, 5));
  getContentPane().add(m_suggestions, BorderLayout.CENTER);

  JPanel p1 = new JPanel();
  p1.setBorder(new EmptyBorder(20, 0, 5, 5));
  p1.setLayout(new FlowLayout());
  p = new JPanel(new GridLayout(3, 2, 8, 2));

  JButton bt = new JButton("Change");
  ActionListener lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      replaceSelection(m_suggestions.getSelected());
      m_continue = true;
      setVisible(false);
    }
  };
  bt.addActionListener(lst);
  bt.setMnemonic('c');
  p.add(bt);

  bt = new JButton("Add");
  lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      addToDB(m_word.toLowerCase());
      m_continue = true;
      setVisible(false);
    }
  };
  bt.addActionListener(lst);
  bt.setMnemonic('a');
  p.add(bt);

  bt = new JButton("Ignore");
  lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      m_continue = true;
      setVisible(false);
    }
  };
  bt.addActionListener(lst);
  bt.setMnemonic('i');
  p.add(bt);

  bt = new JButton("Suggest");
  lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      try {
        m_word = m_suggestions.getSelected();
```

```
          Statement selStmt = m_conn.createStatement();
          ResultSet results = selStmt.executeQuery(
            SELECT_QUERY+"'"+m_word.toLowerCase()+"'");
          boolean toTitleCase = Character.isUpperCase(
            m_word.charAt(0));
          m_suggestions.appendResultSet(results, 1,
            toTitleCase);
        }
        catch (Exception ex) {
          ex.printStackTrace();
          System.err.println("SpellChecker error: "+
            ex.toString());
        }
      }
    };
    bt.addActionListener(lst);
    bt.setMnemonic('s');
    p.add(bt);

    bt = new JButton("Ignore All");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_ignoreAll.put(m_word.toLowerCase(), m_word);
        m_continue = true;
        setVisible(false);
      }
    };
    bt.addActionListener(lst);
    bt.setMnemonic('g');
    p.add(bt);

    bt = new JButton("Close");
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        m_continue = false;
        setVisible(false);
      }
    };
    bt.addActionListener(lst);
    bt.setDefaultCapable(true);
    p.add(bt);
    p1.add(p);
    getContentPane().add(p1, BorderLayout.EAST);

    pack();
    setResizable(false);
    Dimension d1 = getSize();
    Dimension d2 = owner.getSize();
    int x = Math.max((d2.width-d1.width)/2, 0);
    int y = Math.max((d2.height-d1.height)/2, 0);
    setBounds(x + owner.getX(),
      y + owner.getY(), d1.width, d1.height);
  }

  public boolean suggest(String word, ResultSet results) {
    m_continue = false;
    m_word = word;
    m_txtNotFound.setText(word);
    boolean toTitleCase = Character.isUpperCase(
      word.charAt(0));
    m_suggestions.appendResultSet(results, 1, toTitleCase);
    show();
```

```
      return m_continue;
    }
  }
}

class DocumentTokenizer
{
  protected Document m_doc;
  protected Segment  m_seg;
  protected int m_startPos;
  protected int m_endPos;
  protected int m_currentPos;

  public DocumentTokenizer(Document doc, int offset) {
    m_doc = doc;
    m_seg = new Segment();
    setPosition(offset);
  }

  public boolean hasMoreTokens() {
    return (m_currentPos < m_doc.getLength());
  }

  public String nextToken() {
    StringBuffer s = new StringBuffer();
    try {
      // Trim leading seperators
      while (hasMoreTokens()) {
        m_doc.getText(m_currentPos, 1, m_seg);
        char ch = m_seg.array[m_seg.offset];
        if (!Utils.isSeparator(ch)) {
          m_startPos = m_currentPos;
          break;
        }
        m_currentPos++;
      }

      // Append characters
      while (hasMoreTokens()) {
        m_doc.getText(m_currentPos, 1, m_seg);
        char ch = m_seg.array[m_seg.offset];
        if (Utils.isSeparator(ch)) {
          m_endPos = m_currentPos;
          break;
        }
        s.append(ch);
        m_currentPos++;
      }
    }
    catch (BadLocationException ex) {
      System.err.println("nextToken: "+ex.toString());
      m_currentPos = m_doc.getLength();
    }
    return s.toString();
  }

  public int getStartPos() { return m_startPos; }

  public int getEndPos() { return m_endPos; }

  public void setPosition(int pos) {
    m_startPos = pos;
```

```
    m_endPos = pos;
    m_currentPos = pos;
  }
}

class Utils
{
  // Unchanged code from section 20.8

  public static String soundex(String word) {
    char[] result = new char[4];
    result[0] = word.charAt(0);
    result[1] = result[2] = result[3] = '0';
    int index = 1;

    char codeLast = '*';
    for (int k=1; k<word.length(); k++) {
      char ch = word.charAt(k);
      char code = ' ';
      switch (ch) {
        case 'b': case 'f': case 'p': case 'v':
          code = '1';
          break;
        case 'c': case 'g': case 'j': case 'k':
        case 'q': case 's': case 'x': case 'z':
          code = '2';
          break;
        case 'd': case 't':
          code = '3';
          break;
        case 'l':
          code = '4';
          break;
        case 'm': case 'n':
          code = '5';
          break;
        case 'r':
          code = '6';
          break;
        default:
          code = '*';
          break;
      }
      if (code == codeLast)
        code = '*';
      codeLast = code;
      if (code != '*') {
        result[index] = code;
        index++;
        if (index > 3)
          break;
      }
    }
    return new String(result);
  }

  public static boolean hasDigits(String word) {
    for (int k=1; k<word.length(); k++) {
      char ch = word.charAt(k);
      if (Character.isDigit(ch))
        return true;
    }
```

```
    return false;
  }

  public static String titleCase(String source) {
    return Character.toUpperCase(source.charAt(0)) +
      source.substring(1);
  }
}
```

Understanding the Code

Class WordProcessor

This class now imports the `java.sql` package to make use of JDBC functionality. The `createMenuBar()` method now creates a new menu titled "Tools," which contains one menu item titled "Spelling..." This menu item can be invoked with keyboard accelerator F7, or by pressing the coresponding toolbar button. When selected, this new menu item creates and starts the `SpellChecker` thread (see below), passing a reference to the main application frame as parameter.

Class OpenList

This custom component receives new functionality for use in our new spell checker dialog. First, we add a new constructor which assigns a given number of columns to the text field, and does not initialize the list component.

Second, we add the `appendResultSet()` method which populates the list component with the data supplied in the given `ResultSet` instance at the given position. If the third parameter is set to `true`, this tells the method to convert all string data to the 'title case' (which means that the first letter is in upper case, and the rest of the string is unchanged). This is accomplished through use of our new `titleCase()` method in our `Utils` class (see below).

Class SpellChecker

This class extends `Thread` to perform spell checking of the current document from the current caret position moving downward. Two class variables are declared (their use will become more clear below):

`String SELECT_QUERY`: SQL query text used to select a word equal to a given string.

`String SOUNDEX_QUERY`: SQL query text used to select a word matching a given SOUNDEX value.

Five instance variables are declared:

`WordProcessor m_owner`: a reference to the main application frame.

`Connection m_conn`: JDBC connection to a database.

`DocumentTokenizer m_tokenizer`: a custom object used to retrieve each word in a document.

`Hashtable m_ignoreAll`: a collection of words to ignore in a search, added to with the "Ignore All" button.

`SpellingDialog m_dlg`: our custom dialog used for processing spelling mistakes.

The `SpellChecker` constructor takes a reference to the application's frame as a parameter and stores it in the `m_owner` instance variable.

The `run()` method is responsible for the most significant activity of this thread. To prevent the user from modifying the document during spell checking we first disable the main application frame and our text pane contained within it.

Note: Unlike AWT, Swing containers do not disable their child components when they themselves are disabled. It is not clear whether this is a bug or an intended feature.

Then we create a new `SpellingDialog` instance to provide the user interface, and instantiate the `m_ignoreAll` collection. In a try/catch block we process all JDBC interactions to allow proper handling of any potential errors. This code creates a JDBC connection to our Shakespeare database, retrieves the current caret position, and creates an instance of `DocumentTokenizer` to parse the document from the current caret position. In a while loop we perform spell checking on each word fetched until there are no more tokens. Words in all upper case, containing only one letter, or containing digits, are skipped (this behavior can easily be customized). Then we convert the word under examination to lower case and search for it in the `m_ignoreAll` collection. If it is not found, we try to find it in the database. If the SQL query does not return any results, we try to locate a similar word in the database with the same SOUNDEX value to suggest to the user in the dialog. The word in question is then selected in our text pane to show the user which word is currently under examination. Finally we call our `SpellingDialog`'s `suggest()` method to request that the user make a decision about what to do with this word. If the `suggest()` method returns `false`, the user has chosen to terminate the spell checking process, so we exit the loop. Once outside the loop we close the JDBC connection, restore the original caret position, explicitly call the garbage collector, and reenable the main application frame and our text pane editor contained within it.

The following two methods are invoked by the `SpellingDialog` instance associated with this `SpellChecker`:

`replaceSelection()` is used to replace the most recently parsed word by the `DocumentTokenizer` instance, with the given replacement string.

`addToDB()` adds a given word, and it's SOUNDEX value, to the database by executing an insert query.

Class SpellChecker.SpellingDialog

This inner class represents a dialog which prompts the user to verify or correct a certain word if it is not found in the database. The user can select one of several actions in response: ignore the given word, ignore all occurrences of that word in the document, replace that word with another word, add this word to the database and consider it correct in any future matches, or cancel the spell check. Four instance variables are declared:

`JTextField m_txtNotFound`: used to display the word under investigation.

`OpenList m_suggestions`: editable list component to select or enter a replacement word.

`String m_word`: the word under investigation.

`boolean m_continue`: a flag indicating that spell checking should continue.

The `SpellingDialog` constructor places the `m_txtNotFound` component and corresponding label at the top of the dialog window. The `m_suggestions` `OpenList` is placed in the center, and six buttons discussed below, are grouped to the right.

The button titled "Change" replaces the word under investigation with the word currently selected in the list or

entered by the user. Then it stores `true` in the `m_continue` flag and hides the dialog window. This terminates the modal state of the dialog and makes the `show()` method return, which in turn allows the program's execution to continue (recall that modal dialogs block the calling thread until they are dismissed).

The button titled "Add" adds the word in question to the spelling database. This word will then be considered correct in future queries. In this way we allow the spell checker to "learn" new words (i.e. add them to the dictionary).

The button titled "Suggest" populates the `m_suggestions` list with all `SOUNDEX` matches to the word under investigation. This button is intended for use in situations where the user is not satisfied with the initial suggestions.

The button titled "Ignore" simply skips the current word and continues spell checking the remaining text.

The button titled "Ignore All" does the same as the "Ignore" button, but also stores the word in question in the collection of words to ignore, so the next time the spell checker finds this word it will be deemed correct. The difference between "Ignore All" and "Add" is that ignored words will only be ignored during a single spell check, whereas words added to the database will persist as long as the database data does.

The button titled "Close" stores `false` in the `m_continue` flag and hides the dialog window. This results in the termination of the spell checking process (see the `suggest()` method).

The `suggest()` method is used to display this `SpellingDialog` each time a questionable word is located during the spell checking process. It takes a `String` and a `ResultSet` containing suggested substitutions as parameters. It sets the text of the `m_txtNotFound` component to the `String` passed in, and calls `appendResultSet()` on the `OpenList` to display an array of suggested corrections. Note that the first character of these suggestions will be converted to upper case if the word in question starts with a capital letter. Finally, the `show()` method displays this dialog in the modal state. As soon as this state is terminated by one of the push buttons, or by directly closing the dialog, the `suggest()` method returns the `m_continue` flag. If this flag is set to `false`, this indicates that the calling program should terminate the spell checking cycle.

Class DocumentTokenizer

This helper class was built to parse the current text pane document. Unfortunately we cannot use the standard `StreamTokenizer` class for this purpose, because it provides no way of querying the position of a token within the document (we need this information to allow word replacement). Several instance variables are declared:

- `Document m_doc`: a reference to the document to be parsed.
- `Segment m_seg`: used for quick delivery of characters from the document being parsed.
- `int m_startPos`: the start position of the current word from the beginning of the document.
- `int m_endPos`: the end position of the current word from the beginning of the document.
- `int m_currentPos`: the current position of the parser from the beginning of the document.

The `DocumentTokenizer` constructor takes a reference to the document to be parsed and the offset to start at as parameters. It initializes the instance variables described above.

The `hasMoreTokens()` method returns `true` if the current parsing position lies within the document.

The `nextToken()` method extracts the next token (a group of characters separated by one or more characters defined in the `WORD_SEPARATORS` array from our `Utils` class) and returns it as a `String`. The positions of the beginning and the end of the token are stored in the `m_startPos` and `m_endPos` instance variables respectively. To access a portion of document text with the least possible overhead we use the `Document.getText()` method which takes three parameters: offset from the beginning of the document, length of the text fragment, and a reference to an instance of the `Segment` class (recall from chapter 19 that the `Segment` class provides an efficient means of directly accessing an array of document characters).

We look at each character in turn, passing over separator characters until the first non-separator character is reached. This position is marked as the beginning of a new word. Then a `StringBuffer` is used to accumulate characters until a separator character, or the end of document, is reached. The resulting characters are returned as a `String`.

Note: This variant of the `getText()` method gives us direct access to the characters contained in the document through a `Segment` instance. These characters should not be modified.

Class Utils

Three new static methods are added to this class. The `soundex()` method calculates and returns the SOUNDEX code of the given word. To calculate that code we use the first character of the given word and add a three-digit code that represents the first three remaining consonants. The conversion is made according to the following table:

Code	Letters
1	B, P, F, V
2	C, S, G, J, K, Q, X, Z
3	D, T
4	L
5	M, N
6	R
*	(all others)

The `hasDigits()` method returns `true` if a given string contains digits, and the `titleCase()` method converts the first character of a given string to upper case.

Running the Code

Open an existing RTF file and try running a complete spell check. Try adding some words to the dictionary and use the "Ignore All" button to avoid questioning a word again during that spell check. Try using the "Suggest" button to query the database for more suggestions based on our SOUNDEX algorithm. Click the "Change" button to accept a suggestion or a change typed into the text field. Click the "Ignore" button to ignore the current word being questioned.

Note: The Shakespeare vocabulary database supplied for this example is neither complete nor contemporary. It does not include such words as "software" or "Internet." However, you can easily add them, when encountered during a spell check, by clicking the "Add" button.

Chapter 21. Pluggable Look & Feel

In this chapter:

- Pluggable Look & Feel overview
- Custom L&F: part I - Using custom resources
- Custom L&F: part II - Creating custom UI delegates
- L&F for custom components: part I - Implementing L&F support
- L&F for custom components: part II - Third party L&F support

21.1 Pluggable Look & Feel overview

The pluggable look-and-feel architecture is one of Swing's greatest milestones. It allows seamless changes in the appearance of an application and the way an application interacts with the user. This can occur without modification or recompilation of the application, and can be invoked programmatically during any single JVM session. In this chapter we'll discuss how L&F works, how custom L&F can be implemented for standard Swing components, and how L&F support (both existing and custom) can be added to custom components.

Note: In chapter 1 we introduced the basic concepts behind L&F and UI delegates, and it may be helpful to review this material before moving on.

In examining Swing component source code, you will quickly notice that these classes do not contain any code for sophisticated rendering. All this drawing code is stored somewhere else. As we learned in chapter 1, this code is defined within various UI delegates, which act as both a component's view and controller. (In chapter 6 we learned how to customize a tabbed pane and its UI delegate. This was only a small taste of the flexibility offered by pluggable look-and-feel that we will be discussing here.) Before jumping into the examples we need to discuss how the most significant L&F-related classes and interfaces function and interact in more detail.

21.1.1 LookAndFeel

abstract class javax.swing.LookAndFeel

This abstract class serves as the superclass of the central class of any pluggable look-and-feel implementation. The `getDefaults()` method returns an instance of `UIDefaults` (see 21.1.2). The `getDescription()` method returns a one-to-two sentence description of the L&F. `getID()`[1] returns a simple, unique string that identifies an L&F. `getName()` returns a short string representing the name of that L&F, such as "Malachite," or "Windows."

The `isNativeLookAndFeel()` method querie's the `System` class to determine whether the given `LookAndFeel` corresponds to that which emulates the operating system platform the running VM is designed for. The `isSupportedLookAndFeel()` method is used to determine whether the given `LookAndFeel` is supported by the operating system the running VM is designed for. Due to legal issues, some `LookAndFeels` will not be supported by certain operating systems, even though they have the ability to function perfectly fine.

---

[1] This method is actually not used by Swing, but as a rule it is a good idea to provide `LookAndFeel` implementations with a unique identifier.

Note: We will not go into the details of how to work around this imposed limitation (although it is relatively easy), specifically because of the legal issues involved.

The `initialize()` and `uninitialize()` methods are called when a `LookAndFeel` is installed and uninstalled, respectively. The `toString()` method is defined to return the description returned by `getDescription()`, as well as the fully qualified class name.

Several convenient static methods are also provided for assigning and unassigning borders, colors, and fonts to components: `installBorder()`, `installColors()`, `installColorsAndFont()`, and `uninstallBorder()`. `LookAndFeel` implements these so that the specified properties only change if the current property value of the given component is a `UIResource` (see 21.1.4) or `null`. Static methods `makeKeyBindings()` and `makeIcon()` are convenience methods for building a list of text component key bindings, and creating a `UIDefaults.LazyValue` (see 21.1.2) which can create an `ImageIcon` `UIResource`.

21.1.2 UIDefaults

class javax.swing.UIDefaults

This class extends `Hashtable` and manages a collection of custom resources (objects, primitives, etc) used in this L&F. Methods `put(Object key, Object value)` and `putDefaults(Object[] keyValueList)` can be used to store data (in the latter case they must be placed in a one-dimensional array in the order: key1, value1, key2, value2, etc.). Method `get(Object key)` can be used to retrieve a stored resource.

UIDefaults also defines two inner classes: `LazyValue` and `ActiveValue`. A `LazyValue` is an entry in the UIDefaults hashtable that is not instantiated until it is looked up with its associated key name. Large objects that take a long time to instantiate, and are rarely used, can benefit from being implemented as `LazyValues`. An `ActiveValue` is one that is instantiated each time it is looked up with its associated key name. Those resources that must be unique in each place they are used are often implemented as `ActiveValues`.

Both interfaces require the definition of the `createValue()` method. The following code shows a simple `LazyValue` that constructs a new border.

```
Object myBorderLazyValue = new UIDefaults.LazyValue() {
  public Object createValue(UIDefaults table) {
    return new BorderFactory.createLoweredBevelBorder();
  }
};
myUIDefaults.put("MyBorder", borderLazyValue);
```

Note that the `createValue()` method will only be called once for `LazyValues`, whereas with `ActiveValues` it will be called each time that resource is requested.

21.1.3 UIManager

public class javax.swing.UIManager

This class provides a set of static methods used to manage the current look-and-feel. The current look-and-feel is actually made up of a three-level `UIDefaults` hierarchy: user defaults, current look-and-feel defaults, and system defaults. Particularly important methods are `getUI(JComponent target)`, which retrieves an instance of `ComponentUI` for the specified component, and `getDefaults()`, which retrieves a shared instance of the `UIDefaults` class.

21.1.4 The UIResource interface

abstract interface javax.swing.plaf.UIResource

This interface has no implementation and is used solely to mark resource objects created for a component's UI delegate. Several classes used to wrap component UI resources implement this interface. For example: `InsetsUIResource`, `FontUIResource`, `IconUIResource`, `BorderUIResource`, and `ColorUIResource`. These wrapper classes are used for assigning resources that will be relinquished when a component's UI delegate is changed. In other words, if we were to assign an instance of `JLabel` a background of `Color.Yellow`, this background setting would persist even through a UI delegate change. However, if we were to assign that `JLabel` a background of `new ColorUIResource(Color.Yellow)`, the background would only persist until another UI delegate is installed. When the next UI delegate is installed, the label will receive a new label background based on the L&F the new UI delegate belongs to.

21.1.5 ComponentUI

abstract class javax.swing.plaf.ComponentUI

This abstract class represents a common superclass of all component UI delegate classes implemented by each different L&F packages. The `createUI(JComponent c)` static method creates an instance of `ComponentUI` for a given component. See chapter 1, section 1.4.1, for a description of each `ComponentUI` method.

Abstract classes in the `javax.swing.plaf` package extend `ComponentUI` to represent the base class from which each Swing component's UI should extend: `ButtonUI`, `TreeUI`, etc. Each of these classes have a concrete default implementation in the `javax.swing.plaf.basic` package: `BasicButtonUI`, `BasicTreeUI`, etc. In turn, these basic UI classes can be, and are intended to be, extended by other L&F implementations. For example, the classes mentioned above are extended by `MetalButtonUI` and `MetalTreeUI` defined in the `javax.swing.plaf.metal` package.

21.1.6 BasicLookAndFeel

class javax.swing.plaf.basic.BasicLookAndFeel

This class provides the basic implementation of `javax.swing.LookAndFeel`. It creates all resources used by UI classes defined in the `basic` package. Custom L&F classes are expected to extend this class, rather than `LookAndFeel` directly, to replace only those resources that need to be customized.

Note: Though we will not go into the details of each basic UI delegate implementation in this book (indeed this is a large topic and deserves a whole volume unto itself), note that the basic package contains a class called BasicGraphicsUtils which consists of several static methods used for drawing various types of rectangles most commonly used for borders. The basic package also contains several other quite useful utility-like classes, and a quick browse through the basic package API docs will reveal some of these interesting members.

21.1.7 How L&F works

Now we'll discuss how the pluggable look-and-feel mechanism works and what actually happens when a Swing component is created, painted, and when the user changes the application's L&F during a Java session.

All Swing component constructors call the `updateUI()` method inherited from `JComponent`. This method also may be called with the `SwingUtilities.updateComponentTreeUI()` helper method. The latter method recursively updates the UI delegate of each child of the specified component (we've already seen how this is used in chapters 1 and 16).

The `updateUI()` method overridden by most Swing components typically has an implementation similar to the following:

```
setUI((MenuUI)UIManager.getUI(this));
```

This invokes the static `UIManager.getUI()` method and passes a `this` component reference as parameter. This method, in turn, triggers a call to `getUI()` on the shared `UIDefaults` instance retrieved with the `getDefaults()` method.

The `UIDefaults.getUI()` method actually creates the `ComponentUI` object for a given `JComponent`. First it calls `getUIClassID()` on that component to discover the unique string ID associated with that class. For example, the `JTree.getUIClassID()` call returns the string: "TreeUI."

Prior to the process described above, the `UIDefaults` instance (which extends `Hashtable`) is initialized by the sub-class of `LookAndFeel` which is currently in charge. For instance, the Java look-and-feel (also referred to as "Metal") is defined by `javax.swing.plaf.metal.MetalLookAndFeel`. This class fills that look-and-feel's shared `UIDefaults` instance with key-value pairs. For each component which has a corresponding UI delegate implementation in the current L&F, a component ID String and a fully qualified UI delegate class name is added as a key/value pair to `UIDefaults`. For instance, the "TreeUI" ID key corresponds to the "javax.swing.plaf.metal.MetalTreeUI" value in `MetalLookAndFeel`'s L&F `UIDefaults`. If a particular `LookAndFeel` implementation does not specify a UI delegate for some component, a value from the parent `javax.swing.plaf.BasicLookAndFeel` class is used.

Using these key/value pairs, the `UIDefaults.getUI()` method determines the fully qualified class name and calls the `createUI()` method on that class using the Java reflection API. This static method returns an instance of the proper UI delegate, for instance, `MetalTreeUI`.

Back to the `updateUI()` method, the retrieved `ComponentUI` object is passed to the `setUI()` method and stored into `protected` variable, `ComponentUI ui`, inherited from the `JComponent` base class. This completes the creation of a UI delegate.

Recall that UI delegates are normally in charge of performing the associated component's rendering, as well as processing user input directed to that component. The `update()` method of a UI delegate is normally responsible for painting a component's background, if it is opaque, and then calling `paint()`. A UI delegate's `paint()` method is what actually paints a component's content, and is the method we most often override when building our own delegates.

Now let's review this process from a higher level perspective:

1. The currently installed look-and-feel provides an application with information about UI delegates to be used for all Swing components instantiated in that application.

2. Using this information, an instance of a UI delegate class can be instantiated on demand for a given component.

3. This UI delegate is passed to the component and generally takes responsibility for providing the complete user interface (view and controller).

4. The UI delegate can be easily replaced with another one at run-time without affecting the underlying component or its data (e.g. its model).

21.1.8 Selecting an L&F

The Swing API shipped with Java 2 includes three standard L&Fs: Metal, Motif, and Windows (available only for Microsoft Windows users). The first one is not associated with any existing platform and is considered the

"Java L&F." This look-and-feel is the default, and will be used automatically unless we directly change L&Fs in our app.

Reference: Apple provides the MacOS look-and-feel available for download at http://www.apple.com/macos/java/text/download.html.

Note: Swing also provides a Multiplexing look-and-feel which allows more than one UI delegate to be associated with a component at the same time. This L&F is intended for, but not limited to, use with accessible technologies.

To select a particular L&F we call the UIManager.setLookAndFeel() method and specify the fully qualified class name of a subclass of javax.swing.LookAndFeel which defines the desired L&F. The following code shows how to force an application to use the Motif L&F:

```
try {
  UIManager.setLookAndFeel(
    "com.sun.java.swing.plaf.motif.MotifLookAndFeel");
}
catch (Exception e) {
  System.out.println ("Couldn't load Motif L&F " + e);
}
```

Note that this should be called before we instantiate any components. Alternatively we can call this method and then use the SwingUtilities updateComponentTree() method to change the current L&F of a container and all its children, as discussed previously.

UI Guideline: Design Balance is affected by Look & Feel Selection

Beware! Although it is technically possible to update Look & Feel on-the-fly, this may often be visually undesirable. Different L&Fs use different graphical weights for each component e.g. bezel thickness on buttons. Therefore, a display which is designed to look good in a particular Look and Feel may be visually unbalanced and inelegant when switched to another L&F. This could be due to the change in White Space which balances against the graphical weight of elements such as bezels or it may be a change in alignment. For example, the "Malachite" L&F is visually heavy, as a rough guide, more white space will be required compared to Metal L&F for a well balanced effect.

21.1.9 Creating a custom LookAndFeel implementation

Swing provides the complete flexibility of implementing our own custom L&F, and distributing it with our application. This task usually involves overriding the rendering functionality of all Swing components supported by our L&F (default implementations are then used for each remaining component we are not interested in customizing). In general this is not a simple project, and will almost always require referencing Swing plaf source code.

The first step is to establish a basic idea of how we will provide consistent UI delegate appearances. This includes some thought as to what colors and icons will be used for each component, and whether or not these choices fit well together.

Then we move on to the most significant step in creating a custom look-and-feel, which is the implementation of a javax.swing.LookAndFeel sub-class. The following five abstract methods are the minimum that must be defined:

String getID(): returns the string ID of this L&F (e.g. "Motif").

String getName(): returns a short string describing this L&F (e.g. "CDE/Motif").

`String getDescription()`: returns a one line string description of this L&F.

`boolean isNativeLookAndFeel()`: returns `true` if the L&F corresponds to the current underlying native platform.

`boolean isSupportedLookAndFeel()`: returns `true` if the the current underlying native platform supports and/or permits this L&F.

`UIDefaults getDefaults()`: returns the L&F-specific `Hashtable` of resources (discussed above). This is the most important method of any `LookAndFeel` implementation.

However, to make implementation simpler, it is normally expected that we extend `javax.swing.plaf.basic.BasicLookAndFeel` instead of `javax.swing.LookAndFeel` directly. In this case we override some of the following `BasicLookAndFeel` methods (along with a few `LookAndFeel` methods above):

`void initClassDefaults(UIDefaults table)`: fills a given `UIDefaults` instance with key/value pairs specifying IDs and fully qualified class names of UI delegates for each component supported by this L&F.

`void initComponentDefaults(UIDefaults table)`: fills a given `UIDefaults` instance with key/value pairs using information (typically drawing resources, e.g. colors, images, borders, etc.) specific to this L&F.

`void initSystemColorDefaults(UIDefaults table)`: fills a given `UIDefaults` instance with color information specific to this L&F.

`void loadSystemColors(UIDefaults table, String[] systemColors, boolean useNative)`: fills a given `UIDefaults` instance with color information specific to the underlying platform.

The first two methods are the most significant, and we will discuss them in a bit more detail here.

21.1.10 Defining default component resources

The following code shows how to override the `initComponentDefaults()` method to store custom resources in a given `UIDefaults` instance. These resources will be used to construct a `JButton` UI delegate corresponding to this L&F (an imaginary implementation for now):

```
protected void initComponentDefaults(UIDefaults table) {
  super.initComponentDefaults(table);
  Object[] defaults = {
    "Button.font", new FontUIResource("Arial", Font.BOLD, 12 ),
    "Button.background", new ColorUIResource(4, 108, 2),
    "Button.foreground", new ColorUIResource(236, 236, 0),
    "Button.margin", new InsetsUIResource(8, 8, 8, 8)
  };
  table.putDefaults( defaults );
}
```

Note that the super-class `initComponentDefaults()` method is called before putting our custom information in the table, since we only want to override button UI resources. Also note that the resource objects are encapsulated in special wrapper classes defined in `javax.swing.plaf` package (`Font` instances are placed in `FontUIResources`, `Colors` in `ColorUIResources`, etc.). This is necessary to correctly load and unload resources when the current L&F is changed.

Note: Resources keys start with the component name, minus the "J" prefix. So "Button.font" defines the font resource for JButtons, while "RadioButton.font" defines the font resource for JRadioButtons. Unfortunately these standard resource keys are not documented, but they can all be found directly in Swing

L&F source code. For example, see MetalLookAndFeel.java in package javax.swing.plaf.metal.

21.1.11 Defining class defaults

Providing custom resources, such as colors and fonts, is the simplest way to create a custom L&F. However, to provide more powerful customizations we need to develop custom extensions of `ComponentUI` classes for specific components. We also need to provide a means of locating our custom UI delegate classes so that `UIManager` can successfully switch a component's L&F on demand.

The following code overrides the `initClassDefaults()` method to store information about our imaginary `myLF.MyLFButtonUI` class (a member of the imaginary `myLF` look-and-feel package), which extends `javax.swing.plaf.ButtonUI`, and will be used to provide a custom L&F for `JButton`:

```
  protected void initClassDefaults(UIDefaults table) {
    super.initClassDefaults(table);
    try {
        String className = "myLF.MyLFButtonUI";
        Class buttonClass = Class.forName(className);
        table.put("ButtonUI", className);
        table.put(className, buttonClass);
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
  }
```

The `initClassDefaults()` implementation of the super-class is called before (not after) we populate the table with our custom information. This is because we don't intend to override all UI class mappings for all components. Instead we use the default settings for all but "ButtonUI." (We did a similar thing above in `initComponentDefaults()`.) Also note that we place both the fully qualified class name of the delegate, as well as the `Class` instance itself, in the table.

Note: Placing only the class name in the defaults table does not provide correct functionality. As of Java 2 FCS, without a corresponding `Class` instance in the table as well, `getUI()` will not be able to retrieve instances of custom L&F delegates. We will see that this is the case in the examples below.

21.1.12 Creating custom UI delegates

Now it's time to show a simple pseudo-code implemention of our imaginary `myLF.MyLFButtonUI` class we've been relating our discussion to:

```
package myLF;

public class MyLFButtonUI extends  BasicButtonUI {
  private final static MyLFButtonUI m_buttonUI =
    new MyLFButtonUI();

  protected Color  m_backgroundNormal = null;
  // Declare variables for other resources.

  public static ComponentUI createUI( JComponent c ) {
    return m_buttonUI;
  }

  public void installUI(JComponent c) {
    super.installUI(c);
```

```
    m_backgroundNormal = UIManager.getColor(
      "Button.background");
    // Retrieve other resources and store them
    // as instance variables.
    // Add listeners. These might be registered to receive
    // events from a component's model or the component itself.
  }

  public void uninstallUI(JComponent c) {
    super.uninstallUI(c);
    // Provide cleanup.
    // Remove listeners.
  }

  public void update(Graphics g, JComponent c) {
    // Provide custom background painting if the component is
    // opaque, then call paint().
  }

  public void paint(Graphics g, JComponent c) {
    // Provide custom rendering for the given component.
  }

  // Provide implementation for listeners.
}
```

This class extends `javax.swing.plaf.basic.BasicButtonUI` to override some of its functionality and relies on basic L&F defaults for the rest. The shared instance, `MyLFButtonUI m_buttonUI`, is created once and retrieved using the `createUI()` method. Thus, only one instance of this delegate will exist, and it will act as the view and controller for all `JButton` instances with the `myLF` look-and-feel.

The `installUI()` method retrieves `myLF`-specific resources corresponding to `JButton` (refer back to our discussion of `initComponentDefaults()` above). We might also use this method to add mouse and key listeners to provide L&F-specific functionality. For instance, we might design our button UI such that an associated `JButton`'s text changes color each time the mouse cursor rolls over it. An advantage of this approach is that we don't need to modify our application--we can still use normal `JButtons`. Once `myLF` is installed, this functionality will automatically appear.

The `uninstallUI()` method performs all necessary cleanup, including the removal of any listeners that this delegate might have attached to the component or its model.

The `update()` method will paint the given component's background if it is opaque, and then immediately call `paint()` (do not confuse this method with `JComponent`'s `paint()` method).

Note: We always recommend to implement painting functionality in this way, but in reality the background of Swing components are more often painted directly within the `paint()` method (a quick skim through Swing UI delegate source code illustrates--for example, see `BasicRadioButtonUI.java`). If this is not the case the resulting background will be that painted by `JComponent`'s painting routine. For this reason we often find no background rendering code at all in UI delegates.

This is a relatively minor issue. Just make sure that if you do want to take control of a component's background rendering, it is best to do so in UI delegate `update()` methods. This rendering should only occur if the associated component's opaque property is set to true, and it should be called before the main detail of its view is painted (`update()` should end with a call to `paint()`).

The `paint()` method renders a given component using the given graphical context. Note that to use an L&F successfully, the component class should not implement any rendering functionality for itself. Instead, it

should allow its painting to be controlled by UI delegate classes so that all rendering is L&F-specific (refer back to chapter 2 for further discussion of painting issues).

Note: Implementing a custom L&F will make much more sense once we step through the first two examples. We suggest that you reference the above discussion often as you make your way through this chapter. Reviewing the discussion of MVC in chapter 1 may also be helpful at this point.

21.1.13 MetalTheme

class javax.swing.plaf.metal.MetalTheme

Themes are sets of color and font definitions that can be dynamically plugged into `MetalLookAndFeel`, and immediately used by a Swing app on-the-fly if Metal is the current L&F. To create a theme we simply subclass `MetalTheme` (or `DefaultMetalTheme`) and override a selection of its numerous `getXX()` methods to return a specific font or color. A quick browse through these methods shows implementations for all the colors and fonts used throughout the MetalL&F, allowing us to customize Metal appearance however we like. `MetalLookAndFeel` contains `createDefaultTheme()`, a protected method used to create the default metal theme, and provides us with the `setCurrentTheme()` method which allows us to plug in a new theme. The effects of plugging in a new theme are seen immediately. Themes offer a simple alternative to building a custom `LookAndFeel`, when all that is desired are some simple appearance changes.

21.2 Custom L&F: part I - Using custom resources

UI Guideline: When to consider a Custom L&F

Developing a Custom L&F is not a trivial undertaking. Almost certainly, there is more effort needed for the design rather than the coding. Consider a Custom Look & Feel in these situations:

You are designing a single use system, such as a self-service kiosk

You are intending to roll-out a suite of enterprise applications which will work together and you want the L&F to reflect the corporate image or identity.

You are developing a family of software products and want to develop a unique environment or corporate identity. This was exactly Sun's intention with MetalL&F which closely reflects the colours and styles used in the Sun Corporate identity. Other examples of custom designed environments are Lotus Notes, Lotus eSuite and Sun HotJava Views.

The easiest way to create a custom look-and-feel is to simply customize default component resources (colors, borders, fonts, etc.) without actually implementing any custom UI delegates. In this case, the only thing we need to do is extend `BasicLookAndFeel` (see above discussion), or another existing `LookAndFeel` implementation, and provide a set of resources. The following example demonstrates how this can be done by beginning the implementation of our custom "Malachite" L&F.

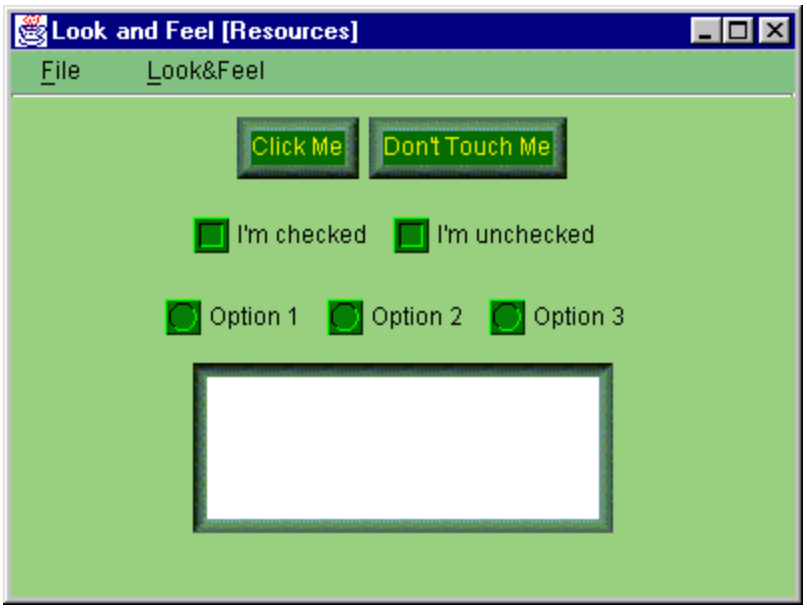


Figure 21.1 Malachite L&F in action.
<<file figure21-1.gif>>

The Code: Button1.java
see \Chapter21\1

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

import javax.swing.*;
import javax.swing.event.*;

import Malachite.*;

public class Button1 extends JFrame
{
  protected Hashtable  m_lfs;

  public Button1() {
    super("Look and Feel [Resources]");
    setSize(400, 300);
    getContentPane().setLayout(new FlowLayout());

    JMenuBar menuBar = createMenuBar();
    setJMenuBar(menuBar);

    JPanel p = new JPanel();
    JButton bt1 = new JButton("Click Me");
    p.add(bt1);

    JButton bt2 = new JButton("Don't Touch Me");
    p.add(bt2);
    getContentPane().add(p);

    p = new JPanel();
    JCheckBox chk1 = new JCheckBox("I'm checked");
    chk1.setSelected(true);
    p.add(chk1);
```

```
    JCheckBox chk2 = new JCheckBox("I'm unchecked");
    chk2.setSelected(false);
    p.add(chk2);
    getContentPane().add(p);

    p = new JPanel();
    ButtonGroup grp = new ButtonGroup();
    JRadioButton rd1 = new JRadioButton("Option 1");
    rd1.setSelected(true);
    p.add(rd1);
    grp.add(rd1);

    JRadioButton rd2 = new JRadioButton("Option 2");
    p.add(rd2);
    grp.add(rd2);

    JRadioButton rd3 = new JRadioButton("Option 3");
    p.add(rd3);
    grp.add(rd3);
    getContentPane().add(p);

    JTextArea txt = new JTextArea(5, 30);
    getContentPane().add(txt);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  protected JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();
    JMenu mFile = new JMenu("File");
    mFile.setMnemonic('f');

    JMenuItem mItem = new JMenuItem("Exit");
    mItem.setMnemonic('x');
    ActionListener lstExit = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    };
    mItem.addActionListener(lstExit);
    mFile.add(mItem);
    menuBar.add(mFile);

    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String str = e.getActionCommand();
        Object obj = m_lfs.get(str);
        if (obj != null)
        try {
          String className = (String)obj;
          Class lnfClass = Class.forName(className);
          UIManager.setLookAndFeel(
            (LookAndFeel)(lnfClass.newInstance()));
          SwingUtilities.updateComponentTreeUI(
            Button1.this);
```

```
        }
        catch (Exception ex) {
          ex.printStackTrace();
          System.err.println(ex.toString());
        }
      }
    };

    m_lfs = new Hashtable();
    UIManager.LookAndFeelInfo lfs[] =
      UIManager.getInstalledLookAndFeels();
    JMenu mLF = new JMenu("Look&Feel");
    mLF.setMnemonic('l');
    for (int k = 0; k < lfs.length; k++ ) {
      String name = lfs[k].getName();
      JMenuItem lf = new JMenuItem(name);
      m_lfs.put(name, lfs[k].getClassName());
      lf.addActionListener(lst);
      mLF.add(lf);
    }
    menuBar.add(mLF);

    return menuBar;
  }

  public static void main(String argv[]) {
    try {
      LookAndFeel malachite = new Malachite.MalachiteLF();
      UIManager.LookAndFeelInfo info =
        new UIManager.LookAndFeelInfo(malachite.getName(),
      malachite.getClass().getName());
      UIManager.installLookAndFeel(info);
      UIManager.setLookAndFeel(malachite);
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.err.println(ex.toString());
    }
    new Button1();
  }
}
```

The Code: MalachiteLF.java
see \Chapter21\1\Malachite

```
package Malachite;

import java.awt.*;

import javax.swing.*;
import javax.swing.plaf.*;
import javax.swing.plaf.basic.*;

public class MalachiteLF extends BasicLookAndFeel
 implements java.io.Serializable
{
  public String getID() { return "Malachite"; }
  public String getName() { return "Malachite Look and Feel"; }
  public String getDescription() { return "Sample L&F from Swing"; }
  public boolean isNativeLookAndFeel() { return false; }
  public boolean isSupportedLookAndFeel() { return true; }
```

```
protected void initComponentDefaults(UIDefaults table) {
  super.initComponentDefaults(table);

  ColorUIResource commonBackground =
    new ColorUIResource(152, 208, 128);
  ColorUIResource commonForeground =
    new ColorUIResource(0, 0, 0);
  ColorUIResource buttonBackground =
    new ColorUIResource(4, 108, 2);
  ColorUIResource buttonForeground =
    new ColorUIResource(236, 236, 0);
  ColorUIResource menuBackground =
    new ColorUIResource(128, 192, 128);

  BorderUIResource borderRaised = new
    BorderUIResource(new MalachiteBorder(
  MalachiteBorder.RAISED));
  BorderUIResource borderLowered = new
    BorderUIResource(new MalachiteBorder(
      MalachiteBorder.LOWERED));

  FontUIResource commonFont = new
    FontUIResource("Arial", Font.BOLD, 12 );

  Icon ubox = new ImageIcon("Malachite/ubox.gif");
  Icon ubull = new ImageIcon("Malachite/ubull.gif");

  Object[] defaults = {
    "Button.font", commonFont,
    "Button.background", buttonBackground,
    "Button.foreground", buttonForeground,
    "Button.border", borderRaised,
    "Button.margin", new InsetsUIResource(8, 8, 8, 8),
    "Button.textIconGap", new Integer(4),
    "Button.textShiftOffset", new Integer(2),

    "CheckBox.font", commonFont,
    "CheckBox.background", commonBackground,
    "CheckBox.foreground", commonForeground,
    "CheckBox.icon", new IconUIResource(ubox),

    "MenuBar.font", commonFont,
    "MenuBar.background", menuBackground,
    "MenuBar.foreground", commonForeground,

    "Menu.font", commonFont,
    "Menu.background", menuBackground,
    "Menu.foreground", commonForeground,
    "Menu.selectionBackground", buttonBackground,
    "Menu.selectionForeground", buttonForeground,

    "MenuItem.font", commonFont,
    "MenuItem.background", menuBackground,
    "MenuItem.foreground", commonForeground,
    "MenuItem.selectionBackground", buttonBackground,
    "MenuItem.selectionForeground", buttonForeground,
    "MenuItem.margin", new InsetsUIResource(2, 2, 2, 2),

    "Panel.background", commonBackground,
    "Panel.foreground", commonForeground,
```

```
      "RadioButton.font", commonFont,
      "RadioButton.background", commonBackground,
      "RadioButton.foreground", commonForeground,
      "RadioButton.icon", new IconUIResource(ubull),

      "TextArea.margin", new InsetsUIResource(8, 8, 8, 8),
      "TextArea.border", borderLowered
    };

    table.putDefaults( defaults );
  }
}
```

The Code: MalachiteBorder.java
see \Chapter21\1\Malachite

```
package Malachite;

import java.awt.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class MalachiteBorder implements Border
{
  public static final int RAISED = 0;
  public static final int LOWERED = 1;

  static final String IMAGE_DIR = "Malachite/";
  static final ImageIcon IMAGE_NW = new ImageIcon(
    IMAGE_DIR+"nw.gif");
  static final ImageIcon IMAGE_N  = new ImageIcon(
    IMAGE_DIR+"n.gif");
  static final ImageIcon IMAGE_NE = new ImageIcon(
    IMAGE_DIR+"ne.gif");
  static final ImageIcon IMAGE_E  = new ImageIcon(
    IMAGE_DIR+"e.gif");
  static final ImageIcon IMAGE_SE = new ImageIcon(
    IMAGE_DIR+"se.gif");
  static final ImageIcon IMAGE_S  = new ImageIcon(
    IMAGE_DIR+"s.gif");
  static final ImageIcon IMAGE_SW = new ImageIcon(
    IMAGE_DIR+"sw.gif");
  static final ImageIcon IMAGE_W  = new ImageIcon(
    IMAGE_DIR+"w.gif");

  static final ImageIcon IMAGE_L_NW = new ImageIcon(
    IMAGE_DIR+"l_nw.gif");
  static final ImageIcon IMAGE_L_N  = new ImageIcon(
    IMAGE_DIR+"l_n.gif");
  static final ImageIcon IMAGE_L_NE = new ImageIcon(
    IMAGE_DIR+"l_ne.gif");
  static final ImageIcon IMAGE_L_E  = new ImageIcon(
    IMAGE_DIR+"l_e.gif");
  static final ImageIcon IMAGE_L_SE = new ImageIcon(
    IMAGE_DIR+"l_se.gif");
  static final ImageIcon IMAGE_L_S  = new ImageIcon(
    IMAGE_DIR+"l_s.gif");
  static final ImageIcon IMAGE_L_SW = new ImageIcon(
    IMAGE_DIR+"l_sw.gif");
```

```
  static final ImageIcon IMAGE_L_W  = new ImageIcon(
    IMAGE_DIR+"l_w.gif");

  protected int m_w = 7;
  protected int m_h = 7;

  protected boolean m_isRaised = true;

  public MalachiteBorder() {}

  public MalachiteBorder(int type) {
    if (type != RAISED && type != LOWERED)
      throw new IllegalArgumentException(
        "Type must be RAISED or LOWERED");
    m_isRaised = (type == RAISED);
  }

  public Insets getBorderInsets(Component c) {
    return new Insets(m_h, m_w, m_h, m_w);
  }

  public  boolean isBorderOpaque() { return true; }

  public void paintBorder(Component c, Graphics g,
   int x, int y, int w, int h)
  {
    int x1 = x+m_w;
    int x2 = x+w-m_w;
    int y1 = y+m_h;
    int y2 = y+h-m_h;
    int xx, yy;

    if (m_isRaised) {
      for (xx=x1; xx<=x2; xx += IMAGE_N.getIconWidth())
        g.drawImage(IMAGE_N.getImage(), xx, y, c);
      for (yy=y1; yy<=y2; yy += IMAGE_E.getIconHeight())
        g.drawImage(IMAGE_E.getImage(), x2, yy, c);
      for (xx=x1; xx<=x2; xx += IMAGE_S.getIconWidth())
        g.drawImage(IMAGE_S.getImage(), xx, y2, c);
      for (yy=y1; yy<=y2; yy += IMAGE_W.getIconHeight())
        g.drawImage(IMAGE_W.getImage(), x, yy, c);
      g.drawImage(IMAGE_NW.getImage(), x, y, c);
      g.drawImage(IMAGE_NE.getImage(), x2, y, c);
      g.drawImage(IMAGE_SE.getImage(), x2, y2, c);
      g.drawImage(IMAGE_SW.getImage(), x, y2, c);
    }
    else {
      for (xx=x1; xx<=x2; xx += IMAGE_L_N.getIconWidth())
        g.drawImage(IMAGE_L_N.getImage(), xx, y, c);
      for (yy=y1; yy<=y2; yy += IMAGE_L_E.getIconHeight())
        g.drawImage(IMAGE_L_E.getImage(), x2, yy, c);
      for (xx=x1; xx<=x2; xx += IMAGE_L_S.getIconWidth())
        g.drawImage(IMAGE_L_S.getImage(), xx, y2, c);
      for (yy=y1; yy<=y2; yy += IMAGE_L_W.getIconHeight())
        g.drawImage(IMAGE_L_W.getImage(), x, yy, c);
      g.drawImage(IMAGE_L_NW.getImage(), x, y, c);
      g.drawImage(IMAGE_L_NE.getImage(), x2, y, c);
      g.drawImage(IMAGE_L_SE.getImage(), x2, y2, c);
      g.drawImage(IMAGE_L_SW.getImage(), x, y2, c);
    }
  }
}
```

Class Button1

This class represents a simple frame container populated by several components: `JButtons`, `JCheckBoxes`, `JRadioButtons`, and `JTextArea`. Code in the constructor should be familiar and requires no special explanation here.

The `createMenuBar()` method is responsible for creating this frame's menu bar. A menu titled "Look&Feel" is populated by menu items corresponding to `LookAndFeel` implementations available on the current JVM. An array of `UIManager.LookAndFeelInfo` instances is retrieved using the `UIManager.getInstalledLookAndFeels()` method. L&F class names stored in each info object are placed into the `m_lfs Hashtable` for future use. A brief text description of a particular L&F retrieved using the `getName()` method is used to create each corresponding menu item.

When a menu item is selected, the corresponding `ActionListener` updates the L&F for our application. This listener locates the class name corresponding to the selected menu item, and a new instance of that class is created, through reflection, and set as the current L&F using the `UIManager.setLookAndFeel()` method.

Note: We can also use the overloaded `UIManager.setLookAndFeel(String className)` method which takes a fully qualified `LookAndFeel` class name as parameter. However, as of Java 2 FCS, this does not work properly on all platforms.

The `main()` method creates an instance of our custom L&F, `MalachiteLF` (defined in the `Malachite` package), makes it available to the the Java session using `UIManager.installLookAndFeel()`, and sets is as the current L&F using `UIManager.setLookAndFeel()`. Our example frame is then created, which uses Malachite resources initially.

Class Malachite.MalachiteLF

This class defines our Malachite look-and-feel. Note that it extends `BasicLookAndFeel` to override its functionality and resources only where necessary. This L&F is centered around a green malachite palette.

Note: Malachite is a green mineral containing copper. This mineral can be found in the Ural Mountains of Russia, in Australia, and in Arizona in the US. Since ancient times it has been used as gem stone.

Methods `getID()`, `getName()`, and `getDescription()` return a short ID, name, and a text description of this L&F respectively. As we've discussed earlier, method `initComponentDefaults()` fills a given `UIDefaults` instance with key/value pairs representing information specific to this L&F. In our implementation we customize resources for the following components (recall that the "J" prefix is not used): Button, CheckBox, RadioButton, TextArea, MenuBar, Menu, MenuItem, and Panel.

Note that we did not define the `initClassDefaults()` method because we have not implemented any custom UIdelegates (we will do this in the next section).

Class Malachite.MalachiteBorder

This class defines our custom Malachite implementation of the `Border` interface. This border is intended to provide the illusion of a 3D frame cut out of a green gem stone. It can be drawn in two forms: lowered or raised. A 3D effect is produced through the proper combination of previously prepared images. The actual rendering is done in the `paintBorder()` method, which simply draws a set of these images to render the border.

Running the Code

Figure 21.1 shows our `Button1` example frame populated with controls using the Malachite L&F. Note that these controls are lifeless. We cannot click buttons, check or uncheck boxes, or select radio buttons. Try using the menu to select another L&F available on your system and note the differences.

The components are actually fully functional when using the Malachite L&F, but they do not have the ability to change their appearance in response to user interaction. More functionality needs to be added to provide mouse and key listener capabilities, as well as additional resources for use in representing the selected state of the button components. We will do this in the next section.

Note: The UI delegate used for each of these components is the corresponding `basic` L&F version, because we did not override any class defaults in `MalachiteLF`. A quick look in the source code for these delegates shows that the rendering functionality for selected and focused states is not implemented. All sub-classes corresponding to specific L&Fs are responsible for implementing this functionality themselves.

Note: The text area in this example is not placed in a scrolling pane specifically because we want to emphasize the use of our custom border. This is the reason it resizes when a significant amount of text is entered.

21.3 Custom L&F: part II - Creating custom UI delegates

The next step in the creation of a custom L&F is the implementation of custom UI delegates corresponding to each supported component. In this section we'll show how to implement Malachite delegates for three relatively simple Swing components: `JButton`, `JCheckBox`, and `JRadioButton`.

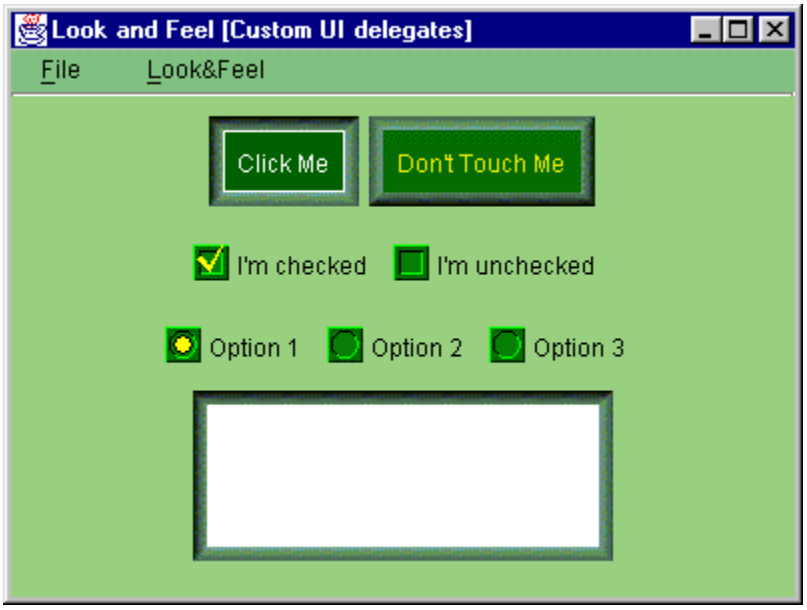


Figure 21.2 Custom Malachite UI delegates in action.
<<file figure21-2.gif>>

The Code: MalachiteLF.java
see \Chapter21\2\Malachite

```
package Malachite;

import java.awt.*;
```

```
import javax.swing.*;
import javax.swing.plaf.*;
import javax.swing.plaf.basic.*;

public class MalachiteLF extends BasicLookAndFeel
 implements java.io.Serializable
{
  // Unchanged code from section 21.1

  protected void initClassDefaults(UIDefaults table) {
    super.initClassDefaults(table);
    putDefault(table, "ButtonUI");
    putDefault(table, "CheckBoxUI");
    putDefault(table, "RadioButtonUI");
  }

  protected void putDefault(UIDefaults table, String uiKey) {
    try  {
      String className = "Malachite.Malachite"+uiKey;
      Class buttonClass = Class.forName(className);
      table.put(uiKey, className);
      table.put(className, buttonClass);
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  protected void initComponentDefaults(UIDefaults table) {
    super.initComponentDefaults(table);

    // Unchanged code from section 21.1

    Icon ubox = new ImageIcon("Malachite/ubox.gif");
    Icon ubull = new ImageIcon("Malachite/ubull.gif");

    Icon cbox = new ImageIcon("Malachite/cbox.gif");
    Icon pcbox = new ImageIcon("Malachite/p_cbox.gif");
    Icon pubox = new ImageIcon("Malachite/p_ubox.gif");

    Icon cbull = new ImageIcon("Malachite/cbull.gif");
    Icon pcbull = new ImageIcon("Malachite/p_cbull.gif");
    Icon pubull = new ImageIcon("Malachite/p_ubull.gif");

    Object[] defaults = {
      "Button.font", commonFont,
      "Button.background", buttonBackground,
      "Button.foreground", buttonForeground,
      "Button.border", borderRaised,
      "Button.margin", new InsetsUIResource(8, 8, 8, 8),
      "Button.textIconGap", new Integer(4),
      "Button.textShiftOffset", new Integer(2),

      "Button.focusBorder", focusBorder,
      "Button.borderPressed", borderLowered,
      "Button.activeForeground", new
        ColorUIResource(255, 255, 255),
      "Button.pressedBackground", new
        ColorUIResource(0, 96, 0),

      "CheckBox.font", commonFont,
      "CheckBox.background", commonBackground,
```

```
      "CheckBox.foreground", commonForeground,
      "CheckBox.icon", new IconUIResource(ubox),

      "CheckBox.focusBorder", focusBorder,
      "CheckBox.activeForeground", activeForeground,
      "CheckBox.iconPressed", new IconUIResource(pubox),
      "CheckBox.iconChecked", new IconUIResource(cbox),
      "CheckBox.iconPressedChecked", new IconUIResource(pcbox),
      "CheckBox.textIconGap", new Integer(4),

      // Unchanged code from section 21.1

      "RadioButton.font", commonFont,
      "RadioButton.background", commonBackground,
      "RadioButton.foreground", commonForeground,
      "RadioButton.icon", new IconUIResource(ubull),

      "RadioButton.focusBorder", focusBorder,
      "RadioButton.activeForeground", activeForeground,
      "RadioButton.iconPressed", new IconUIResource(pubull),
      "RadioButton.iconChecked", new IconUIResource(cbull),
      "RadioButton.iconPressedChecked", new IconUIResource(pcbull),
      "RadioButton.textIconGap", new Integer(4),

      "TextArea.margin", new InsetsUIResource(8, 8, 8, 8),
      "TextArea.border", borderLowered
    };

    table.putDefaults( defaults );
  }
}
```

The Code: MalachiteButtonUI.java
see \Chapter21\2\Malachite

```
package Malachite;

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.*;
import javax.swing.plaf.basic.*;

public class MalachiteButtonUI extends BasicButtonUI
 implements java.io.Serializable, MouseListener, KeyListener
{
  private final static MalachiteButtonUI m_buttonUI =
    new MalachiteButtonUI();

  protected Border m_borderRaised = null;
  protected Border m_borderLowered = null;
  protected Color  m_backgroundNormal = null;
  protected Color  m_backgroundPressed = null;
  protected Color  m_foregroundNormal = null;
  protected Color  m_foregroundActive = null;
  protected Color  m_focusBorder = null;

  public MalachiteButtonUI() {}
```

```
  public static ComponentUI createUI( JComponent c ) {
    return m_buttonUI;
  }

  public void installUI(JComponent c) {
    super.installUI(c);

    m_borderRaised = UIManager.getBorder(
      "Button.border");
    m_borderLowered = UIManager.getBorder(
      "Button.borderPressed");
    m_backgroundNormal = UIManager.getColor(
      "Button.background");
    m_backgroundPressed = UIManager.getColor(
      "Button.pressedBackground");
    m_foregroundNormal = UIManager.getColor(
      "Button.foreground");
    m_foregroundActive = UIManager.getColor(
      "Button.activeForeground");
    m_focusBorder = UIManager.getColor(
      "Button.focusBorder");

    c.addMouseListener(this);
    c.addKeyListener(this);
  }

  public void uninstallUI(JComponent c) {
    super.uninstallUI(c);
    c.removeMouseListener(this);
    c.removeKeyListener(this);
  }

  public void paint(Graphics g, JComponent c) {
    AbstractButton b = (AbstractButton) c;
    Dimension d = b.getSize();

    g.setFont(c.getFont());
    FontMetrics fm = g.getFontMetrics();

    g.setColor(b.getForeground());
    String caption = b.getText();
    int x = (d.width - fm.stringWidth(caption))/2;
    int y = (d.height + fm.getAscent())/2;
    g.drawString(caption, x, y);

    if (b.isFocusPainted() && b.hasFocus()) {
      g.setColor(m_focusBorder);
      Insets bi = b.getBorder().getBorderInsets(b);
      g.drawRect(bi.left, bi.top, d.width-bi.left-bi.right-1,
      d.height-bi.top-bi.bottom-1);
    }
  }

  public Dimension getPreferredSize(JComponent c) {
    Dimension d = super.getPreferredSize(c);
    if (m_borderRaised != null) {
      Insets ins = m_borderRaised.getBorderInsets(c);
      d.setSize(d.width+ins.left+ins.right,
        d.height+ins.top+ins.bottom);
    }
    return d;
  }
```

```
  public void mouseClicked(MouseEvent e) {}

  public void mousePressed(MouseEvent e) {
    JComponent c = (JComponent)e.getComponent();
    c.setBorder(m_borderLowered);
    c.setBackground(m_backgroundPressed);
  }

  public void mouseReleased(MouseEvent e) {
    JComponent c = (JComponent)e.getComponent();
    c.setBorder(m_borderRaised);
    c.setBackground(m_backgroundNormal);
  }

  public void mouseEntered(MouseEvent e) {
    JComponent c = (JComponent)e.getComponent();
    c.setForeground(m_foregroundActive);
    c.repaint();
  }

  public void mouseExited(MouseEvent e) {
    JComponent c = (JComponent)e.getComponent();
    c.setForeground(m_foregroundNormal);
    c.repaint();
  }

  public void keyTyped(KeyEvent e) {}

  public void keyPressed(KeyEvent e) {
    int code = e.getKeyCode();
    if (code == KeyEvent.VK_ENTER || code == KeyEvent.VK_SPACE) {
      JComponent c = (JComponent)e.getComponent();
      c.setBorder(m_borderLowered);
      c.setBackground(m_backgroundPressed);
    }
  }

  public void keyReleased(KeyEvent e) {
    int code = e.getKeyCode();
    if (code == KeyEvent.VK_ENTER || code == KeyEvent.VK_SPACE) {
      JComponent c = (JComponent)e.getComponent();
      c.setBorder(m_borderRaised);
      c.setBackground(m_backgroundNormal);
    }
  }
}
```

The Code: MalachiteCheckBoxUI.java
see \Chapter21\2\Malachite

```
package Malachite;

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.*;
import javax.swing.plaf.basic.*;
```

```
public class MalachiteCheckBoxUI extends BasicCheckBoxUI
 implements java.io.Serializable, MouseListener
{
  private final static MalachiteCheckBoxUI m_buttonUI =
    new MalachiteCheckBoxUI();

  protected Color  m_backgroundNormal = null;
  protected Color  m_foregroundNormal = null;
  protected Color  m_foregroundActive = null;
  protected Icon   m_checkedIcon = null;
  protected Icon   m_uncheckedIcon = null;
  protected Icon   m_pressedCheckedIcon = null;
  protected Icon   m_pressedUncheckedIcon = null;
  protected Color  m_focusBorder = null;
  protected int    m_textIconGap = -1;

  public MalachiteCheckBoxUI() {}

  public static ComponentUI createUI( JComponent c ) {
    return m_buttonUI;
  }

  public void installUI(JComponent c) {
    super.installUI(c);
    m_backgroundNormal = UIManager.getColor(
      "CheckBox.background");
    m_foregroundNormal = UIManager.getColor(
      "CheckBox.foreground");
    m_foregroundActive = UIManager.getColor(
      "CheckBox.activeForeground");
    m_checkedIcon = UIManager.getIcon(
      "CheckBox.iconChecked");
    m_uncheckedIcon = UIManager.getIcon(
      "CheckBox.icon");
    m_pressedCheckedIcon = UIManager.getIcon(
      "CheckBox.iconPressedChecked");
    m_pressedUncheckedIcon = UIManager.getIcon(
      "CheckBox.iconPressed");
    m_focusBorder = UIManager.getColor(
      "CheckBox.focusBorder");
    m_textIconGap = UIManager.getInt(
      "CheckBox.textIconGap");

    c.setBackground(m_backgroundNormal);
    c.addMouseListener(this);
  }

  public void uninstallUI(JComponent c) {
    super.uninstallUI(c);
    c.removeMouseListener(this);
  }

  public void paint(Graphics g, JComponent c) {
    AbstractButton b = (AbstractButton)c;
    ButtonModel model = b.getModel();
    Dimension d = b.getSize();

    g.setFont(c.getFont());
    FontMetrics fm = g.getFontMetrics();

    Icon icon = m_uncheckedIcon;
    if (model.isPressed() && model.isSelected())
```

```
      icon = m_pressedCheckedIcon;
    else if (model.isPressed() && !model.isSelected())
      icon = m_pressedUncheckedIcon;
    else if (!model.isPressed() && model.isSelected())
      icon = m_checkedIcon;

    g.setColor(b.getForeground());
    int x = 0;
    int y = (d.height - icon.getIconHeight())/2;
    icon.paintIcon(c, g, x, y);

    String caption = b.getText();
    x = icon.getIconWidth() + m_textIconGap;
    y = (d.height + fm.getAscent())/2;
    g.drawString(caption, x, y);

    if (b.isFocusPainted() && b.hasFocus()) {
      g.setColor(m_focusBorder);
      Insets bi = b.getBorder().getBorderInsets(b);
      g.drawRect(x-2, y-fm.getAscent()-2, d.width-x,
      fm.getAscent()+fm.getDescent()+4);
    }
  }

  public void mouseClicked(MouseEvent e) {}
  public void mousePressed(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}

  public void mouseEntered(MouseEvent e) {
    JComponent c = (JComponent)e.getComponent();
    c.setForeground(m_foregroundActive);
    c.repaint();
  }

  public void mouseExited(MouseEvent e) {
    JComponent c = (JComponent)e.getComponent();
    c.setForeground(m_foregroundNormal);
    c.repaint();
  }
}
```

The Code: MalachiteRadioButtonUI.java
see \Chapter21\2\Malachite

```
package Malachite;

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

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.*;
import javax.swing.plaf.basic.*;

public class MalachiteRadioButtonUI extends MalachiteCheckBoxUI
 implements java.io.Serializable, MouseListener
{
  private final static MalachiteRadioButtonUI m_buttonUI =
    new MalachiteRadioButtonUI();

  public MalachiteRadioButtonUI() {}
```

```
  public static ComponentUI createUI( JComponent c ) {
    return m_buttonUI;
  }

  public void installUI(JComponent c) {
    super.installUI(c);
    m_backgroundNormal = UIManager.getColor(
      "RadioButton.background");
    m_foregroundNormal = UIManager.getColor(
      "RadioButton.foreground");
    m_foregroundActive = UIManager.getColor(
      "RadioButton.activeForeground");
    m_checkedIcon = UIManager.getIcon(
      "RadioButton.iconChecked");
    m_uncheckedIcon = UIManager.getIcon(
      "RadioButton.icon");
    m_pressedCheckedIcon = UIManager.getIcon(
      "RadioButton.iconPressedChecked");
    m_pressedUncheckedIcon = UIManager.getIcon(
      "RadioButton.iconPressed");
    m_focusBorder = UIManager.getColor(
      "RadioButton.focusBorder");
    m_textIconGap = UIManager.getInt(
      "RadioButton.textIconGap");

    c.setBackground(m_backgroundNormal);
    c.addMouseListener(this);
  }
}
```

Understanding the Code

Class Malachite.MalachiteLF

The `initClassDefaults()` method inherited from `BasicLookAndFeel` is now overridden. As we've discussed above, this method will be called to fill a given `UIDefaults` instance with information about the specific classes responsible for providing a component's UI delegate for this L&F. Our implementation calls the super-class's `initClassDefaults()` method to provide all default options. It then replaces the delegate classes for our three supported button components by calling our `putDefault()` custom method. This helper method puts two entries into the given `UIDefaults` instance: the UI delegate fully qualified class name, and a corresponding instance of `java.lang.Class` (see 21.1.1.1).

The `initComponentDefaults()` method now places more custom resources into the given `UIDefaults` instance, including six custom icons. These resources are needed by our custom Malachite UI delegates, as we will see below.

Class Malachite.MalachiteButtonUI

This class provides a custom UI delegate for `JButton`. It extends `BasicButtonUI` to reuse much of its functionality, and implements `MouseListener` and `KeyListener` to capture and process user input.

Class variable:

`MalachiteButtonUI m_buttonUI`: a shared instance of this class which is returned by `createUI()`.

Instance variables:

`Border m_borderRaised`: border when not pressed.

`Border m_borderLowered`: border when pressed.

`Color m_backgroundNormal`: background color when not pressed.

`Color m_backgroundPressed`: background color when pressed.

`Color m_foregroundNormal`: foreground color.

`Color m_foregroundActive`: foreground color when mouse cursor rolls over.

`Color m_focusBorder`: focus rectangle color.

The `installUI()` method retrieves rendering resources from the defaults table by calling static methods defined in the `UIManager` class (these resources were stored by `MalichiteLF` as described above). It also attaches `this` as a `MouseListener` and `KeyListener` to the specified component. The `uninstallUI()` simply removes these listeners.

The `paint()` method renders a given component using the given graphical context. Rendering of the background and border is done automatically by `JComponent` (see 21.1.12), so the responsibility of this method is to simply render a button's text and focus rectangle.

The `getPreferredSize()` method is overridden since the default implementation in the `BasicButtonUI` class does not take into account the button's border (interestingly enough). Since we use a relatively thick border in Malachite, we need to override this method and add the border's insets to the width and height returned by the superclass implementation.

The next five methods represent an implementation of the `MouseListener` interface. To indicate that a button component is currently pressed, the `mousePressed()` method changes a button's background and border, which in turn causes that component to be repainted. Method `mouseReleased()` restores these attributes. To provide an additional rollover effect, method `mouseEntered()` changes the associated button's foreground color, which is then restored in the `mouseExited()` method.

The remaining three methods represent an implementation of the `KeyListener` interface. Pressing the space-bar or Enter keys while the button is in focus produces the same effect as performing a button click.

Class Malachite.MalachiteCheckBoxUI

This class extends `BasicCheckBoxUI` to provide a custom UI delegate for our `JCheckBox` component.

Class variable:

`MalachiteCheckBoxUI m_buttonUI`: a shared instance of this class which is returned by `createUI()` method.

Instance variables:

`Color m_backgroundNormal`: component's background.

`Color m_foregroundNormal`: foreground color.

`Color m_foregroundActive`: rollover foreground color.

`Icon m_checkedIcon`: icon displayed when checked and not pressed.

`Icon m_uncheckedIcon`: icon displayed when not checked and not pressed.

`Icon m_pressedCheckedIcon`: icon displayed when checked and pressed.

`Icon m_pressedUncheckedIcon`: icon displayed when not checked and pressed.

`Color m_focusBorder`: focus rectangle color.

`int m_textIconGap`: gap between icon and text.

Similar to `MalachiteButtonUI`, the `installUI()` method retrieves rendering resources from the defaults table and stores them in instance variables. It also attatches `this` as a `MouseListener` to the given component.

The `paint()` method renders the given component using a given graphical context. It draws an icon, text, and focus rectangle when appropriate (this code is fairly straightforward and does not require detailed explanation here).

The next five methods represent an implementation of the `MouseListener` interface which provides a similar rollover effect to that of `MalachiteButtonUI`.

Class Malachite.MalachiteRadioButtonUI

This class extends `MalachiteCheckBoxUI`. The only major difference between this class and its parent is that this class uses a different set of icons to render the radio button. The `paint()` method is not overriden. The `installUI()` method is modified to retrieve the necessary resources.

Running the Code

Figure 21.2 shows our example frame from the previous section with our new Malachite UI delegates in action. You can see that the push buttons have became bigger because their size now properly includes the border thickness. The most significant difference appears when the buttons are clicked, and boxes are checked/unchecked using using the mouse and keyboard.

At this point we leave the implementation of Malachite UI delegates for other existing Swing components up to you. You should now have a good idea of how to approach the task for any component. Switching gears, we will now discuss L&F customization from the opposite point of view: providing existing L&F capabilities for custom components.

21.4 L&F for custom components: part I - Implementing L&F support

In this section we'll add support for existing L&Fs to a custom component, namely our `InnerFrame` developed in chapter 15. We'll show how to modify this component so it complies with the look-and-feel paradigm and behaves accordingly. It will use shared resources provided by the installed L&F and use these resources for rendering itself. This requires creation of a custom UI delegate as well as modification of the component itself.

To allow direct comparison with `JInternalFrame`, we create a desktop pane container with a menu bar to allow the creation of an arbitrary number of `InnerFrames` and `JInternalFrames` in a cascaded fashion. We also allow L&F switching at run-time (as we did in the previous examples).

Note: We use a `JDesktopPane` instead of our `mdi` package's custom `MDIPane` component because `JInternalFrame` generates a null pointer exception when activated (in Java 2 FCS) if its parent is not a `JDesktopPane`.

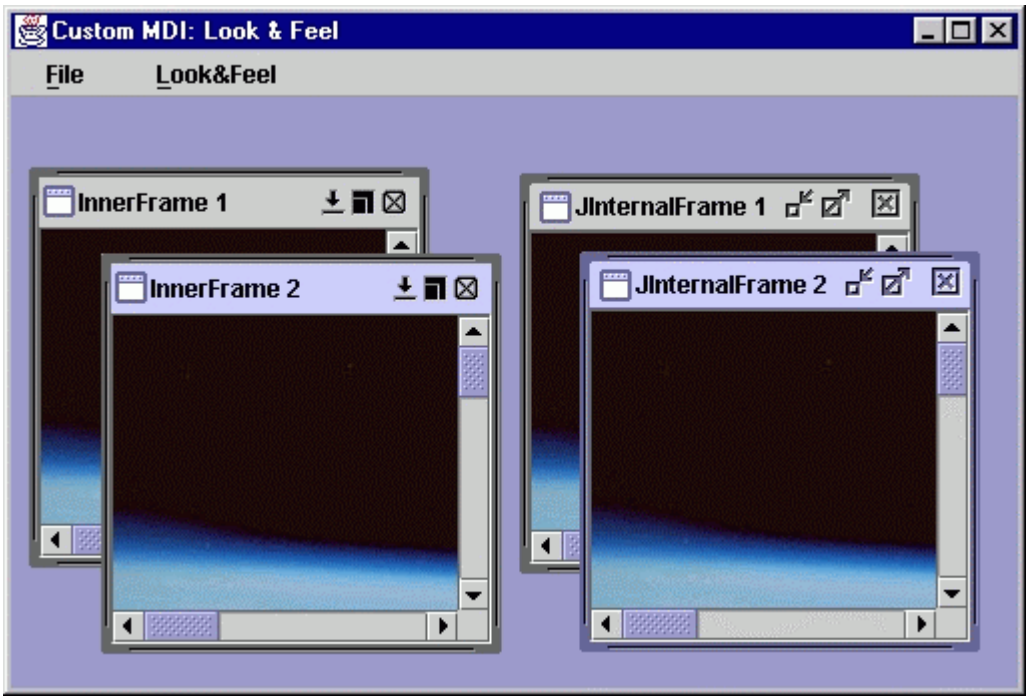


Figure 21.4 InnerFrame and JInternalFrame in the MetalL&F.
<<file figure21-4.gif>>

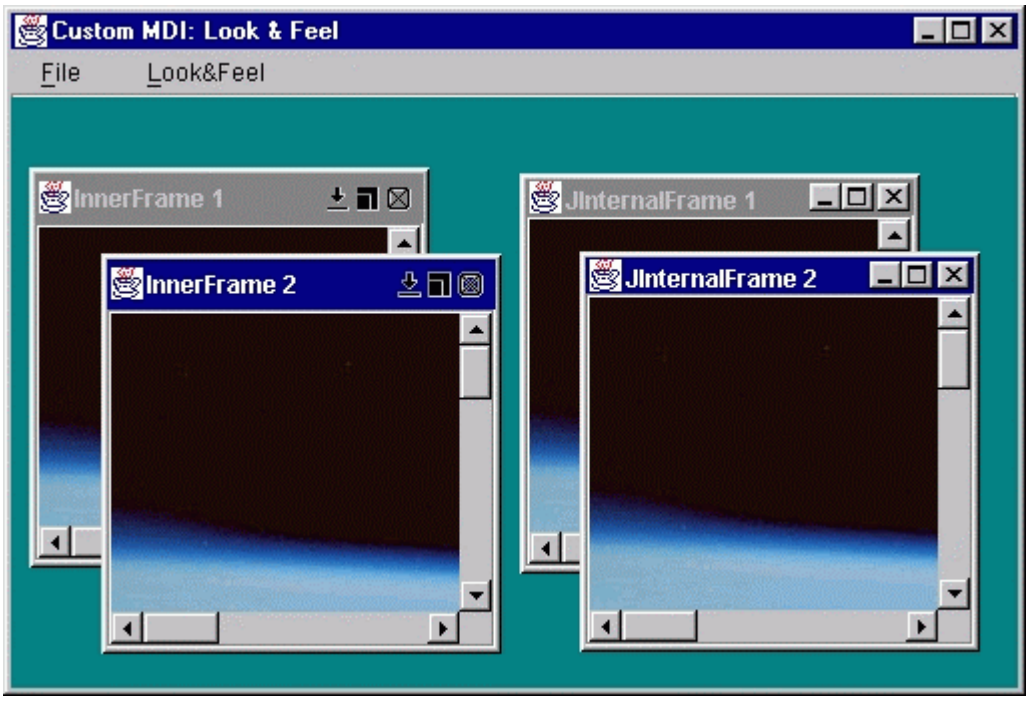


Figure 21.5 InnerFrame and JInternalFrame in the Windows L&F.
<<file figure21-5.gif>>

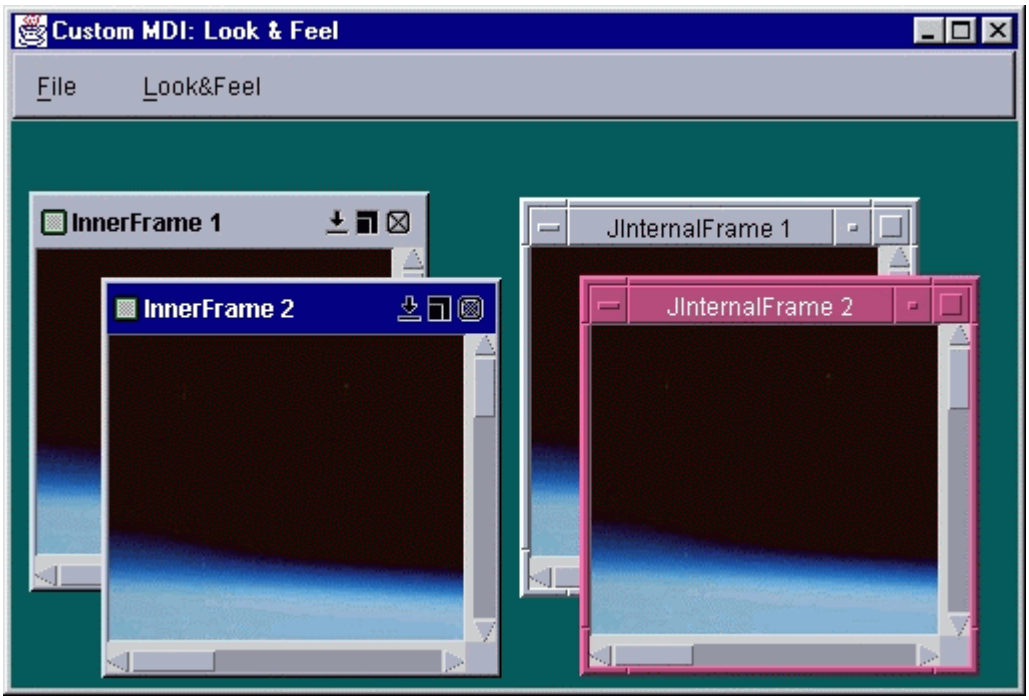


Figure 21.6 InnerFrame and JInternalFrame in the MotifL&F.
<<file figure21-6.gif>>

The Code: MdiContainer.java
see \Chapter21\3

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

import javax.swing.*;

import mdi.*;

public class MdiContainer extends JFrame
{
  protected ImageIcon m_icon;
  protected Hashtable m_lfs;

  public MdiContainer() {
    super("Custom MDI: Look & Feel");

    setSize(570,400);
    setContentPane(new JDesktopPane());

    m_icon = new ImageIcon("earth.jpg");
    JMenuBar menuBar = createMenuBar();
    setJMenuBar(menuBar);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);
    setVisible(true);
  }
```

```
protected JMenuBar createMenuBar() {
  JMenuBar menuBar = new JMenuBar();

  JMenu mFile = new JMenu("File");
  mFile.setMnemonic('f');

  JMenuItem mItem = new JMenuItem("New InnerFrame");
  mItem.setMnemonic('i');
  ActionListener lst = new ActionListener() {
    int m_counter = 0;
    public void actionPerformed(ActionEvent e) {
      m_counter++;
      InnerFrame frame = new InnerFrame("InnerFrame " +
        m_counter);
      int i = m_counter % 5;
      frame.setBounds(20+i*20, 20+i*20, 200, 200);
      frame.getContentPane().add(
        new JScrollPane(new JLabel(m_icon)));
      getContentPane().add(frame);
      frame.toFront();
    }
  };
  mItem.addActionListener(lst);
  mFile.add(mItem);

  mItem = new JMenuItem("New JInternalFrame");
  mItem.setMnemonic('j');
  lst = new ActionListener() {
    int m_counter = 0;
    public void actionPerformed(ActionEvent e) {
      m_counter++;
      JInternalFrame frame = new JInternalFrame(
        "JInternalFrame " + m_counter);
      frame.setClosable(true);
      frame.setMaximizable(true);
      frame.setIconifiable(true);
      frame.setResizable(true);

      int i = m_counter % 5;
      frame.setBounds(50+i*20, 50+i*20, 200, 200);
      frame.getContentPane().add(
        new JScrollPane(new JLabel(m_icon)));
      getContentPane().add(frame);
      frame.toFront();
    }
  };
  mItem.addActionListener(lst);
  mFile.add(mItem);
  mFile.addSeparator();

  mItem = new JMenuItem("Exit");
  mItem.setMnemonic('x');
  lst = new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      System.exit(0);
    }
  };
  mItem.addActionListener(lst);
  mFile.add(mItem);
  menuBar.add(mFile);
```

```
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        String str = e.getActionCommand();
        Object obj = m_lfs.get(str);
        if (obj != null)
          try {
            String className = (String)obj;
            Class lnfClass = Class.forName(className);
            UIManager.setLookAndFeel(
              (LookAndFeel)(lnfClass.newInstance()));
            SwingUtilities.updateComponentTreeUI(
              MdiContainer.this);
          }
          catch (Exception ex) {
            ex.printStackTrace();
            System.err.println(ex.toString());
          }
      }
    };

    m_lfs = new Hashtable();
    UIManager.LookAndFeelInfo lfs[] =
      UIManager.getInstalledLookAndFeels();
    JMenu mLF = new JMenu("Look&Feel");
    mLF.setMnemonic('l');
    for (int k = 0; k < lfs.length; k++ ) {
      String name = lfs[k].getName();
      JMenuItem lf = new JMenuItem(name);
      m_lfs.put(name, lfs[k].getClassName());
      lf.addActionListener(lst);
      mLF.add(lf);
    }
    menuBar.add(mLF);

    return menuBar;
  }

  public static void main(String argv[]) {
    new MdiContainer();
  }
}
```

The Code: InnerFrame.java
see \Chapter21\3\mdi

```
package mdi;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.border.*;

import javax.swing.plaf.ComponentUI;
import mdi.plaf.*;

public class InnerFrame extends JPanel
 implements RootPaneContainer, Externalizable
{
  // Unchanged code from section 15.7
```

```
  private Color m_titleBarBackground;
  private Color m_selectedTitleBarBackground;
  private Color m_titleBarForeground;
  private Color m_selectedTitleBarForeground;
  private Font  m_titleBarFont;
  private Border m_frameBorder;

  private Icon m_frameIcon;

  public InnerFrame() {
    this("");
  }

  // Unchanged code from section 15.7

  ////////////////////////////////////////////
  /////////////// L&F Support ////////////////
  ////////////////////////////////////////////

  static {
    UIManager.getDefaults().put(
      "InnerFrameUI", "mdi.plaf.InnerFrameUI");
    UIManager.getDefaults().put("InnerFrameButtonUI",
      "javax.swing.plaf.basic.BasicButtonUI");
  }

  public static ComponentUI createUI(JComponent a) {
    ComponentUI mui = new InnerFrameUI();
    return mui;
  }

  public void setUI(InnerFrameUI ui) {
    if ((InnerFrameUI)this.ui != ui) {
      super.setUI(ui);
      repaint();
    }
  }

  public InnerFrameUI getUI() {
    return (InnerFrameUI)ui;
  }

  public void updateUI() {
    setUI((InnerFrameUI)UIManager.getUI(this));
    invalidate();
  }

  public String getUIClassID() {
    return "InnerFrameUI";
  }

  // Unchanged code from section 15.7

  public void setSelectedTitleBarForeground(Color c) {
    m_selectedTitleBarForeground = c;
    updateTitleBarColors();
  }

  public Color getSelectedTitleBarForeground() {
    return m_selectedTitleBarForeground;
  }
```

```
  public void setTitleBarFont(Font f) {
    m_titleBarFont = f;
    updateTitleBarColors();
  }

  public Font getTitleBarFont() {
    return m_titleBarFont;
  }
```

```
  public void setBorder(Border b) {
    m_frameBorder = b;
    if (b != null) {
      Insets ins = b.getBorderInsets(this);
      if (m_northResizer != null)
        m_northResizer.setHeight(ins.top);
      if (m_southResizer != null)
        m_southResizer.setHeight(ins.bottom);
      if (m_eastResizer != null)
        m_eastResizer.setWidth(ins.right);
      if (m_westResizer != null)
        m_westResizer.setWidth(ins.left);
      if (isShowing())
        validate();
    }
  }

  public Border getBorder() {
    return m_frameBorder;
  }
```

```
  protected void updateTitleBarColors() {
    if (isShowing())
        repaint();
  }
```

```
  public void setFrameIcon(Icon fi) {
    m_frameIcon = fi;
    if (fi != null) {
      if (m_frameIcon.getIconHeight() > TITLE_BAR_HEIGHT)
        setTitleBarHeight(m_frameIcon.getIconHeight() + 2*FRAME_ICON_PADDING);
      if (m_iconLabel != null)
        m_iconLabel.setIcon(m_frameIcon);
    }
    else
        setTitleBarHeight(TITLE_BAR_HEIGHT);
    if (isShowing())
        revalidate();
  }

  public Icon getFrameIcon() {
    return m_frameIcon;
  }

  // Unchanged code from section 15.7

  protected void createTitleBar() {
  m_titlePanel = new JPanel() {
    public Dimension getPreferredSize() {
      return new Dimension(InnerFrame.this.getWidth(),
        m_titleBarHeight);
    }
```

```
    public Color getBackground() {
      if (InnerFrame.this == null)
        return super.getBackground();
      if (isSelected())
        return getSelectedTitleBarBackground();
      else
        return getTitleBarBackground();
    }
  };
  m_titlePanel.setLayout(new BorderLayout());
  m_titlePanel.setOpaque(true);

  m_titleLabel = new JLabel() {
    public Color getForeground() {
      if (InnerFrame.this == null)
        return super.getForeground();
      if (isSelected())
        return getSelectedTitleBarForeground();
      else
        return getTitleBarForeground();
    }

    public Font getFont() {
      if (InnerFrame.this == null)
        return super.getFont();
      return m_titleBarFont;
    }
  };

  // Unchanged code from section 15.7

  class InnerFrameButton extends JButton
  {
    // Unchanged code from section 15.7

    public void setBorder(Border b) { }

    public Border getBorder() { return null; }

    public String getUIClassID() {
        return "InnerFrameButtonUI";
    }
  }

  // Unchanged code from section 15.7

  class EastResizeEdge extends JPanel
   implements MouseListener, MouseMotionListener
  {
    private int WIDTH = BORDER_THICKNESS;
    private int MIN_WIDTH = ICONIZED_WIDTH;
    private boolean m_dragging;
    private JComponent m_resizeComponent;

    protected EastResizeEdge(JComponent c) {
      m_resizeComponent = c;
      setOpaque(false);
      if (m_frameBorder != null)
        WIDTH =
          m_frameBorder.getBorderInsets(InnerFrame.this).right;
    }
```

```
    public void setWidth(int w) {
      WIDTH = w;
    }

    // Unchanged code from section 15.7
  }

  // Classes WestResizeEdge, NorthResizeEdge, and SouthResizeEdge
  // are modified similarly

  public void writeExternal(ObjectOutput out) throws IOException {
    out.writeObject(m_titleBarBackground);
    out.writeObject(m_titleBarForeground);
    out.writeObject(m_selectedTitleBarBackground);
    out.writeObject(m_selectedTitleBarForeground);
    out.writeObject(m_frameBorder);

    // Unchanged code from section 15.7
  }

  public void readExternal(ObjectInput in)
   throws IOException, ClassNotFoundException {
    setTitleBarBackground((Color)in.readObject());
    setTitleBarForeground((Color)in.readObject());
    setSelectedTitleBarBackground((Color)in.readObject());
    setSelectedTitleBarForeground((Color)in.readObject());
    setBorder((Border)in.readObject());

    setTitle((String)in.readObject());

    // Unchanged code from section 15.7

    setFrameIcon((Icon)in.readObject());
    // Unchanged code from section 15.7
  }
}
```

The Code: InnerFrameUI.java
see \Chapter21\3\mdi\plaf

```
package mdi.plaf;

import java.awt.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.*;

import mdi.*;

public class InnerFrameUI extends javax.swing.plaf.PanelUI
{
  private static InnerFrameUI frameUI;

  protected static Color DEFAULT_TITLE_BAR_BG_COLOR;
  protected static Color DEFAULT_SELECTED_TITLE_BAR_BG_COLOR;
  protected static Color DEFAULT_TITLE_BAR_FG_COLOR;
  protected static Color DEFAULT_SELECTED_TITLE_BAR_FG_COLOR;
  protected static Font  DEFAULT_TITLE_BAR_FONT;
```

```
protected static Border DEFAULT_INNER_FRAME_BORDER;
protected static Icon  DEFAULT_FRAME_ICON;

private static Hashtable m_ownDefaults = new Hashtable();

static {
  m_ownDefaults.put("InternalFrame.inactiveTitleBackground",
    new ColorUIResource(108,190,116));
  m_ownDefaults.put("InternalFrame.inactiveTitleForeground",
    new ColorUIResource(Color.black));
  m_ownDefaults.put("InternalFrame.activeTitleBackground",
    new ColorUIResource(91,182,249));
  m_ownDefaults.put("InternalFrame.activeTitleForeground",
    new ColorUIResource(Color.black));
  m_ownDefaults.put("InternalFrame.titleFont",
    new FontUIResource("Dialog", Font.BOLD, 12));
  m_ownDefaults.put("InternalFrame.border",
    new BorderUIResource(new MatteBorder(4, 4, 4, 4, Color.blue)));
  m_ownDefaults.put("InternalFrame.icon",
    new IconUIResource(new ImageIcon("mdi/default.gif")));
}

public static ComponentUI createUI(JComponent c) {
  if(frameUI == null)
    frameUI = new InnerFrameUI();
  try {
    frameUI.installDefaults();
    InnerFrame frame = (InnerFrame)c;
    frame.setTitleBarBackground(DEFAULT_TITLE_BAR_BG_COLOR);
    frame.setSelectedTitleBarBackground(
      DEFAULT_SELECTED_TITLE_BAR_BG_COLOR);
    frame.setTitleBarForeground(DEFAULT_TITLE_BAR_FG_COLOR);
      frame.setSelectedTitleBarForeground(
        DEFAULT_SELECTED_TITLE_BAR_FG_COLOR);
    frame.setTitleBarFont(DEFAULT_TITLE_BAR_FONT);
    frame.setBorder(DEFAULT_INNER_FRAME_BORDER);
    frame.setFrameIcon(DEFAULT_FRAME_ICON);
    if (frame.isShowing())
      frame.repaint();
  }
  catch (Exception ex) {
    System.err.println(ex);
    ex.printStackTrace();
  }

  return frameUI;
}

public void installUI(JComponent c) {
  InnerFrame frame = (InnerFrame)c;
  super.installUI(frame);
}

public void uninstallUI(JComponent c) {
  super.uninstallUI(c);
}

protected void installDefaults() {
  DEFAULT_TITLE_BAR_BG_COLOR = (Color)findDefaultResource(
    "InternalFrame.inactiveTitleBackground");
  DEFAULT_TITLE_BAR_FG_COLOR = (Color)findDefaultResource(
    "InternalFrame.inactiveTitleForeground");
```

```
    DEFAULT_SELECTED_TITLE_BAR_BG_COLOR = (Color)findDefaultResource(
      "InternalFrame.activeTitleBackground");
    DEFAULT_SELECTED_TITLE_BAR_FG_COLOR = (Color)findDefaultResource(
      "InternalFrame.activeTitleForeground");
    DEFAULT_TITLE_BAR_FONT = (Font)findDefaultResource(
      "InternalFrame.titleFont");
    DEFAULT_INNER_FRAME_BORDER = (Border)findDefaultResource(
      "InternalFrame.border");
    DEFAULT_FRAME_ICON = (Icon)findDefaultResource(
      "InternalFrame.icon");
  }

  protected Object findDefaultResource(String id) {
    Object obj = null;
    try {
      UIDefaults uiDef = UIManager.getDefaults();
      obj = uiDef.get(id);
    }
    catch (Exception ex) {
      System.err.println(ex);
    }
    if (obj == null)
      obj = m_ownDefaults.get(id);
    return obj;
  }

  public void paint(Graphics g, JComponent c) {
    super.paint(g, c);
    if (c.getBorder() != null)
      c.getBorder().paintBorder(
    c, g, 0, 0, c.getWidth(), c.getHeight());
  }

  public Color getTitleBarBkColor() {
    return DEFAULT_TITLE_BAR_BG_COLOR;
  }

  public Color getSelectedTitleBarBkColor() {
    return DEFAULT_SELECTED_TITLE_BAR_BG_COLOR;
  }

  public Color getTitleBarFgColor() {
    return DEFAULT_TITLE_BAR_FG_COLOR;
  }

  public Color getSelectedTitleBarFgColor() {
    return DEFAULT_SELECTED_TITLE_BAR_FG_COLOR;
  }

  public Font getTitleBarFont() {
    return DEFAULT_TITLE_BAR_FONT;
  }

  public Border getInnerFrameBorder() {
    return DEFAULT_INNER_FRAME_BORDER;
  }
}
```

Understanding the Code

Class MdiContainer

This class represents a simple frame container similar to the one used in chapter 15 to demonstrate our custom MDI interface. An instance of `JDesktopPane` is set as the content pane for this frame to support `JInternalFrames` as well as our `InnerFrame` component.

The `createMenuBar()` method creates and populates a menu bar for this example. A "File" menu is constructed with three menu items:

- Menu item "New InnerFrame" creates a new instance of `InnerFrame` and adds it to the desktop pane. The `closable, maximizable, iconifiable, and resizable` properties are set by default to demonstrate the maximum number of UI elements used in this component.
- Menu item "New JInternalFrame" creates a new instance of Swing's `JInternalFrame` and adds it to the desktop pane.
- Menu item "Exit" quits this application.

A "Look&Feel" menu is constructed with an array of menu items corresponding to the L&Fs currently installed. These items are handled the same way they were in previous examples and do not require additional explanation here.

Class mdi.InnerFrame

This class was introduced in chapter 15 and requires some minor modifications to support L&F. First note that the new `mdi.plaf` package and `javax.swing.plaf.ComponentUI` class are imported.

As we know, it is typical for L&F-compliant components to not perform any rendering by themselves and not explicitly hold any resources for this process. Instead all rendering and necessary resources involved (colors, icons, borders etc.) should be maintained by a UI delegate corresponding to that component. To conform to this design pattern, we remove several class variables from our `InnerFrame` class and move them to our custom `InnerFrameUI` class (see below). Specifically: `DEFAULT_TITLE_BAR_BG_COLOR, DEFAULT_SELECTED_TITLE_BAR_BG_COLOR,` and `DEFAULT_FRAME_ICON`. We will use resources provided by the currently installed L&F for these variables instead of hard-coded values defined in the component class.

Two class variables (`DEFAULT_BORDER_COLOR` and `DEFAULT_SELECTED_BORDER_COLOR`) have been removed. We don't need them any more since we'll use a `Border` instance to render `InnerFrame`'s border. Eight default icon variables (`ICONIZE_BUTTON_ICON, RESTORE_BUTTON_ICON, CLOSE_BUTTON_ICON, MAXIMIZE_BUTTON_ICON, MINIMIZE_BUTTON_ICON,` and their pressed variants) are intentionally left unchanged. We could move them into our UI delegate class as well, and use standard icons provided by L&F for these variables. But we've decided to preserve some individuality for our custom component.

Note: Swing's `JInternalFrame` delegates provide only one standard icon for the frame controls, whereas our implementation uses two icons for pressed and not pressed states. Since we will use `JInternalFrame` UI delegate resources, this would require us to either remove our two-icon functionality, or construct separate icons for use with each L&F.

Now let's take a look at the instance variables that have been modified:

- `Color m_titleBarBackground`: background color for the title bar of an inactive frame; now initialized by the `InnerFrameUI` delegate.
- `Color m_selectedTitleBarBackground`: background color for the title bar of an active frame; now initialized by the `InnerFrameUI` delegate.
- `Color m_titleBarForeground`: foreground color for the title bar of an inactive frame; now initialized

by the InnerFrameUI delegate. The previous version supported only one foreground color. This version distinguishes between the foreground of active and inactive frames (that is necessary for support of various L&Fs).

- Color m_selectedTitleBarForeground: new variable for foreground color of the title bar of an active frame. Two new methods, setSelectedTitleBarForeground() and getSelectedTitleBarForeground(), support this variable.
- Font m_titleBarFont: new variable for the title bar's font. The previous version used a default font for to render frame's title, but this may not be acceptable for all L&Fs. Two new methods, setTitleBarFont() and getTitleBarFont(), support this variable.
- Border m_frameBorder: new variable for frame's border. The previous version's border was made from the four resizable edge components which were colored homogeneously. In this example we use a shared Border instance provided by the current L&F, and store it in the m_frameBorder variable. Two new methods, setBorder() and getBorder(), support this variable.
- Icon m_frameIcon: this variable was formerly defined as an ImageIcon instance. This may not be acceptable for L&F implementations which provide a default frame icon as a different instance of the Icon interface. So we now declare m_frameIcon as an Icon, resulting in several minor modifications throughout the code.

We have also removed two instance variables: m_BorderColor and m_selectedBorderColor. Their corresponding set/get methods have also been removed. Method updateBorderColors() is left without implementation and can also be removed from the code. The reason for this change is that we no longer support direct rendering of the resize edge components (they are now non-opaque, see below), and instead delegate this functionality to the m_frameBorder instance retrieved from the currently installed L&F.

Note: We also no longer support different borders for active and inactive frames since not all L&Fs provide two Border instances for internal frames.

A significant amount of code needs to be added for look-and-feel support in this component. First note that a static block places two values into UIManager's resource defaults storage. These key/value pairs allow retrieval of the fully-qualified InnerFrameUI class name, and that of the custom delegate of it's inner class title bar button component, InnerFrameButtonUI (for this we simply use BasicButtonUI):

```
static {
  UIManager.getDefaults().put(
    "InnerFrameUI", "mdi.plaf.InnerFrameUI");
  UIManager.getDefaults().put("InnerFrameButtonUI",
    "javax.swing.plaf.basic.BasicButtonUI");
}
```

As we've discussed in the previous examples, for all provided Swing components this information is provided by the concrete sub-classes of LookAndFeel class (added to the defaults table in the initClassDefaults() method). However, these implementations have no knowledge of our custom component, so we must place it in the table ourselves.

Note: In this case we do not need to add a corresponding java.lang.Class instance to to the defaults table, as was necessary when implementing our own LookAndFeel (see 21.1.11).

The createUI() method will be called to create and return an instance of InnerFrame's UI delegate, InnerFrameUI (see below). The setUI() method installs a new InnerFrameUI instance, and the getUI() method retrieves the current delegate (both use the protected ui variable inherited from the JComponent class).

Method updateUI() will be called to notify the component whenever the current L&F changes. Our implementation requests an instance of InnerFrameUI from UIManager (using getUI()), which is then passed to setUI(). Then invalidate() is called to mark this component for revalidation because the new L&F's resources will most likely change the sizing and position of InnerFrame's constituents.

The getUIClassID() method returns a unique String ID identifying this component's base UI delegate name. This String must be consistent with the string used as the key for the fully qualified class name of this component's delegate that was placed in UIManager's resource defaults table (see above):

```
    public String getUIClassID() { return "InnerFrameUI"; }
```

Several simple setXX()/getXX() methods have been added that do not require any explanation. The only exception is setBorder(), which overrides the corresponding JComponent method. The Border parameter is stored in our m_frameBorder variable, to be painted manually by InnerFrameUI, and we do not call the super class implementation. Thus, the border is not used in the typical way we are used to. Specifically, it does not define any insets for InnerFrame. Instead, it is just painted directly on top of it. We do this purposefully because we desire the border to be painted over each resize edge child.

To do this, we make the resize edge components NorthResizeEdge, SouthResizeEdge, EastResizeEdge, and WestResizeEdge transparent, and preserve all functionality (changing the mouse cursor, resizing the frame, etc.). Thus, they form an invisible border whose width is synchronized (in the setBorder() method) with the width of the Border instance stored in our m_frameBorder variable. As you'll see below in our InnerFrameUI class, this Border instance is drawn explicitly on the frame's surface. Figure 21.3 illustrates. In this way we can preserve any decorative elements (usually 3D effects specific to each L&F), and at the same time leave our child resize edge components directly below this border to intercept and process mouse events as normal.

Figure 21.3 Invisible resize edge components around InnerFrame.
<<file figure21-3.gif>>

Note: The setBorder() method checks whether the component is shown on the screen (if isShowing() returns true) before calling validate(). This precaution helps avoid exceptions during creation of the application. A similar check needs to be made before update(), repaint(), etc., calls if they can possibly be invoked before InnerFrame is displayed on the screen.

Method updateTitleBarColors() has lost his importance, as the overridden methods in the title bar provide the current color information (so we don't have to call setXX() methods explicitly each time the color palette is changed).

Method createTitleBar() creates and initializes the title bar for this frame. The anonymous inner class defining our m_titlePanel component receives a new getBackground() method, which returns

getSelectedTitleBarBackground() or getTitleBarBackground() depending on whether the frame is selected or not. Take note of the following code, which may seem strange:

```
    public Color getBackground() {
      if (InnerFrame.this == null)
        return super.getBackground();
      if (isSelected())
        return getSelectedTitleBarBackground();
      else
        return getTitleBarBackground();
    }
```

The reason that a check against null is made here is because this child component will be created before the creation of the corresponding InnerFrame instance is completed. By doing this we can avoid any possible NullPointerExceptions that would occur from calling parent class methods too early.

The m_titleLabel component is also now defined as an anonymous inner class. It overrides two JComponent methods, getForeground() and getFont(). These methods return instance variables from the parent InnerFrame instance. Thus we can avoid keeping track of the font and foreground of this child component, as long as the corresponding instance variables are properly updated. Note that we check the InnerFrame.this reference for null here as well.

Finally, methods writeExternal() and readExternal() have been modified to reflect the changes in the instance variables we have discussed.

Class mdi.InnerFrame.InnerFrameButton

This inner class has received two minor changes. First, we override JComponent's setBorder() and getBorder() methods to hide the default implementation and eliminate the possibility of assigning any border to these small custom buttons (otherwise every time the L&F is changed the standard JButton border will be set).

We also override the getUIClassID() method which returns a string ID representing this child component's UI delegate. Referring back to the static block in the InnerFrame class, we see that this ID is associated with javax.swing.plaf.basic.BasicButtonUI class. Thus, we directly assign the BasicButtonUI class as the delegate for this component instead of allowing the current L&F to take control.

Class mdi.InnerFrame.EastResizeEdge, WestResizeEdge, NorthResizeEdge, SouthResizeEdge

This EastResizeEdge inner class has received a few minor changes. The opaque property is now set to false, and we drop any code for the background color since this component is now transparent. Second, the WIDTH variable is initially set to the right inset of the frame's border. This variable is also accessible using the new setWidth() method, which was constructed primarily for synchronizing the size of this component with the width of the Border drawn around the frame (see InnerFrame's setBorder() method above). Each resize edge component is modified similarly.

Class mdi.plaf.InnerFrameUI

This class extends javax.swing.plaf.PanelUI and defines the custom UI delegate for our InnerFrame component. The basic idea behind this class is to retrieve and reuse the rendering resources defined by the current L&F for JInternalFrame. It is perfectly reasonable to use resources already defined in the LookAndFeel implementations for standard Swing components. By doing so we can more easily provide consistent views of custom components under different L&Fs, and reduce the amount of required coding. However, if we cannot rely on pre-defined resources in a particular LookAndFeel, we need to define our own custom resources. The next section will show how to deal with this situation (which might arise when using third party L&Fs).

Class variables:

`InnerFrameUI frameUI`: a shared instance of this class returned by `createUI()`.

`Color DEFAULT_TITLE_BAR_BG_COLOR`: default background color for the title bar of an inactive frame.

`Color DEFAULT_SELECTED_TITLE_BAR_BG_COLOR`: default background color for the title bar of an active frame.

`Color DEFAULT_TITLE_BAR_FG_COLOR`: default foreground color for the title bar of an inactive frame.

`Color DEFAULT_SELECTED_TITLE_BAR_FG_COLOR`: default foreground color for the title bar of an active frame.

`Font DEFAULT_TITLE_BAR_FONT`: default title bar font.

`Border DEFAULT_INNER_FRAME_BORDER`: default frame border.

`Icon DEFAULT_FRAME_ICON`: default frame icon.

`Hashtable m_ownDefaults`: a collection of resources stored in this UI delegate, and used when a particular resource is not implemented in the current L&F.

Note: The resource variables listed above are default in the sense that they're used unless other values are set explicitly using `setXX()` methods of `InnerFrame` (e.g. `setTitleBarFont()`, `setBorder()`, etc.).

The `static` block defined in this class creates and populates `Hashtable m_ownDefaults`. Method `createUI()` creates the `InnerFrameUI` shared instance (if it is not created yet) and calls `installDefaults()` to refresh its attributes with resources provided by the current L&F. Several `setXX()` methods are then called to update the properties of the specified `InnerFrame` instance.

Methods `installUI()` and `uninstallUI()` simply delegate their calls to the super-class implementation. They are included here because, in some cases, we may need to override these methods (as we saw in the previous section).

Method `installDefaults()` retrieves resources provided by the current L&F by calling our custom `findDefaultResource()` method, and stores them in the class variables. Remember that the keys for these resources are not documented, but can easily be found in the Swing `LookAndFeel` implementation source code. "InternalFrame.inactiveTitleBackground" is used to retrieve the background for `JInternalFrame`'s inactive title bar, "InternalFrame.icon" is used for the `JInternalFrame`'s icon, etc.

The custom method `findDefaultResource()` takes a resource ID `String` and searches for it in `UIManager`'s `UIDefaults` table. If the resource is not found or an exception occurs, a resource stored in our `m_ownDefaults` collection under the same ID is used.

The `paint()` method renders a given component using a given graphical context. As we've discussed above, it explicitly paints the border around `InnerFrame` by calling the border's `paintBorder()` method.

Several remaining `getXX()` methods allow simple retrieval of the default rendering resources defined in this class, and do not require explanation here.

Running the Code

Create several `InnerFrames` and `JInternalFrames` using the "File" menu. Select different L&Fs and note that the appearance of `InnerFrame` changes accordingly. Compare the appearance of `InnerFrame` to `JInternalFrame` in each available L&F. Figures 21.4, 21.5, and 21.6 shows `MdiContainer` displaying two `InnerFrames` and two `JInternalFrames` in the Metal, Windows, and Motif L&Fs respectively.

Bug Alert! The MotifL&F does not supply proper resources for JInternalFrame rendering. A quick look at the MotifLookAndFeel source code shows that no resources are placed in the defaults table corresponding to JInternalFrame. This is why the appearance of InnerFrame is not at all consistent with JInternalFrame under the MotifL&F. (This is actually more of a design flaw than a bug.)

Note: The selection of JInternalFrames and InnerFrames is not synchronized. A single instance of each can be selected at any given time. Also note that because of known flaws in JDesktopPane, InnerFrame (as well as JInternalFrame) will not receive resize events from it, which cripples maximize functionality. Placing these components in an instance of our MDIPane would fix this. However, we would then see null pointer exceptions each time a JInternalFrame's title bar is pressed.

21.5 L&F for custom components: part II - third party L&F support

The previous example demonstrated how to exploit existing L&F to provide custom components with an appearance consistent with the currently installed look-and-feel. But what if we want to support a custom L&F which does not provide any suitable resources? The most direct way, of course, is to adjust the code in the L&F package. This can be done if we have developed both a custom component and a custom L&F ourselves (as we did in 21.2 and 21.3). However, in the case of a custom L&F supplied by a third party vendor, we would need to provide all necessary resources inside our custom component's UI delegate.

The following example shows how to add support for a custom L&F, namely Malachite, to our InnerFrame component. (Recall that MalachiteLF only provides rendering resources for three types of buttons and a border.) We intentionally will not modify, or add to, the original Malachite package (acting as if it is supplied by a third party vendor), and will include all necessary modifications to our component's UI delegate itself.

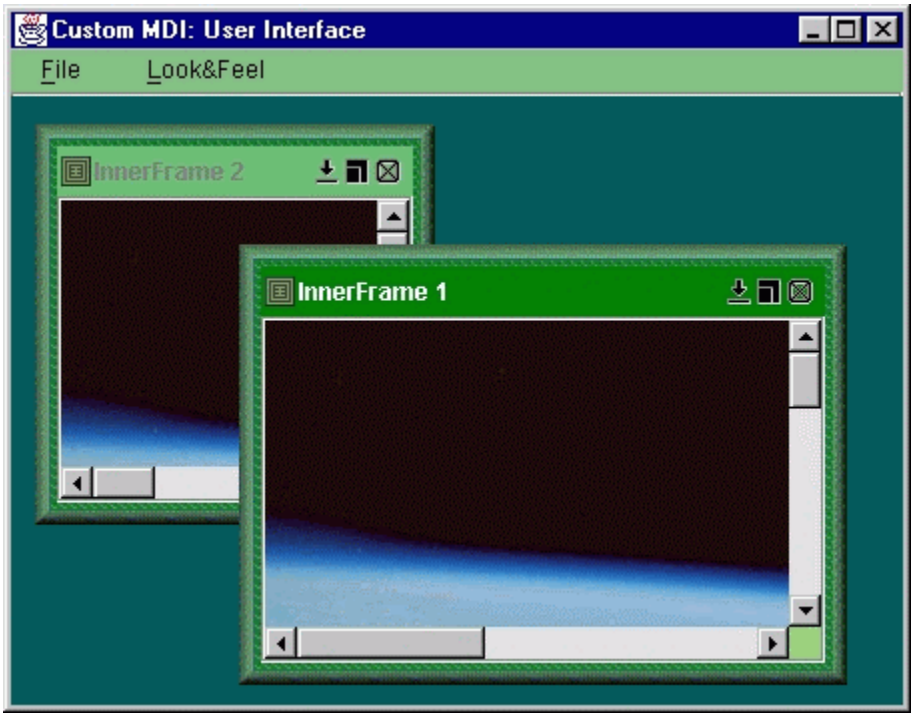


Figure 21.7 InnerFrames in the Malachite L&F.
<<file figure21-7.gif>>

The Code: MdiContainer2.java
see \Chapter21\4

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

import javax.swing.*;

import mdi.*;

public class MdiContainer2 extends JFrame
{
  public MdiContainer2() {
    super("Custom MDI: User Interface");
    getContentPane().setLayout(new FlowLayout());
    // Unchanged code from section 21.4
  }
  // Unchanged code from section 21.4

  public static void main(String argv[]) {
    try {
      LookAndFeel malachite = new Malachite.MalachiteLF();
      UIManager.LookAndFeelInfo info =
        new UIManager.LookAndFeelInfo(malachite.getName(),
        malachite.getClass().getName());
      UIManager.installLookAndFeel(info);
      UIManager.setLookAndFeel(malachite);
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.err.println(ex.toString());
    }

    new MdiContainer2();
  }
}
```

The Code: InnerFrameUI.java
see \Chapter21\4\mdi\plaf

```
package mdi.plaf;

import java.awt.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.*;

import mdi.*;

public class InnerFrameUI extends javax.swing.plaf.PanelUI
{
  // Unchanged code from section 21.4

  public static ComponentUI createUI(JComponent c) {
    LookAndFeel currentLF = UIManager.getLookAndFeel();
    if (currentLF != null && currentLF.getID().equals("Malachite"))
      return mdi.plaf.Malachite.MalachiteInnerFrameUI.createUI(c);

// Remaining code unchanged from section 21.4
```

The Code: MalachiteInnerFrameUI.java
see \Chapter21\4\mdi\plaf\Malachite

```
package mdi.plaf.Malachite;

import java.awt.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.*;

import mdi.*;
import Malachite.*;

public class MalachiteInnerFrameUI extends mdi.plaf.InnerFrameUI
{
  private static MalachiteInnerFrameUI frameUI;

  public static ComponentUI createUI(JComponent c) {
    if(frameUI == null)
      frameUI = new MalachiteInnerFrameUI();
    try {
      frameUI.installDefaults();

      InnerFrame frame = (InnerFrame)c;
      frame.setTitleBarBackground(DEFAULT_TITLE_BAR_BG_COLOR);
      frame.setSelectedTitleBarBackground(
        DEFAULT_SELECTED_TITLE_BAR_BG_COLOR);
      frame.setTitleBarForeground(DEFAULT_TITLE_BAR_FG_COLOR);
      frame.setSelectedTitleBarForeground(
        DEFAULT_SELECTED_TITLE_BAR_FG_COLOR);
      frame.setTitleBarFont(DEFAULT_TITLE_BAR_FONT);
      frame.setBorder(DEFAULT_INNER_FRAME_BORDER);
      frame.setFrameIcon(DEFAULT_FRAME_ICON);
      if (frame.isShowing())
        frame.repaint();
    }
    catch (Exception ex) {
      System.err.println(ex);
      ex.printStackTrace();
    }

    return frameUI;
  }

  protected void installDefaults() {
    DEFAULT_TITLE_BAR_BG_COLOR = new ColorUIResource(108,190,116);
    DEFAULT_TITLE_BAR_FG_COLOR = new ColorUIResource(Color.gray);
    DEFAULT_SELECTED_TITLE_BAR_BG_COLOR =
      new ColorUIResource(0,128,0);
    DEFAULT_SELECTED_TITLE_BAR_FG_COLOR =
      new ColorUIResource(Color.white);
    DEFAULT_TITLE_BAR_FONT = new FontUIResource(
      "Dialog", Font.BOLD, 12);
    Border fb1 = new MalachiteBorder();
    Border fb2 = new MatteBorder(4, 4, 4, 4, new ImageIcon(
      "mdi/plaf/Malachite/body.gif"));
    DEFAULT_INNER_FRAME_BORDER = new BorderUIResource(
      new CompoundBorder(fb1, fb2));
    DEFAULT_FRAME_ICON = new IconUIResource(new ImageIcon(
```

```
      "mdi/plaf/Malachite/icon.gif"));
  }
}
```

Understanding the Code

Class MdiContainer2

This container class is similar to that in the previous example, but now the Malachite L&F is installed just as it was in the examples of sections 21.2 and 21.3.

Class mdi.plaf.InnerFrameUI

Method `createUI()`, which is responsible for creation of this `InnerFrame` UI delegate, has been modified to check the ID of the current L&F. If it matches "Malachite", `createUI()` is invoked in the new `mdi.plaf.Malachite.MalachiteInnerFrameUI` class.

Class mdi.plaf.Malachite.MalachiteInnerFrameUI

This new class, a sub-class of `InnerFrameUI`, provides `InnerFrame` with support for our custom Malachite L&F. Note that it is defined in a separate package, which is recommended for organized support of various different L&Fs.

Since this class extends `mdi.plaf.InnerFrameUI`, only two methods need to be overridden. The `createUI()` method now creates an instance of `MalachiteInnerFrameUI` as a shared instance, and calls `installDefaults()` on that instance. It then assigns the class variables inherited from `InnerFrameUI` using rendering resources appropriate for the Malachite L&F. Particularly the background colors form a green palette, and the frame's border is built as a `CompoundBorder` consisting of a `MalachiteBorder` (on the outside) and a `MatteBorder` drawn using a small green dashed image (on the inside).

Running the Code

Create several `InnerFrames` using the "File" menu, and note that their appearance is in fact consistent with the Malachite L&F. Figure 21.7 illustrates.

Part IV - Special Topics

In the following six chapters we cover several different topics which relate directly to the use of Swing. Chapter 22 discusses the powerful new Java 2 printing API. We construct examples showing how to print an image on multiple pages, construct a print preview component, print styled text, and print `JTable` data (in both portrait and landscape modes). Chapter 23 introduces a few Java2D features. Examples include a generic 2D chart class, a 2D label class, and the beginnings of a Pac-man game. Chapter 24 introduces Accessibility and shows how easy it is to integrate this functionality into existing apps. Chapter 25 covers the basics of the JavaHelp API, and includes examples showing how we can customize the Swing-based help viewer to our liking. Chapter 26 introduces CORBA and contains an example of a client-server, Swing-based app based on our `StocksTable` example from chapter 18. Chapter 27 consists of two examples contributed by experienced Swing developers: constructing custom multi-line labels and tooltips and an internet browser application. Unfortunately, due to space limitations, chapters 24-27 were not included in this edition. However, they remain freely available to all readers on the book's web site.

Chapter 22. Printing

In this chapter:

- Java 2 Printing API overview
- Printing images
- Print preview
- Printing styled text
- Printing tables

22.1 Java 2 Printing API overview

With Java 2 comes a considerably advanced printing API. Java veterans may recall that JDK 1.0 didn't provide printing capabilities at all. JDK 1.1 provided access to native print jobs, but multi-page printing was a real problem for that API.

Now Java developers are able to perform multi-page printing using page count selection and other typical specifications in the native Print dialog, as well as page format selection in the native platform-specific Page Setup dialog. The printing-related API is concentrated in the `java.awt.print` package, and we'll start this chapter with an overview of these classes and interfaces.

Note: At this point the underlying communication with the native printing system is not yet matured. You will notice that some of the examples in this chapter run extremely slow, especially when dealing with `Image`s. We expect these deficiencies to decrease, and the material presented here should be equally applicable in future releases of Java.

22.1.1 PrinterJob

class java.awt.print.PrinterJob

This is the main class which controls printing in Java 2. It is used to store print job properties, to initiate printing when necessary, and to control the display of Print dialogs. A typical printing process is shown in the following code:

```
PrinterJob prnJob = PrinterJob.getPrinterJob();
prnJob.setPrintable(myPrintable);
if (!prnJob.printDialog())
  return;
prnJob.print();
```

This code retrieves an instance of `PrinterJob` with the static `getPrinterJob()` method, passes a `Printable` instance to it (used to render a specific page on demand--see below), invokes a platform-dependent Print dialog by calling `PrinterJob`'s `printDialog()` method, and, if this method returns `true` (indicating the "ok" to print), starts the actual printing process by calling the `print()` method on that `PrinterJob`.

The Print dialog will look familiar, as it is the typical dialog used by most other applications on the user's system. For example, figure 22.1 shows a Windows NT Print dialog:

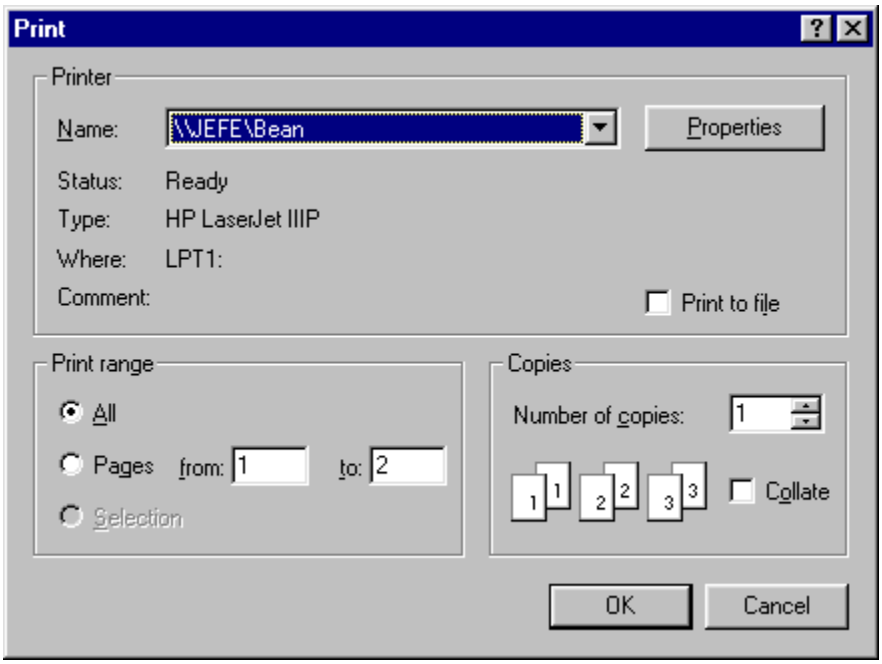


Figure 22.1 Windows NT Print dialog: about to print a pageable job.
<<file figure22-1.gif>>

Though the `PrinterJob` is the most important constituent of the printing process, it can do nothing without a `Printable` instance that specifies how to actually perform the necessary rendering for each page.

22.1.2 The Printable interface

abstract interface java.awt.print.Printable

This interface defines only one method: `print()`, which takes three parameters:

`Graphics graphics`: the graphical context into which the page will be drawn.

`PageFormat pageFormat`: an object containing information about the size and orientation of the page being drawn (see below).

`int pageIndex`: the zero based index of the page to be drawn.

The `print()` method will be called to print a portion of the `PrinterJob` corresponding to a given `pageIndex`. An implementation of this method should perform rendering of a specified page, using a given graphical context and a given `PageFormat`. The return value from this method should be `PAGE_EXISTS` if the page is rendered successfully, or `NO_SUCH_PAGE` if the given page index is too large and does not exist. (These are static `int`s defined in `Printable`.)

Note: we never call a `Printable`'s `print()` method ourselves. This is handled deep inside the actual platform-specific `PrinterJob` implementation which we aren't concerned with here.

A class that implements `Printable` is said to be a page painter. When a `PrinterJob` uses only one page painter to print each page it is referred to as a printable job. The notion of a document as being separated into a certain number of pages is not predefined in a printable job. In order to print a specific page, a printable job will actually render all pages leading up to that page first, and then it will print the specified page. This is because it does not maintain information about how much space each page will occupy when rendered with the given page painter. For example, if we specify, in our Print dialog, that we want to print pages 3 and 5 only, then pages 0 through 4 (because pages are 0-indexed) will be rendered with the `print()` method, but only 2

and 4 will actually be printed.

Warning: Since the system only knows how many pages a printable job will span after the rendering of the complete document takes place (i.e. after `paint()` has been called), Print dialogs will not display the correct number of pages to be printed. This is because there is no pre-print communication between a `PrinterJob` and the system that determines how much space the printable job requires. For this reason you will often see a range such as 1 to 9999 in Print dialogs when printing printable jobs. (This is not the case for pageable jobs--see below.)

In reality, it is often the case that `print()` will be called for each page more than once. From a draft of an overview of the Java Printing API: "This *callback* printing model is necessary to support printing on a wide range of printers and systems...This model also enables printing to a bitmap printer from a computer that doesn't have enough memory or disk space to buffer a full-page bitmap. In this situation, a page is printed as a series of small bitmaps or *bands*. For example, if only enough memory to buffer one tenth of a page is available, the page is divided into ten bands. The printing system asks the application to render each page ten times, once to fill each band. The application does not need to be aware of the number or size of the bands; it simply must be able to render each page when requested." -- http://java.sun.com/printing/jdk1.2/index.html

Though this explains some of the performance problems that we will see in the coming examples, it seems that the model described above is not exactly what we are dealing with in Java 2 FCS. In fact, after some investigation*, it turns out that the division into bands is not based on available memory. Rather, a hard-coded 512k buffer is used. By increasing the size of this buffer, it is feasible to increase performance significantly. However, this would involve modification of peer-level classes; something that we are certainly not encouraged to do. We hope to see this limitation accounted for in future releases.

22.1.3 The Pageable interface

abstract interface java.awt.print.Pageable

It is possible to support multiple page painters in a single `PrinterJob`. As we know, each page printer can correspond to a different scheme of printing because each `Printable` implements its own `print()` method. Implementations of the `Pageable` interface are designed to manage groups of page painters, and a print job that uses multiple page painters is referred to as a pageable job. Each page in a pageable job can use a different page printer and `PageFormat` (see below) to perform its rendering.

Unlike printable jobs, pageable jobs do maintain the predefined notion of a document as a set of separate pages. For this reason pages of a pageable job can be printed in any order without the necessity of rendering all pages leading up to a specific page (as is the case with printable jobs). Also, because a `Pageable` instance carries with it an explicit page count, this can be communicated to the native printing system when a `PrinterJob` is established. So when printing a pageable job the native Print dialog will know the correct range of pages to display, unlike a printable job. (Note that this does not mean pageable jobs are not subject to the inherent limitations described above; we will see the same repetitive calling of `print()` that we do in printable jobs.)

When constructing a pageable `PrinterJob`, instead of calling `PrinterJob`'s `setPrintable()` method (see section 22.1.1 above), we call its `setPageable()` method. Figure 22.1 shows a Windows NT Print dialog about to print a pageable job. Notice that the range of pages is not 1 to 9999.

We won't be working with pageable jobs in this chapter because all the documents we will be printing only require one `Printable` implementation, even if documents can span multiple pages. In most real-world applications, each page of a document is printed with identical orientation, margins, and other sizing

---

* Thanks to John Sullivan of WebScope, Inc. for his valuable detective work.

characterstics. However, if greater flexibility is desired, `Pageable` implementations such as `Book` (see below) can be useful.

22.1.4 The PrinterGraphics interface

abstract interface java.awt.print.PrinterGraphics

This interface defines only one method: `getPrinterJob()`, which retrieves the `PrinterJob` instance controlling the current printing process. It is implemented by `Graphics` objects that are passed to `Printable` objects to render a page. (We will not need to use this interface at all, as it is used deep inside `PrinterJob` instances to define `Graphics` objects passed to each `Printable`'s `paint()` method during printing.)

22.1.5 PageFormat

class java.awt.print.PageFormat

This class encapsulates a `Paper` object and adds to it an orientation property ('landscape or portrait). We can force a `Printable` to use a specific `PageFormat` by passing one to `PrinterJob`'s overloaded `setPrintable()` method. For instance, the following would force a printable job to use a specific `PageFormat` with a landscape orientation:

```
PrinterJob prnJob = PrinterJob.getPrinterJob();
PageFormat pf = job.defaultPage();
pf.setOrientation(PageFormat.LANDSCAPE);
prnJob.setPrintable(myPrintable, pf);
if (!prnJob.printDialog())
  return;
prnJob.print();
```

`PageFormat` defines three orientations:

- `LANDSCAPE`: The origin is at the bottom left-hand corner of the paper with x axis pointing up and y axis pointing to the right.
- `PORTRAIT` (most common): The origin is at the top left-hand corner of the paper with x axis pointing to the right and y axis pointing down.
- `REVERSE_LANDSCAPE`: The origin is at the top right-hand corner of the paper with x axis pointing down and y axis pointing to the left.

We can optionally display a page setup dialog in which the user can specify page characteristics such as orientation, paper size, margin size, etc. This dialog will return a new `PageFormat` to use in printing. The page setup dialog is meant to be presented before the Print dialog and can be displayed using `PrinterJob`'s `pageDialog()` method. The following code brings up a page setup dialog, and uses the resulting `PageFormat` for printing a printable job:

```
PrinterJob prnJob = PrinterJob.getPrinterJob();
PageFormat pf = job.pageDialog(job.defaultPage());
prnJob.setPrintable(myPrintable, pf);
if (!prnJob.printDialog())
  return;
prnJob.print();
```

Note that we need to pass the `pageDialog()` method a `PageFormat` instance, as it uses it to clone and modify as the user specifies. If the changes are accepted the cloned and modifed version is returned. If they are not, the original version passed in is returned. Figure 22.2 shows a Windows NT page setup dialog:

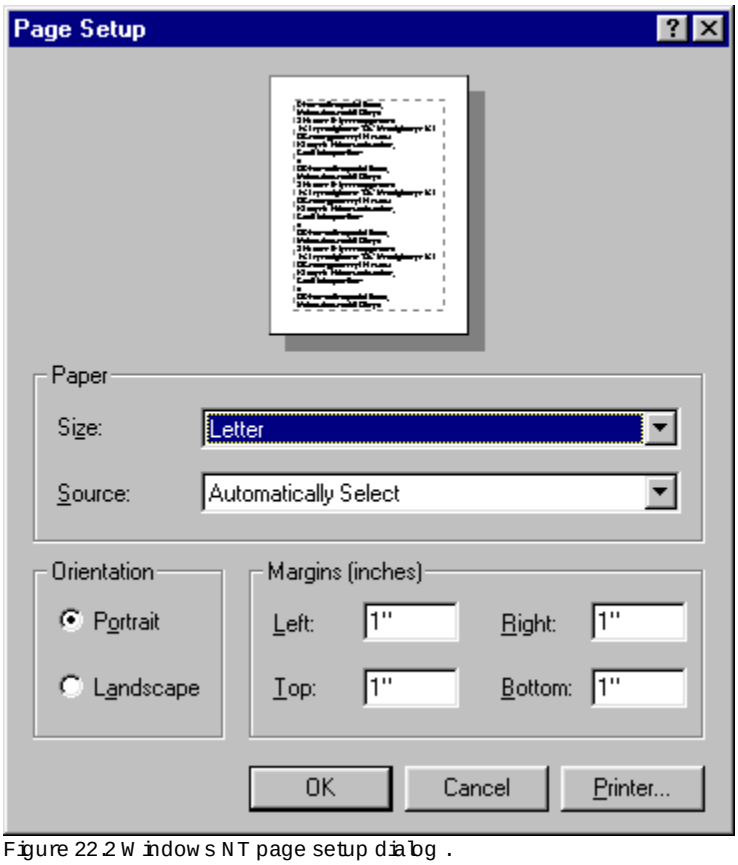


Figure 22.2 Windows NT page setup dialog.
<<file figure22-2.gif>>

22.1.6 Paper

class java.awt.print.Paper

This class holds the size and margins of the paper used for printing. Methods `getImageableX()` and `getImageableY()` retrieve the coordinates of the top-left corner of the printable area in 1/72nds of an inch (which is approximately equal to one screen pixel--referred to as a "point" in typography). Methods `getImageableWidth()` and `getImageableHeight()` retrieve the width and height of the printable area (also in 1/72nds of an inch). We can also change the size of the useable region of the paper using its `setImageableArea()` method.

We can access the `Paper` object associated with a `PageFormat` using `PageFormat's getPaper()` and `setPaper()` methods.

22.1.7 Book

class java.awt.print.Book

This class represents a collection of `Printable` instances with corresponding `PageFormats` to represent a complex document whose pages may have different formats. The `Book` class implements the `Pageable` interface, and `Printables` are added to a `Book` using one of its `append()` methods. This class also defines several methods allowing for the manipulation and replacement of specific pages. (A page in terms of a `Book` is a `Printable-PageFormat` pair. Each page does correspond to an actual printed page.) See the API docs and the Java Tutorial for more information about this class.

22.1.8 PrinterException

class java.awt.print.PrinterException

This exception may be thrown to indicate an error during a printing procedure. It has two concrete sub-classes: `PrinterAbortException` and `PrinterIOException`. The former indicates that a print job was terminated by the application or user while printing, and the latter indicates that there was a problem outputting to the printer.

Reference: For more information about the printing API and features that are expected to be implemented in future versions, refer to the Java tutorial.

22.2 Printing images

In this section we add printing capabilities to the `JPEGEditor` application introduced in chapter 13. This example will form a solid basis for the subsequent printing examples. Here we show how to implement the `Printable` interface to construct a custom panel with a `print()` method that can manage the printing of large images by splitting them up into a matrix of pages.

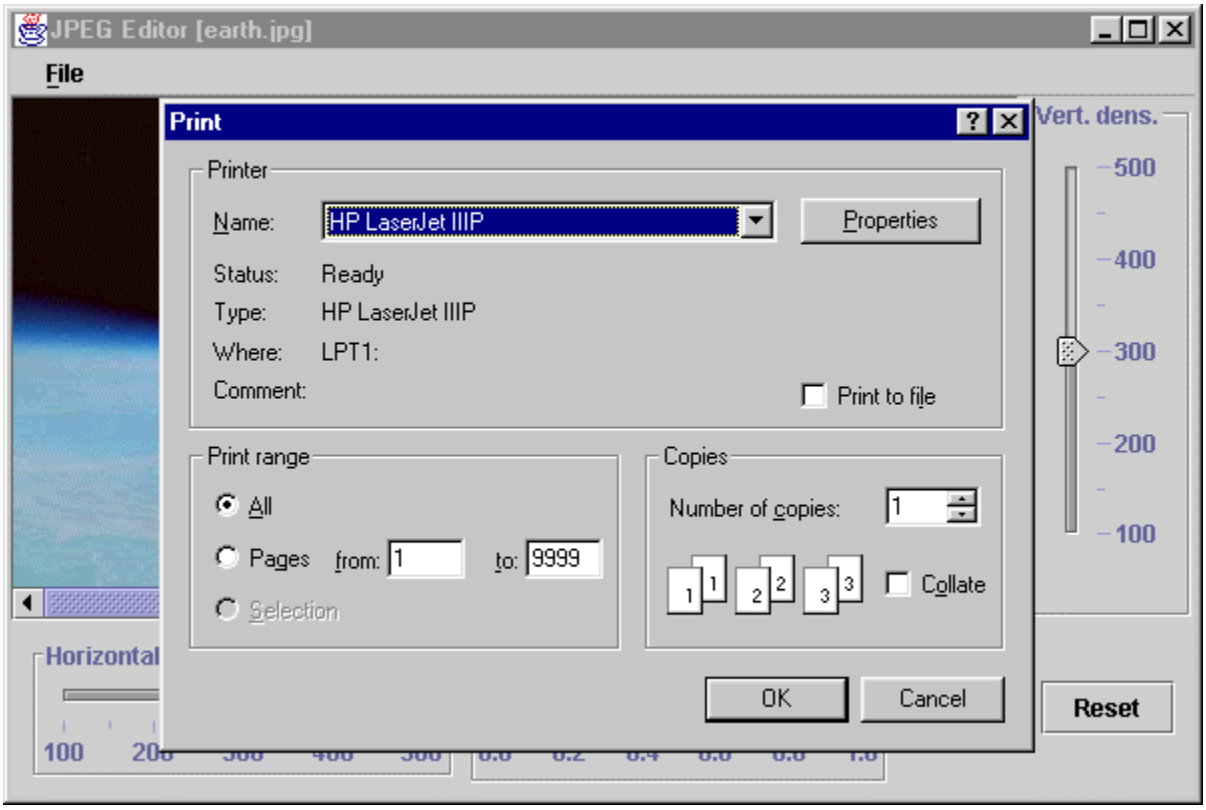


Figure 22.3 Running JPEGEditor example displaying native Print dialog.
<<figure22-3.gif>>

The Code: JPEGEditor.java
see \Chapter22\1

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

```
import java.io.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.filechooser.*;

import com.sun.image.codec.jpeg.*;

import java.awt.print.*;

// Unchanged code from section 13.4

public class JPEGEditor extends JFrame
{
  // Unchanged code from section 13.4

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 13.4

    mItem = new JMenuItem("Print...");
    mItem.setMnemonic('p');
    ActionListener lstPrint = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Thread runner = new Thread() {
          public void run() {
            if (m_panel.getBufferedImage() != null)
              printData();
          }
        };
        runner.start();
      }
    };
    mItem.addActionListener(lstPrint);
    mFile.add(mItem);
    mFile.addSeparator();

    mItem = new JMenuItem("Exit");
    mItem.setMnemonic('x');
    lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        System.exit(0);
      }
    };
    mItem.addActionListener(lst);
    mFile.add(mItem);
    menuBar.add(mFile);
    return menuBar;
  }

  // Unchanged code from section 13.4

  public void printData() {
    getJMenuBar().repaint();
    try {
      PrinterJob prnJob = PrinterJob.getPrinterJob();
      prnJob.setPrintable(m_panel);
      if (!prnJob.printDialog())
        return;
      setCursor( Cursor.getPredefinedCursor(
        Cursor.WAIT_CURSOR));
      prnJob.print();
```

```
      setCursor( Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
      JOptionPane.showMessageDialog(this,
        "Printing completed successfully", "JPEGEditor2",
        JOptionPane.INFORMATION_MESSAGE);
    }
    catch (PrinterException e) {
      e.printStackTrace();
      System.err.println("Printing error: "+e.toString());
    }
  }

  public static void main(String argv[]) {
    new JPEGEditor();
  }
}

class JPEGPanel extends JPanel implements Printable
{
  protected BufferedImage m_bi = null;

  public int m_maxNumPage = 1;

  // Unchanged code from section 13.4

  public int print(Graphics pg, PageFormat pageFormat,
   int pageIndex) throws PrinterException {
    if (pageIndex >= m_maxNumPage || m_bi == null)
      return NO_SUCH_PAGE;

    pg.translate((int)pageFormat.getImageableX(),
      (int)pageFormat.getImageableY());
    int wPage = (int)pageFormat.getImageableWidth();
    int hPage = (int)pageFormat.getImageableHeight();

    int w = m_bi.getWidth(this);
    int h = m_bi.getHeight(this);
    if (w == 0 || h == 0)
      return NO_SUCH_PAGE;
    int nCol = Math.max((int)Math.ceil((double)w/wPage), 1);
      int nRow = Math.max((int)Math.ceil((double)h/hPage), 1);
    m_maxNumPage = nCol*nRow;

    int iCol = pageIndex % nCol;
    int iRow = pageIndex / nCol;
    int x = iCol*wPage;
    int y = iRow*hPage;
    int wImage = Math.min(wPage, w-x);
    int hImage = Math.min(hPage, h-y);

    pg.drawImage(m_bi, 0, 0, wImage, hImage,
      x, y, x+wImage, y+hImage, this);
    System.gc();

    return PAGE_EXISTS;
  }
}
```

Understanding the Code

Class JPEGEditor

The `java.awt.print` package is imported to provide printing capabilities. A new menu item titled "Print..." has been added to the "File" menu of this application. If this item is selected and an image has been loaded, our new custom `printData()` method is called.

The `printData()` method retrieves a `PrinterJob` instance and passes it our `m_panel` component (an instance of `JPEGPanel` -- which now implements the `Printable` interface, see below). It then invokes a native Print dialog and initializes printing by calling `print()`. If no exception was thrown, a "Printing completed successfully" message is displayed when printing completes. Otherwise the exception trace is printed.

Class JPEGPanel

This class, which was originally designed to just display an image, now implements the `Printable` interface and is able to print a portion of its displayed image upon request. A new instance variable, `m_maxNumPage`, holds a maximum page number available for this printing. This number is set initially to one and its actual value is calculated in the `print()` method (see below).

The `print()` method prints a portion of the current image corresponding to the given page index. If the current image is larger than a single page, it will be split into several pages which are arranged as several rows and columns (a matrix). When printed they can be placed in this arrangement to form one big printout.

First this method shifts the origin of the graphics context to take into account the page's margins, and calculates the width and height of the area available for drawing: `wPage` and `hPage`.

```
        pg.translate((int)pageFormat.getImageableX(),
            (int)pageFormat.getImageableY());
        int wPage = (int)pageFormat.getImageableWidth();
        int hPage = (int)pageFormat.getImageableHeight();
```

Local variables `w` and `h` represent the width and height of the whole `BufferedImage` to be printed. (If any of these happens to be `0` we return `NO_SUCH_PAGE`.) Comparing these dimensions with the width and height of a single page, we can calculate the number of columns (not less than `1`) and rows (not less than `1`) in which the original image should be split to fit to the page's size:

```
        int nCol = Math.max((int)Math.ceil((double)w/wPage), 1);
        int nRow = Math.max((int)Math.ceil((double)h/hPage), 1);
        m_maxNumPage = nCol*nRow;
```

The product of rows and columns gives us the number of pages in the print job, `m_maxNumPage`.

Now, because we know the index of the current page to be printed (it was passed as parameter `pageIndex`) we can determine the current column and row indices (note that enumeration is made from left to right and then from top to bottom), `iCol` and `iRow`:

```
        int iCol = pageIndex % nCol;
        int iRow = pageIndex / nCol;
        int x = iCol*wPage;
        int y = iRow*hPage;
        int wImage = Math.min(wPage, w-x);
        int hImage = Math.min(hPage, h-y);
```

We also can calculate the coordinates of the top-left corner of the portion of the image to be printed on this page (`x` and `y`), and the width and height of this region (`wImage` and `hImage`). Note that in the last column or row of our image matrix, the width and/or height of a portion can be less then the maximum values (which we calculated above--`wPage` and `hPage`).

Now we have everything ready to actually print a region of the image to the specified graphics context. We now need to extract this region and draw it at (0,0), as this will be the origin (upper-left hand corner) of our printed page. The `Graphics drawImage()` method does the job. It takes ten parameters: an `Image` instance, four coordinates of the destination area (top-left and bottom-right--not width and height), four coordinates of the source area, and an `ImageObserver` instance.

```
        pg.drawImage(m_bi, 0, 0, wImage, hImage,
            x, y, x+wImage, y+hImage, this);
        System.gc();
```

Note: Because the `print()` method may be called many times for the same page (see below), it makes good sense to explicitly invoke the garbage collector in this method. Otherwise we may run out of memory.

Running the Code

Figure 22.2 shows a Page Setup dialog brought up by our program when run on a Windows NT platform. Be aware that the print job could take up to 15 minutes to print (and this assumes you don't run out of memory first)!

As we mentioned in the beginning of this chapter, the Java 2 printing environment is not yet fully matured. It doesn't work with all printers as expected, so writing and debugging printing applications may be difficult in many cases (a print preview capability is great help, as we will see in the next section). Also note that because we are using a printable job and not a pageable job, the page range is displayed as 1 to 9999 (see section 22.1.2--this may differ depending on your platform).

The most annoying thing with Java 2 printing is that is terribly slow. This is mainly because we are dealing with an `Image` (`BufferedImage` is a subclass of `Image`). Images and printing clash severely. As we will see in later examples, printing is much faster when `Image`s are not involved.

The size of a relatively simple print job spooled to the printer may be unreasonably large. This makes Java 2 printing applications hardly comparable with native applications (at least at the time of this writing). Be sure to have plenty of memory, time, and patience when running this example. Or, alternatively, wait for the next Java 2 release.

Note: It is recommended that the `DoubleBuffered` property of components be set to `false` during printing if the `print()` method directly calls a component's `paint()` method. Note that it is only safe to call `paint()` from the `print()` method if we are sure that `print()` is executing in the AWT event dispatching thread. Refer back to chapter 2 for how to shut off double-buffering, and how to check if a method is running within the AWT event-dispatching thread.

22.3 Print preview

Print preview functionality has became a standard service provided by most modern print-enabled applications. It only makes sense to include this service in Java 2 applications. The example in this section shows how to construct a print preview component.

Note: An additional reason for Java developers to add print preview to their programs is that this service can be very useful for debugging print code. Slow performance of the Java printing API can make debugging impractical using an actual printer.

The print preview component displays small images of the printed pages as they would appear after printing.

A GUI attached to the preview component typically allows for changing the scale of the preview images and invoking a print. The following example demonstrates such a component which can be easily added to any print-aware Swing application.

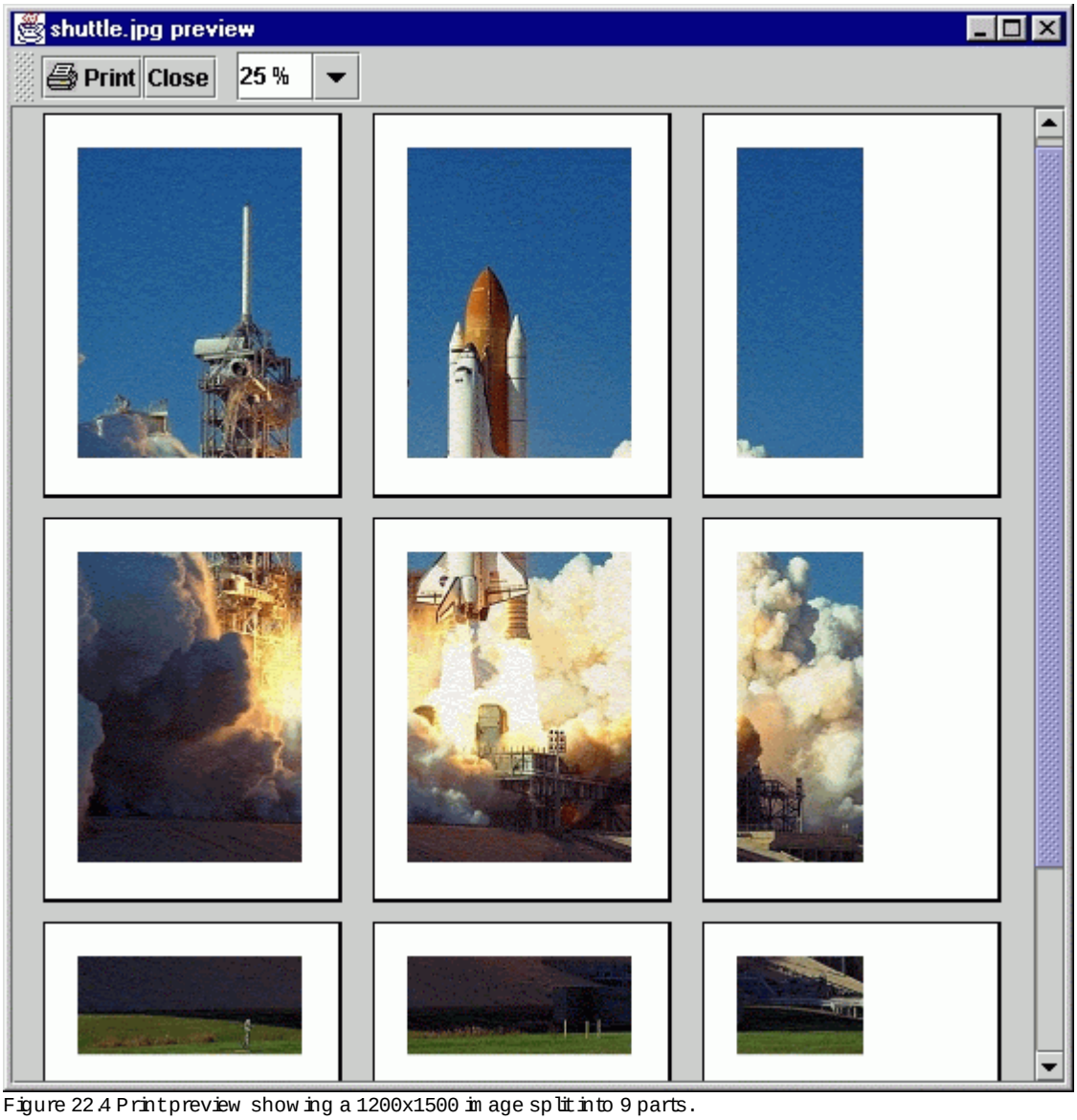


Figure 22.4 Print preview showing a 1200x1500 image split into 9 parts.

<<figure22-4.gif>>

The Code: JPEGEditor.java
see \Chapter22\2

```
public class JPEGEditor extends JFrame
{
  // Unchanged code from section 22.2

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 22.2

    mItem = new JMenuItem("Print Preview");
    mItem.setMnemonic('v');
    ActionListener lstPreview = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
```

```
        Thread runner = new Thread() {
          public void run() {
            setCursor(Cursor.getPredefinedCursor(
              Cursor.WAIT_CURSOR));
            if (m_panel.getBufferedImage() != null)
              new PrintPreview(m_panel,
               m_currentFile.getName()+" preview");
            setCursor(Cursor.getPredefinedCursor(
              Cursor.DEFAULT_CURSOR));
          }
        };
        runner.start();
      }
    };
    mItem.addActionListener(lstPreview);
    mFile.add(mItem);

    mFile.addSeparator();

// The rest of the code is unchanged from section 22.2
```

The Code: PrintPreview.java
see \Chapter22\2

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.util.*;
import java.awt.print.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

public class PrintPreview extends JFrame
{
  protected int m_wPage;
  protected int m_hPage;
  protected Printable m_target;
  protected JComboBox m_cbScale;
  protected PreviewContainer m_preview;

  public PrintPreview(Printable target) {
    this(target, "Print Preview");
  }

  public PrintPreview(Printable target, String title) {
    super(title);
    setSize(600, 400);
    m_target = target;

    JToolBar tb = new JToolBar();
    JButton bt = new JButton("Print", new ImageIcon("print.gif"));
    ActionListener lst = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        try {
          // Use default printer, no dialog
          PrinterJob prnJob = PrinterJob.getPrinterJob();
          prnJob.setPrintable(m_target);
          setCursor( Cursor.getPredefinedCursor(
            Cursor.WAIT_CURSOR));
```

```
      prnJob.print();
      setCursor( Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
      dispose();
    }
    catch (PrinterException ex) {
      ex.printStackTrace();
      System.err.println("Printing error: "+ex.toString());
    }
  }
};
bt.addActionListener(lst);
bt.setAlignmentY(0.5f);
bt.setMargin(new Insets(4,6,4,6));
tb.add(bt);

bt = new JButton("Close");
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    dispose();
  }
};
bt.addActionListener(lst);
bt.setAlignmentY(0.5f);
bt.setMargin(new Insets(2,6,2,6));
tb.add(bt);

String[] scales = { "10 %", "25 %", "50 %", "100 %" };
m_cbScale = new JComboBox(scales);
lst = new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    Thread runner = new Thread() {
      public void run() {
        String str = m_cbScale.getSelectedItem().
          toString();
        if (str.endsWith("%"))
          str = str.substring(0, str.length()-1);
        str = str.trim();
          int scale = 0;
        try { scale = Integer.parseInt(str); }
        catch (NumberFormatException ex) { return; }
        int w = (int)(m_wPage*scale/100);
        int h = (int)(m_hPage*scale/100);

        Component[] comps = m_preview.getComponents();
        for (int k=0; k<comps.length; k++) {
          if (!(comps[k] instanceof PagePreview))
            continue;
          PagePreview pp = (PagePreview)comps[k];
            pp.setScaledSize(w, h);
        }
        m_preview.doLayout();
        m_preview.getParent().getParent().validate();
      }
    };
    runner.start();
  }
};
m_cbScale.addActionListener(lst);
m_cbScale.setMaximumSize(m_cbScale.getPreferredSize());
m_cbScale.setEditable(true);
tb.addSeparator();
```

```
    tb.add(m_cbScale);
    getContentPane().add(tb, BorderLayout.NORTH);

    m_preview = new PreviewContainer();

    PrinterJob prnJob = PrinterJob.getPrinterJob();
    PageFormat pageFormat = prnJob.defaultPage();
    if (pageFormat.getHeight()==0 || pageFormat.getWidth()==0) {
      System.err.println("Unable to determine default page size");
        return;
    }
    m_wPage = (int)(pageFormat.getWidth());
    m_hPage = (int)(pageFormat.getHeight());
    int scale = 10;
    int w = (int)(m_wPage*scale/100);
    int h = (int)(m_hPage*scale/100);

    int pageIndex = 0;
    try {
      while (true) {
        BufferedImage img = new BufferedImage(m_wPage,
          m_hPage, BufferedImage.TYPE_INT_RGB);
        Graphics g = img.getGraphics();
        g.setColor(Color.white);
        g.fillRect(0, 0, m_wPage, m_hPage);
        if (target.print(g, pageFormat, pageIndex) !=
         Printable.PAGE_EXISTS)
          break;
        PagePreview pp = new PagePreview(w, h, img);
        m_preview.add(pp);
        pageIndex++;
      }
    }
    catch (PrinterException e) {
      e.printStackTrace();
      System.err.println("Printing error: "+e.toString());
    }

    JScrollPane ps = new JScrollPane(m_preview);
    getContentPane().add(ps, BorderLayout.CENTER);

    setDefaultCloseOperation(DISPOSE_ON_CLOSE);
    setVisible(true);
  }

  class PreviewContainer extends JPanel
  {
    protected int H_GAP = 16;
    protected int V_GAP = 10;

    public Dimension getPreferredSize() {
      int n = getComponentCount();
      if (n == 0)
        return new Dimension(H_GAP, V_GAP);
      Component comp = getComponent(0);
      Dimension dc = comp.getPreferredSize();
      int w = dc.width;
      int h = dc.height;

      Dimension dp = getParent().getSize();
      int nCol = Math.max((dp.width-H_GAP)/(w+H_GAP), 1);
      int nRow = n/nCol;
```

```
    if (nRow*nCol < n)
      nRow++;

    int ww = nCol*(w+H_GAP) + H_GAP;
    int hh = nRow*(h+V_GAP) + V_GAP;
    Insets ins = getInsets();
    return new Dimension(ww+ins.left+ins.right,
      hh+ins.top+ins.bottom);
  }

  public Dimension getMaximumSize() {
    return getPreferredSize();
  }

  public Dimension getMinimumSize() {
    return getPreferredSize();
  }

  public void doLayout() {
    Insets ins = getInsets();
    int x = ins.left + H_GAP;
    int y = ins.top + V_GAP;

    int n = getComponentCount();
    if (n == 0)
      return;
    Component comp = getComponent(0);
    Dimension dc = comp.getPreferredSize();
    int w = dc.width;
    int h = dc.height;

    Dimension dp = getParent().getSize();
    int nCol = Math.max((dp.width-H_GAP)/(w+H_GAP), 1);
    int nRow = n/nCol;
    if (nRow*nCol < n)
      nRow++;

    int index = 0;
    for (int k = 0; k<nRow; k++) {
      for (int m = 0; m<nCol; m++) {
        if (index >= n)
          return;
        comp = getComponent(index++);
        comp.setBounds(x, y, w, h);
        x += w+H_GAP;
      }
      y += h+V_GAP;
      x = ins.left + H_GAP;
    }
  }
}

class PagePreview extends JPanel
{
  protected int m_w;
  protected int m_h;
  protected Image m_source;
  protected Image m_img;

  public PagePreview(int w, int h, Image source) {
    m_w = w;
    m_h = h;
```

```
      m_source= source;
      m_img = m_source.getScaledInstance(m_w, m_h,
        Image.SCALE_SMOOTH);
      m_img.flush();
      setBackground(Color.white);
      setBorder(new MatteBorder(1, 1, 2, 2, Color.black));
    }

    public void setScaledSize(int w, int h) {
      m_w = w;
      m_h = h;
      m_img = m_source.getScaledInstance(m_w, m_h,
        Image.SCALE_SMOOTH);
      repaint();
    }

    public Dimension getPreferredSize() {
      Insets ins = getInsets();
      return new Dimension(m_w+ins.left+ins.right,
        m_h+ins.top+ins.bottom);
    }

    public Dimension getMaximumSize() {
      return getPreferredSize();
    }

    public Dimension getMinimumSize() {
      return getPreferredSize();
    }

    public void paint(Graphics g) {
      g.setColor(getBackground());
      g.fillRect(0, 0, getWidth(), getHeight());
      g.drawImage(m_img, 0, 0, this);
      paintBorder(g);
    }
  }
}
```

Understanding the Code

Class JPEGEditor

Compared to the previous example this class has only one difference: it creates a menu item titled "Print Preview." When selected, this item creates an instance of the `PrintPreview` class (see below). This class's constructor takes two parameters: a reference to a `Printable` instance and a text string for the frame's title. As we have seen in the previous example, our `m_panel` component implements the `Printable` interface and provides the actual printing functionality, so we use it to create the `PrintPreview` instance. Note that this call is wrapped in a thread because, when used with large images, creation of a `PrintPreview` instance can take a significant amount of time.

Note: As you can see, we only need to have a reference to an instance of the `Printable` interface to create a `PrintPreview` component. Thus, this component can be added to any print-aware application with only a couple lines of code. We will use it in the remaining examples as well, because it is such a simple feature to add.

Class PrintPreview

This class represents a `JFrame`-based component which is capable of displaying the results of printing before

actual printing occurs. Several instance variables are used:

- `Printable m_target`: an object whose printout will be previewed.
- `int m_wPage`: width of the default printing page.
- `int m_hPage`: height of the default printing page.
- `JComboBox m_cbScale`: combobox which selects a scale for preview.
- `PreviewContainer m_preview`: container which holds previewing pages.

Two `public` constructors are provided. The first one takes an instance of the `Printable` interface and passes control to the second constructor, using the `Printable` along with the "Print Preview" `String` as parameters. The second constructor takes two parameters: an instance of the `Printable` interface and the title string for the frame. This second constructor is the one that actually sets up the `PrintPreview` component.

First, a toolbar is created and a button titled "Print" is added to perform printing of the `m_target` instance as described in the previous example. The only difference is that no Print dialog is invoked, and the default system printer is used (this approach is typical for print preview components). When the printing is complete, this print preview component is disposed. The second button added to the toolbar is labeled "Close" and merely disposes of this frame component.

The third (and the last) component added to the toolbar is the editable combobox `m_cbScale`, which selects a percent scale to zoom the previewed pages. Along with several pre-defined choices (`10%`, `25%`, `50%`, and `100%`) any percent value can be entered. As soon as that value is selected and the corresponding `ActionListener` involved, the zoom scale value is extracted and stored in the local variable `scale`. This determines the width and height of each `PreviewPage` component we will be creating:

```
int w = (int)(m_wPage*scale/100);
int h = (int)(m_hPage*scale/100);
```

Then all child components of the `m_preview` container in turn are cast to `PagePreview` components (each child is expected to be a `PagePreview` instance, but `instanceof` is used for precaution), and the `setScaledSize()` method is invoked to assign a new size to the preview pages. Finally `doLayout()` is invoked on `m_preview` to lay out the resized child components, and `validate()` is invoked on the scroll pane. This scroll pane is the parent of the `m_preview` component in the second generation (the first parent is a `JViewport` component--see chapter 7). This last call is necessary to display/hide scrollbars as needed for the new size of the `m_preview` container. This whole process is wrapped in a thread to avoid clogging up the AWT event-dispatching thread.

When toolbar construction is complete, the `m_preview` component is created and filled with the previewed pages. To do so we first retrieve a `PrinterJob` instance for a default system printer without displaying a Page Setup dialog, and retrieve a default `PageFormat` instance. We use this to determine the initial size of the previewed pages by multiplying its dimensions by the computed scaling percentile (which is 10% at initialization time, because `scale` is set to 10).

To create these scalable preview pages we set up a `while` loop to continuously call the `print()` method of the given `Printable` instance, using a page index that gets incremented each iteration, until it returns something other than `Printable.PAGE_EXISTS`.

Each page is rendered into a separate image in memory. To do this, an instance of `BufferedImage` is created with width `m_wPage` and height `m_hPage`. A `Graphics` instance is retrieved from that image using `getGraphics()`:

```
BufferedImage img = new BufferedImage(m_wPage,
```

```
            m_hPage, BufferedImage.TYPE_INT_RGB);
        Graphics g = img.getGraphics();
        g.setColor(Color.white);
        g.fillRect(0, 0, m_wPage, m_hPage);
        if (target.print(g, pageFormat, pageIndex) !=
            Printable.PAGE_EXISTS)
            break;
```

After filling the image's area with a white background (most paper is white), this `Graphics` instance, along with the `PageFormat` and current page index, `pageIndex`, are passed to the `print()` method of the `Printable` object.

Note: The `BufferedImage` class in the `java.awt.image` package allows direct image manipulation in memory. This class will be discussed in more detail in Chapter 23, as well as other classes from the Java 2 2D API.

If the call to the `print()` method returns `PAGE_EXISTS`, indicating success in the rendering of the new page, a new `PagePreview` component is created:

```
        PagePreview pp = new PagePreview(w, h, img);
        m_preview.add(pp);
        pageIndex++;
```

Note that our newly created `BufferedImage` is passed to the `PagePreview` constructor as one of the parameters. This is so that we can use it now and in the future for scaling each `PagePreview` component separately. The other parameters are the width and height to use, which, at creation time, are 10% of the page size (as discussed above).

Each new component is added to our `m_preview` container. Finally, when the `Printable`'s `print()` method finishes, our `m_preview` container is placed in a `JScrollPane` to provide scrolling capabilities. This scrollpane is then added to the center of the `PrintPreview` frame, and our frame is then made visible.

Class PrintPreview.PreviewContainer

This inner class extends `JPanel` to serve as a container for `PagePreview` components. The only reason this custom container is developed is because we have specific layout requirements. What we want here is a layout which places its child components from left to right, without any resizing (using their preferred size), and leaves equal gaps between them. When the available container's width is filled, a new row should be started from the left edge, without regard to the available height (we assume scrolling functionality will be made available).

You may want to refer back to our discussion of layouts in chapter 4. The code constituting this class does not require much explanation and provides a good exercise for custom layout development (even though this class is not explicitly a layout manager).

Class PrintPreview.PagePreview

This inner class extends `JPanel` to serve as a placeholder for the image of each printed page preview. Four instance variables are used:

- `int m_w`: the current component's width (without insets).
- `int m_h`: the current component's height (without insets).
- `Image m_source`: the source image depicting the previewed page in full scale.
- `Image m_img`: the scaled image currently used for rendering.

The constructor of the `PagePreview` class takes its initial width, height, and the source image. It creates a scaled image by calling the `getScaledInstance()` method and sets its border to `MatteBorder(1, 1, 2, 2, Color.black)` to imitate a page laying on a flat surface.

The `setScaledSize()` method may be called to resize this component. It takes a new width and height as parameters and creates a new scaled image corresponding to the new size. Usage of the `SCALE_SMOOTH` option for scaling is essential to get a preview image which looks like a zoomed printed page (although it is not the fastest option).

The `paint()` method draws a scaled image and draws a border around the component.

Running the Code

At this point you can compile and execute this example. Figure 22.2 shows a preview of the large image which will be printed on the nine pages. Select various zoom factors in the combobox and see how the size of the previewed pages is changed. Then press the "Print" button to print to the default printer directly from the preview frame.

22.4 Printing styled text

In this section we'll add printing capabilities to the RTF word processor application developed in chapter 20. The printing of styled text would be easy if `JTextComponent` or `JTextPane` implemented the `Printable` interface and provided the capability to print their content. Unfortunately this is not the case (at least as of Java 2 FCS). So we have to get fairly clever, and create our own `BoxView` subclass to specifically handle printing.

Our styled editor class will now implement the `Printable` interface and delegate the mechanics of printing of each page to our custom `BoxView` subclass. Note that this custom view is not actually displayed on the screen as the editor. It sits in the background and is used only for printing and display in our print preview component.

Recall, from our discussion in chapters 11 and 19, that styled documents consist of a hierarchy of elements: paragraphs, images, components, etc. Each element is rendered by an associated view, which are all children of the root view. A `BoxView` in particular, arranges all its child views along either the x or y axis (typically the y axis). So, in this example, when we need to render a page, we start from the first child view of our custom `BoxView`, and render each of the child views sequentially, placing each below the previous in the vertical direction. When the next page should be rendered, we start from the first remaining view and continue in this fashion until all child views have been rendered. (This process will be explained in greater detail below.)

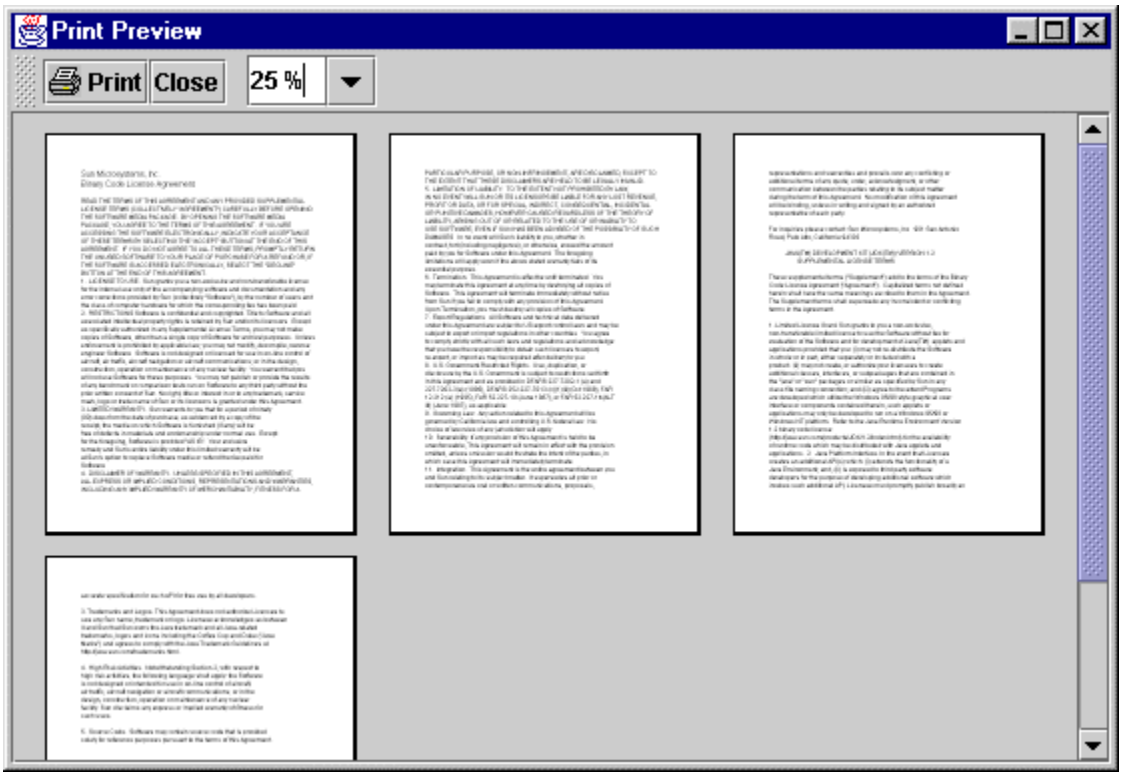


Figure 22.5 Printpreview showing a four page RTF document.
<<figure22-5.gif>>

The Code: WordProcessor.java
see \Chapter22\3

```
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.sql.*;

import java.awt.print.*;
import javax.swing.plaf.basic.*;

import javax.swing.*;
import javax.swing.text.*;
import javax.swing.event.*;
import javax.swing.border.*;
import javax.swing.text.rtf.*;
import javax.swing.undo.*;

import dl.*;

public class WordProcessor extends JFrame implements Printable
{
  // Unchanged code from section 20.9

  protected PrintView m_printView;

  // Unchanged code from section 20.9

  protected JMenuBar createMenuBar() {
    // Unchanged code from section 20.9
```

```
    mFile.addSeparator();

    Action actionPrint = new AbstractAction("Print...",
     new ImageIcon("print.gif")) {
      public void actionPerformed(ActionEvent e) {
        Thread runner = new Thread() {
          public void run() {
            printData();
          }
        };
        runner.start();
      }
    };
    item =  mFile.add(actionPrint);
    item.setMnemonic('p');

    item = new JMenuItem("Print Preview");
    item.setMnemonic('v');
    ActionListener lstPreview = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Thread runner = new Thread() {
          public void run() {
            setCursor(Cursor.getPredefinedCursor(
              Cursor.WAIT_CURSOR));
            new PrintPreview(WordProcessor.this);
            setCursor(Cursor.getPredefinedCursor(
              Cursor.DEFAULT_CURSOR));
          }
        };
        runner.start();
      }
    };
    item.addActionListener(lstPreview);
    mFile.add(item);

    mFile.addSeparator();

    // Unchanged code from section 20.9
  }

  // Unchanged code from section 20.9

  public void printData() {
    getJMenuBar().repaint();
    try {
      PrinterJob prnJob = PrinterJob.getPrinterJob();
      prnJob.setPrintable(this);
      if (!prnJob.printDialog())
        return;
      setCursor( Cursor.getPredefinedCursor(
        Cursor.WAIT_CURSOR));
      prnJob.print();
      setCursor( Cursor.getPredefinedCursor(
        Cursor.DEFAULT_CURSOR));
      JOptionPane.showMessageDialog(this,
        "Printing completed successfully", "Info",
        JOptionPane.INFORMATION_MESSAGE);
    }
    catch (PrinterException e) {
      e.printStackTrace();
      System.err.println("Printing error: "+e.toString());
```

```
    }
  }

  public int print(Graphics pg, PageFormat pageFormat,
   int pageIndex) throws PrinterException {
    pg.translate((int)pageFormat.getImageableX(),
      (int)pageFormat.getImageableY());
    int wPage = (int)pageFormat.getImageableWidth();
    int hPage = (int)pageFormat.getImageableHeight();
    pg.setClip(0, 0, wPage, hPage);

    // Only do this once per print
    if (m_printView == null) {
      BasicTextUI btui = (BasicTextUI)m_monitor.getUI();
      View root = btui.getRootView(m_monitor);
      m_printView = new PrintView(
        m_doc.getDefaultRootElement(),
        root, wPage, hPage);
    }

    boolean bContinue = m_printView.paintPage(pg,
      hPage, pageIndex);
    System.gc();

    if (bContinue)
      return PAGE_EXISTS;
    else {
      m_printView = null;
      return NO_SUCH_PAGE;
    }
  }

  public static void main(String argv[]) {
    new WordProcessor();
  }

  // Unchanged code from section 20.9

  class PrintView extends BoxView
  {
    protected int m_firstOnPage = 0;
    protected int m_lastOnPage = 0;
    protected int m_pageIndex = 0;

    public PrintView(Element elem, View root, int w, int h) {
      super(elem, Y_AXIS);
      setParent(root);
      setSize(w, h);
      layout(w, h);
    }

    public boolean paintPage(Graphics g, int hPage,
     int pageIndex) {
      if (pageIndex > m_pageIndex) {
        m_firstOnPage = m_lastOnPage + 1;
        if (m_firstOnPage >= getViewCount())
          return false;
        m_pageIndex = pageIndex;
      }
      int yMin = getOffset(Y_AXIS, m_firstOnPage);
      int yMax = yMin + hPage;
      Rectangle rc = new Rectangle();
```

```
      for (int k = m_firstOnPage; k < getViewCount(); k++) {
        rc.x = getOffset(X_AXIS, k);
        rc.y = getOffset(Y_AXIS, k);
        rc.width = getSpan(X_AXIS, k);
        rc.height = getSpan(Y_AXIS, k);
        if (rc.y+rc.height > yMax)
          break;
        m_lastOnPage = k;
        rc.y -= yMin;
        paintChild(g, rc, k);
      }
      return true;
    }
  }

// Remaining code is unchanged from section 20.9
```

Class WordProcessor

In comparison to the example of section 20.9, this class imports two new packages: `java.awt.print` and `javax.swing.plaf.basic`. The first one provides the necessary printing API, while the second is used to gain access to text component UI delegates (we will soon see why this is necessary).

One new instance variable, `PrintView m_printView`, represents our custom view used to print the styled document (see below). The `createMenuBar()` method now creates and adds to the "File" menu two new menu items titled "Print..." and "Print Preview". When the first one is selected it calls the `printData()` method, while the second one creates a `PrintPreview` instance by passing `WordProcessor.this` as the `Printable` reference. The `printData()` method obtains a `PrinterJob` instance, invokes a native Print dialog, and initializes printing the same way as we've seen in previous examples.

The `print()` method is called to print a given page of the current styled document. First, this method determines the size and origin of the printable area using a `PageFormat` instance as we've seen before. Next we need to set a clip area of the graphics context to the size of this printable area. This is necessary for the rendering of text component Views because they do clipping area intersection detection for optimized painting. If we don't set the clipping area, they won't know how to render themselves.

Unfortunately, the `Printable` interface does not provide any methods which can be called to initialize specific resources before printing, and release these resources after printing. So we must implement this functionality ourselves. The actual job of rendering the styled document is done by the `m_printView` object, which must be instantiated before printing begins, and released when it ends. Being forced to do all this in a single method, we first check if the `m_printView` reference is `null`. If it is then we assign it a new instance of `PrintVew`. If it isn't null we don't modify it (this indicates that we are in the midst of a printing session). When printing ends, we then set it to `null` so that the remaining `PrintView` instance can be garbage collected.

```
    // Only do this once per print
    if (m_printView == null) {
      BasicTextUI btui = (BasicTextUI)m_monitor.getUI();
      View root = btui.getRootView(m_monitor);
      m_printView = new PrintView(
        m_doc.getDefaultRootElement(),
        root, wPage, maxHeight);
    }
```

To create an `m_printView` object we need to access the `BasicTextUI` instance for our `m_monitor`

JTextPane component, and retrieve its root View (which sits on the top of the hierarchy of views--see chapter 19) using BasicTextUI's getRootView() method. At this point the PrintView instance can be created. Its constructor takes four parameters: the root element of the current document, the root view, and the width and height of the entire document's printing bounds.

As soon as we're sure that the m_printView object exists, we call its custom paintPage() method to render a page with the given index to the given graphical context. Then the garbage collector is called explicitly in an attempt to cut down on the heavy memory usage.

Finally if the paintPage() call returns true, the PAGE_EXISTS value is returned to indicate a successful render. Otherwise we set the m_printView reference to null, and return NO_SUCH_PAGE to indicate that no more pages can be rendered.

Class WordProcessor.PrintView

This inner class extends BoxView and is used to render the content of a styled document. (Note that since this class extends BoxView, we have access to some of its protected methods, such as getOffset(), getSpan(), layout(), and paintChild().)

Three instance variables are defined:

- int m_firstOnPage: index of the first view to be rendered on the current page.
- int m_lastOnPage: index of the last view to be rendered on the current page.
- int m_pageIndex: index of the current page.

The PrintView constructor creates the underlying BoxView object for a given root Element instance (this should be the root element in the document model of the text component we are printing) and the specified axis used for format/break operations (this is normally Y_AXIS). A given View instance is then set as the parent for this PrintView (this should be the root View of the text component we are printing). The setSize() method is called to set the size of this view, and layout() is called to lay out the child views based on the specified width and height (this is done to calculate the coordinates of all views used in the rendering of this document). These operations may be time consuming for large documents. Fortunately they are only performed at construction time:

```
    public PrintView(Element elem, View root, int w, int h) {
      super(elem, Y_AXIS);
      setParent(root);
      setSize(w, h);
      layout(w, h);
    }
```

Note: We found that setParent() must be called prior to setSize() and layout() to avoid undesirable side effects.

Our paintPage() method renders a single page of a styled document. It takes three parameters:

- Graphics g: the graphical context to render the page in.
- int hPage: the height of the page.
- int pageIndex: the index of the page to render.

This method will return true if the page with the given index is rendered successfully, or false if the end of the document is reached. We assume that the pages to be rendered will be fetched in sequential order (although more than one call can be made to print the most recently rendered page). If a new page index is

greater than `m_pageIndex` (which holds the index of the last rendered page), we begin rendering from the next view after the last one rendered on the previous page, and set `m_firstOnPage` to `m_lastOnPage + 1`. If this exceeds the number of child views, no more rendering can be done, so we return `false`.

```
      m_firstOnPage = m_lastOnPage + 1;
      if (m_firstOnPage >= getViewCount())
        return false;
```

Local variables `yMin` and `yMax` denote top and bottom coordinates of the page being rendered relative to the top of the document. `yMin` is determined by the offset of the first view to be rendered, and `yMax` is then `yMin` plus the height of the page:

```
      int yMin = getOffset(Y_AXIS, m_firstOnPage);
      int yMax = yMin + hPage;
```

All child views, from `m_firstOnPage` to the last view that will fit on the current page, are examined sequentially in a loop. In each iteration, local variable, `Rectangle rc`, is assigned the coordinates of where the associated child view is placed in the document (not on the current page). Based on the height of this view, if there is enough room horizontally to render it (note that it is guaranteed to fit vertically, since the page's width was specified in the `layout()` call above), the `paintChild()` method is called to render it into the graphics context. Also note that we offset the y-coordinate of the view by `yMin` because, as we just mentioned, each child view is positioned in terms of the whole document, and we are only concerned with its position on the current page. If at any point a view will not fit within the remaining page space we exit the loop.

```
      for (int k = m_firstOnPage; k < getViewCount(); k++) {
        rc.x = getOffset(X_AXIS, k);
        rc.y = getOffset(Y_AXIS, k);
        rc.width = getSpan(X_AXIS, k);
        rc.height = getSpan(Y_AXIS, k);
        if (rc.y+rc.height > yMax)
          break;
        m_lastOnPage = k;
          rc.y -= yMin;
        paintChild(g, rc, k);
      }
      return true;
```

Note: A more sophisticated and precise implementation might examine the y coordinates of all views in the hierarchy, not only the children of the root view. It might be the case that a large paragraph should be split between two or more pages. Our simple approach is not this flexible. In fact, in the case of a paragraph that spans a height larger than the page size, we could be in real trouble with this implementation. Although this is not common, it must be accounted for in professional implementations.

Running the Code

At this point you can compile and execute this example. Figure 22.3 shows a preview of a text document which will occupy four pages. Try previewing and printing a styled document. We've included the `License.rtf` file for you to experiment with.

22.5 Printing tables

In this section we'll add printing capabilities to the `JTable` application developed earlier in chapter 18. Unlike other examples in this chapter, a printed table should not resemble the `JTable` component as displayed on the screen. This requires us to add detailed code for the rendering of the table's contents as it should be displayed in a printout. The resulting code, however, does not depend on the table's structure and can be easily used for printing any table component. Thus, the code presented here can be plugged into any `JTable` application that

needs printing functionality. Combined with our print preview component (see previous examples), the amount of work we need to do to support printing of tables in professional applications is minimal.

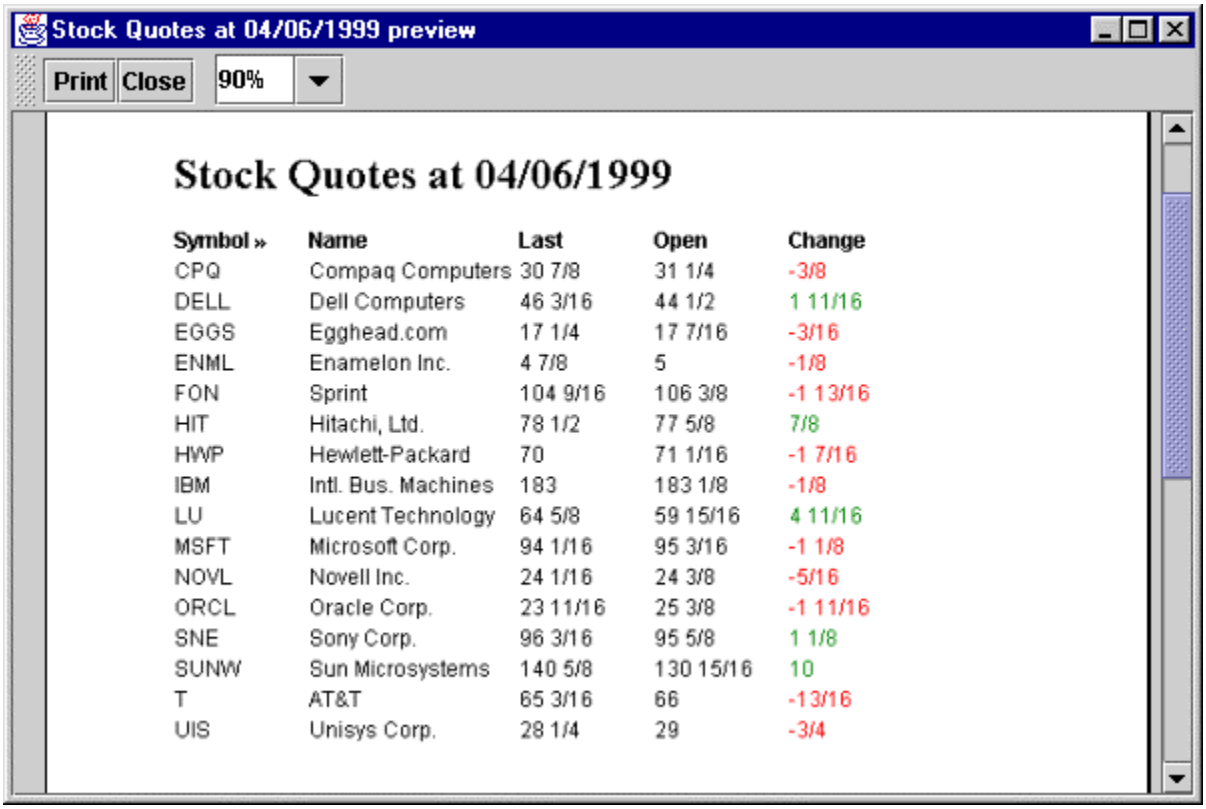


Figure 22.6 Print preview of JTable data.

<<figure22-6.gif>>

The Code: StocksTable.java

see \Chapter22\4

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;
import java.sql.*;
import java.awt.print.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

public class StocksTable extends JFrame implements Printable
{
  protected JTable m_table;
  protected StockTableData m_data;
  protected JLabel m_title;

  protected int m_maxNumPage = 1;

  // Unchanged code from section 18.6

  protected JMenuBar createMenuBar() {
```

```
    // Unchanged code from section 18.6

    JMenuItem mPrint = new JMenuItem("Print...");
    mPrint.setMnemonic('p');
    ActionListener lstPrint = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Thread runner = new Thread() {
          public void run() {
            printData();
          }
        };
        runner.start();
      }
    };
    mPrint.addActionListener(lstPrint);
    mFile.add(mPrint);

    JMenuItem mPreview = new JMenuItem("Print Preview");
    mPreview.setMnemonic('v');
    ActionListener lstPreview = new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        Thread runner = new Thread() {
          public void run() {
            setCursor(Cursor.getPredefinedCursor(
              Cursor.WAIT_CURSOR));
            new PrintPreview(Table7.this,
            m_title.getText()+" preview");
            setCursor(Cursor.getPredefinedCursor(
              Cursor.DEFAULT_CURSOR));
          }
        };
        runner.start();
      }
    };
    mPreview.addActionListener(lstPreview);
    mFile.add(mPreview);
    mFile.addSeparator();

    // Unchanged code from section 18.6
  }

  public void printData() {
    try {
      PrinterJob prnJob = PrinterJob.getPrinterJob();
      prnJob.setPrintable(this);
      if (!prnJob.printDialog())
        return;
      m_maxNumPage = 1;
      prnJob.print();
    }
    catch (PrinterException e) {
      e.printStackTrace();
      System.err.println("Printing error: "+e.toString());
    }
  }

  public int print(Graphics pg, PageFormat pageFormat,
   int pageIndex) throws PrinterException {
    if (pageIndex >= m_maxNumPage)
      return NO_SUCH_PAGE;

    pg.translate((int)pageFormat.getImageableX(),
```

```
      (int)pageFormat.getImageableY());
    int wPage = 0;
    int hPage = 0;
    if (pageFormat.getOrientation() == pageFormat.PORTRAIT) {
      wPage = (int)pageFormat.getImageableWidth();
      hPage = (int)pageFormat.getImageableHeight();
    }
    else {
      wPage = (int)pageFormat.getImageableWidth();
      wPage += wPage/2;
      hPage = (int)pageFormat.getImageableHeight();
      pg.setClip(0,0,wPage,hPage);
    }

    int y = 0;
    pg.setFont(m_title.getFont());
    pg.setColor(Color.black);
    Font fn = pg.getFont();
    FontMetrics fm = pg.getFontMetrics();
    y += fm.getAscent();
    pg.drawString(m_title.getText(), 0, y);
    y += 20; // space between title and table headers

    Font headerFont = m_table.getFont().deriveFont(Font.BOLD);
    pg.setFont(headerFont);
    fm = pg.getFontMetrics();

    TableColumnModel colModel = m_table.getColumnModel();
    int nColumns = colModel.getColumnCount();
    int x[] = new int[nColumns];
    x[0] = 0;

    int h = fm.getAscent();
    y += h; // add ascent of header font because of baseline
            // positioning (see figure 2.10)

    int nRow, nCol;
    for (nCol=0; nCol<nColumns; nCol++) {
      TableColumn tk = colModel.getColumn(nCol);
      int width = tk.getWidth();
      if (x[nCol] + width > wPage) {
        nColumns = nCol;
        break;
      }
      if (nCol+1<nColumns)
        x[nCol+1] = x[nCol] + width;
      String title = (String)tk.getIdentifier();
      pg.drawString(title, x[nCol], y);
    }

    pg.setFont(m_table.getFont());
    fm = pg.getFontMetrics();

    int header = y;
    h = fm.getHeight();
    int rowH = Math.max((int)(h*1.5), 10);
    int rowPerPage = (hPage-header)/rowH;
    m_maxNumPage = Math.max((int)Math.ceil(m_table.getRowCount()/
      (double)rowPerPage), 1);

    TableModel tblModel = m_table.getModel();
    int iniRow = pageIndex*rowPerPage;
```

```
    int endRow = Math.min(m_table.getRowCount(),
      iniRow+rowPerPage);

    for (nRow=iniRow; nRow<endRow; nRow++) {
      y += h;
      for (nCol=0; nCol<nColumns; nCol++) {
        int col = m_table.getColumnModel().getColumn(nCol).getModelIndex();
        Object obj = m_data.getValueAt(nRow, col);
        String str = obj.toString();
        if (obj instanceof ColorData)
          pg.setColor(((ColorData)obj).m_color);
        else
          pg.setColor(Color.black);
          pg.drawString(str, x[nCol], y);
      }
    }

    System.gc();
    return PAGE_EXISTS;
  }

// Remaining code unchanged from section 18.6
```

Understanding the Code

Class StocksTable

In comparison with the table examples of chapter 18, we now implement the `Printable` interface. In our `createMenuBar()` method we add a "Print..." menu item, which calls our new `printData()` method which acts just like the `printData()` methods we implemented in the examples above.

In our implementation of the `print()` method, we first determine whether a valid page index has been specified by comparing it to the maximum number of pages, `m_maxNumPage`:

```
        if (pageIndex > m_maxNumPage)
            return NO_SUCH_PAGE;
```

The catch is that we don't know this maximum number in advance. So we assign an initial value of 1 to `m_maxNumPage` (the code above works for the 0-th page), and adjust `m_maxNumPage` to the real value later in the code, just as we've done in the examples above.

We then translate the origin of the graphics context to the origin of the given `PageFormat` instance and determine the width and height of the area available for printing. These dimensions are used to determine how much data can fit on the given page. This same technique was also used in the previous examples. However, in this example we've added the ability to print with a landscape orientation because tables can be quite wide, and we normally don't want table data to span multiple pages (at least horizontally). In order to do this we have to first check the orientation of the given `PageFormat` instance. If it is `PORTRAIT` we determine its width and height as we have always done. If it is not `PORTRAIT`, then it must be either `LANDSCAPE` or `REVERSE_LANDSCAPE` (see section 22.1.5). In this case we need to increase the width of the page because the default is not adequate. After increasing the width we must also explicitly set the size of the graphics clip,

This is all we have to do to allow printing in either orientation.

Local variable `y` is created to keep track of the current vertical position on the page, and we are now ready to actually start the rendering, and begin with the the table's title. Note that we use the same font as is used in the table application for consistency. We add some white space below the title (by increasing `y`) and then we make preparations for printing our table's headers and body. A bold font is used for our table's header. An array, `x[]`, is created which will be used to store the x coordinate of each column's upper left corner (taking into account

that they may be resized and moved). Variable `nColumns` contains the total number of columns in our table.

Now we actually iterate through the columns and print each column header while filling our `x[]` array. We check each iteration to see if the x coordinate of the previous column, combined with the width of the column under consideration, will be more than the width of the page. If so we set the total number of columns, `nColumns`, to the number that will actually fit on the page, and then break out of the loop. If not we set the x coordinate corresponding to the current column, print its title, and continue on to the next iteration.

Since we've completed the printing of our table's title and headers, we know how much space is left for printing our table's body. We also know the font's height, so we can calculate how many rows can be printed on one page, which is `rowPerPage` below (the height of the page minus the current y offset, all divided by the height of the current font or 10, whichever is larger). Finally we calculate the real number of pages, `m_maxNumPage`, by dividing the total row count of our table by the number of rows per page we just calculated as `rowPerPage`. The minimum page count will be 1.

Now we need to actually print the table data. First we calculate the initial `iniRow` and final `endRow` rows to be printed on this page:

```
        TableModel tblModel = m_table.getModel();
        int iniRow = pageIndex*rowPerPage;
        int endRow = Math.min(m_table.getRowCount(),
            iniRow+rowPerPage);
```

Then, in a double `for` loop, iterating through each column of each row in turn, we print the table's contents. This is done by extracting each cell's data as an `Object` (using `getValueAt()`). We store its `toString()` `String` representation in a local variable and check if the object is an instance of our custom inner class, `ColorData` (defined in earlier chapter 18 examples). This class is designed to associate a color with a given data object. So if the object is a `ColorData` instance we grab its color and assign it as the current color of the graphics context. If it isn't we use black. Finally, we print that object's `toString()` representation and continue on to the remaining cells.

Note: We are assuming that each object's `toString()` representation is what we want to print. For more complex `TableCellRenderer` implementations, this printing code will need to be customized.

We end by explicitly invoking the garbage collector and returning `PAGE_EXISTS` to indicate a successful print.

Running the Code

At this point you can compile and execute this example. Figure 22.4 shows a print preview of our table application. Try manipulating the table's contents (by choosing different dates if you have JDBC and ODBC -- see chapter 18) and column orders to see how it affects the table's printout and print preview.

You will notice that in order to fit the whole table on the paper it must be condensed considerably. It is natural at this point to want to print it with a landscape orientation. When we choose landscape from the Page Setup dialog this modifies the `PageFormat` object that will be sent to our `print()` method when printing begins. However, this will not actually tell the printer to print in landscape mode. In order to do this, we have to explicitly choose landscape mode from the Print dialog as well. Unfortunately, the Page Setup information does not inform the printer, but it is necessary to inform our application.

Though our application can print successfully with a landscape orientation, our print preview component is not designed to display anything but portrait-oriented previews. Because of the way our `PrintPreview` component has been constructed, it is quite easy to add the ability to preview landscape-oriented pages if desired. The only modification that is necessary, is the addition of a parameter to its constructor which

specifyies the orientation to use. This parameter can then be assigned to the `PageFormat` object used in constructing each `PagePreview` object. We will not show the code here, but we have included a modified version of `PrintPreview` and the `StocksTable` application to demonstrate how you can implement this functionality. See \Chapter22\5. Figure 22.7 illustrates.

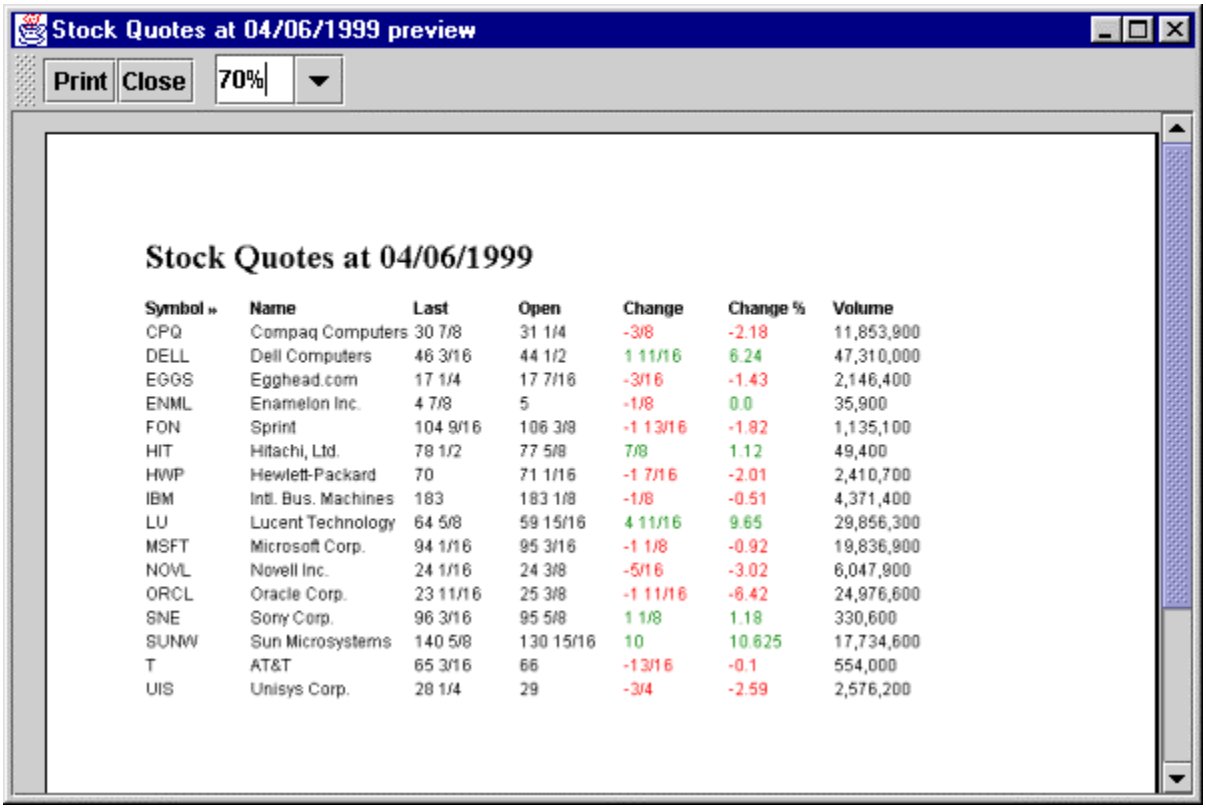


Figure 22.7 Printpreview component modified for landscape orientation.
<<figure22-7.gif>>

Chapter 23. Java2D

In this chapter:

- Java2D API overview
- Rendering charts
- Rendering text strings
- Rendering images

23.1 Java2D API overview

Java 2 offers a very powerful new rendering model known as Java2D. This model consists of a set of classes and interfaces for advanced 2D line art, text, and image rendering. Although this API is not considered a part of Swing, it is closely related to Swing and may be effectively used to develop sophisticated Swing applications.

Note: Packages java.awt.print (discussed in chapter 22) and com.sun.image.codec.jpeg (discussed in chapter 13) are also considered part of the Java2D API.

This chapter includes a Java2D API overview and shows how to use this API for chart rendering, enhanced label creation, and advanced image rendering. Below we briefly discuss the classes and interfaces that are fundamental to the 2D API. Note, however, that a complete description of all Java2D features lies beyond the scope of this book.

23.1.1 The Shape interface

abstract interface java.awt.Shape

This interface provides the definition for a 2D geometrical object. Most of the classes contained in the java.awt.geom package implement this interface. These classes define such things as points, lines, arcs, rectangles, round rectangles, ellipses, and more complex shapes such as cubic and quadratic parametric curves. These geometries allow a high degree of flexibility and with them, we can create almost any shape imaginable. We can render these geometries into a 2D graphics context using its draw() or fill() methods (see below). We can also perform boolean operations on multiple shapes such as union, intersection, exclusive or, etc. using the java.awt.geom.Area class. The geometry of each Shape's boundary is defined as a path which is represented by a set of line segments and curves encapsulated in a PathIterator instance (we will not discuss the details of this here).

Several overloaded contains() methods determine whether a given point or rectangle lies inside a Shape, and the getBounds() method returns a Shape's minimum bounding rectangle.

23.1.2 GeneralPath

class java.awt.geom.GeneralPath

This class implements the Shape interface and represents a geometric path constructed from several line segments, or quadratic and cubic curves. Particularly important is its append(Shape s, boolean connect) method which provides us with a way to append one shape to another by optionally connecting their paths with a line segment.

GeneralPath maintains a current coordinate at all times which represents the coordinate that, if we were to add a line segment, would be the beginning of the added line segment. To do this we use its lineTo() method passing it two floats representing the destination coordinate. Similarly, we can use its moveTo() and quadTo() methods to add a point or curve to the path.

23.1.3 Rectangle2D

class java.awt.geom.Rectangle2D

This class serves as a superclass for three classes: the well-known java.awt.Rectangle class, Rectangle2D.Double, and Rectangle2D.Float. These classes not only provide new ways to work with rectangles, but also allow us to specify a rectangle's coordinates in int, float, or double form. Along with Rectangle2D, the java.awt.geom package also includes a set of classes which provide new functionality to familiar graphical primitives, such as Dimension2D, Line2D, and Point2D. Each of these classes allow us to specify their coordinates using ints, floats, or doubles through appropriate subclasses.

23.1.4 AffineTransform

class java.awt.geom.AffineTransform

This class encapsulates a general form affine transformation between two coordinate systems. This transformation is essentially a coordinate transformation represented by a 3x3 matrix with an implied last row ([0 0 1]) mapping each x and y in the bounding rectangle of a Shape to a new x' and y' according to the following:

$$x' = m_{00}x + m_{01}y + m_{02}$$
$$y' = m_{10}x + m_{11}y + m_{12}$$

The m_{xx}'s represent the first two rows of a 3x3 matrix. These formulas are quite simple to understand and can be rewritten, for most operations, as the following:

```
x' = (scaleX * x) + (shearX * y) + offsetX
y' = (scaleY * x) + (shearY * y) + offsetY
```

Thess transformations preserve lines and parallelism (i.e. parallel lines are mapped to parallel lines). We use them to perform scaling, shearing, and translation. To construct an `AffineTransform` we use either the `double` or `float` version of the following constructor:

```
AffineTransform(m00, m10, m01, m11, m02, m12)
```

Note the order of the parameters. This directly corresponds to the columns of our matrix described above.

Rotation also preserves parallelism. Given an angle of rotation in radians, θ:

```
x' = x*(cosθ) + y*(-sinθ) + offsetX
y' = x*(sinθ) + y*(cosθ) + offsetY
```

Note that (degrees * π/180) = radians.

The Java2D graphics context (see below) maintains a transform attribute, just as it maintains a color and font attribute. Whenever we draw or fill a shape, this operation will be performed according to the current state of the transform attribute. We can create an instance of `AffineTransform` by specifying the first two rows of the matrix as described above. Alternatively, we can use static methods to create specific types of transformations: `getRotateInstance()`, `getScaleInstance()`, `getShearInstance()`, or `getTranslateInstance()`. We can use the `concatenate()` method to concatenate multiple transformations successively. We can also compose specific transformations with an existing `AffineTransform` using its `rotate()`, `scale()`, `shear()`, and `translate()` methods.

`AffineTransforms` are widely used throughout the 2D API as parameters to methods requiring a transformation to produce various visual effects.

23.1.5 The Stroke interface

abstract interface java.awt.Stroke

This interface defines only one method: `createStrokedShape(Shape p)`, which generates a `Shape` that is the outline of the given `Shape` parameter. This outline can be of various size, shape, and décor. The only implementing class is `BasicStroke` (see below). We use `Strokes` to define line styles for drawing in the Java2D graphics context. To set the stroke attribute of a given `Graphics2D` we use its `setStroke()`

method.

23.1.6 BasicStroke

class java.awt.BasicStroke

This class implements the `Stroke` interface and defines a set of rendering attributes specifying how to render the outline of a `Shape`. These attributes consist of line width, join style, end-cap style, and dash style:

- The line width (often called the pen width) is the thickness measured perpendicular to its trajectory.
- The end-cap style specifies whether round, butt, or square ends are used to render the ends of line segments: `CAP_ROUND`, `CAP_BUTT`, and `CAP_SQUARE`.
- The join style specifies how to render the joints between segments. This can be one of bevel, miter, or round: `JOIN_BEVEL`, `JOIN_MITER`, and `JOIN_ROUND`.
- The dash style defines a pattern of opaque and transparent regions rendered along a line segment.

23.1.7 The Paint interface

abstract interface java.awt.Paint

This interface defines how colors and color patterns may be assigned to the 2D graphics context for use in drawing and filling operations. Some important implementing classes are `Color`, `GradientPaint`, and `TexturePaint`. We use `Paints` to define fill patterns for filling in `Shapes` in the Java2D graphics context. To set the paint attribute of a given `Graphics2D` we use its `setPaint()` method.

23.1.8 GradientPaint

class java.awt.GradientPaint

This class implements the `Paint` interface and renders a shape by using a linear color gradient. The gradient is determined by two 2D points and two colors associated with them. The gradient can optionally be cyclical which means that between both points it will cycle through shades of each color several times, rather than just once. We use the `Graphics2D.setPaint()` method to assign a `GradientPaint` instance to a `Graphics2D` object. We can then call the `fill()` method to fill a specified `Shape` with this gradient. Note that this class provides an easy way to produce remarkable visual effects using only a few lines of code.

23.1.9 TexturePaint

class java.awt.TexturePaint

This class implements the `Paint` interface and is used to fill `Shapes` with a texture stored in a `BufferedImage`. We use the `Graphics2D.setPaint()` method to assign a `TexturePaint` instance to a `Graphics2D` object. We can call the `fill()` method to fill a specified `Shape` with this texture. Note that the `BufferedImages` used for a texture are expected to be small, as a `TexturePaint` object makes a copy of its data and stores it internally; it does not reference the provided `BufferedImage`. It is also important to reuse `TexturePaint` objects, rather than create new ones, whenever possible.

23.1.10 Graphics2D

class java.awt.Graphics2D

This class extends the `java.awt.Graphics` class to provide a more sophisticated API for working with geometry, transformations, colors, fill patterns and line styles, and text layout. In Java 2, the `Graphics`

object passed to a component's `paint()` method is really a `Graphics2D` object. So we can use this class in our `paint()` implementation by simply casting our `Graphics` object to a `Graphics2D`:

```
public void paint(Graphics g)
{
  Graphics2D g2 = (Graphics2D) g;
  // Use Graphics2D ...
```

We can assign attributes to a `Graphics2D` instance using methods such as `setTransform()`, `setStroke()` or `setPaint()`, as we discussed above. We can then call `draw()` to outline a given `Shape` instance using the assigned `Stroke`, and we can call `fill()` to fill a given `Shape` with the assigned `Color`, `GradientPaint`, or `TexturePaint`. Depending on the state of the transform attribute, `Shapes` will be translated, rotated, scaled, or sheared appropriately as they are drawn (see `AffineTransform`). We can modify the current transform directly with methods `rotate()`, `scale()`, `shear()`, and `translate()`. We can also assign it a new transform using its `setTransform()` method, or compose the current transform with a given one using its `transform()` method.

A `Graphics2D` object can maintain preferences for specific rendering algorithms to use depending on whether speed or quality is the priority. These are called rendering hints. They can be assigned using the `setRenderingHint()` method and are stored as key/value pairs. Valid keys and values are defined in the `RenderingHints` class. Two of these pairs are especially important to us, as the examples in this chapter will always use them:

- By setting the `KEY_ANTIALIASING` property to `VALUE_ANTIALIAS_ON` you can take advantage of a technique used to render objects with smoothly blended edges (by using intermediate colors to render a border between, say, black and white areas).
- By setting the the `KEY_RENDERING` property to `VALUE_RENDER_QUALITY`, appropriate rendering algorithms will always be chosen to ensure the best output quality.

23.1.11 GraphicsEnvironment

class java.awt.GraphicsEnvironment

This class is capable of retrieving the collection of `GraphicsDevice` and `Font` instances available to a Java application on the running platform. `GraphicsDevices` can reside on the local machine or any number of remote machines. A `GraphicsDevice` instance describes, surprisingly, a graphics device such as a screen or printer.

Recall from chapter 2 that we normally reference `GraphicsEnvironment` to retrieve the names of all available fonts:

```
String[] fontNames = GraphicsEnvironment.getLocalGraphicsEnvironment().
  getAvailableFontFamilyNames();
```

23.1.12 BufferedImage

class java.awt.image.BufferedImage

This class represents an `Image` stored in memory providing methods for storing, interpreting, and rendering pixel data. It is used widely throughout the 2D API and we've already seen it in chapters 13 and 22. In particular, you can create a `BufferedImage`, retrieve its associated `Graphics2D` instance to render into, perform the rendering, and use the result as an image for, among other things, painting directly into another graphics context (we used this technique in the construction of our print preview component). This is also similar to how `RepaintManager` handles the buffering of all Swing components, as we discussed in chapter

2.

23.1.13 FontRenderContext

class java.awt.font.FontRenderContext

Instances of this class encapsulate information needed to correctly measure text. This includes rendering hints and target device specific information such as resolution (dots-per-inch). A `FontRenderContext` instance representing the current state of the 2D graphics context can be retrieved using `Graphics2D`'s `getFontRenderContext()` method. `FontRenderContext` is usually used in association with text formatting using `Fonts` and `TextLayouts`.

23.1.14 TextLayout

class java.awt.font.TextLayout

Instances of this class represent an immutable graphical representation of styled text--that is, they cannot change (this class does not contain any `set` accessors). Only new instances can be created, and a `FontRenderContext` instance is required to do this. We render a `TextLayout` in the 2D graphcis context using that `TextLayout`'s `draw()` method. This class is very powerful and supports such things as hit detection, which will return the character a mouse press occurs on, as well as support for bi-directional text and split cursors. A particularly noteworthy method is `getOutline(AffineTransform tx)`, which returns a `Shape` instance outlining the text.

23.2 Rendering charts

In this section we'll demonstrate the advantages of using the Java2D API for rendering charts. The following example introduces a custom component which is capable of rendering line graphs, bar charts, and pie charts using strokes, color gradients, and background images. This application demonstrates how to build such charts taking into account issues such as axis positioning and scaling based on the given coordinate data. Be prepared for a bit of math.

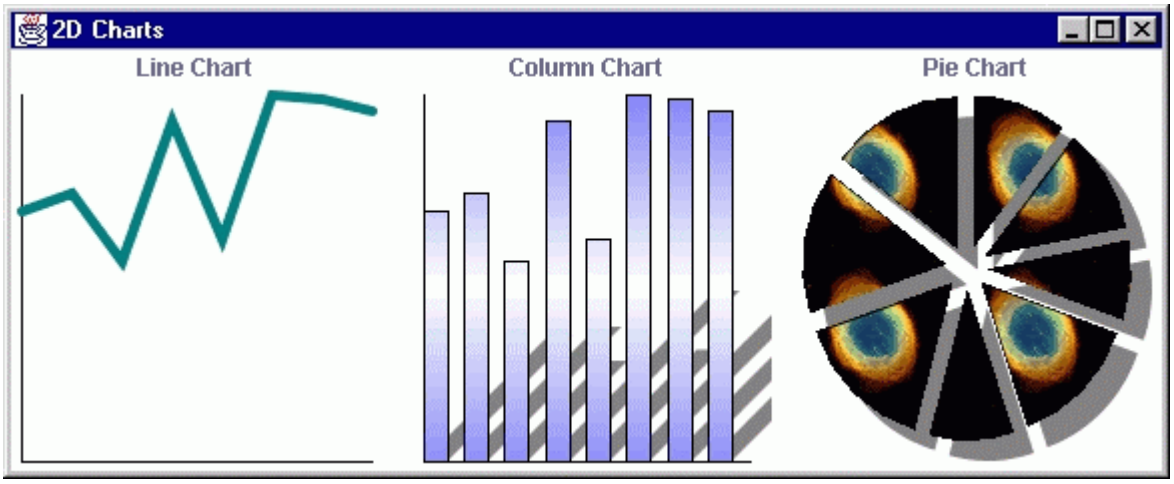


Figure 23.1 Charts2D displaying the three available JChart2D charts with various visual effects.
<<file figure23-1.gif>>

The Code: Charts2D.java
see \Chapter23\1

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

```
import java.awt.geom.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;

public class Charts2D extends JFrame
{
  public Charts2D() {
    super("2D Charts");
    setSize(720, 280);
    getContentPane().setLayout(new GridLayout(1, 3, 10, 0));
    getContentPane().setBackground(Color.white);

    int nData = 8;
    int[] xData = new int[nData];
    int[] yData = new int[nData];
    for (int k=0; k<nData; k++) {
      xData[k] = k;
      yData[k] = (int)(Math.random()*100);
      if (k > 0)
        yData[k] = (yData[k-1] + yData[k])/2;
    }

    JChart2D chart = new JChart2D(
      JChart2D.CHART_LINE, nData, xData,
      yData, "Line Chart");
    chart.setStroke(new BasicStroke(5f, BasicStroke.CAP_ROUND,
      BasicStroke.JOIN_MITER));
    chart.setLineColor(new Color(0, 128, 128));
    getContentPane().add(chart);

    chart = new JChart2D(JChart2D.CHART_COLUMN,
      nData, xData, yData, "Column Chart");
    GradientPaint gp = new GradientPaint(0, 100,
      Color.white, 0, 300, Color.blue, true);
    chart.setGradient(gp);
    chart.setEffectIndex(JChart2D.EFFECT_GRADIENT);
    chart.setDrawShadow(true);
    getContentPane().add(chart);

    chart = new JChart2D(JChart2D.CHART_PIE, nData, xData,
      yData, "Pie Chart");
    ImageIcon icon = new ImageIcon("hubble.gif");
    chart.setForegroundImage(icon.getImage());
    chart.setEffectIndex(JChart2D.EFFECT_IMAGE);
    chart.setDrawShadow(true);
    getContentPane().add(chart);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  public static void main(String argv[]) {
    new Charts2D();
  }
```

```
}

class JChart2D extends JPanel
{
  public static final int CHART_LINE = 0;
  public static final int CHART_COLUMN = 1;
  public static final int CHART_PIE = 2;

  public static final int EFFECT_PLAIN = 0;
  public static final int EFFECT_GRADIENT = 1;
  public static final int EFFECT_IMAGE = 2;

  protected int m_chartType = CHART_LINE;
  protected JLabel m_title;
  protected ChartPanel m_chart;

  protected int m_nData;
  protected int[] m_xData;
  protected int[] m_yData;
  protected int m_xMin;
  protected int m_xMax;
  protected int m_yMin;
  protected int m_yMax;
  protected double[] m_pieData;

  protected int  m_effectIndex = EFFECT_PLAIN;
  protected Stroke m_stroke;
  protected GradientPaint m_gradient;
  protected Image  m_foregroundImage;
  protected Color  m_lineColor = Color.black;
  protected Color  m_columnColor = Color.blue;
  protected int  m_columnWidth = 12;
  protected boolean m_drawShadow = false;

  public JChart2D(int type, int nData,
   int[] yData, String text) {
    this(type, nData, null, yData, text);
  }

  public JChart2D(int type, int nData, int[] xData,
   int[] yData, String text) {
    super(new BorderLayout());
    setBackground(Color.white);
    m_title = new JLabel(text, JLabel.CENTER);
    add(m_title, BorderLayout.NORTH);

    m_chartType = type;

    if (xData==null) {
      xData = new int[nData];
      for (int k=0; k<nData; k++)
        xData[k] = k;
    }
    if (yData == null)
      throw new IllegalArgumentException(
      "yData can't be null");
    if (nData > yData.length)
      throw new IllegalArgumentException(
      "Insufficient yData length");
    if (nData > xData.length)
      throw new IllegalArgumentException(
      "Insufficient xData length");
```

```
    m_nData = nData;
    m_xData = xData;
    m_yData = yData;

    m_xMin = m_xMax = 0;  // To include 0 into the interval
    m_yMin = m_yMax = 0;
    for (int k=0; k<m_nData; k++) {
      m_xMin = Math.min(m_xMin, m_xData[k]);
      m_xMax = Math.max(m_xMax, m_xData[k]);
      m_yMin = Math.min(m_yMin, m_yData[k]);
      m_yMax = Math.max(m_yMax, m_yData[k]);
    }
    if (m_xMin == m_xMax)
      m_xMax++;
    if (m_yMin == m_yMax)
      m_yMax++;

    if (m_chartType == CHART_PIE) {
      double sum = 0;
      for (int k=0; k<m_nData; k++) {
        m_yData[k] = Math.max(m_yData[k], 0);
        sum += m_yData[k];
      }
      m_pieData = new double[m_nData];
      for (int k=0; k<m_nData; k++)
        m_pieData[k] = m_yData[k]*360.0/sum;
    }

    m_chart = new ChartPanel();
    add(m_chart, BorderLayout.CENTER);
  }

  public void setEffectIndex(int effectIndex) {
    m_effectIndex = effectIndex;
    repaint();
  }

  public int getEffectIndex() { return m_effectIndex; }

  public void setStroke(Stroke stroke) {
    m_stroke = stroke;
    m_chart.repaint();
  }

  public void setForegroundImage(Image img) {
    m_foregroundImage = img;
    repaint();
  }

  public Image getForegroundImage() { return m_foregroundImage; }

  public Stroke getStroke() { return m_stroke; }

  public void setGradient(GradientPaint gradient) {
    m_gradient = gradient;
    repaint();
  }

  public GradientPaint getGradient() { return m_gradient; }

  public void setColumnWidth(int columnWidth) {
    m_columnWidth = columnWidth;
```

```
  m_chart.calcDimensions();
  m_chart.repaint();
}

public int getColumnWidth() { return m_columnWidth; }

public void setColumnColor(Color c) {
  m_columnColor = c;
  m_chart.repaint();
}

public Color getColumnColor() { return m_columnColor; }

public void setLineColor(Color c) {
  m_lineColor = c;
  m_chart.repaint();
}

public Color getLineColor() { return m_lineColor; }

public void setDrawShadow(boolean drawShadow) {
  m_drawShadow = drawShadow;
  m_chart.repaint();
}

public boolean getDrawShadow() { return m_drawShadow; }

class ChartPanel extends JComponent
{
  int m_xMargin = 5;
  int m_yMargin = 5;
  int m_pieGap = 10;

  int m_x;
  int m_y;
  int m_w;
  int m_h;

  ChartPanel() {
    enableEvents(ComponentEvent.COMPONENT_RESIZED);
  }

  protected void processComponentEvent(ComponentEvent e) {
    calcDimensions();
  }

  public void calcDimensions() {
    Dimension d = getSize();
    m_x = m_xMargin;
    m_y = m_yMargin;
    m_w = d.width-2*m_xMargin;
    m_h = d.height-2*m_yMargin;
    if (m_chartType == CHART_COLUMN) {
      m_x += m_columnWidth/2;
      m_w -= m_columnWidth;
    }
  }

  public int xChartToScreen(int x) {
    return m_x + (x-m_xMin)*m_w/(m_xMax-m_xMin);
  }
```

```
public int yChartToScreen(int y) {
  return m_y + (m_yMax-y)*m_h/(m_yMax-m_yMin);
}

public void paintComponent(Graphics g) {
  int x0 = 0;
  int y0 = 0;
  if (m_chartType != CHART_PIE) {
    g.setColor(Color.black);
    x0 = xChartToScreen(0);
    g.drawLine(x0, m_y, x0, m_y+m_h);
    y0 = yChartToScreen(0);
    g.drawLine(m_x, y0, m_x+m_w, y0);
  }

  Graphics2D g2 = (Graphics2D) g;
  g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
  g2.setRenderingHint(RenderingHints.KEY_RENDERING,
    RenderingHints.VALUE_RENDER_QUALITY);

  if (m_stroke != null)
    g2.setStroke(m_stroke);

  GeneralPath path = new GeneralPath();
  switch (m_chartType) {
  case CHART_LINE:
    g2.setColor(m_lineColor);
    path.moveTo(xChartToScreen(m_xData[0]),
      yChartToScreen(m_yData[0]));
    for (int k=1; k<m_nData; k++)
      path.lineTo(xChartToScreen(m_xData[k]),
        yChartToScreen(m_yData[k]));
    g2.draw(path);
    break;

  case CHART_COLUMN:
    for (int k=0; k<m_nData; k++) {
      m_xMax ++;
      int x = xChartToScreen(m_xData[k]);
      int w = m_columnWidth;
      int y1 = yChartToScreen(m_yData[k]);
      int y = Math.min(y0, y1);
      int h = Math.abs(y1 - y0);
      Shape rc = new Rectangle2D.Double(x, y, w, h);
      path.append(rc, false);
      m_xMax --;
    }

    if (m_drawShadow) {
      AffineTransform s0 = new AffineTransform(
        1.0, 0.0, 0.0, -1.0, x0, y0);
      s0.concatenate(AffineTransform.getScaleInstance(
        1.0, 0.5));
      s0.concatenate(AffineTransform.getShearInstance(
        0.5, 0.0));
      s0.concatenate(new AffineTransform(
        1.0, 0.0, 0.0, -1.0, -x0, y0));
      g2.setColor(Color.gray);
      Shape shadow = s0.createTransformedShape(path);
      g2.fill(shadow);
    }
```

```
  if (m_effectIndex==EFFECT_GRADIENT &&
    m_gradient != null) {
    g2.setPaint(m_gradient);
    g2.fill(path);
  }
  else if (m_effectIndex==EFFECT_IMAGE &&
    m_foregroundImage != null)
    fillByImage(g2, path, 0);
  else {
    g2.setColor(m_columnColor);
    g2.fill(path);
  }
  g2.setColor(m_lineColor);
  g2.draw(path);
  break;

case CHART_PIE:
  double start = 0.0;
  double finish = 0.0;
  int ww = m_w - 2*m_pieGap;
  int hh = m_h - 2*m_pieGap;
  if (m_drawShadow) {
    ww -= m_pieGap;
    hh -= m_pieGap;
  }

  for (int k=0; k<m_nData; k++) {
    finish = start+m_pieData[k];
    double f1 = Math.min(90-start, 90-finish);
    double f2 = Math.max(90-start, 90-finish);
    Shape shp = new Arc2D.Double(m_x, m_y, ww, hh,
      f1, f2-f1, Arc2D.PIE);
    double f = (f1 + f2)/2*Math.PI/180;
    AffineTransform s1 = AffineTransform.
      getTranslateInstance(m_pieGap*Math.cos(f),
      -m_pieGap*Math.sin(f));
    s1.translate(m_pieGap, m_pieGap);
    Shape piece = s1.createTransformedShape(shp);
    path.append(piece, false);
    start = finish;
  }

  if (m_drawShadow) {
    AffineTransform s0 = AffineTransform.
      getTranslateInstance(m_pieGap, m_pieGap);
    g2.setColor(Color.gray);
    Shape shadow = s0.createTransformedShape(path);
    g2.fill(shadow);
  }

  if (m_effectIndex==EFFECT_GRADIENT && m_gradient != null) {
    g2.setPaint(m_gradient);
    g2.fill(path);
  }
  else if (m_effectIndex==EFFECT_IMAGE &&
   m_foregroundImage != null)
    fillByImage(g2, path, 0);
  else {
    g2.setColor(m_columnColor);
    g2.fill(path);
  }
```

```
        g2.setColor(m_lineColor);
        g2.draw(path);
        break;
      }
    }

    protected void fillByImage(Graphics2D g2,
     Shape shape, int xOffset) {
      if (m_foregroundImage == null)
        return;
      int wImg = m_foregroundImage.getWidth(this);
      int hImg = m_foregroundImage.getHeight(this);
      if (wImg <=0 || hImg <= 0)
        return;
      g2.setClip(shape);
      Rectangle bounds = shape.getBounds();
      for (int xx = bounds.x+xOffset;
       xx < bounds.x+bounds.width; xx += wImg)
        for (int yy = bounds.y; yy < bounds.y+bounds.height;
         yy += hImg)
          g2.drawImage(m_foregroundImage, xx, yy, this);
    }
  }
}
```

Understanding the Code

Class Charts2D

This class provides the frame encompassing this example. It creates an array of equidistant x-coordinates and random y-coordinates to be drawn in the charts. Three instances of our custom `JChart2D` class (see below) are created and placed in the frame using a `GridLayout`. The methods used to provide setup and initialization for our chart are built into the `JChart2D` class and will be explained below.

Class JChart2D

Several constants are defined for use as the available chart type and visual effect options:

`int CHART_LINE`: specifies a line chart.

`int CHART_COLUMN`: specifies a column chart.

`int CHART_PIE`: specifies a pie chart.

`int EFFECT_PLAIN`: use no visual effects (homogeneous chart).

`int EFFECT_GRADIENT`: use a color gradient to fill the chart.

`int EFFECT_IMAGE`: use an image to fill the chart.

Several instance variables are defined to hold data used by this class:

`JLabel m_title`: label used to display a chart's title.

`ChartPanel m_chart`: custom component used to display a chart's body (see below).

`int m_nData`: number of points in the chart.

`int[] m_xData`: array of x-coordinates in the chart.

`int[] m_yData`: array of y-coordinates in the chart.

`int m_xMin`: minimum x-coordinate.

`int m_xMax`: maximum x-coordinate.

`int m_yMin`: minimum y-coordinate.

`int m_yMax`: maximum y-coordinate.

`double[] m_pieData`: angles for each piece of the pie chart.

`int m_chartType`: maintains the chart's type (one of the constants listed above).

`int m_effectIndex`: maintains the chart's effect index (one of the constants listed above).

`Stroke m_stroke`: stroke instance used to outline the chart.

`GradientPaint m_gradient`: color gradient used to fill the chart (this only takes effect when `m_effectIndex` is set to `EFFECT_GRADIENT`).

`Image m_foregroundImage`: image used to fill the chart (this only takes effect when `m_effectIndex` is set to `EFFECT_IMAGE`).

`Color m_lineColor`: color used to outline the chart.

`Color m_columnColor`: color used to fill the chart (this only takes effect when `m_effectIndex` is set to `EFFECT_PLAIN` -- this is its default setting).

`int m_columnWidth`: width of columns in the column chart.

`boolean m_drawShadow`: flag to draw a shadow for column or pie chart.

Two constructors are provided in the `JChart2D` class. The first one takes four parameters and simply calls the second, passing it the given parameters and using a `null` value for a fifth. This second constructor is where a `JChart2D` is actually created and its five parameters are:

`int type`: the type of this chart (`CHART_LINE`, `CHART_COLUMN`, or `CHART_PIE`).

`int nData`: number of data points in this chart.

`int[] xData`: an array of x-coordinates for this chart (may be `null` -- this is passed as `null` from the first constructor).

`int[] yData`: an array of y-coordinates for this chart.

`String text`: this chart's title.

The constructor validates the input data and initializes all instance variables. In the case of a pie chart, an array, `m_pieData`, is created, which contains sectors with angles normalized to 360 degrees (the `sum` value used here was calculated previous to this code as the sum of all `m_yData[]` values):

```
m_pieData = new double[m_nData];
for (int k=0; k<m_nData; k++)
  m_pieData[k] = m_yData[k]*360.0/sum;
```

This chart component extends `JPanel` and contains two child components managed using a `BorderLayout`: `JLabel m_title`, which displays the chart's title in the `NORTH` region, and an instance of our custom `ChartPanel` component, `m_chart`, which is placed in the `CENTER` region.

The rest of the code for this class consists of `set`/`get` methods supporting instance variables declared in this class and does not require further explanation.

Class JChart2D.ChartPanel

This inner class extends `JComponent` and represents the custom component that is actually responsible for

rendering our charts. Several instance variables are declared:

`int m_xMargin`: the left and right margin size of the rendering area.

`int m_yMargin`: the top and bottom margin size of the rendering area.

`int m_pieGap`: radial shift for pieces of pie (i.e. spacing between each).

`int m_x`: left coordinate of the rendering area.

`int m_y`: top coordinate of the rendering area.

`int m_w`: width of the rendering area.

`int m_h`: height of the rendering area.

The `ChartPanel` constructor enables the processing of component resize events. When such an event occurs, the `processComponentEvent()` method triggers a call to `calcDimensions()` (note that this event will normally be generated when `ChartPanel` is added to a container for the first time). This method retrieves the current component's size, calculates the coordinates of the rendering area, and stores them in the appropriate instance variables listed above. In the case of a column chart, we offset the rendering area by an additional half of the column width, and then shrink it by a full column width. Otherwise, the first and the last columns will be rendered on top of the chart's border.

Methods `xChartToScreen()` and `yChartToScreen()` calculate screen coordinates from chart coordinates as illustrated in figure 23.2. We need to scale the chart data so the chart will occupy the entire component region, taking into account the margins. To get the necessary scaling ratios we divide the dimensions of the chart component (minus the margins) by the difference between max and min values of the chart data. These methods are used in rendering the line and column charts because they are based on coordinate data. The only sizing information the pie chart needs is `m_w` and `m_h`, as it does not rely on coordinate data.

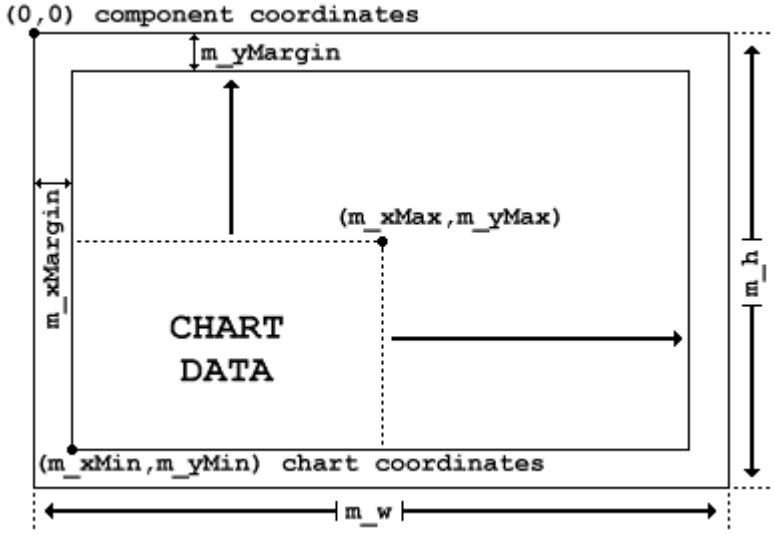


Figure 23.2 Screen coordinates vs. chart coordinates.
<<file figure23-2.gif>>

The `paintComponent()` method performs the actual chart rendering. The coordinate axes are drawn first for line and column charts. Then we cast the `Graphics` instance to a `Graphics2D` so we have access to Java2D features. As we discussed earlier, we use two rendering hints and assign them with the `setRenderingHint()` method: anti-aliasing and the preference to render quality over speed. If the `m_stroke` instance variable has been initialized, the `Graphics2D` stroke attribute is set using the `setStroke()` method. The rest of the `paintComponent()` method is placed into a `switch` block with

cases for each chart type. Before the switch block is entered we create a `GeneralPath` which we will use to construct each chart using the methods we described in section 23.1.2.

The line chart is the simplest case. It is drawn as a broken line through the array of points representing the chart data. First we start the `GeneralPath` out by passing the first coordinate of data using `moveTo()`. Then we iterate through the chart data adding lines to the path using its `lineTo()` method. Once we've done this we are ready to render it and use the `Graphics2D draw()` method to do so.

Note: The Java2D API provides ways to draw quadratic and cubic curves passing through 3 and 4 given points respectively. Unfortunately this functionality is not suitable for drawing a smooth line chart with interpolation.

The column chart is drawn as a set of vertical bars with a common baseline corresponding to the `0`-value of the chart, `y0` (note that this value is always included in the `[m_yMin, m_yMax]` interval). The `GeneralPath` instance accumulates these bars as `Rectangle2D.Double` instances using its `append()` method, passing `false` for the line connection option.

If the `m_drawShadow` flag is set, the next step forms and draws a shadow from these bars, which should be viewed as standing vertically. `AffineTransform s0` is constructed to accomplish this in four steps:

1. Transform from screen coordinates to chart coordinates.
2. Scale y-axis by a factor of 0.5.
3. Shear x-axis by a factor of 1.0.
4. Transform chart coordinates back to screen coordinates.

As soon as this `AffineTransform` is constructed, we create a corresponding transformed version of our path `Shape` using `AffineTransform`'s `createTransformedShape()` method. We then set the current color to gray and render it into the 2D graphics context using the `fill()` method. Finally the set of bars is drawn on the screen. Depending on the `m_effectIndex` setting we fill this shape with the gradient color, image (by calling our custom `fillByImage()` method), or with a solid color.

The pie chart is drawn as pieces of a circle with a common center. The larger the chart's value is for a given point, the larger the corresponding angle of that piece is. For an interesting resemblance with a cut pie, all pieces are shifted apart from the common center in the radial direction. To draw such a pie we first build each piece by iterating through the chart's data. Using class `Arc2D.Double` with its `PIE` setting provides a convenient way to build a slice of pie. We then translate this slice away from the pie's center in the radial direction using an `AffineTransform` and its `createTransformShape()` method. Each resulting shape is appended to our `GeneralPath` instance.

If the `m_drawShadow` flag is set, we form and draw a shadow from these pieces. Since this chart can be viewed as laying on a flat surface, the shadow has the same shape as the chart itself, but is translated in the south-east direction. Finally the set of pie pieces is drawn on the screen using the selected visual effect. Since at this point we operate with the chart as a single `Shape` (remember a `GeneralPath` is a `Shape`), the code is the same as for the column chart.

The custom `fillByImage()` method uses the given `Shape` instance's bounds as the `Graphics2D` clipping area, and, in a doubly nested `for` loop, fills this region using our previously assigned `m_foregroundImage`. (Note that the third parameter to this method, `int xOffset`, is used for horizontal displacement which we do not make use of in this example. However, we will see this method again in the next example where we will need this functionality.)

Running the Code

Figure 23.1 shows our `Charts2D` application containing three charts: line, column, and pie. Try modifying the settings specified in `Charts2D` class to try charts with various combinations of available visual effects. Also try resizing the frame container and note how each chart is scaled accordingly.

Our `JChart2D` component can easily be plugged into any Swing application. Since we have implemented full scalability and correct coordinate mapping, we have the beginnings of a professional chart component. The next step would be to add informative strings to the axis as well as pie pieces, bars, and data points of the line chart.

23.3 Rendering strings

In this section we'll demonstrate advantages of using Java2D for rendering strings. This is especially useful for relatively big fonts used to display titles. The following example introduces a custom label component which is capable of rendering strings with various visual effects, including such things as animation using an image, continuously changing foreground color, and outlining.

Figure 23.3 JLabel2Ds with various visual effects, and a plain JLabel for comparison.

<<figure23-3.gif>>

The Code: Labels2D.java

see \Chapter23\2

```
import java.awt.*;
import java.awt.event.*;
import java.awt.font.*;
import java.awt.geom.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;

public class Labels2D extends JFrame
{
  public Labels2D() {
    super("2D Labels");
    setSize(600, 250);
    getContentPane().setLayout(new GridLayout(6, 1, 5, 5));
    getContentPane().setBackground(Color.white);
    Font bigFont = new Font("Helvetica",Font.BOLD, 24);
```

```
JLabel2D lbl = new JLabel2D("Simple JLabel2D With Outline",
  JLabel.CENTER);
lbl.setFont(bigFont);
lbl.setForeground(Color.blue);
lbl.setBorder(new LineBorder(Color.black));
lbl.setBackground(Color.cyan);
lbl.setOutlineColor(Color.yellow);
lbl.setStroke(new BasicStroke(5f));
lbl.setOpaque(true);
lbl.setShearFactor(0.3);
getContentPane().add(lbl);

lbl = new JLabel2D("JLabel2D With Color Gradient",
  JLabel.CENTER);
lbl.setFont(bigFont);
lbl.setOutlineColor(Color.black);
lbl.setEffectIndex(JLabel2D.EFFECT_GRADIENT);
GradientPaint gp = new GradientPaint(0, 0,
  Color.red, 100, 50, Color.blue, true);
lbl.setGradient(gp);
getContentPane().add(lbl);

lbl = new JLabel2D(
  "JLabel2D Filled With Image", JLabel.CENTER);
lbl.setFont(bigFont);
lbl.setEffectIndex(JLabel2D.EFFECT_IMAGE);
ImageIcon icon = new ImageIcon("mars.gif");
lbl.setForegroundImage(icon.getImage());
lbl.setOutlineColor(Color.red);
getContentPane().add(lbl);

lbl = new JLabel2D("JLabel2D With Image Animation",
  JLabel.CENTER);
lbl.setFont(bigFont);
lbl.setEffectIndex(JLabel2D.EFFECT_IMAGE_ANIMATION);
icon = new ImageIcon("ocean.gif");
lbl.setForegroundImage(icon.getImage());
lbl.setOutlineColor(Color.black);
lbl.startAnimation(300);
getContentPane().add(lbl);

lbl = new JLabel2D("JLabel2D With Color Animation",
  JLabel.CENTER);
lbl.setFont(bigFont);
lbl.setEffectIndex(JLabel2D.EFFECT_COLOR_ANIMATION);
lbl.setGradient(gp);
lbl.setOutlineColor(Color.black);
lbl.startAnimation(300);
getContentPane().add(lbl);

JLabel lbl1 = new JLabel("Plain JLabel For Comparison",
  JLabel.CENTER);
lbl1.setFont(bigFont);
lbl1.setForeground(Color.black);
getContentPane().add(lbl1);

WindowListener wndCloser = new WindowAdapter() {
  public void windowClosing(WindowEvent e) {
    System.exit(0);
  }
};
addWindowListener(wndCloser);
```

```
    setVisible(true);
  }

  public static void main(String argv[]) { new Labels2D(); }
}

class JLabel2D extends JLabel
{
  public static final int EFFECT_PLAIN = 0;
  public static final int EFFECT_GRADIENT = 1;
  public static final int EFFECT_IMAGE = 2;
  public static final int EFFECT_IMAGE_ANIMATION = 3;
  public static final int EFFECT_COLOR_ANIMATION = 4;

  protected int m_effectIndex = EFFECT_PLAIN;
  protected double m_shearFactor = 0.0;
  protected Color  m_outlineColor;
  protected Stroke m_stroke;
  protected GradientPaint m_gradient;
  protected Image  m_foregroundImage;
  protected Thread m_animator;
  protected boolean m_isRunning = false;
  protected int m_delay;
  protected int m_xShift;

  public JLabel2D() { super(); }

  public JLabel2D(String text) { super(text); }

  public JLabel2D(String text, int alignment) {
    super(text, alignment);
  }

  public void setEffectIndex(int effectIndex) {
    m_effectIndex = effectIndex;
    repaint();
  }

  public int getEffectIndex() { return m_effectIndex; }

  public void setShearFactor(double shearFactor) {
    m_shearFactor = shearFactor;
    repaint();
  }

  public double getShearFactor() { return m_shearFactor; }

  public void setOutlineColor(Color outline) {
    m_outlineColor = outline;
    repaint();
  }

  public Color getOutlineColor() { return m_outlineColor; }

  public void setStroke(Stroke stroke) {
    m_stroke = stroke;
    repaint();
  }

  public Stroke getStroke() { return m_stroke; }
```

```
public void setGradient(GradientPaint gradient) {
  m_gradient = gradient;
  repaint();
}

public GradientPaint getGradient() { return m_gradient; }

public void setForegroundImage(Image img) {
  m_foregroundImage = img;
  repaint();
}

public Image getForegroundImage() { return m_foregroundImage; }

public void startAnimation(int delay) {
  if (m_animator != null)
    return;
  m_delay = delay;
  m_xShift = 0;
  m_isRunning = true;
  m_animator = new Thread() {
    double arg = 0;
    public void run() {
      while(m_isRunning) {
        if (m_effectIndex==EFFECT_IMAGE_ANIMATION)
          m_xShift += 10;
        else if (
         m_effectIndex==EFFECT_COLOR_ANIMATION &&
         m_gradient != null) {
          arg += Math.PI/10;
          double cos = Math.cos(arg);
          double f1 = (1+cos)/2;
          double f2 = (1-cos)/2;
          arg = arg % (Math.PI*2);

          Color c1 = m_gradient.getColor1();
          Color c2 = m_gradient.getColor2();
          int r = (int)(c1.getRed()*f1+c2.getRed()*f2);
          r = Math.min(Math.max(r, 0), 255);
          int g = (int)(c1.getGreen()*f1+c2.getGreen()*f2);
          g = Math.min(Math.max(g, 0), 255);
          int b = (int)(c1.getBlue()*f1+c2.getBlue()*f2);
          b = Math.min(Math.max(b, 0), 255);
          setForeground(new Color(r, g, b));
        }
        repaint();
        try { sleep(m_delay); }
        catch (InterruptedException ex) { break; }
      }
    }
  };
  m_animator.start();
}

public void stopAnimation() {
  m_isRunning = false;
  m_animator = null;
}

public void paintComponent(Graphics g) {
  Dimension d= getSize();
  Insets ins = getInsets();
```

```
int x = ins.left;
int y = ins.top;
int w = d.width-ins.left-ins.right;
int h = d.height-ins.top-ins.bottom;

if (isOpaque()) {
  g.setColor(getBackground());
  g.fillRect(0, 0, d.width, d.height);
}
paintBorder(g);

Graphics2D g2 = (Graphics2D) g;
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
  RenderingHints.VALUE_ANTIALIAS_ON);
g2.setRenderingHint(RenderingHints.KEY_RENDERING,
  RenderingHints.VALUE_RENDER_QUALITY);

FontRenderContext frc = g2.getFontRenderContext();
TextLayout tl = new TextLayout(getText(), getFont(), frc);

AffineTransform shear = AffineTransform.
  getShearInstance(m_shearFactor, 0.0);
Shape src = tl.getOutline(shear);
Rectangle rText = src.getBounds();

float xText = x - rText.x;
switch (getHorizontalAlignment()) {
case CENTER:
  xText = x + (w-rText.width)/2;
  break;
case RIGHT:
  xText = x + (w-rText.width);
  break;
}
float yText = y + h/2 + tl.getAscent()/4;

AffineTransform shift = AffineTransform.
  getTranslateInstance(xText, yText);
Shape shp = shift.createTransformedShape(src);

if (m_outlineColor != null) {
  g2.setColor(m_outlineColor);
  if (m_stroke != null)
    g2.setStroke(m_stroke);
  g2.draw(shp);
}

switch (m_effectIndex) {
  case EFFECT_GRADIENT:
    if (m_gradient == null)
      break;
    g2.setPaint(m_gradient);
    g2.fill(shp);
    break;
  case EFFECT_IMAGE:
    fillByImage(g2, shp, 0);
    break;
  case EFFECT_COLOR_ANIMATION:
    g2.setColor(getForeground());
    g2.fill(shp);
    break;
  case EFFECT_IMAGE_ANIMATION:
```

```
        if (m_foregroundImage == null)
          break;
        int wImg = m_foregroundImage.getWidth(this);
        if (m_xShift > wImg)
          m_xShift = 0;
        fillByImage(g2, shp, m_xShift-wImg);
        break;
      default:
        g2.setColor(getForeground());
        g2.fill(shp);
        break;
    }
  }

  // Method fillByImage taken from JChart2D (section 23.2)
}
```

Understanding the Code

Class Labels2D

This class extends `JFrame` and provides the main container for our example. Its constructor creates five instances of our `JLabel2D` custom component, and one `JLabel` used for comparison. All labels are placed in a `GridLayout` containing one column, and each uses the same font. Various settings are used to render these custom labels, as defined in our `JLabel2D` class:

- The first label uses border and background settings to demonstrate that they can be used the same way as with any other Swing components. (Note that we still have to set the opaque property to true for the background to be filled.) The `outlineColor` and `stroke` properties are set to outline the label's text. Finally the `shearFactor` property is set to make the text lean to the left.
- The second label uses a `GradientPaint` instance (using red and blue colors) to fill the text's interior.
- The third label uses an image to fill the text's interior.
- The fourth label uses an image to fill the text's interior with animation of that image (cyclical shifting of the image in the horizontal direction).
- The fifth label uses the same `GradientPaint` as the second one, but to produce an effect of color animation (a solid foreground color which is changed cyclically).
- The sixth label is a plain `JLabel` component without any special effects.

Class JLabel2D

This class extends `JLabel` and provides various visual effects in text rendering. Constants are defined representing each available type of visual effect:

- `int EFFECT_PLAIN`: no special effects used.
- `int EFFECT_GRADIENT`: use gradient painting of the text's foreground.
- `int EFFECT_IMAGE`: use an image to fill the text's foreground.
- `int EFFECT_IMAGE_ANIMATION`: use a moving image to fill the text's foreground.
- `int EFFECT_COLOR_ANIMATION`: use a cyclically changing color to fill the text's foreground.

Several instance variables are needed to hold data used by this class:

- `int m_effectIndex`: type of the current effect used to render the text (defaults to `EFFECT_PLAIN`).
- `double m_shearFactor`: shearing factor determining how the text will lean (positive: lean to the left; negative: lean to the right).

`Color m_outlineColor`: color used to outline the text.

`Stroke m_stroke`: stroke instance used to outline the text.

`GradientPaint m_gradient`: color gradient used to fill the text (this only takes effect when `m_effectIndex` is set to `EFFECT_GRADIENT`).

`Image m_foregroundImage`: image used to fill the text (this only takes effect when `m_effectIndex` is set to `EFFECT_IMAGE`).

`Thread m_animator`: thread which produces animation and color cycling.

`boolean m_isRunning`: `true` if the `m_animator` thread is running, `false` otherwise.

`int m_delay`: delay time in ms which determines the speed of animation.

`int m_xShift`: the current image offset in the horizontal direction (this only takes effect when `m_effectIndex` is set to `EFFECT_IMAGE_ANIMATION`).

There are three `JLabel2D` constructors, and each calls the corresponding superclass (`JLabel`) constructor. This class also defines several self-explanatory `set/get` accessors for our instance variables listed above.

The `startAnimation()` method starts our `m_animator` thread which executes every `m_delay` ms. In the case of image animation, this thread periodically increases our `m_xShift` variable which is used in the rendering routine below as an offset. Depending on how smooth we want our animation we can increase or decrease this shift value. Increasing it would give the appearance of speeding up the animation but would make it jumpier. Decreasing it would slow it down but it would appear much smoother.

In the case of color animation, the `startAnimation()` method determines the two colors of the current `m_gradient`, and calculates an intermediate color by 'drawing' a cosine curve between the red, green, and blue components of these colors. Local variable `arg` is incremented by `Math.PI/10` each iteration. We take the cosine of this value and calculate two new values, `f1` and `f2`, based on this result:

```
arg += Math.PI/10;
double cos = Math.cos(arg);
double f1 = (1+cos)/2;
double f2 = (1-cos)/2;
arg = arg % (Math.PI*2);
```

`f1` and `f2` will always sum to 1, and because `arg` is incremented by `Math.PI/10`, we will obtain a consistent cycle of 20 different combinations of `f1` and `f2` as shown below.

arg	f1	f2
0.3141592653589793	0.9755282581475768	0.024471741852423234
0.6283185307179586	0.9045084971874737	0.09549150281252627
0.9424777960769379	0.7938926261462366	0.20610737385376343
1.2566370614359172	0.6545084971874737	0.3454915028125263
1.5707963267948966	0.5	0.49999999999999994
1.8849555921538759	0.34549150281252633	0.6545084971874737
2.199114857512855	0.20610737385376346	0.7938926261462365
2.5132741228718345	0.09549150281252632	0.9045084971874737
2.827433388230814	0.02447174185242323	0.9755282581475768
3.141592653589793	0.0	1.0
3.4557519189487724	0.024471741852423193	0.9755282581475768
3.7699111843077517	0.09549150281252625	0.9045084971874737
4.084070449666731	0.20610737385376338	0.7938926261462367
4.39822971502571	0.3454915028125262	0.6545084971874737
4.71238898038469	0.4999999999999999	0.5000000000000001
5.026548245743669	0.6545084971874736	0.3454915028125264
5.340707511102648	0.7938926261462365	0.20610737385376354
5.654866776461628	0.9045084971874736	0.09549150281252633
5.969026041820607	0.9755282581475767	0.024471741852423234
0.0	1.0	0.0

We then use f1 and f2 as factors for determining how much of the red, green, and blue component of each of the gradient's colors to use for the foreground image:

```
Color c1 = m_gradient.getColor1();
Color c2 = m_gradient.getColor2();
int r = (int)(c1.getRed()*f1+c2.getRed()*f2);
r = Math.min(Math.max(r, 0), 255);
int g = (int)(c1.getGreen()*f1+c2.getGreen()*f2);
g = Math.min(Math.max(g, 0), 255);
int b = (int)(c1.getBlue()*f1+c2.getBlue()*f2);
b = Math.min(Math.max(b, 0), 255);
setForeground(new Color(r, g, b));
```

This gives us 20 distinct colors. We can always increase or decrease this count by increasing or decreasing the denominator of arg respectively (e.g. arg = Math.PI/20 will give 40 distinct colors, and arg = Math.PI/5 will give 10 distinct colors).

The paintComponent() method first calculates the coordinates of the area available for rendering. Since we avoid the call to super.paintComponent() we are responsible for filling the background and border ourselves. So we first check the opaque property (inherited from JLabel) and, if it is set to true, the background is painted manually by filling the component's available region with its background color. We then call the paintBorder() method to render the border (if any) around the component.

Then the Graphics parameter is cast to a Graphics2D instance to obtain access to Java2D features. As we discussed above, two rendering hints are specified using the setRenderingHint() method: anti-aliasing and the priority of quality over speed (see 23.1.10).

A FontRenderContext instance is retrieved from the Graphics2D context and used to create a TextLayout instance using our label's text (refer back to 23.1.13 and 23.1.14). TextLayout's getOutline() method retrieves a Shape which outlines the given text. This method takes an AffineTransform instance as an optional parameter. We use this parameter to shear the text (if our m_shearFactor property is set to a non-zero value).

We then calculate the location of the text depending on the horizontal alignment of our label (center, or right; left needs no special handling). The y-coordinate of the text's baseline is calculated using the top margin, y, the height over 2, h/2, and the ascent over 4, tl.getAscent()/4 (see Figure 23.4). Why ascent/4? See chapter 2, section 2.8.3 for an explanation.

```
float yText = y + h/2 + tl.getAscent()/4;
```

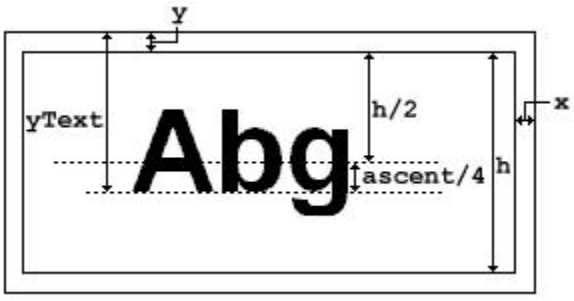


Figure 23.4 Vertical positioning of the label text.
<<figure23-4.gif>>

We then create an AffineTransform for translation to the calculated position. The

createTransformedShape() method is then used to get a Shape representing the outline of the text at that position.

If the m_outlineColor property is set, we draw the text's outline using that color and the assigned stroke instance, m_stroke (if set). The rest of the code fills the interior of the text's Shape we retrieved above using the specified visual effect:

- For EFFECT_GRADIENT we use a GradientPaint instance to fill the interior.
- For EFFECT_IMAGE we use our fillByImage() method to fill the interior with the specified image.
- For EFFECT_COLOR_ANIMATION we simply fill the interior with the foreground color (which changes cyclically in the animation thread).
- For EFFECT_IMAGE_ANIMATION we use our fillByImage() method with the shift parameter (which is periodically incremented by the animation thread).
- For EFFECT_PLAIN we simply fill the interior with the foreground color.

Running the Code

Figure 23.3 shows our JLabel2D demo application in action. You can modify the settings of each label specified in the Labels2D class to try out different combinations of available visual effects. Try out different animation speeds and note how it affects the performance of the application. Choosing too small of a delay may virtually freeze the application. We might consider enforcing a lower bound to ensure that this does not happen.

Note that our JLabel2D component can be easily plugged into any Swing application and used by developers without specific knowledge of Java2D.

23.4 Rendering images

In this section we'll demonstrate the advantages of using Java2D for rendering images and having some fun. The following example is a simple implementation of the well-known Pac-man arcade game. We have designed it such that creating custom levels is very simple, and customizing the appearance is only a matter of building new 20x20 images (you can also change the size of the game's cells to accommodate images of different dimensions). Though there are no monsters, level changes, or sounds, these features are ready and waiting to be implemented by the inspired Pac-man enthusiast. We hope to provide you with a solid base to start from. Otherwise, if you've made it this far through the book without skipping any material, you surely deserve a Pac-man break.

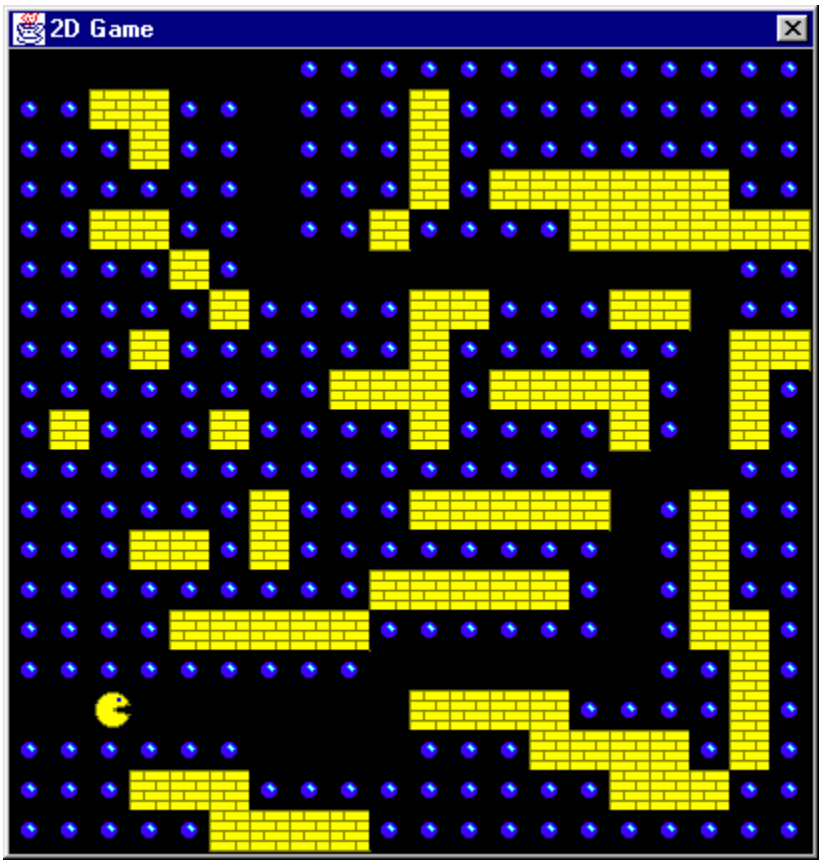


Figure 23.5 Game2D with Pac-man in action.
<<figure23-5.gif>>

The Code: Game2D.java
see \Chapter23\3

```
import java.awt.*;
import java.awt.event.*;
import java.awt.image.*;
import java.awt.geom.*;
import java.util.*;

import javax.swing.*;
import javax.swing.border.*;

public class Game2D extends JFrame
{
  public Game2D() {
    super("2D Game");
    getContentPane().setLayout(new BorderLayout());

    PacMan2D field = new PacMan2D(this);
    getContentPane().add(field, BorderLayout.CENTER);

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);
```

```
    pack(); // no pun intended
    setResizable(false);
    setVisible(true);
  }

  public static void main(String argv[]) { new Game2D(); }
}

class PacMan2D extends JPanel
{
  static final int N_IMG = 4;
  static final int X_CELL = 20;
  static final int Y_CELL = 20;
  static final int N_STEP = 10;
  static final double STEP = 0.1f;

  protected int m_nx;
  protected int m_ny;
  protected int[][] m_flags;
  protected double m_creatureX = 0;
  protected double m_creatureY = 0;
  protected int m_posX = 0;
  protected int m_posY = 0;
  protected int m_creatureVx = 1;
  protected int m_creatureVy = 0;
  protected int m_creatureDir = 0;
  protected int m_creatureInd = 0;
  protected int m_creatureNewVx = 1;
  protected int m_creatureNewVy = 0;
  protected int m_creatureNewDir = 0;

  protected String[] m_data = {
    "00000000000000000000",    // 0
    "00110000001000000000",    // 1
    "00010000001000000000",    // 2
    "00000000001011111100",    // 3
    "00110000010000111111",    // 4
    "00001000000000000000",    // 5
    "00000100001100011000",    // 6
    "00010000001000000011",    // 7
    "00000000111011110010",    // 8
    "01000100001000010010",    // 9
    "00000000000000000000",    // 10
    "00000010001111100100",    // 11
    "00011010000000000100",    // 12
    "00000000011111000100",    // 13
    "00001111100000000110",    // 14
    "00000000000000000010",    // 15
    "00000000001111000010",    // 16
    "00000000000001111010",    // 17
    "00011100000000011100",    // 18
    "00000111100000000000" };  // 19

  protected Image m_wallImage;
  protected Image m_ballImage;
  protected Image[][] m_creature;
  protected Thread m_runner;
  protected JFrame m_parent;

  public PacMan2D(JFrame parent) {
    setBackground(Color.black);
    m_parent = parent;
```

```
AffineTransform[] at = new AffineTransform[3];
at[0] = new AffineTransform(0, 1, -1, 0, Y_CELL, 0);
at[1] = new AffineTransform(-1, 0, 0, 1, X_CELL, 0);
at[2] = new AffineTransform(0, -1, -1, 0, Y_CELL, X_CELL);

ImageIcon icon = new ImageIcon("wall.gif");
m_wallImage = icon.getImage();
icon = new ImageIcon("ball.gif");
m_ballImage = icon.getImage();
m_creature = new Image[N_IMG][4];
for (int k=0; k<N_IMG; k++) {
  int kk = k + 1;
  icon = new ImageIcon("creature"+kk+".gif");
  m_creature[k][0] = icon.getImage();

  for (int d=0; d<3; d++) {
    BufferedImage bi = new BufferedImage(X_CELL, Y_CELL,
      BufferedImage.TYPE_INT_RGB);
    Graphics2D g2 = bi.createGraphics();
    g2.drawImage(m_creature[k][0], at[d], this);
    m_creature[k][d+1] = bi;
  }
}

m_nx = m_data[0].length();
m_ny = m_data.length;
m_flags = new int[m_ny][m_nx];
for (int i=0; i<m_ny; i++)
for (int j=0; j<m_nx; j++)
  m_flags[i][j] = (m_data[i].charAt(j)=='0' ? 0 : 1);

m_runner = new Thread() {
  public void run() {
    m_flags[m_posY][m_posX] = -1;

    while (!m_parent.isShowing())
      try { sleep(150); }
      catch (InterruptedException ex) { return; }

    while (true) {
      m_creatureVx = m_creatureNewVx;
      m_creatureVy = m_creatureNewVy;
      m_creatureDir = m_creatureNewDir;
      int j = m_posX+m_creatureVx;
      int i = m_posY+m_creatureVy;
      if (j >=0 && j < m_nx && i >= 0 && i < m_ny &&
       m_flags[i][j] != 1) {
        for (int k=0; k<N_STEP; k++) {
          m_creatureX += STEP*m_creatureVx;
          m_creatureY += STEP*m_creatureVy;
          m_creatureInd++;
          m_creatureInd = m_creatureInd % N_IMG;
          final int x = (int)(m_creatureX*X_CELL);
          final int y = (int)(m_creatureY*Y_CELL);
          Runnable painter = new Runnable() {
            public void run() {
              PacMan2D.this.paintImmediately(
                x-1, y-1, X_CELL+3, Y_CELL+3);
            }
          };
          try {
```

```
              SwingUtilities.invokeAndWait(painter);
            } catch (Exception e) {}
            try { sleep(40); }
            catch (InterruptedException ex) { break; }
          }
          if (m_flags[i][j] == 0)
            m_flags[i][j] = -1;
          m_posX += m_creatureVx;
          m_posY += m_creatureVy;
          m_creatureX = m_posX;
          m_creatureY = m_posY;
        }
        else
          try { sleep(150); }
          catch (InterruptedException ex) { break; }
      }
    }
  };
  m_runner.start();

  KeyAdapter lst = new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
      switch (e.getKeyCode()) {
        case KeyEvent.VK_RIGHT:
          m_creatureNewVx = 1;
          m_creatureNewVy = 0;
          m_creatureNewDir = 0;
          break;
        case KeyEvent.VK_DOWN:
          m_creatureNewVx = 0;
          m_creatureNewVy = 1;
          m_creatureNewDir = 1;
          break;
        case KeyEvent.VK_LEFT:
          m_creatureNewVx = -1;
          m_creatureNewVy = 0;
          m_creatureNewDir = 2;
          break;
        case KeyEvent.VK_UP:
          m_creatureNewVx = 0;
          m_creatureNewVy = -1;
          m_creatureNewDir = 3;
          break;
      }
    }
  };
  parent.addKeyListener(lst);
}

public Dimension getPreferredSize() {
  return new Dimension(m_nx*X_CELL, m_ny*Y_CELL);
}

public Dimension getMaximumSize() {
  return getPreferredSize();
}

public Dimension getMinimumSize() {
  return getPreferredSize();
}

public void paintComponent(Graphics g) {
```

```
    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
      RenderingHints.VALUE_ANTIALIAS_ON);
    g2.setRenderingHint(RenderingHints.KEY_RENDERING,
      RenderingHints.VALUE_RENDER_QUALITY);

    g2.setColor(getBackground());
    g2.fill(g.getClip());

    int x, y;
    for (int i=0; i<m_ny; i++)
    for (int j=0; j<m_nx; j++) {
      x = j*X_CELL;
      y = i*Y_CELL;
      if (m_flags[i][j] == 1)
        g2.drawImage(m_wallImage, x, y, this);
      else if (m_flags[i][j] == 0)
        g2.drawImage(m_ballImage, x, y, this);
    }

    x = (int)(m_creatureX*X_CELL);
    y = (int)(m_creatureY*Y_CELL);
    g2.drawImage(
      m_creature[m_creatureInd][m_creatureDir], x, y, this);
  }
}
```

Understanding the Code

Class Game2D

This class merely creates a `JFrame` and places an instance of our `PacMan2D` component in it.

Class PacMan2D

This component represents a simple Pac-man implementation. Several class constants are defined:

- `int N_IMG`: number of slides to produce animation of moving Pac-man.
- `int X_CELL`: horizontal size of a cell in the arcade.
- `int Y_CELL`: vertical size of a cell in the arcade.
- `int N_STEP`: number of steps used to produce smooth movement from one cell to another.
- `double STEP`: the length of one step (in terms of cells).

Instance variables:

- `int m_nx`: number of cell columns in a level.
- `int m_ny`: number of cell rows in a level.
- `int[][] m_flags`: two-dimensional array describing the current state of each cell.
- `double m_creatureX`: current x-coordinate of Pac-man in terms of cells. May be fractional because of smooth motion.
- `double m_creatureY`: current y-coordinate of Pac-man in terms of cells.
- `int m_posX`: current x-coordinate of the cell in which Pac-man resides.
- `int m_posY`: current y-coordinate of the cell in which Pac-man resides.
- `int m_creatureVx`: current x-component of Pac-man's velocity (may be `1`, `0`, or `-1`).

`int m_creatureVy`: current y-component of Pac-man's velocity (may be 1, 0, or -1).

`int m_creatureDir`: current direction of Pac-man's motion (0-east, 1-south, 2-west, 3-north). Used for quick selection of the proper image from our `m_creature` 2D array (see below).

`int m_creatureInd`: the index of the current slide in Pac-man's animation.

`int m_creatureNewVx`: a new value for `m_creatureVx` (assigned by the player via keyboard, but not yet accepted by the game).

`int m_creatureNewVy`: a new value for `m_creatureVy`.

`int m_creatureNewDir`: a new value for `m_creatureDir`.

`String[] m_data`: an array of `Strings` which determine the location of walls in a level.

`Image m_wallImage`: an image of a wall occupying one cell.

`Image m_ballImage`: an image of a pellet (to be eaten by Pac-man) occupying one cell.

`Image[][] m_creature`: 2D array of Pac-man images.

`Thread m_runner`: the thread which runs this game.

The constructor of `PacMan2D` performs all necessary initialization. First we create three `AffineTransforms` to be used for flipping our Pac-man images:

```
AffineTransform[] at = new AffineTransform[3];
at[0] = new AffineTransform(0, 1, -1, 0, Y_CELL, 0);
at[1] = new AffineTransform(-1, 0, 0, 1, X_CELL, 0);
at[2] = new AffineTransform(0, -1, -1, 0, Y_CELL, X_CELL);
```

Then, we read in our wall and ball images, and create a 2D array of Pac-man's images. The first row in this array is filled with animated slide images of Pac-man read from four prepared image files. Each of these images represents Pac-man facing east in one of his chomping positions. The next three rows are filled with the same images, but each are transformed to face south (second row), west (third row), and north (fourth row). These flipped images are created in three steps:

Create an empty `BufferedImage` instance the size of the original image.

Retrieve its `Graphics2D` context to draw into that image.

Use the overloaded `drawImage()` method to use an `AffineTransform` instance (prepared above) as a parameter and render the transformed image.

Each resulting image is stored in our `m_creature` array.

The configuration of the game's level is encoded in the `m_data String` array: '0' characters correspond to pellets, '1' characters correspond to walls. This information is then parsed and stored in the `m_flags` array. Thus, the size of the `m_data` array also determines the size of the level (the product of `m_nx` and `m_ny`).

Note: The `m_data String` array can be easily modified to produce a new level. We also can easily modify the program to read this information from an external file. These features would be natural enhancements to make if we were to expand upon this game.

The thread `m_runner` represents the engine of this game. First, it waits while the parent frame is shown on the screen (otherwise wild visual effects may appear). The endless `while` loop manages Pac-man's movement on the screen. The direction of Pac-man's motion may have been changed by the user since the last cycle, so we reassign three parameters which determine that direction from storage variables (`m_creatureVx` from `m_creatureNewVx` etc.). This insures that Pac-man's direction will not change in the middle of a cycle.

We then calculate the coordinates of the next cell `i` and `j` Pac-man will visit. If these coordinates lie inside the level and do not correspond to a wall cell (a 1), we smoothly move Pac-man to the new position. Otherwise we wait until the user provides a new direction.

The movement of Pac-man from the current cell to a new one is split into `N_STEP` steps. On each step we determine the fractional coordinates, `m_creatureX` and `m_creatureY` (in cell units). Then we call `paintImmediately()` to redraw a portion of the level surrounding Pac-man's current location, and pause for 40 ms:

```
final int x = (int)(m_creatureX*X_CELL);
final int y = (int)(m_creatureY*Y_CELL);
Runnable painter = new Runnable() {
  public void run() {
    PacMan2D.this.paintImmediately(
      x-1, y-1, X_CELL+3, Y_CELL+3);
  }
};
try {
  SwingUtilities.invokeAndWait(painter);
} catch (Exception e) {}
try { sleep(40); }
catch (InterruptedException ex) { break; }
```

The `paintImmediately()` method can be used to force very quick repaints but should only be called from within the AWT event dispatching thread. Additionally, because we do not want any other painting or movement to occur while this paint takes place, we wrap the call in a `Runnable` and send it to the event queue with `invokeAndWait()` (refer back to chapter 2 for a discussion of painting and multithreading issues).

When the creature's relocation is over, we eat a pellet (by setting the `m_flags` array element corresponding to the current cell to -1) and adjust Pac-man's coordinate variables.

To listen for the user's keyboard activity we create a `KeyAdapter` instance and add it to the parent component. This `KeyAdapter` processes arrow keys (up, down, left, and right) and assigns new values to the `m_creatureNewVx`, `m_creatureNewVy`, and `m_creatureNewDir` variables accordingly. The program flow is not interrupted, these new values will be requested only on the next thread cycle as discussed above. Note that if the keypads are pressed too fast, only the last typed value will affect the Pac-man's direction.

The `getPreferredSize()` method determines the size of the level, which is simply based on the number and size of cells. Finally the `paintComponent()` method is responsible for rendering the whole game. This process is relatively simple: we render the level using two images (wall and ball) and draw Pac-man's image (taken from our 2D `m_creature` array) in the proper location.

Running the Code

Figure 23.5 shows Pac-man in action. Try the game and have some fun. Experiment with modifying the level and icons for the wall, ball, and Pac-man himself. If you like this example, you might go further and add monsters, score, sound effects, various levels and level changing, and other full-featured game characteristics.

Chapter 24. Accessibility

In this chapter:

- Java 2 Accessibility API overview
- Following accessibility guidelines
- Accessibility support for custom components

24.1 Java 2 Accessibility API overview

When developing an application, it is important to keep in mind that many of it's users may be disabled to some degree. Disabilities can range anywhere from poor vision to blindness, hearing impared to deafness, or a physical inability to use a standard input device such as a mouse or keyboard. Many of these users may often rely on special devices plugged into the computer to help offset specific disabilities. Some of these devices can substitute printed text with audio output, ease keyboard input or point-and-click functionality, perform large scale screen magnification, etc. To do their job correctly, most of these devices need to find connection points within applications, and substitute some of the original functionality with accessible variations.

Java offers sets of classes and interfaces to provide the needed connection points for such special devices. These are referred to as assistive technologies, and are normally packages in a JAR archive. Java loads assistive technologies when the VM is started by looking for special entries in a file called accessibility.properties located in the jdk1.2\jre\lib directory. We will discuss how to use this file below.

24.1.1 Implementing accessibile applications

Fortunately, Swing provides an extremely rich API for accessibility needs. If we use Swing "as is", there are only a few simple things we need to do in order to provide an accessible interface for assistive technologies:

Use tooltips to provide a text description of a component's purpose. The tooltip text may be used by accessible devices to provide a description of that component.

Enable keyboard access. Ideally, all functionality should be available without using the mouse. Use mnemonics for buttons and menus, set keyboard accelerators in menus.

Associate labels with components. By setting both the `displayedMnemonic` and `labelFor` properties, a label will transfer focus to the specified component with the `labelFor` property when the `displayedMnemonic` is activated (i.e. when ALT + mnemonic character is pressed).

Logically group components with labels refering to each when necessary.

Throughout this book we have adhered to most of these rules without explicitly being aware that we were also building accessible-compliant applications. Let's review a bit more about the association of labels with components. Consider the following code:

```
JPanel p = new JPanel();
p.add(new JLabel("IP address:"));
JTextField user = new JTextField();
p.add(user);
```

To make this code accessible we would need to associate the label with the textfield above as follows:

```
JPanel p = new JPanel();
```

```
JLabel lbl = new JLabel("IP address:");
lbl.setDisplayedMnemonic('n');
p.add(lbl);
JTextField user = new JTextField();
user.setToolTipText("IP address");
lbl.setLabelFor(user);
p.add(user);
```

The changes made should be familiar and are relatively simple. First, the `setDisplayedMnemonic()` method assigns a mnemonic character to the label component (this character will be underlined in the label if the label's text contains that character). Second, a tooltip text is assigned to the text field to provide an accessible text description. Finally, the `setLabelFor()` method is called to associate the text field component with the label. When the mnemonic is activated, the label will notify the text field to request the focus.

Note: As we've mentioned in chapter 19, due to a Swing bug or oversite, a mnemonic character will be passed to the text field as input. We will not concentrate on a workaround in this chapter, assuming that it will be fixed in future Swing releases.

Reference: To find out more about writing accessible Java applications, we refer you to IBM's guidelines at http://www.austin.ibm.com/sns/access.html.

Java 2 ships with the `javax.accessibility` package, which includes classes and interfaces constituting the Java Accessibility API. In the remainder of this section we'll give a brief introduction to some of its most important constituents.

Note: You may not need to explicitly use this package in your Swing application. Almost all of Swing has accessibile functionaliy built into it.

24.1.3 The Accessible interface

abstract interface javax.accessibility.Accessible

This interface must be implemented by all Java classes designed to support accessibility. All Swing components implement this interface either directly or indirectly. It declares only one method, `getAccessibleContext()`, which returns an instance of `AccessibleContext` (see below).

24.1.4 AccessibleContext

abstract class javax.accessibility.AccessibleContext

This class provides the minimum information all accessible objects should expose. This information includes the accessible: name, description, role, and state of the object, as well as information about the parent and children of the object. To obtain more specific accessibility information about a component this class exposes methods to retrieve the instances of interfaces defined in the `accessibility` package (see below).

Typically Swing components define an inner class which extends `AccessibleContext` and provides information specific to that component. For instance, `JComponent` provides the inner class `AccessibleJComponent`, `JComboBox` provides `AccessibleJComboBox` etc.

24.1.5 AccessibleRole

class javax.accessibility.AccessibleRole

This class defines several static constants used in providing information about the role a particular component plays in an application. Normally the `getAccessibleRole()` method of `AccessibleContext` will return one of these constants.

24.1.6 The AccessibleAction interface

abstract interface javax.accessibility.AccessibleAction

This interface declares methods to perform one or more accessible actions, retrieve descriptions of these actions, and find out exactly how many actions have been exposed to assistive technologies.

- `int getAccessibleActionCount()`: returns the number of actions exposed through the `doAccessibleAction()` method.
- `String getAccessibleActionDescription(int i)`: should be defined to return a description of the code executed by `doAccessibleAction()` based on the given index.
- `boolean doAccessibleAction(int i)`: When implementing this interface we are expected to define code to be executed based on a given index. Normally this is involves invoking event handling code defined for a specific component, however, there is nothing stopping us from defining accessible-specific code here. It should return a boolean value representing whether or not the given index corresponds to a valid action.

24.1.7 The AccessibleComponent interface

abstract interface javax.accessibility.AccessibleComponent

This interface should be supported by any accessible component that is rendered to a graphical context. It provides the standard mechanism for an assistive technology to determine and assign certain aspects of the graphical representation of that component. Most methods will call the corresponding component's methods where applicable. (Note that no painting methods are members of this interface because we are expected to define our own look-and-feel if a more customized view is desired.)

24.1.8 The AccessibleHypertext interface

abstract interface javax.accessibility.AccessibleHypertext

This interface provides standard mechanisms for determining and manipulating hyperlinks with an assistive technology. Normally only text components will provide implementations of this interface, however, any other component supporting hypertext may do the same.

24.1.9 The AccessibleSelection interface

abstract interface javax.accessibility.AccessibleSelection

This interface should be implemented by any accessible component that has selectable children, whether individually or in groups, such as trees, tables, lists, tabbed panes, menu bars, etc.

24.1.10 The AccessibleText interface

abstract interface javax.accessibility.AccessibleText

This interface should be implemented by any accessible text component, and is intended to provide access to that component's document content and attributes (where applicable).

24.1.11 The AccessibleValue interface

abstract interface javax.accessibility.AccessibleValue

This interface should be implemented by any accessible component that supports a numerical value (e.g. a scrollbar). It is intended to be used by an assistive technology to determine and assign current, minimum, and maximum values.

24.1.12 Accessibility Utilities

In order to provide accessibility compliance between an application and an assistive technology, we cannot just rely on our use of the accessibility package. We also require support for locating the objects that implement accessibility functionality, as well as support for loading assistive technologies into the Java virtual machine and tracking all events sent to the system event queue. For these reasons, Sun provides the Java Accessibility Utilities which can be downloaded from the JavaSoft web site. Along with utility classes to perform these services, a set of sample applications is included which can be used to test and debug accessible Java programs. These applications are capable of linking with the system event queue and displaying information about selected types of events.

The Java Accessibility Utilities package is an assistive technology itself. To install it we must place the jaccess.jar and jaccess-examples.jar files into our jdk1.2\jre\lib\ext directory.

To use one of its assistive-technology examples, we need to include a string such as the following in our accessibility.properties file:

```
assistive_technologies=Explorer
```

This will cause the Explorer utility to start each time we start a new JVM. (Note that if this file does not already exits we must create it ourselves.) In the examples below we assume that you have enabled the Explorer utility, and it is shown in the corresponding example figures.

Reference: A detailed description of the Java Accessibility Utilities lies beyond the scope of this book. Please see http://java.sun.com/products/jfc/jaccess-1.2/doc/guide.html for more information.

Note: Accessibility Utilities can be used by any Java application to monitor the content of the system event queue for debugging purposes.

24.2 Following accessibility guidelines

Let's take a look at how the guidelines for writing an accessible application can be fully adhered to in one of our previous examples: the FTP Client application from chapter 13. As you will see, only minor changes need to be made for full compliance.

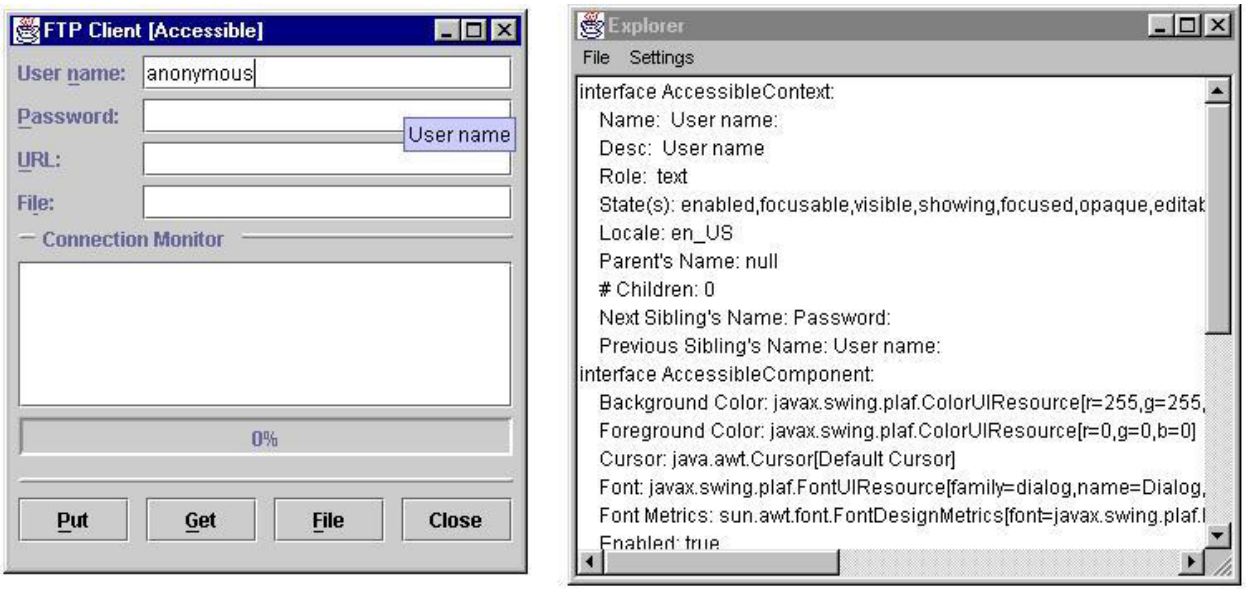


Figure 24.1 FTP Client application with added accessibility support, and monitored using the Explorer utility.
<<file figure24-1.gif>>

The Code: Access1.java
see \Chapter24\1

```
// Unchanged imports from section 13.5

public class Access1 extends JFrame
{
  // Unchanged code from section 13.5

  public Access1()
  {
    super("FTP Client [Accessible]");
    getContentPane().setLayout(new BorderLayout());

    JPanel p = new JPanel();
    p.setLayout(new DialogLayout2(10, 5));
    p.setBorder(new EmptyBorder(5, 5, 5, 5));

    JLabel lbl = new JLabel("User name:");
    lbl.setDisplayedMnemonic('n');
    p.add(lbl);
    m_txtUser = new JTextField("anonymous");
    m_txtUser.setToolTipText("User name");
    p.add(m_txtUser);
    lbl.setLabelFor(m_txtUser);

    lbl = new JLabel("Password:");
    lbl.setDisplayedMnemonic('p');
    p.add(lbl);
    m_txtPassword = new JPasswordField();
    m_txtPassword.setToolTipText("Password");
    p.add(m_txtPassword);
    lbl.setLabelFor(m_txtPassword);

    lbl = new JLabel("URL:");
    lbl.setDisplayedMnemonic('u');
    p.add(lbl);
```

```
    m_txtURL = new JTextField();
    m_txtURL.setToolTipText("URL");
    p.add(m_txtURL);
    lbl.setLabelFor(m_txtURL);

    lbl = new JLabel("File:");
    lbl.setDisplayedMnemonic('l');
    p.add(lbl);
    m_txtFile = new JTextField();
    m_txtFile.setToolTipText("File");
    p.add(m_txtFile);
    lbl.setLabelFor(m_txtFile);

    // Unchanged code from section 13.5

    p.add(new DialogSeparator());
    m_btPut = new JButton("Put");
    m_btPut.setToolTipText("Upload file");
    // Unchanged code from section 13.5
    p.add(m_btPut);

    m_btGet = new JButton("Get");
    m_btGet.setToolTipText("Download file");
    // Unchanged code from section 13.5
    p.add(m_btGet);

    m_btFile = new JButton("File");
    m_btFile.setToolTipText("Select file");
    // Unchanged code from section 13.5
    p.add(m_btFile);

    m_btClose = new JButton("Close");
    m_btClose.setToolTipText("Quit application");
    // Unchanged code from section 13.5
    p.add(m_btClose);

    // Unchanged from section 13.5
```

Understanding the Code

Class Access1

Compared to the example in chapter 13 we have made some a very minor amount of changes, and those we did make are quite simple. All labels receive a `displayedMnemonic` property and are associated with the corresponding text fields using the `setLabelFor()` method. We've also added tooltip text for each text field and label.

Running the Code

At this point you can compile and execute this example. Figure 24.1 shows our FTP Client application and the Explorer accessible utility displaying events generating by that application. Unfortunately we cannot implement actual support for disabled users because no real assistive technologies are currently available.

24.3 Accessibility for custom components

In developing professional custom Swing components, special care should be taken to implement the `Accessible` interface and provide a custom extension of the `AccessibleContext` class. The following example extends the functionality of our `OpenList` component, developed and used in the examples of chapter 20, to comply with the accessibility standards.

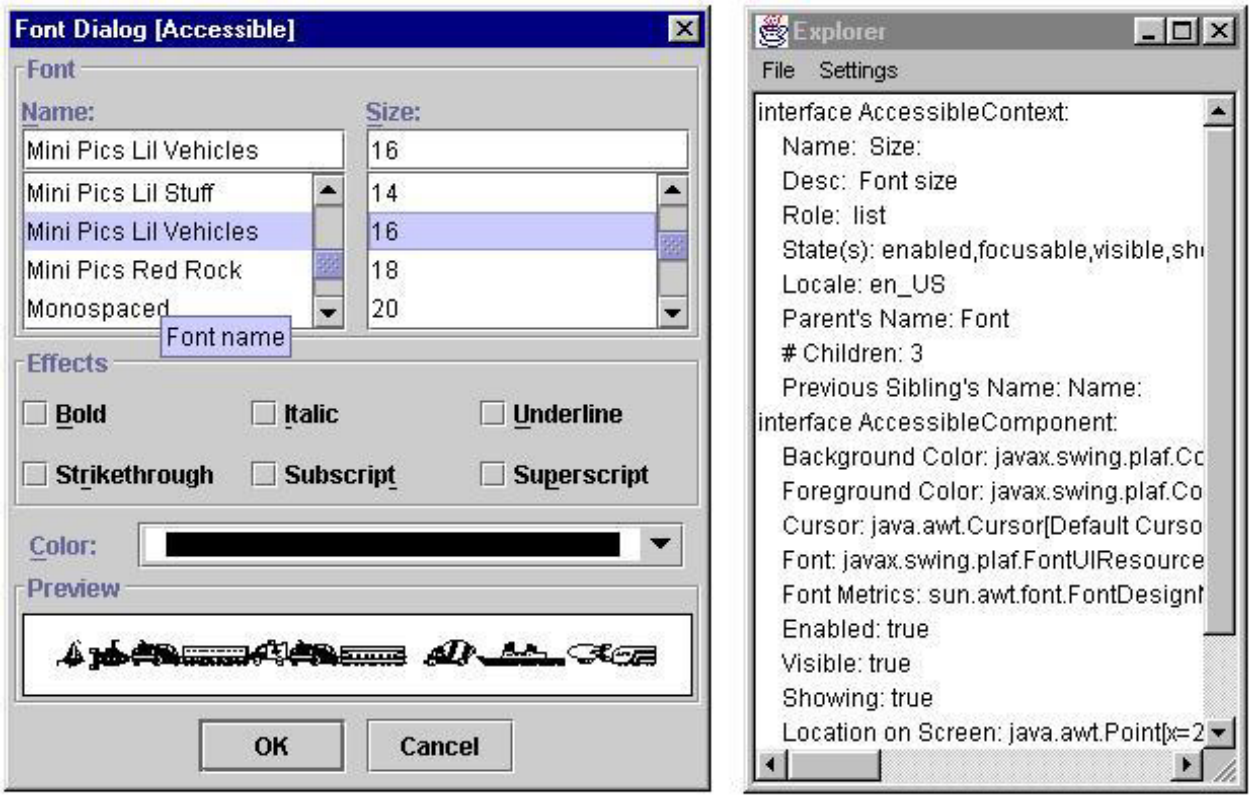


Figure 24.2 Custom fontdialog using accessible OpenList components, and monitored using the Explorer utility.
<<file figure24-2.gif>>

The Code: Access2.java
see \Chapter24\2

```
import javax.accessibility.*;

public class Access2 extends JDialog
{
  // added mnemonics and tooltip text to all relevant components
  // ...this class is used to set up a font dialog and has been
  // extracted from code in chapter 20. It requires little
  // presentation or explanation here.

  public static void main(String argv[]) {
    GraphicsEnvironment ge = GraphicsEnvironment.
      getLocalGraphicsEnvironment();
    m_fontNames = ge.getAvailableFontFamilyNames();
    m_fontSizes = new String[] {"8", "9", "10", "11", "12", "14",
      "16", "18", "20", "22", "24", "26", "28", "36", "48", "72"};

    Access2 dlg = new Access2(new JFrame());
    SimpleAttributeSet a = new SimpleAttributeSet();
    StyleConstants.setFontFamily(a, "Monospaced");
    StyleConstants.setFontSize(a, 12);
    dlg.setAttributes(a);
    dlg.show();
  }
}

class OpenList extends JPanel
 implements ListSelectionListener, ActionListener
```

```
{
  protected JLabel m_title;
  protected JTextField m_text;
  protected JList m_list;
  protected JScrollPane m_scroll;

  public OpenList(String[] data, String title)
  {
    setLayout(null);
    m_title = new OpenListLabel(title, JLabel.LEFT);
    add(m_title);
    m_text = new OpenListText();
    m_text.addActionListener(this);
    m_title.setLabelFor(m_text);
    add(m_text);
    m_list = new OpenListList(data);
    m_list.setVisibleRowCount(4);
    m_list.addListSelectionListener(this);
    m_scroll = new JScrollPane(m_list);
    add(m_scroll);
  }

  public OpenList(String title, int numCols) {
    setLayout(null);
    m_title = new OpenListLabel(title, JLabel.LEFT);
    add(m_title);
    m_text = new OpenListText(numCols);
    m_text.addActionListener(this);
    m_title.setLabelFor(m_text);
    add(m_text);
    m_list = new OpenListList();
    m_list.setVisibleRowCount(4);
    m_list.addListSelectionListener(this);
    m_scroll = new JScrollPane(m_list);
    add(m_scroll);
  }

  public void setToolTipText(String text) {
    super.setToolTipText(text);
    m_title.setToolTipText(text);
    m_text.setToolTipText(text);
    m_list.setToolTipText(text);
  }

  public void setDisplayedMnemonic(char ch) {
    m_title.setDisplayedMnemonic(ch);
  }

  public AccessibleContext getAccessibleContext() {
    if (accessibleContext == null)
      accessibleContext = new AccessibleOpenList();
    return accessibleContext;
  }

  // Unchanged code from section 20.9

  class OpenListLabel extends JLabel {
    public OpenListLabel(String text, int alignment) {
      super(text, alignment);
    }

    public AccessibleContext getAccessibleContext() {
```

```
      return OpenList.this.getAccessibleContext();
    }
  }

  class OpenListText extends JTextField
  {
    public OpenListText() {}

    public OpenListText(int numCols) {
      super(numCols);
    }

    public AccessibleContext getAccessibleContext() {
      return OpenList.this.getAccessibleContext();
    }
  }

  class OpenListList extends JList
  {
    public OpenListList() {}

    public OpenListList(String[] data) { super(data); }

    public AccessibleContext getAccessibleContext() {
      return OpenList.this.getAccessibleContext();
    }
  }

  protected class AccessibleOpenList extends AccessibleJComponent
  {
    public String getAccessibleName() {
      if (accessibleName != null)
        return accessibleName;
      return m_title.getText();
    }

    public AccessibleRole getAccessibleRole() {
      return AccessibleRole.LIST;
    }
  }
}
```

Understanding the Code

Class Access2

This class is mainly extracted from the `Editor6` class of chapter 20 and represents an extension of our `FontDialog` component. We have simply added tooltip text and mnemonics to each of its children and have defined a `main` method to construct and display an instance of this component.

Class OpenList

Compared to the example in chapter 20 we haven't changed much. We now sub-class three components to build inner class, accessible counterparts: `OpenListLabel` extends `JLabel`, `OpenListText` extends `JTextField`, and `OpenListList` extends `JList` (see below for more about these sub-classes). Also note that the label component is linked with the text component using the `setLabelFor()` method as discussed above.

The `setToolTipText()` method is overridden to assign a given tooltip text to all three sub-components. The `setDisplayedMnemonic()` method is exposed assign the `displayedMnemonic` property of the label component. In conjunction with the `setLabelFor()` method call included in the constructors, this

provides additional keyboard access to our `OpenList` component.

The `getAccessibleContext()` method is the only remaining change made to our `OpenList` class. It is required for implementation of the `Accessible` interface inherited from its `JPanel` ancestor. Our implementation fills the `accessibleContext` instance variable (inherited from `JComponent`) by creating an instance of our `AccessibleOpenList` inner class, and returns it as an `AccessibleContext`.

Class OpenList.OpenListLabel

This inner class extends `JLabel` to override the `getAccessibleContext()` method and delegates the call to `OpenList`'s `getAccessibleContext()` method. The same is true for inner `OpenListText` and `OpenListList` sub-classes of `JTextField` and `JList` respectively. This guarantees that the `AccessibleOpenList` will be used as the accessible context, no matter which sub-component currently has the focus. In this way, information about `OpenList` and each of its constituent child components is always grouped together and treated as one component by assistive technologies.

Class OpenList.AccessibleOpenList

This inner class extends the `AccessibleJComponent` class (an extension of `AccessibleContext` provided by the `JComponent` ancestor) to provide accessible information about our component. The `getAccessibleName()` method returns the text of the `m_title` label as the accessible name (unless this has been set directly with the `setAccessibleName()` method). The `getAccessibleRole()` method returns `AccessibleRole.LIST` as the closest pre-defined accessible role (see 24.1.5).

Note: We could have gone much further in developing a custom accessible context. However, in the absence of real assistive technologies this does not make much sense. We hope to have provided a sufficient background to get you started when these are made available.

Running the Code

At this point you can compile and execute this example. Figure 24.2 shows the font dialog from our word processor application of chapter 20 now using an accessible variant of the `OpenList` component. Notice that when focus is transferred to an `OpenList` component an instance of `AccessibleOpenList`, described above, is used as the accessible context (the Explorer window in figure 24.2 illustrates).

Chapter 25. JavaHelp

In this chapter:

- JavaHelp introduction
- JavaHelp API overview
- Basic help example
- Adding dialog-style help
- Customizing the help viewer
- Creating a custom help view

25.1 JavaHelp introduction

Most software developers do not look forward to spending time documenting and explaining their product. However, the commercial success of any modern software project greatly depends on the quality and availability of the help information provided to end users. As Java apps evolve into more extensive and sophisticated solutions, they require a robust, uniform, and effective mechanism for providing help to these users. For this reason JavaSoft has developed the JavaHelp API: a full-featured, platform-independent, extensible help system that gives developers and authors the ability to efficiently incorporate an online help system into applications (both network and stand-alone), applets, JavaBean components, HTML pages, operating systems, and devices. The 1.0 release of this API is available for download from the JavaSoft web site (see http://java.sun.com/products/javahelp/).

Note: You will need to download and install the JavaHelp API before running the examples in this chapter.

JavaHelp includes a standard Swing-based help viewer illustrated in figure 25.1 below. This viewer contains a toolbar in its northern region and a split pane in the center. The toolbar contains two buttons allowing navigation between help topics that have been visited previously in a forward/backward style. The left split pane component is a tabbed pane containing a table of contents, help index, and search navigation facilities. The right split pane component is an editor pane used to display the content of the currently selected help topic. Help content is expected to be in the form of HTML documents, enabling rich display capabilities and greatly simplifying the delivery of help files across the Internet.

Note: JavaHelp also provides a way to deliver multi-lingual help depending on the currently selected locale. This capability, along with the more comprehensive internationalization support provided by Java 2, is very powerful, currently under major development, and lie beyond the scope if this text.

To use JavaHelp in an application we need to prepare a set of files described below. With the exception of content files and search files, each file is expected to be in the XML format. (In describing each XML-based file below, we present a simple example implementation which we will use in the section 25.3 application.)

Reference: For more information about XML see http://www.w3.org/TR/WD-xml-lang-970331.html. It is likely that in the near future, we will have graphical tools available to facilitate quick and easy construction of these system files. However, at this point we are forced to delve into some of the details of XML coding to accomplish this.

25.1.1 HelpSet file

This system file uses the XML format to deliver information about other help components and files. XML tags used in this file have the following meaning:

- `<helpset>`: top level of a HelpSet document. The `<version>` and `<lang>` attributes can specify file version and language respectively.
- `<title>`: title to be displayed in the help viewer's title bar.
- `<maps>`: defines a section representing a HelpSet map (discussed below).
  - `<homeID>`: ID of the default help topic, which should be listed in the map file.
  - `<mapref>`: specifies a map file or URL with its `<location>` attribute.
- `<view>`: defines a section representing a help view contained in the help viewer tabbed pane. A HelpSet file may contain an arbitrary number of views.
  - `<name>`: a name identifying the view.

`<label>`: used for displaying the tooltip text of the view's corresponding tabbed pane tab (more descriptive than a name).

`<type>`: the view type (fully qualified name of the class to be instantiated to create this view component). Normally a subclass of `NavigatorView` (see below).

`<data>`: specifies a data file or URL used in this view. The optional `<engine>` attribute can be used to define the fully qualified name of the class defining a search engine.

The following is a typical example of a HelpSet file, `Layout.hs`:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<helpset version="1.0">
  <title>Basic Layout Example - Help</title>
  <maps>
     <homeID>top</homeID>
     <mapref location="Layout.jhm"/>
  </maps>
  <view>
    <name>TOC</name>
    <label>Table Of Contents</label>
    <type>javax.help.TOCView</type>
    <data>LayoutTOC.xml</data>
  </view>
  <view>
    <name>Index</name>
    <label>Help Index</label>
    <type>javax.help.IndexView</type>
    <data>LayoutIndex.xml</data>
  </view>
  <view>
    <name>Search</name>
    <label>Search</label>
    <type>javax.help.SearchView</type>
    <data>JavaHelpSearch</data>
  </view>
</helpset>
```

25.1.2 Map file

This system file uses the XML format to provide a mapping of help topic IDs (text strings) to URLs (which can be either local or remote and of the form `file:`, `http:`, `ftp:`, or `jar:`). Multiple map files can be used in a HelpSet. In such a case the mappings will be merged and we are expected to always provide unique IDs because of this. XML tags used in this file have the following meaning:

`<map>`: top level of the map document. The `<version>` and `<lang>` attributes can specify file version and language respectively.

`<mapID>`: represents a single map entry. The `<target>` attribute specifies the string ID of the help topic and the `<url>` attribute specifies the associated URL of the HTML content file constituting its documentation. The optional `<lang>` attribute allows specification of a language to use when displaying this topic.

The following is a typical example of a map file, `Layout.jhm`:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<map version="1.0">
    <mapID target="top" url="LayoutFrame.html" />
    <mapID target="FlowLayout" url="FlowLayout.html" />
    <mapID target="GridLayout" url="GridLayout.html" />
```

```
    <mapID target="BorderLayout" url="BorderLayout.html" />
    <mapID target="BoxLayout" url="BoxLayout.html" />
</map>
```

25.1.3 Table of Contents (TOC) file

This system file uses the XML format to deliver information about a help topics table of contents. This represents a view that typically contains references to help topic IDs in a logically grouped order. XML tags used in this file have the following meaning:

- `<toc>`: top level of TOC document. The `<version>` and `<lang>` attributes can specify file version and language respectively.
- `<tocitem>`: defines a TOC entry. The `<target>` attribute specifies the string ID of the target help item listed in a map file. The `<text>` attribute specifies the TOC view text to represent the corresponding help topic.

The following is an example of a TOC file, `LayoutTOC.xml`:

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<toc version="1.0">
    <tocitem>Basic Layout Example - TOC
        <tocitem text="Frame" target="top"/>
        <tocitem text="FlowLayout" target="FlowLayout"/>
        <tocitem text="GridLayout" target="GridLayout"/>
        <tocitem text="BorderLayout" target="BorderLayout"/>
        <tocitem text="BoxLayout" target="BoxLayout"/>
    </tocitem>
</toc>
```

25.1.4 Index file

This system file uses the XML format to deliver information representing a help index view. This view is very similar to a TOC view described above, but typically organizes references to help topics in alphabetical order. XML tags used in this file have the following meaning:

- `<index>`: top level of a help index document. The `<version>` and `<lang>` attributes can specify file version and language respectively.
- `<indexitem>`: defines index entry. Attribute `<target>` contains string ID of the target help item listed in the map file. Attribute `<text>` contains text to be displayed in the index.

  Here's an example of the index file, `LayoutIndex.xml`:

```
<?xml version='1.0' encoding='ISO-8859-1'  ?>
<index version="1.0">
  <indexitem text="Index">
    <indexitem text="B">
      <indexitem target="BorderLayout" text="BorderLayout"/>
      <indexitem target="BoxLayout" text="BoxLayout"/>
    </indexitem>
    <indexitem text="F">
      <indexitem target="FlowLayout" text="FlowLayout"/>
      <indexitem target="top" text="Frame"/>
    </indexitem>
    <indexitem text="G">
      <indexitem target="GridLayout" text="GridLayout"/>
    </indexitem>
  </indexitem>
</index>
```

25.1.5 Help Content files

JavaHelp uses the HTML format for data files making up the actual help topic content. We can place help topics in one or more HTML files using typical fonts, images, and links to create a robust and efficient help system. We can also embed `JComponents` using the <OBJECT> tag (see chapter 19 for more about Swing's HTML support).

25.1.6 Search files

JavaHelp supports access to search engines, and provides a default search engine of its own: `com.sun.java.help.search.DefaultSearchEngine`. Search engines used to perform searches are subclasses of `javax.help.search.SearchEngine` (as is `DefaultSearchEngine`). A JavaHelp search engine uses a search database residing in the directory specified by the `<data>` tag of the search view section in the HelpSet file (see above). This directory contains necessary search data pre-generated based on given content. This data is searched using a specified engine pointed at by the `<data>` tag's `engine` attribute (if none is specified the default is used).

To generate binary files constituting the necessary search data, we first create content files for our help system (usually HTML), change to the directory containing these files, and run the following command listing each of our help content files as parameters:

```
com.sun.java.help.search.Indexer file1 file2 ...
```

This generates several files and places them in the default "JavaHelpIndex" sub-directory (see the JavaHelp documentation for more options available in generating search data).

The JavaHelp search mechanism can be used to support client-side searching, server-side searching, and stand-alone seaching. As with the whole JavaHelp system, we are provided with very flexible choices in both the presentation and deployment of this information.

25.2 JavaHelp API overview

25.2.1 HelpSet

class javax.help.HelpSet

This class represents a collection of help information files we discussed above: HelpSet, table of contents (TOC), index, content, and map files. To create a `HelpSet` instance we must supply two parameters: a `ClassLoader` instance used to locate any classes required by the navigators in the help set, and the `URL` of the main HelpSet file. The following code shows a typical `HelpSet` instantiation, assuming the main HelpSet file, `MyHelpSet.hs`, is contained in the current running directory:

```
ClassLoader loader = this.getClass().getClassLoader();
URL url = HelpSet.findHelpSet(loader, "MyHelpSet.hs");
HelpSet hs = new HelpSet(loader, url);
```

25.2.2 The HelpBroker interface

abstract interface javax.help.HelpBroker

This interface describes a component used to manage the presentation and interaction with a `HelpSet`. We can use the `HelpSet.createHelpBroker()` method to retrieve an instance of `HelpBroker`. We can then use `HelpBroker`'s `enableHelpKey()` method, specifying the `String` ID of a desired help topic, to

enable JavaHelp on a given Component. This is often used on a JRootPane (see chapter 3):

```
HelpBroker hb = hs.createHelpBroker();
hb.enableHelpKey(myFrame.getRootPane(), "MyTopicID", hs);
```

When F1 is pressed while that component, or one of its children, has the focus, the JavaHelp viewer will appear displaying the appropriate help topic. If a child component has the focus and has an associated help ID assigned, the help viewer will display the help topic specific to that component. Otherwise the default topic (specified as "MyTopicID" above) will be displayed.

We can also use the enableHelpOnButton() method to enable the display of JavaHelp when a specific button is pressed:

```
JButton btHelp = new JButton("Help");
hb.enableHelpOnButton(btHelp, "MyTopicID", null);
```

A default implementation is provided by the javax.help.DefaultHelpBroker class.

25.2.3 The Map interface

abstract interface javax.help.Map

This interface defines a String ID to URL mapping. The inner class Map.ID is used to identify a help topic. It encapsulates a HelpSet reference /String ID pair. Given an ID we can get the associated URL and vice versa. Each HelpSet has an associated Map instance.

25.2.4 JHelp

class javax.help.JHelp

This class represents the main JavaHelp viewer capable of displaying help set data using JTree or JList navigators in a JTabbedPane, and a content viewer (normally a JEditorPane for HTML display). The current implementation of JavaHelp does not provide public access to this class via HelpSet or HelpBroker instances. Later in this chapter we'll see how to gain access to this component for customization.

25.2.5 The HelpModel interface

abstract interface javax.help.HelpModel

This interface represents the model of a JHelp viewer component. It maintains IDs and URLs corresponding to the HelpSet being displayed in a JHelp instance, and fires HelpModelEvents (see below) when the current help topic is changed.

25.2.6 JHelpNavigator

class javax.help.JHelpNavigator

This class represents a navigation control (JTree or JList) for the GUI presentation of help topic data. Instances of JHelpNavigator reside in the tabbed pane on the left side of the JHelp viewer. Any number of navigators can be present in a HelpSet. JavaHelp provides three sub-classes of JHelpNavigator used in JHelp by default: JHelpIndexNavigator, JHelpSearchNavigator, and JHelpTOCNavigator. Each JHelpNavigator displays data retrieved through a specific NavigatorView instance.

25.2.7 NavigatorView

abstract class javax.help.NavigatorView

Instances of `NaviagtorView` define what type of data a view accepts and how it is parsed. They aso define how data will be presented visually by specifying a corresponding component used to view it. JavaHelp provides three sub-classes of `NavigatorView` used in `JHelp` by default: `IndexView`, `SeachView`, and `TOCView`--each of which specify the corresponding `JHelpNavigator` subclass discussed above for representation.

25.2.8 CSH

class javax.help.CSH

This class provides simple access to context-sensitive help by supplying several static methods. Context-sensitive help is the provision of certain help information based on the user's current task. Implementing this type of help often involves associating help topics with each component in a GUI and tracking context-sensitive events. Particularly important is the `CSH.setHelpIDString(JComponent comp, String helpID)` method, which assigns a `String` ID to a given Swing component. When all IDs are assigned, we can create a button or menu item to trigger content sensitive help, and attach a special predefined `ActionListener`:

```
JButton btItemHelp = new JButton("Item Help");
btItemHelp.addActionListener(new CSH.DisplayHelpAfterTracking(hb));
```

When this button is pressed, a custom mouse cursor displaying a pointer and a question mark appears. When a component is clicked the JavaHelp viewer is displayed showing the help topic corresponding to the selected component's assigned ID (if it has one).

25.2.9 TreeItem

class javax.help.TreeItem

Instances of this class are used as user objects in a navigational view's `JTree` or `JList`. Subclasses of `TreeItem` include `IndexItem`, used in the Index navigator, and `TOCItem`, used in the TOC navigator. A `SearchTOCItem` class extends `TOCItem` and is used in the Search navigator.

25.2.9 The HelpModelListener interface

abstract interface javax.help.event.HelpModelListener

This interface describes a listener which receives `javax.help.event.HelpModelEvents` when the current help topic is changed (i.e. whenever a new topic is selected in a navigator). `HelpModellListener` implementations must implement the `idChanged()` method which takes a `HelpModelEvent` as parameter. We can add a `HelpModelListeners` to a `HelpModel` using its `addHelpModelListener()` method.

25.2.10 Setting up JavaHelp

To run all examples in this chapter you will need to do the following (this will vary depending on your platform):

1. Download and install JavaHelp 1.0 (or the most recent release).
2. Copy all files in your `javahelp\bin` directory to your `jdk1.2\bin` directory.
3. Copy the contents of your `javahelp\lib` directory to your `jdk1.2\lib` directory.

4. Each example includes Windows batch files set up with the assumption that you have installed Java 2, and the above files, in `c:\jdk1.2`. If you are not using Windows or do not have Java 2 installed in this directory, you will need to write your own scripts or modify these batch files accordingly.

Note: The JavaHelp API now exists as a Java extension. In future Java 2 releases it is possible that JavaHelp will become part of the core API.

25.3 Basic JavaHelp example

In this section we will add help capabilities to the introductory layout example from chapter 4, section 4.2, demonstrating five `JInternalFrames` each using a different layout manager. Each of these internal frames will receive a help topic placed in a separate HTML file. The HelpSet, map, TOC, and index files for this example were listed in section 25.1, and are used to create a `HelpSet` instance. We will assign `String` IDs to each internal frame using the `CSH` class and attach an `InternalFrameListener` to transfer focus to a the first component in a selected frame's content pane (because focus does not get transferred by default when a new internal frame is selected). By using our `HelpSet`'s `HelpBroker` we will then enable help on each frame, which can be invoked through the F1 key.

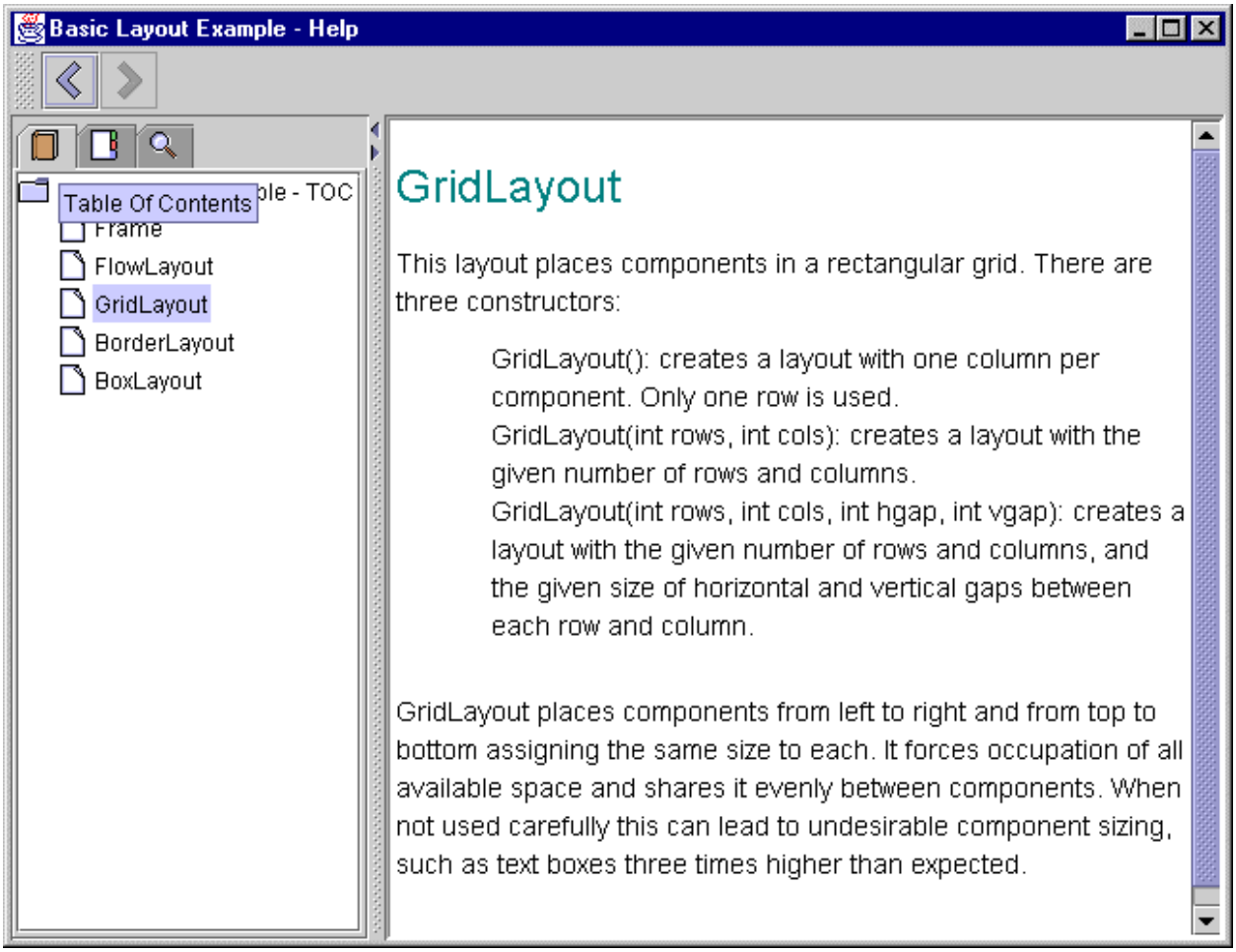


Figure 25.1 JavaHelp help viewer displaying the TOC view.

<<file figure25-1.gif>>

The Code: CommonLayoutsDemo.java
see \Chapter25\1

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.net.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.help.*;

public class CommonLayoutsDemo extends JFrame
{
  protected HelpSet m_hs;
  protected HelpBroker m_hb;
  static final String HELPSETNAME = "Help/Layout.hs";

  public CommonLayoutsDemo() {
    super("Layout Managers [Help]");
    setSize(500, 380);

    createHelp();
    InternalFrameAdapter activator = new InternalFrameAdapter() {
      public void internalFrameActivated(InternalFrameEvent e) {
        JInternalFrame fr = (JInternalFrame)e.getSource();
        Component c = fr.getContentPane().getComponent(0);
        if (c != null)
          c.requestFocus();
      }
    };

    JDesktopPane desktop = new JDesktopPane();
    getContentPane().add(desktop);

    JInternalFrame fr1 =
      new JInternalFrame("FlowLayout", true, true);
    // Unchanged code from section 4.2
    CSH.setHelpIDString(fr1,"FlowLayout");
    fr1.addInternalFrameListener(activator);

    JInternalFrame fr2 =
      new JInternalFrame("GridLayout", true, true);
    // Unchanged code from section 4.2
    CSH.setHelpIDString(fr2,"GridLayout");
    fr2.addInternalFrameListener(activator);

    JInternalFrame fr3 =
      new JInternalFrame("BorderLayout", true, true);
    // Unchanged code from section 4.2
    CSH.setHelpIDString(fr3,"BorderLayout");
    fr3.addInternalFrameListener(activator);

    JInternalFrame fr4 =
      new JInternalFrame("BoxLayout - X", true, true);
    // Unchanged code from section 4.2
    CSH.setHelpIDString(fr4,"BoxLayout");
    fr4.addInternalFrameListener(activator);

    JInternalFrame fr5 =
      new JInternalFrame("BoxLayout - Y", true, true);
    // Unchanged code from section 4.2
```

```
    CSH.setHelpIDString(fr5,"BoxLayout");
    fr5.addInternalFrameListener(activator);

    // Unchanged code from section 4.2
  }

  public static void main(String argv[]) {
    new CommonLayoutsDemo ();
  }

  protected void createHelp() {
    ClassLoader loader = this.getClass().getClassLoader();
    URL url;
    try {
      url = HelpSet.findHelpSet(loader, HELPSETNAME);
      m_hs = new HelpSet(loader, url);
      m_hb = m_hs.createHelpBroker();
      m_hb.enableHelpKey(getRootPane(), "Frame", m_hs);
    }
    catch (Exception ex) { ex.printStackTrace(); }
  }
}
```

Understanding the Code

Class CommonLayoutsDemo

New instance variables:

`HelpSet m_hs`: used to manage the help data for this application.

`HelpBroker m_hb`: used to manage the help viewer for this application.

New class variable:

`String HELPSETNAME`: the path and name of the HelpSet file relative to the location of this class.

The `CommonLayoutsDemo` constructor calls our custom `createHelp()` method to initialize JavaHelp functionality (see below). Each `JInternalFrame` is associated with a proper help ID using the `CSH.setHelpIDString()` method.

An `InternalFrameAdapter` (implementation of `InternalFrameListener`--see chapter 16) is used to detect whenever an internal frame is acivated, and transfer focus to the first component within its content pane. This is done because even though `JInternalFrame` can be activated by a mouse click over it's surface, this does not necessarily transfer focus to it. The focus may still belong to the component in the previously active `JInternalFrame`. This causes undesirable effects with JavaHelp because in this case we need to show a help topic based on the currently selected frame. By manually transfering focus to a child of the active `JInternalFrame` the correct help topic will be displayed when help is invoked. Thus, we add an instance of this listener to each internal frame.

The `createHelp()` method retrieves the `URL` corresponding to the help set for this application, and instantiates our `HelpSet` and `HelpBroker` instance variables. The `enableHelpKey()` method enables help functionality using the F1 key, and sets the default topic ID to "`Frame`" on the application's `JRootPane`.

Running the Code

Activate one of the internal frames and press F1. Figure 25.1 shows the JavaHelp help viewer displaying the help topic associated with the currently active internal frame. Explore and become familiar with the navigation functionality provided by the help viewer.

25.4 Adding dialog-style help

Dialogs in modern applications quite often contain a "Help" button to invoke content-sensitive help for a particular component in that dialog (usually using an altered question mark cursor). In this section we'll show how to add such a feature provided by JavaHelp to the FTP client example we constructed in chapter 13.

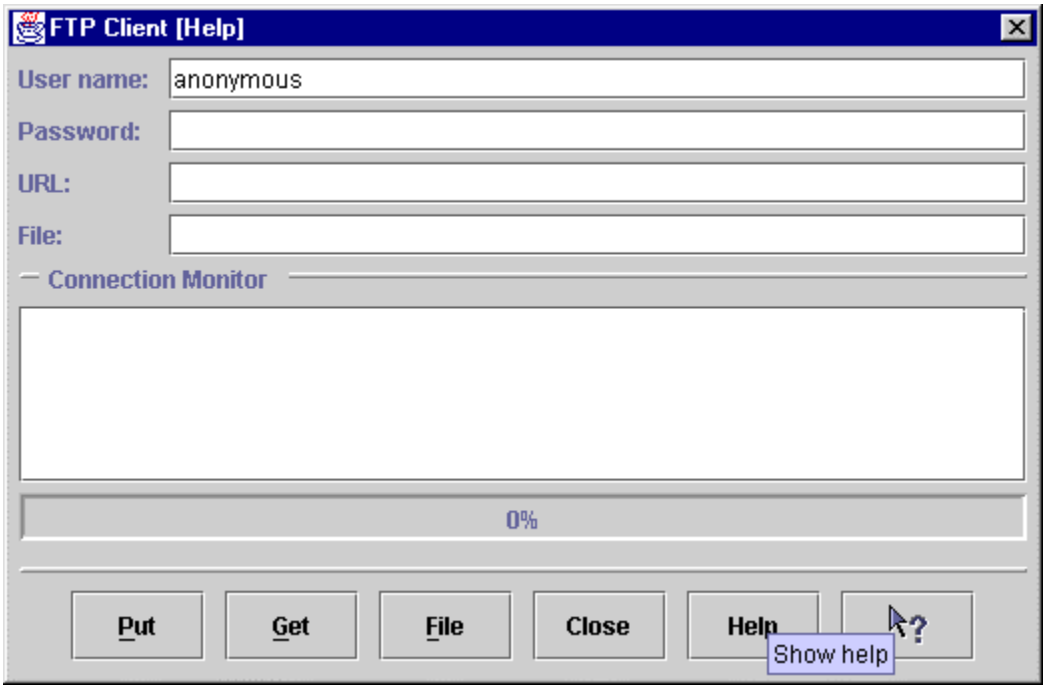


Figure 25.2 FTP Client application with help buttons for default and context-sensitive help.

<<file figure25-2.gif>>

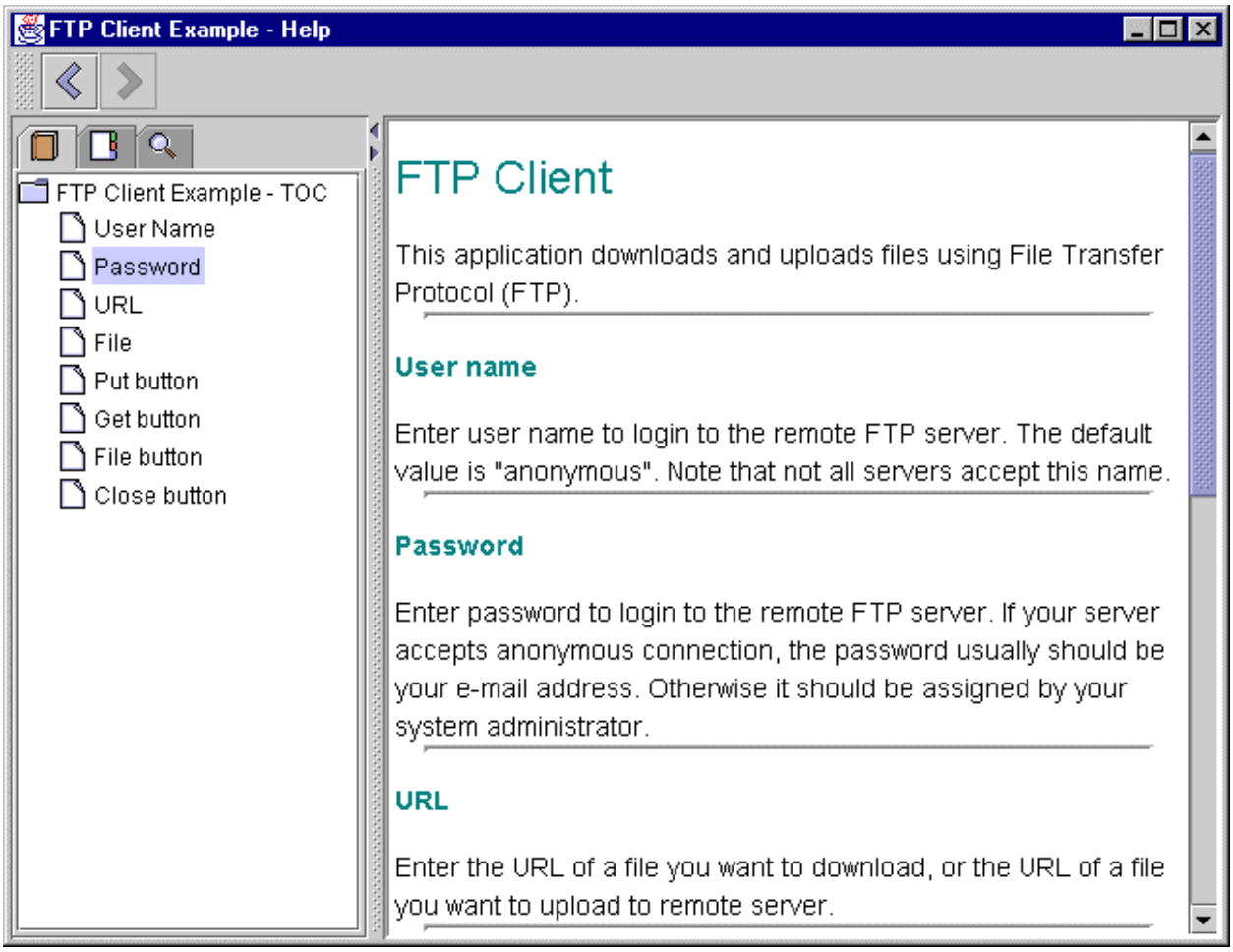


Figure 25.3 JavaHelp viewer displaying custom help for our FTP client application.

<<file figure25-3.gif>>

HelpSet file: FTP.hs
see \Chapter25\2\help

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<helpset>
   <title>FTP Client Example - Help</title>
   <maps>
     <homeID>top</homeID>
     <mapref location="FTP.jhm"/>
   </maps>
   <view>
     <name>TOC</name>
     <label>Table Of Contents</label>
     <type>javax.help.TOCView</type>
     <data>FTPTOC.xml</data>
   </view>
   <view>
     <name>Index</name>
     <label>Help Index</label>
     <type>javax.help.IndexView</type>
     <data>FTPIndex.xml</data>
   </view>
   <view>
     <name>Search</name>
```

```
     <label>Search</label>
     <type>javax.help.SearchView</type>
     <data>JavaHelpSearch</data>
   </view>
</helpset>
```

Map file: FTP.jhm
see \Chapter25\2\help

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<map version="1.0">
  <mapID target="top" url="FTPClient.html" />
  <mapID target="UserName" url="FTPClient.html#UserName" />
  <mapID target="Password" url="FTPClient.html#Password" />
  <mapID target="URL" url="FTPClient.html#URL" />
  <mapID target="File" url="FTPClient.html#File" />
  <mapID target="PutButton" url="FTPClient.html#PutButton" />
  <mapID target="GetButton" url="FTPClient.html#GetButton" />
  <mapID target="FileButton" url="FTPClient.html#FileButton" />
  <mapID target="CloseButton" url="FTPClient.html#CloseButton" />
</map>
```

TOC file: FTPTOC.xml
see \Chapter25\2\help

```
<?xml version='1.0' encoding='ISO-8859-1' ?>
<toc version="1.0">
  <tocitem target=top>FTP Client Example - TOC
    <tocitem text="User Name" target="UserName"/>
    <tocitem text="Password" target="Password"/>
    <tocitem text="URL" target="URL"/>
    <tocitem text="File" target="File"/>
    <tocitem text="Put button" target="PutButton"/>
    <tocitem text="Get button" target="GetButton"/>
    <tocitem text="File button" target="FileButton"/>
    <tocitem text="Close button" target="CloseButton"/>
  </tocitem>
</toc>
```

Index file: FTPIndex.xml
see \Chapter25\2\help

```
<?xml version='1.0' encoding='ISO-8859-1'  ?>
<index version="1.0">
  <indexitem text="Index">
    <indexitem text="C">
      <indexitem target="CloseButton" text="Close button"/>
    </indexitem>
    <indexitem text="F">
      <indexitem target="File" text="File"/>
      <indexitem target="FileButton" text="File button"/>
    </indexitem>
    <indexitem text="G">
      <indexitem target="GetButton" text="Get button"/>
    </indexitem>
    <indexitem text="P">
      <indexitem target="Password" text="Password"/>
      <indexitem target="PutButton" text="Put button"/>
    </indexitem>
    <indexitem text="U">
```

```
      <indexitem target="UserName" text="User Name"/>
      <indexitem target="URL" text="URL"/>
    </indexitem>
  </indexitem>
</index>
```

The Code: FTPApp.java
see \Chapter25\2

```
// Unchanged imports from section 13.5

import javax.help.*;

public class FTPApp extends JFrame
{
  // Unchanged code from section 13.5

  protected HelpSet m_hs;
  protected HelpBroker m_hb;
  static final String HELPSETNAME = "Help/FTP.hs";

  public FTPApp() {
    super("FTP Client [Help]");

    createHelp();

    JPanel p = new JPanel();
    p.setLayout(new DialogLayout2(5, 5));
    p.setBorder(new EmptyBorder(5, 5, 5, 5));

    // Unchanged code from section 13.5

    JButton btHelp = new JButton("Help");
    btHelp.setToolTipText("Show help");
    m_hb.enableHelpOnButton(btHelp, "top", null);
    p.add(btHelp);

    btHelp = new JButton(new ImageIcon("onItem.gif"));
    btHelp.setToolTipText("Component help");
    btHelp.addActionListener(
      new CSH.DisplayHelpAfterTracking(m_hb));
    p.add(btHelp);

    CSH.setHelpIDString(m_txtUser,"UserName");
    CSH.setHelpIDString(m_txtPassword,"Password");
    CSH.setHelpIDString(m_txtURL,"URL");
    CSH.setHelpIDString(m_txtFile,"File");
    CSH.setHelpIDString(m_btPut,"PutButton");
    CSH.setHelpIDString(m_btGet,"GetButton");
    CSH.setHelpIDString(m_btFile,"FileButton");
    CSH.setHelpIDString(m_btClose,"CloseButton");

    setSize(520,340);
    setResizable(false);
    setVisible(true);
  }

  protected void createHelp() {
    ClassLoader loader = this.getClass().getClassLoader();
    URL url;
    try {
```

```
      url = HelpSet.findHelpSet(loader, HELPSETNAME);
      m_hs = new HelpSet(loader, url);
      m_hb = m_hs.createHelpBroker();
      m_hb.enableHelpKey(getRootPane(), "top", m_hs);
    }
    catch (Exception ex) { ex.printStackTrace(); }
  }

  // Unchanged code from section 13.5
}
```

Understanding the Code

Class FTPApp

The mechanism of adding help to this application is very similar to that of the previous example. We add IDs to each button and text field component using CSH's `setHelpIDString()` method, and enable help invocation on the frame's `JRootPane` using an identical `createHelp()` method (see previous example).

Two new buttons are added to the button row. The first one is titled "Help". Method `enableHelpOnButton()` is called on the `HelpBroker` instance to invoke the display of the default help topic when this button is pressed. The second button receives a predefined `ActionListener` retrieved with the `CSH.DisplayHelpAfterTracking()` method. This listener displays a custom question mark cursor and displays the appropriate help topic corresponding to the clicked component. (Note that once a component is clicked and help is shown, the cursor reverts back to normal.)

Running the Code

Figure 25.2 shows our FTP client example with two new buttons to manage help. Press the "Help" button to display the default help topic (which also can be shown using the F1 key). Try using the content-sensitive help button to bring up a help topic for a particular component, and note the use of a custom cursor.

25.5 Customizing the JHelp viewer

So far we've seen how to add the help viewer to an application and invoke it through pressing F1 or a button. However, we may very well want to customize help functionality by adding a component to the help viewer toolbar, or adding a custom navigational view. The current release of the JavaHelp does include a fairly powerful GUI, however, there are certainly cases where this is not enough. In this and the next section we will show how to integrate two new features into the help viewer: a "Home" toolbar button switching to the home help topic (specified by the `<homeID>` HelpSet file tag), and a "History" view providing quick return to a specific previously viewed topic.

The bad news is that (at least in this release) JavaHelp does not provide any direct access to the `JHelp` instance constituting the help viewer. So no customization can be done though provided functionality. The good news is that by taking advantage of the fact that `JHelp` is a `java.awt.Container` subclass, we can gain access to its child components indirectly and do what we need.

Reference: We did a similar thing with `JFileChooser` in chapter 14 to gain access to its embedded `JList`. This was necessary to allow multiple selection, which is not implemented correctly as of Java 2 FCS.

The following example builds off of the previous FTP client example, and shows how to customize the help viewer by adding a "Home" button to its toolbar.

Note: This solution is highly dependant on the construction of `JHelp`, which may very well change in future implemenatations. However, it illustrates some general techniques that can be used when components are not

build as flexible as they should be.

Note: This example, and the example in section 25.6, requires Java2 because new functionality built into `java.awt.Frame` and `java.io.File` is used here (explained below).

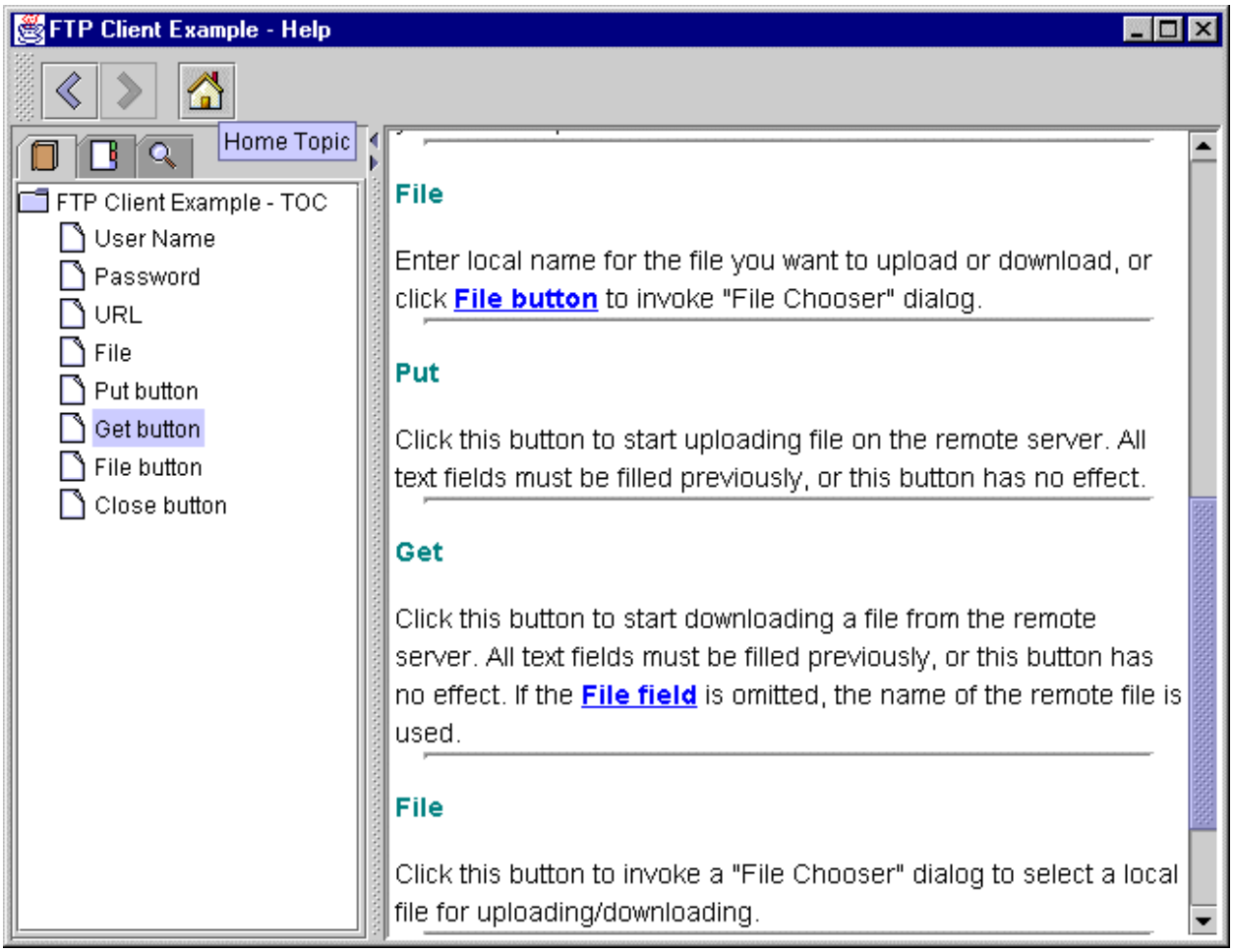


Figure 25.4 JHelp viewer with a custom toolbar button for jumping to the home topic specified in a HelpSet file.
<<file figure25-4.gif>>

The Code: FTPApp.java
see \Chapter25\3

```
// Unchanged imports from section 25.4

public class FTPApp extends JFrame
{
  // Unchanged code from section 25.4

  public FTPApp() {
    super("FTP Client [Customizing Help]");

    // Unchanged code from section 25.4

    WindowListener wndCloser = new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        disconnect();
        System.exit(0);
```

```
      }

      Frame m_helpFrame = null;
      public void windowDeactivated(WindowEvent e) {
        if (m_helpFrame != null)
          return;
        Frame[] frames = getFrames(); // Only available in Java 2
        for (int k = 0; k < frames.length; k++) {
          if (!(frames[k] instanceof JFrame))
            continue;
          JFrame jf = (JFrame)frames[k];
          if (jf.getContentPane().getComponentCount()==0)
            continue;
          Component c = jf.getContentPane().
            getComponent(0);
          if (c == null || !(c instanceof JHelp))
            continue;
          m_helpFrame = jf;
          JHelp jh = (JHelp)c;
          for (int s=0; s<jh.getComponentCount(); s++) {
            c = jh.getComponent(s);
            if (c == null || !(c instanceof JToolBar))
              continue;
            JToolBar jtb = (JToolBar)c;
            jtb.addSeparator();
            JButton home = new JButton(new
              ImageIcon("Home.gif"));
            home.setToolTipText("Home Topic");
            jtb.add(home);

            ActionListener alst = new ActionListener() {
              public void actionPerformed(ActionEvent e) {
                try {
                  m_hb.setCurrentID(m_hs.getHomeID());
                }
                catch (Exception ex) {}
              }
            };
            home.addActionListener(alst);
          }
        }
      }
    };
    addWindowListener(wndCloser);

    // Unchanged code from section 25.4
  }

// Unchanged code from section 25.4
```

Understanding the Code

Class FTPApp

The idea behind this example is that when the application's frame is deactivated, this may have occurred because the help viewer's frame has become activated. So at that moment we can check all existing frames in the current JVM, and check whether they hold an instance of `JHelp` as a first child in the content pane. If so, we gain access to the `JHelp` instance and add our custom component to it.

To accomplish this, significant modification has been made to our `WindowListener` implementation, which

previously only implemented the `windowClosing()` method. The `Frame m_helpFrame` instance variable holds a reference to the help viewer's frame (note that this frame is created only once and is hidden/shown as needed after that). The `windowDeactivated()` method checks this variable, and if it hasn't been determined yet (i.e. if it is null) it retrieves an array of all the frames created by this JVM using the static `Frame.getFrames()` method (another nice new feature in Java 2). All these frames are examined in turn. If one of them is an instance of `JFrame` and has an instance of `JHelp` as its first child, we take a reference to that child, cast it to `JHelp`, and, in turn, examine all its children. We know that one of these children should be an instance of `JToolBar` (since this only occurs before `JHelp` first becomes visible, there is no possibility that the toolbar has been undocked and left floating). As soon as it is found, we `add()` a `JButton` which calls `setCurrentID()` on our `HelpSet` when pressed:

```
    m_hb.setCurrentID(m_hs.getHomeID());
```

The `Map.ID` instance returned by the `getHomeID()` method is supplied as an argument to this call, which sets the current topic in the viewer to the home topic specifed in our HelpSet file.

Running the Code

Figure 25.4 shows the `JHelp` viewer with a new toolbar button. Pressing this button shows the home topic as specified by the `<homeID>` tag in our HelpSet file. We can use a similar technique to add any other component to this toolbar, or to another `JHelp` constituent.

25.6 Creating a custom help view

As we mentioned earlier, JavaHelp allows developers to add a new navigator view by simply adding a new `<view>` tag to the HelpSet file. Unfortunately this navigator view cannot gain an access to the viewer component that displays it (it can only have access to a data file supplied as a parameter with the `<data>` tag, similar to TOC or index views). We can work around this limitation by using the same basic technique used in the previous section to access the `JHelp` toolbar. The following example adds a custom navigational view to the `JHelp` viewer tabbed pane to display a selectable, dynamically changeable help topic history.

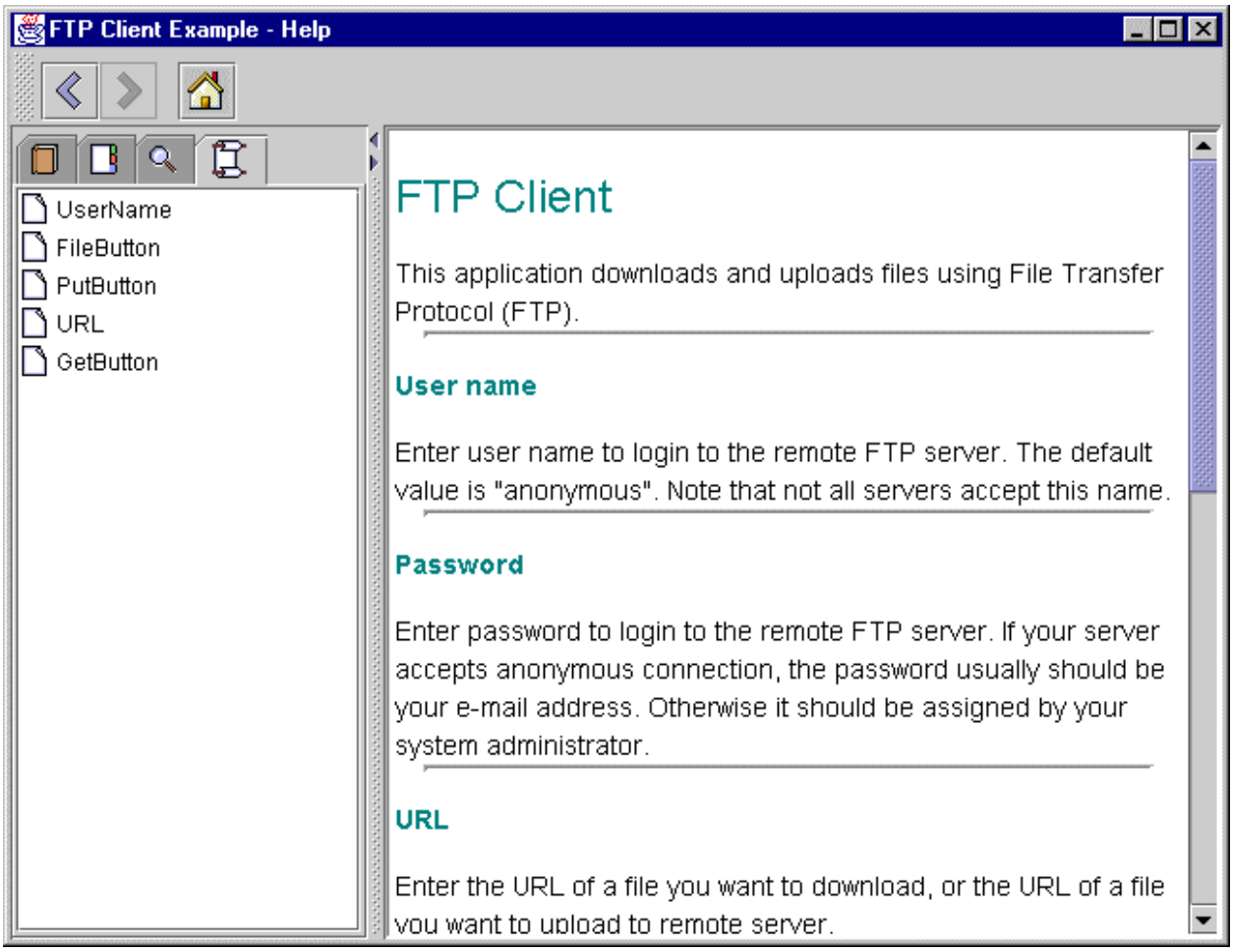


Figure 25.5 Custom JavaHelp History view added to the JavaHelp viewer.

<<file figure25-5.gif>>

HelpSet file: FTP.hs
see \Chapter25\4\help\

```
   // Unchanged from section 25.4

   <view>
     <name>History</name>
     <label>History</label>
     <type>HistoryView</type>
   </view>
</helpset>
```

The Code: FTPApp.java
see \Chapter25\4

```
// Unchanged imports from section 25.5

public class FTPApp extends JFrame
{
  // Unchanged code from section 25.5

  public FTPApp() {
    super("FTP Client [Help History]");
```

```
    // Unchanged code from section 25.5

    WindowListener wndCloser = new WindowAdapter() {
      // Unchanged code from section 25.5

      public void windowDeactivated(WindowEvent e) {
        // Unchanged code from section 25.5

          Enumeration en = jh.getHelpNavigators();
          while (en.hasMoreElements()) {
            JHelpNavigator jhn = (JHelpNavigator)
              en.nextElement();
            NavigatorView view = jhn.getNavigatorView();
            if (view instanceof HistoryView)
              ((HistoryView)view).setNavigator(jhn);
          }
        }
      }
    };
    addWindowListener(wndCloser);

    // Unchanged code from section 25.5
  }

// Unchanged code from section 25.5
```

The Code: HistoryView.java
see \Chapter25\4

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

import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;

import javax.help.*;
import javax.help.event.*;

public class HistoryView extends TOCView
 implements HelpModelListener
{
  protected DefaultMutableTreeNode m_topNode;
  protected JTree m_tree;
  protected DefaultTreeModel m_model;
  protected TreePath m_zeroPath;

  public HistoryView(HelpSet hs, String name, String label,
   Hashtable params) {
    super(hs, name, label, hs.getLocale(), params);
  }

  public HistoryView(HelpSet hs, String name, String label,
   Locale locale, Hashtable params) {
    super(hs, name, label, locale, params);
  }

  public Component createNavigator(HelpModel model) {
    JHelpNavigator navigator =
```

```
      new JHelpTOCNavigator(this, model);
    navigator.getUI().setIcon(new ImageIcon("History.gif"));
    return navigator;
  }

  public static MutableTreeNode parse(URL url, HelpSet hs,
    Locale locale, TreeItemFactory factory) {
      return new DefaultMutableTreeNode("History");
  }

  public void setNavigator(JHelpNavigator jhn) {
    jhn.getModel().addHelpModelListener(this);
    try {
      Container c = jhn;
      while (!(c instanceof JTree))
        c = (Container)(c.getComponent(0));
      m_tree = (JTree)c;
      TOCItem top = new TOCItem();
      top.setName("History");
      m_topNode = new DefaultMutableTreeNode(top);
      m_model = new DefaultTreeModel(m_topNode);
      m_tree.setModel(m_model);
      m_zeroPath = new TreePath(new Object[] { m_topNode });
    }
    catch (Exception ex) {
      ex.printStackTrace();
      System.err.println(ex.toString());
    }
  }

  public void idChanged(HelpModelEvent e) {
    // To avoid conflict with java.util.Map
    javax.help.Map.ID id = e.getID();
    if (m_topNode != null) {
      TOCItem item = new TOCItem(id, null, null);
      item.setName(id.id);
      for (int k=0; k<m_topNode.getChildCount(); k++) {
        DefaultMutableTreeNode child =
          (DefaultMutableTreeNode)m_topNode.getChildAt(k);
        TOCItem childItem = (TOCItem)child.getUserObject();
        if (childItem.getID().equals(id))
          m_topNode.remove(child);
      }
      DefaultMutableTreeNode node =
        new DefaultMutableTreeNode(item);
      m_topNode.insert(node, 0);
      m_model.reload(m_topNode);
      m_tree.expandPath(m_zeroPath);
    }
  }
}
```

Understanding the Code

FTP.hs

One more `<view>` tag is now added to our `FTP.hs` XML HelpSet file to request the creation of a `HistoryView` with a name and label of "History." This results in a new navigator in `JHelp`'s navigational view tabbed pane. This navigator is connected to an instance of our custom `HistoryView` class.

Class FTPApp

To connect the history view to the viewer itself, some code has been added to our `windowDeactivated()` method in the `WindowAdapter` inner class. This code retrieves all help views using the `getHelpNavigators()` method, examines all of them in turn, and, if an instance of our custom `HistoryView` class is found, calls the `setNavigator()` method to pass a reference to `JHelpNavigator` to it.

Note: The fact that we have to add this code to our application in order to use a custom view should immediately stand out as bad design, even though it is only called once (see previous example). We are forced to do this because the parent class, `NavigatorView`, does not provide any means of retrieving the associated instance of `JHelpNavigator` used to display it. This severely limits the abilities of help customization and contradicts the flexible, JavaBeans design of Swing components. We hope to see this limitation accounted for in future JavaHelp releases, making this workaround unnecessary.

Class HistoryView

This class represents a custom implementation of `NavigatorView` to provide easy access to previously viewed help topics. `HistoryView` implements `HelpModelListener` and extends `TOCView` to inherit its functionality and avoid writing our own `NavigatorView` sub-class. Instance variables:

`JTree m_tree`: the tree component used to display historical help topics.

`DefaultMutableTreeNode m_topNode`: the root node in the tree parenting all historical help topics. This node is not visible because the `rootVisible` property is set to `false` by the parent `TOCView` class.

`DefaultTreeModel m_model`: the historical tree model.

`TreePath m_zeroPath`: path to the root node.

The two available constructors delegate their work the corresponding superclass constructors. The `createNavigator()` method creates an instance of `JHelpTOCNavigator` (a sub-class of `JHelpNavigator`) to represent our historical view graphically, and assigns it a custom icon (to be displayed in `JHelp`'s navigational views tabbed pane).

Since we are not intending to parse any external data files in our historical view (all data is generated by the help viewer itself at run-time), the `parse()` method simply creates and returns an instance of `DefaultMutableTreeNode` with user object "History," and no child nodes.

Our custom `setNavigator()` method establishes a connection between this view and the `JHelpNavigator` component supplied as parameter representing our history information graphically. First, `this` view is added as a `HelpModelListener` to the `JHelpNavigator`'s `HelpModel` to receive `HelpModelEvents` when a change in the current help topic occurs. Second, we search for the `JHelpNavigator`'s tree component by iterating through its containment hierarchy. Then the `m_topNode` variable is assigned a `DefaultMutableTreeNode` instance with a default `TOCItem` user object assigned a name of "History." We also assign the `m_model` variable as a reference to this tree's model, and `m_zeroPath` is used to store the path from the tree's root to `m_topNode` (which simply consists of `m_topNode` itself). Both variables are needed in `idChanged()`.

The `idChanged()` method will be called when the current help topic is changed. This method receives an instance of `HelpModelEvent` which encapsulates, among other things, the new help topic ID. It then creates a new `TOCItem` using that ID, and checks all child nodes of `m_topNode` to find out whether the corresponding topic already exists in the history view. If so, it is removed to avoid redundancy. Then the new

topic is inserted at the top of the tree. Thus the previously viewed topics appear listed from most recent at the top, to least recent at the bottom. Finally we call `reload()` on the tree model and `expandPath()` on the tree, using `m_zeroPath`, to make sure that the tree updates its graphical representation correctly after it is reloaded.

Note: Since the root tree node is hidden and the path to it is collapsed by default, no nodes in the view will be visible after a `reload()` unless we expand `m_zeroPath` programmatically.

That's all we need to do to create a history view. As soon as the user clicks on the topic listed in the help view, it will be displayed in the viewer because we inherit this functionality from `TOCView`.

Running the Code

Figure 25.5 shows the customized help viewer for our FTP Client example with the custom history view. Select several help topics in a row using the TOC or index views and then select the history view. Note that all visited help topics are listed in the reverse order that they were visited in. Click on any of these topics and note that viewer jumps to that topic (at the same time the selected topic is displayed it moves to the top of the history view, as expected).

Chapter 26. Swing and CORBA

In this chapter:

- Java 2 and CORBA
- Creating a CORBA interface
- Creating a CORBA server
- Creating a CORBA client

26.1 Java 2 and CORBA

CORBA (Common Object Request Broker Architecture) was developed by the Object Management Group (OMG), a large industry consortium, as an attempt to satisfy the need for interoperability among the rapidly proliferating number of hardware and software products available today. The Java 2 platform includes CORBA-related packages (`org.omg.CORBA`, `org.omg.CosNaming`, and their sub-packages) which can be used to build CORBA/Java applications and applets.

Note: The latest revision of the CORBA architecture specification adopted by OMG (version 2.2 in February 1998) is available at http://www.omg.org/library/c2indx.html.

A detailed description of both CORBA and the CORBA-related Java 2 packages lies far beyond the scope of this book. However, we feel that the potential for this technology is strong enough to warrant the inclusion of an example application showing how to implement a CORBA-enabled back end with Swing up front. For readers completely new to CORBA, we suggest referencing the following URL before moving on: http://www.omg.org/corba/beginners.html.

Note: The current CORBA functionality in Java 2 is not complete. Only a naming service is implemented, and the API docs lack a significant amount of explanation.

The example in this chapter is presented in three parts and demonstrates how we can build a complete CORBA-enabled Java application using Swing as a front end, CORBA as middleware, and a database server as a backend. We will build this application using our stocks table example presented in chapter 18 as a base, and we will use a CORBA-enabled remote server to supply stock data.

26.2 Creating a CORBA interface

To start the construction of a CORBA-enabled application we first need to define an IDL interface which will be used to retrieve data from a remote server. The example in this section presents two such interfaces and two user-defined exceptions in one module IDL file.

Note: IDL stands for Interface Definition Language. This language is specified by OMG and is used define the interfaces to CORBA objects. Java 2 uses a subset of IDL which is called "Java IDL" (see http://www.javasoft.com/products/jdk/idl/index.html for more details).

The Code: MarketData.idl
see \Chapter26\1

```
module MarketDataApp
{
  interface DataUnit {
    string getSymbol();
    string getName();
    double getLast();
    double getOpen();
    double getChange();
    double getChangePr();
    long getVolume();
  };

  exception CorbaSqlException {
    string reason;
  };

  exception LoginException {
  };

  interface MarketData {
    DataUnit getFirstData(in string name, in string password,
      in long year, in long month, in long day)
      raises(LoginException, CorbaSqlException);

    DataUnit getNextData()
      raises(CorbaSqlException);
  };
};
```

Java IDL is similar to Java and C++, but it does have some important distinctions. First, all parameters passed to the declared methods (see the `MarketData` interface) must have a direction attribute: `in`, `out`, or `inout`. Second, the `raises` keyword is used to declare exceptions which may be thrown by the associated method.

The `MarketDataApp` Java IDL file shown above declares the following:

- The `DataUnit` interface holds data for a single row in our stocks table.
- The `MarketData` interface delivers a sequence of `DataUnit` objects to a caller. The `getFirstData()` method returns the first `DataUnit` object available for a given year, month, and day, or `null` if no data is available. This method also takes a user's name and password as parameters, and is capable of throwing

a `LoginException` or a `CorbaSqlException`. The `getNextData()` method returns the next available `DataUnit` object. By calling this method repeatedly we can retrieve all sequences of data objects (used to fill each row of our stocks table).

A `LoginException` exception will be thrown if the user's authentication fails.

The `CorbaSqlException` exception will be thrown to encapsulate an SQL exception thrown on the server side, and pass it to the caller.

To compile this IDL file you need to download the `idltojava` compiler. This tool is not included in the Java 2 release, but is freely available from Java Developers Connection web site at http://developer.java.sun.com/developer/earlyAccess/jdk12/idltojava.html. This tool should be placed in the Java/bin directory. Once there, we can run the following command:

```
idltojava -fverbose -fno-cpp MarketData.idl
```

This command will generate a number of Java source files in a "MarketDataApp" sub-directory:

`DataUnit.java`: Java interface (listed below) corresponding to our `DataUnit` CORBA interface.

`DataUnitHelper.java`: helper class for the `DataUnit` interface. Helpers declare a set of useful static methods to help manage a given class, notably `narrow()`, which is the CORBA counterpart of a Java class cast.

`DataUnitHolder.java`: holder class for the `DataUnit` interface. Holders are required for CORBA operations that take `out` or `inout` arguments. Unlike CORBA `in` arguments, `out` and `inout` don't map directly to Java's pass-by-value semantics.

`_DataUnitImplBase.java`: this abstract class provides a base implementation of a CORBA object which all concrete implementations of the `DataUnit` interface must extend.

`_DataUnitStub.java`: this class represents the implementation of a client stub. It implements the `DataUnit` interface and provides the associated functionality for the client.

`MarketData.java`: Java interface (listed below) corresponding to our `MarketData` CORBA interface.

`MarketDataHelper.java`: helper class for `MarketData` interface.

`MarketDataHolder.java`: holder class for `MarketData` interface.

`_MarketDataImplBase.java`: this abstract class provides a base implementation of a CORBA object which all concrete implementations of the `MarketData` interface must extend.

`_MarketDataStub.java`: this class represents the implementation of client stub. It implements our `MarketData` interface and provides the associated functionality for the client.

`CorbaSqlException.java`: this class implements the SQL server-side exception.

`CorbaSqlExceptionHelper.java`: helper class for `CorbaSqlException`.

`CorbaSqlExceptionHolder.java`: holder class for `CorbaSqlException`.

`LoginException.java`: this class implements the authentication exception.

`LoginExceptionHelper.java`: helper class for `LoginException` class.

`LoginExceptionHolder.java`: holder class for `LoginException` class.

26.4 Creating a CORBA server

Now that we have the compiler-generated source files it's time to implement the CORBA server which will run on a remote computer. First let's take a look at the `DataUnit` interface describing a data object which can be transported to the caller. The following code is the compiler-generated `DataUnit` interface, and the

associated `DataUnitImpl` class which implements this interface:

The Code: DataUnit.java
see \Chapter26\MarketDataApp\1

```
/*
 * File: ./MARKETDATAAPP/DATAUNIT.JAVA
 * From: MARKETDATA.IDL
 * Date: Tue Feb 09 11:31:12 1999
 *   By: idltojava Java IDL 1.2 Aug 18 1998 16:25:34
 */

package MarketDataApp;
public interface DataUnit extends org.omg.CORBA.Object,
 org.omg.CORBA.portable.IDLEntity
{
  String getSymbol();
  String getName();
  double getLast();
  double getOpen();
  double getChange();
  double getChangePr();
  int getVolume();
}
```

The Code: DataUnitImpl.java
see \Chapter26\MarketDataApp\1

```
package MarketDataApp;

import java.sql.*;

public class DataUnitImpl extends _DataUnitImplBase
{
  protected String  m_symbol;
  protected String  m_name;
  protected double  m_last;
  protected double  m_open;
  protected double  m_change;
  protected double  m_changePr;
  protected long    m_volume;

  public DataUnitImpl(ResultSet results)
   throws SQLException {
    m_symbol = results.getString(1);
    m_name = results.getString(2);
    m_last = results.getDouble(3);
    m_open = results.getDouble(4);
    m_change = results.getDouble(5);
    m_changePr = results.getDouble(6);
    m_volume = results.getLong(7);
  }

  public String getSymbol()   { return m_symbol; }
  public String getName()     { return m_name; }
  public double getLast()     { return m_last; }
  public double getOpen()     { return m_open; }
  public double getChange()   { return m_change; }
  public double getChangePr() { return m_changePr; }
  public int getVolume()      { return (int)m_volume; }
}
```

The compiler-generated `DataUnit` interface contains seven `getXX()` methods intended for retrieving data fields from the implementor. Note that these fields directly correspond to the data fields of our `StockData` class from chapter 18.

Class `DataUnitImpl` extends the compiler-generated `_DataUnitImplBase` class (not listed here), which, in turn, implements the `DataUnit` interface. The seven data fields declared in the `DataUnitImpl` class correspond to the `getXX()` methods of the `DataUnit` interface. The `DataUnitImpl` constructor takes an instance of `java.sql.ResultSet` as parameter, and reads data from each record of the given result set.

Note: CORBA does not support 8-byte integers, so we are forced to pass the `m_volume` variable as `int`.

The Code: MarketData.java
see \Chapter26\MarketDataApp\1

```
/*
 * File: ./MARKETDATAAPP/MARKETDATA.JAVA
 * From: MARKETDATA.IDL
 * Date: Tue Feb 09 11:31:12 1999
 *   By: idltojava Java IDL 1.2 Aug 18 1998 16:25:34
 */

package MarketDataApp;

public interface MarketData extends org.omg.CORBA.Object,
 org.omg.CORBA.portable.IDLEntity
{
  MarketDataApp.DataUnit getFirstData(String name,
   String password, int year, int month, int day)
    throws MarketDataApp.LoginException,
    MarketDataApp.CorbaSqlException;

  MarketDataApp.DataUnit getNextData()
   throws MarketDataApp.CorbaSqlException;
}
```

The Code: MarketDataServer.java
see \Chapter26\MarketDataApp\1

```
package MarketDataApp;

import java.sql.*;
import java.util.*;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class MarketDataServer extends _MarketDataImplBase
{
  protected Connection m_conn;
  protected Statement  m_stmt;
  protected ResultSet  m_results;

  public MarketDataServer() {
    try {
      // Load the JDBC-ODBC bridge driver
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

```
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}

public DataUnit getFirstData(String name, String password,
 int year, int month, int day)
 throws MarketDataApp.LoginException, MarketDataApp.CorbaSqlException
{
  if (!name.equals("CORBA") || !password.equals("Swing"))
    throw new MarketDataApp.LoginException();

  String query = "SELECT data.symbol, symbols.name, "+
    "data.last, data.open, data.change, data.changeproc, "+
    "data.volume FROM DATA INNER JOIN SYMBOLS "+
    "ON DATA.symbol = SYMBOLS.symbol WHERE "+
    "month(data.date1)="+month+" AND day(data.date1)="+day+
    " AND year(data.date1)="+year;

  try {
    m_conn = DriverManager.getConnection(
      "jdbc:odbc:Market", "admin", "");

    m_stmt = m_conn.createStatement();
      m_results = m_stmt.executeQuery(query);

    if (m_results.next())
      return new DataUnitImpl(m_results);
    else {
      disconnect();
      return null;
    }
  }
  catch (Exception e) {
    e.printStackTrace();
    throw new CorbaSqlException(e.toString());
  }
}

public DataUnit getNextData()
 throws MarketDataApp.CorbaSqlException {
  try {
    if (m_results != null && m_results.next())
      return new DataUnitImpl(m_results);
    else {
      disconnect();
      return null;
    }
  }
  catch (Exception e) {
    e.printStackTrace();
    throw new CorbaSqlException(e.toString());
  }
}

protected void disconnect() {
  try {
    if (m_results != null)
      m_results.close();
    if (m_stmt != null)
      m_stmt.close();
```

```
      if (m_conn != null)
        m_conn.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
    m_results = null;
    m_stmt = null;
    m_conn = null;
  }

  public static void main(String args[]) {
    try {
      // create and initialize the ORB
      Properties props = new Properties();
      props.put("org.omg.CORBA.ORBInitialPort", "1250");
      ORB orb = ORB.init(args, props);

      // create server and register it with the ORB
      MarketDataServer server = new MarketDataServer();
      orb.connect(server);

      // get the root naming context
      org.omg.CORBA.Object objRef =
        orb.resolve_initial_references("NameService");
      NamingContext ncRef = NamingContextHelper.narrow(objRef);

      // bind the Object Reference in Naming
      NameComponent nc = new NameComponent("MarketData", "");
      NameComponent path[] = {nc};
      ncRef.rebind(path, server);

      // wait for invocations from clients
      java.lang.Object sync = new java.lang.Object();
      synchronized (sync) {
        System.out.println("Waiting for client connection");
        sync.wait();
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

The `MarketData` interface, according to the IDL file described above, defines two methods: `getFirstData()` and `getNextData()`. Class `MarketDataServer` extends the compiler-generated `_MarketDataImplBase` class (not listed here), which, in turn, implements `MarketData` interface.

The `MarketDataServer` constructor loads the JDBC-ODBC bridge driver used to retrieve data from the database.

The `getFirstData()` method creates an SQL query and returns the first data record (in the form of a `DataUnit`) or `null` if no data is available. As a first step, typical for any network application, user authentication is checked. If the user's name or password does not match the required values, a `LoginException` is thrown.

Note: In a more realistic CORBA application we would use a more sophisticated mechanism of user authentication than hard-coding a single username and password! Encryption is an increasingly important topic for addressing

the security of online transactions.

`getFirstData()` then creates an SQL query to retrieve data corresponding to a certain date, establishes a connection to the driver, creates an SQL statement, and executes a query. If at least one record is retrieved by that query, a new `DataUnitImpl` object is created and returned. Otherwise, the method returns `null`. Note that if any exceptions occur, they will be caught and thrown as `CorbaSqlExceptions`.

The `getNextData()` method will most likely be called repeatedly to retrieve remaining data records fetched by an SQL query. If the result set is currently in use and a new record can be retrieved, a new `DataUnitImpl` object is created and returned. Otherwise this method calls `disconnect()` and returns `null`. Similar to `getFirstData()`, all exceptions are caught and thrown as `CorbaSqlExceptions`.

The `disconnect()` method closes an SQL connection and sets all related references to `null`.

The `main()` method creates an instance of this server and registers it with an `ORB`. Let's discuss it step-by-step:

- A call to the `ORB.init()` static method creates an `ORB`. The `ORB` constructor takes two parameters: a `String` array of application parameters, and a `Properties` instance created from within the application. Note that we have explicitly set the `ORBInitialPort` property to 1250, specifying the port for our `ORB` to run on.
- A new `MarketDataServer` instance is created and connected to our `ORB`. At this point the `ORB` server can start receiving remote invocations.
- The `CORBA` naming service is resolved, and a reference to the `NamingContext` instance is retrieved using the `narrow()` method (different implementations of this method serve as the equivalent of class casting).
- A `NameComponent` object is created to encapsulate the name of this service: "`MarketData`". An array of `NameComponents` can hold a fully specified path to an object on any file or disk system. In our case this path contains only one element. The `rebind()` method binds this path and the server object, so a call to our server may be resolved by `CORBA`.
- Finally, a `java.lang.Object` is created and access to it is synchronized to wait for a server connection.

Note: These steps are similar to the typical procedure used to establish an `RMI` connection.

Note: Because of name conflict with `org.omg.CORBA.Object`, all references to `java.lang.Object` must be explicit in Java 2 `CORBA`-related code.

26.5 Creating a CORBA client

Its finally time to take a look at how we can modify our stocks table application from chapter 18 to retrieve data from the `CORBA` server.

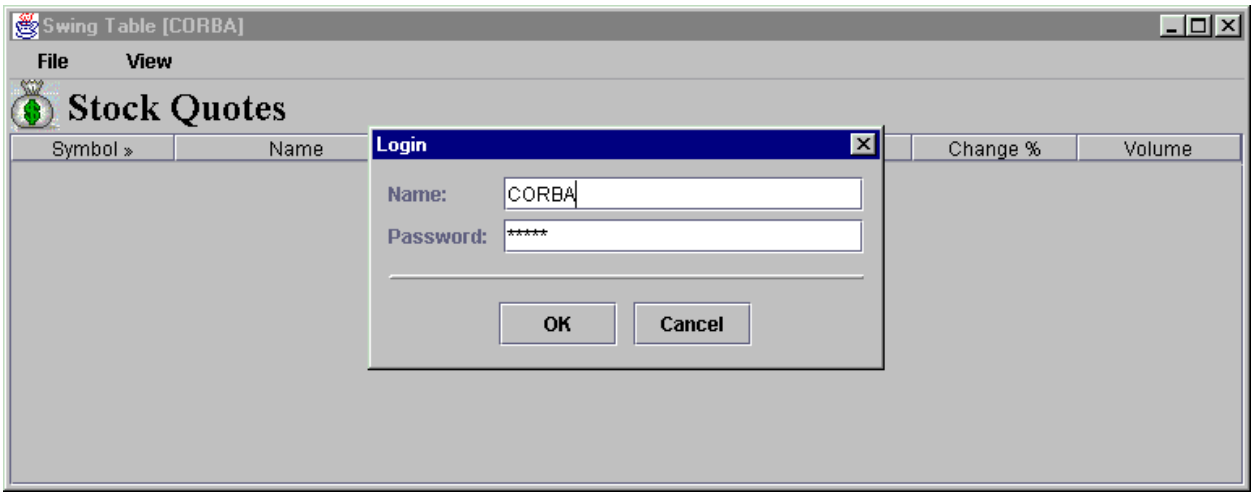


Figure 26.1 Login dialog used to connect to CORBA-enabled remote server.
<<file figure26-1.gif>>

The Code: StocksTable.java
see \Chapter26\1

```
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import java.text.*;

import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import javax.swing.table.*;

import dl.*;
import MarketDataApp.*;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class StocksTable extends JFrame
{
  protected JTable m_table;
  protected StockTableData m_data;
  protected JLabel m_title;

  private String m_name = "CORBA";
  private String m_password = "Swing";
  protected MarketData m_server = null;    // Remote server ref.

  public StocksTable() {
    super("Swing Table [CORBA]");
    setSize(770, 300);
    getContentPane().setLayout(new BorderLayout());

    // Unchanged code
    WindowListener wndCloser = new WindowAdapter() {
      public void windowOpened(WindowEvent e) {
        LoginDialog dlg = new LoginDialog(StocksTable.this);
        dlg.show();
```

```
        if (!dlg.getOkFlag())
          System.exit(0);

        // Connect to the server
        try {
          // create and initialize the ORB
          String[] args = new String[]
            { "-ORBInitialPort", "1250",
              "ORBInitialHost", "localhost" };
          ORB orb = ORB.init(args, null);

          // get the root naming context
          org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
          NamingContext ncRef = NamingContextHelper.narrow(objRef);

          // resolve the Object Reference in Naming
          NameComponent nc = new NameComponent("MarketData", "");
          NameComponent path[] = {nc};
          m_server = MarketDataHelper.narrow(ncRef.resolve(path));
        }
        catch (Exception ex) {
          ex.printStackTrace(System.out);
          System.exit(0);
        }
        retrieveData();
      }

      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    };
    addWindowListener(wndCloser);

    setVisible(true);
  }

  // Unchanged code

  public void retrieveData() {
    SimpleDateFormat frm = new SimpleDateFormat("MM/dd/yyyy");
    String currentDate = (m_data.m_date == null ? "7/22/1998" :
      frm.format(m_data.m_date));
    String result = (String)JOptionPane.showInputDialog(this,
      "Please enter date in form mm/dd/yyyy:", "Input",
    JOptionPane.INFORMATION_MESSAGE, null, null,
      currentDate);
    if (result==null)
      return;

    // Unchanged code
  }

  // Unchanged code

  class LoginDialog extends JDialog
  {
    protected JTextField m_nameTxt;
    protected JPasswordField m_passTxt;
    protected boolean m_okFlag = false;
```

```
    public LoginDialog(Frame owner) {
      super(owner, "Login", true);
      JPanel p = new JPanel(new DialogLayout2());
      p.add(new JLabel("Name:"));
      m_nameTxt = new JTextField(m_name, 20);
      p.add(m_nameTxt);
      p.add(new JLabel("Password:"));
      m_passTxt = new JPasswordField(m_password);
      p.add(m_passTxt);
      p.add(new DialogSeparator());

      JButton btOK = new JButton("OK");
      ActionListener lst = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          m_name = m_nameTxt.getText();
          m_password = new String(m_passTxt.getPassword());
          m_okFlag = true;
          dispose();
        }
      };
      btOK.addActionListener(lst);
      p.add(btOK);

      JButton btCancel = new JButton("Cancel");
      lst = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          dispose();
        }
      };
      btCancel.addActionListener(lst);
      p.add(btCancel);

      p.setBorder(new EmptyBorder(10, 10, 10, 10));
      getContentPane().add(p);
      pack();
      setResizable(false);
      Dimension d1 = getSize();
      Dimension d2 = owner.getSize();
      int x = Math.max((d2.width-d1.width)/2, 0);
      int y = Math.max((d2.height-d1.height)/2, 0);
      setBounds(x, y, d1.width, d1.height);
    }

    public boolean getOkFlag() {
      return m_okFlag;
    }
  }

  public static void main(String argv[]) {
    new StocksTable();
  }
}

class ColoredTableCellRenderer extends DefaultTableCellRenderer
{
  public void setValue(java.lang.Object value) {
    // Unchanged code
  }
}

class StockData
{
```

```
  // Unchanged code

  public StockData(DataUnit unit) {
    this(unit.getSymbol(), unit.getName(), unit.getLast(),
      unit.getOpen(), unit.getChange(), unit.getChangePr(),
      unit.getVolume());
  }
}

class StockTableData extends AbstractTableModel
{
  // Unchanged code

  public StockTableData() {
    m_frm = new SimpleDateFormat("MM/dd/yyyy");
    m_vector = new Vector();
    //setDefaultData();        No longer used
  }

  public int retrieveData(java.util.Date date, String name,
   String password, MarketData server) {
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    int month = calendar.get(Calendar.MONTH)+1;
    int day = calendar.get(Calendar.DAY_OF_MONTH);
    int year = calendar.get(Calendar.YEAR);

    try {
      DataUnit unit = server.getFirstData(name, password,
        year, month, day);

      boolean hasData = false;
      while (unit != null) {
        if (!hasData) {
          m_vector.removeAllElements();
            hasData = true;
        }
        m_vector.addElement(new StockData(unit));
        unit = server.getNextData();
      }

      if (!hasData)    // We've got nothing
        return 1;
    }
    catch (Exception e) {
      e.printStackTrace();
      System.err.println("Load data error: "+e.getMessage());
      return -1;
    }
    m_date = date;
    Collections.sort(m_vector, new
      StockComparator(m_sortCol, m_sortAsc));
    return 0;
  }
}

// Unchanged code
```

Understanding the Code

Class StocksTable

Compared to the final stocks table example in chapter 18, we now import two custom packages (`dl` and `MarketDataApp`), and three CORBA-related packages.

New instance variables:

`String m_name`: user login name for authentication.

`String m_password`: user password for authentication.

`MarketData m_server`: a reference to the remote server obtained from CORBA.

The `WindowAdapter` used in this frame class now receives a new method, `windowOpened()`, which will be invoked when after the frame has been created and dispalyed on the screen (i.e. when `setVisible(true)` is called). We use this method to display a login dialog (see below). Then we connect to the CORBA server by following the same procedure as described above. Note that an additional property, `ORBInitialHost`, is passes to the `ORB.init()` method. This property determines an initial host address which will be checked for the required CORBA server (note that if both a CORBA server and client reside on the same machine, we can omit this property).

A reference to a `MarketData` is retrieved and stored in our `m_server` variable. Then the `retrieveData()` method is called to populate our table with the initial data fetched using the CORBA server. The only change made to this method is that it now sets the initial string for data input before calling `JOptionPane.showInputDialog()`. The rest of the job is delegated to `StockTableData.retrieveData()` method, as we did in the chapter 18 example.

Class LoginDialog

This inner class represents a custom modal dialog which allows input of a user name and password for authentication. Our custom `DialogLayout2` layout manager (discussed in chapter 4) simplifies the creation of this dialog, which is shown in figure 26.1.

If the "OK" button is pressed, the username and password are retrieved and stored in class variables. It also sets the `m_okFlag` to true. This flag can be retrieved with the `getOkFlag()` method and indicates how this dialog was closed (i.e. whether "OK" or "Cancel" was pressed).

Class StockData

This class now receives a new constructor which creates a new `StockData` object from a `DataUnit` instance discussed above.

Class StockTableData

This class no longer uses the `setDefaultData()` method. All data must be retrieved from the remote server. The `retrieveData()` method now receives three new parameters: user name, password, and a reference to the remote server. This method no longer uses SQL queries to retrieve data, but instead calls the `getFirstData()` method on the remote server (as we discussed in the last section). Then it repeatedly calls `getNextData()` until no more data is available. Retrieved data objects are encapsulated in `StockData` instances and stored in our `m_vector` collection.

Running the code

At this point you can compile all code and run the Java 2 name server:

```
tnameserv -ORBInitialPort 1250
```

Then run our CORBA server:

```
java MarketDataApp.MarketDataServer
```

...and wait until it displays a message "Waiting for client connection" in the console. In another session, run our CORBA client:

```
java StocksTable
```

Enter the user name and password, "CORBA" and "Swing" respectively (they are displayed by default). Select a date for retrieval of corresponding stock data from the remote CORBA server. This data will be retrieved and displayed in the table.

Note: Don't forget to verify that your database is correctly listed as an ODBC data source. If your server runs across a network, specify it's host while creating ORB object.

Note: We can improve this application's responsiveness during data retrieval by wrapping the associated code in threads. An even more professional application might implement progress bars to let the user know how far along the procedure is.

Chapter 27. Contributions

In this chapter:

- Multi-line tooltips and labels by Albert L. Ting
- Swingscape browser by Ron Widitz

27.1 Multi-line tooltips and labels

...by Albert L. Ting of Artisan Components, alt@artisan.com

One common feature that was missing in Swing for significant amount of time was multi-line support, in particular, multi-line tooltips and labels. In this section we show how to implement this functionality ourselves, with support for horizontal as well as vertical text. The implementation goals for this example are:

1. Be as lightweight as possible. Concatenating multiple JLabels or using a JTextArea is not acceptable (using an HTML renderer for each multiline button or label is not exactly the most efficient solution).
2. Transparently replace JToolTip and JLabel, such that any Swing component (or third party component) that uses JToolTip or JLabel would automatically have multi-line support (and vertical labels support).
3. Support all L&F's. Moreover, if the user changes L&F, the newly selected L&F will still have multi-line support.

The Swing pluggable L&F interface allows us to accomplish all of these goals, by defining a new ToolTipUI and a new LabelUI and registering them with the UIManager.

Note: As of Swing 1.1.1 beta1, multi-line label and button support has been added by using an html renderer

whenever the text appears to be html-based (see chapter 5 for an example). In this case each label has it's own UI delegate instance and vertical text is not supported. Our implementation uses a single renderer for multi-line text and vertical text. If the text is html-based, we delegate all painting work to an html renderer.

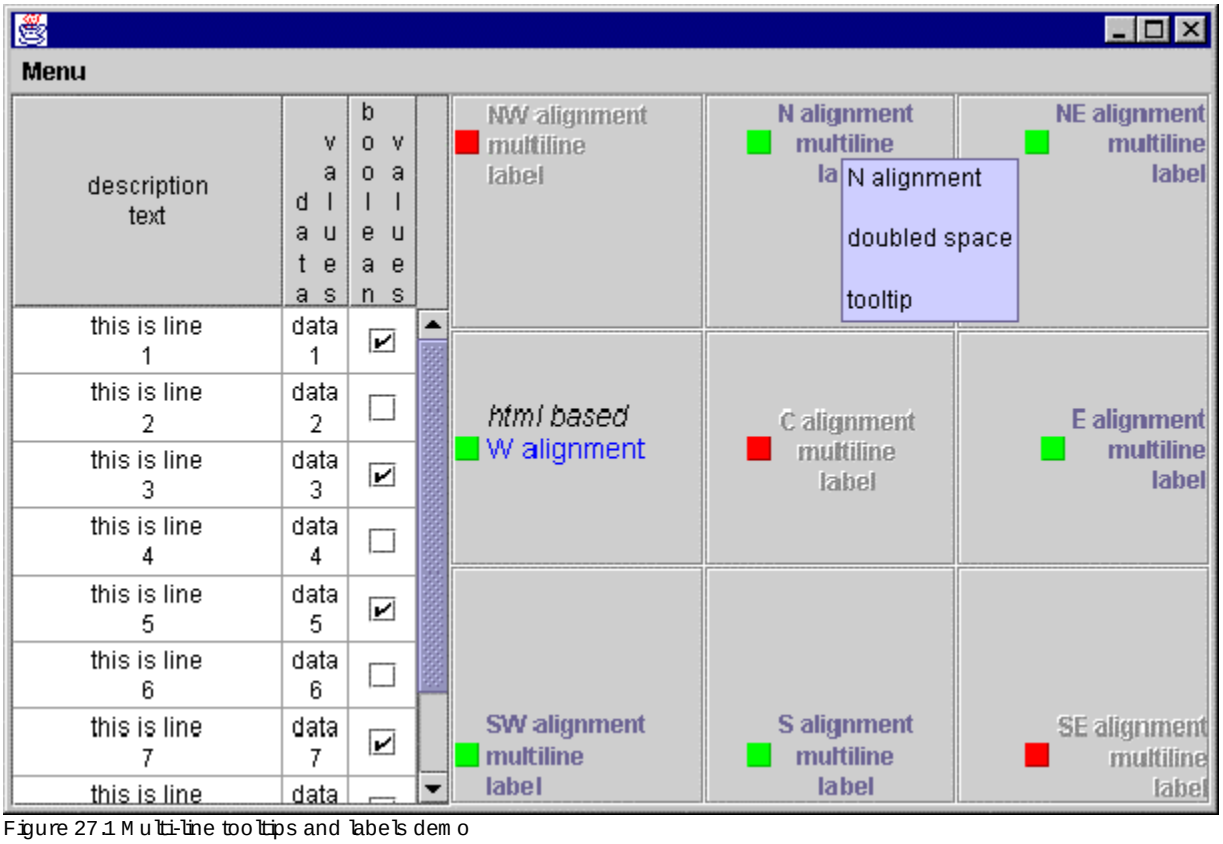


Figure 27.1 Multi-line tooltips and labels demo

<<file figure27-1.gif>>

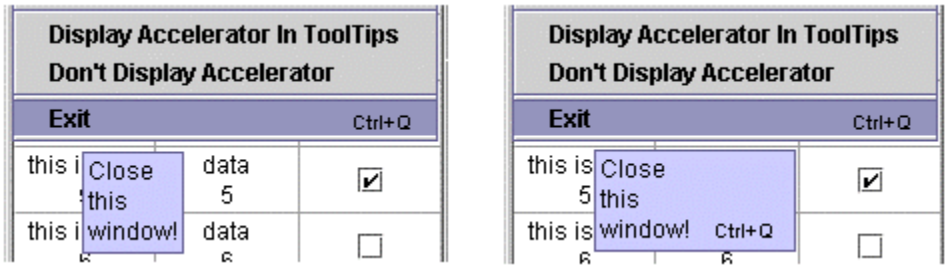


Figure 27.2 Multi-line tooltip with accelerator display option

<<file figure27-2.gif>>

The Code: PlafMacros.java

see \Chapter 27\Ting\jalt

```
package jalt;

import javax.swing.*;
import java.util.*;
import java.io.*;
import java.lang.*;
import java.awt.*;

public class PlafMacros implements SwingConstants
```

```
{
  // don't make these final, since the value is
  // different on each platform
  private static String LINE_SEPARATOR =
    System.getProperty("line.separator");
  private static int LINE_SEPARATOR_LEN =
    LINE_SEPARATOR.length();

  public static String[] breakupLines(String text)
  {
    int len = text.length();
    if (len == 0)
      return new String[] {""};
    else
    {
      Vector data = new Vector(10);
      int start=0;
      int i=0;
      while (i<len) {
        if (text.startsWith(LINE_SEPARATOR,i)) {
          data.addElement(text.substring(start,i));
          start=i+LINE_SEPARATOR_LEN;
          i=start;
        }
        else if (text.charAt(i)=='\n')
        {
          data.addElement(text.substring(start,i));
          start=i+1;
          i=start;
        }
        else { i++; }
      }
      if (start != len)
        data.addElement(text.substring(start));
      int numlines = data.size();
      String lines[] = new String[numlines];
      data.copyInto(lines);
      return lines;
    }
  }

  // See included source for the rest of this class
}
```

The Code: MultiLineToolTipUI.java
see \Chapter 27\Ting\jalt

```
package jalt;

import java.awt.*;
import java.awt.event.*;
import java.util.*;

import javax.swing.*;
import javax.swing.plaf.ToolTipUI;
import javax.swing.plaf.ComponentUI;

public class MultiLineToolTipUI extends ToolTipUI
{
  static MultiLineToolTipUI SINGLETON = new MultiLineToolTipUI();
  static boolean DISPLAY_ACCELERATOR=true;
```

```
  int accelerator_offset = 15;
  int inset = 3;

  private MultiLineToolTipUI() {}

  public static void initialize() {
    // don't hardcode class name, this way we can obfuscate.
    String key = "ToolTipUI";
    Class cls = SINGLETON.getClass();
    String name = cls.getName();
    UIManager.put(key,name);
    UIManager.put(name,cls);
  }

  public static ComponentUI createUI(JComponent c) {
    return SINGLETON;
  }

  public void installUI(JComponent c) {
    LookAndFeel.installColorsAndFont(c, "ToolTip.background",
      "ToolTip.foreground", "ToolTip.font");
    LookAndFeel.installBorder(c, "ToolTip.border");
  }

  public void uninstallUI(JComponent c) {
    LookAndFeel.uninstallBorder(c);
  }

   public static void setDisplayAcceleratorKey(boolean val) {
    displayAccelerator=val;
  }

  public Dimension getPreferredSize(JComponent c) {
    Font font = c.getFont();
    FontMetrics fontMetrics =
      Toolkit.getDefaultToolkit().getFontMetrics(font);
    int fontHeight = fontMetrics.getHeight();
    String tipText = ((JToolTip)c).getTipText();
    if (tipText == null)
      tipText = "";
    String lines[] = PlafMacros.breakupLines(tipText);
    int num_lines = lines.length;

    Dimension dimension;
    int width, height, onewidth;
    height = num_lines * fontHeight;
    width = 0;
    for (int i=0; i<num_lines; i++)
    {
      onewidth = fontMetrics.stringWidth(lines[i]);
      if (displayAccelerator && i == num_lines - 1)
      {
        String keyText = getAcceleratorString((JToolTip)c);
        if (!keyText.equals(""))
           onewidth += fontMetrics.stringWidth(keyText)
            + accelerator_offset;
      }
      width = Math.max(width,onewidth);
    }
    return new Dimension(width+inset*2,height+inset*2);
  }
```

```
public Dimension getMinimumSize(JComponent c) {
  return getPreferredSize(c);
}

public Dimension getMaximumSize(JComponent c) {
  return getPreferredSize(c);
}

public void paint(Graphics g, JComponent c) {
  Font font = c.getFont();
  FontMetrics fontMetrics =
    Toolkit.getDefaultToolkit().getFontMetrics(font);
  Dimension dimension = c.getSize();
  int fontHeight = fontMetrics.getHeight();
  int fontAscent = fontMetrics.getAscent();
  String tipText = ((JToolTip)c).getTipText();
  String lines[] = PlafMacros.breakupLines(tipText);
  int num_lines = lines.length;
  int height;
  int i;

  g.setColor(c.getBackground());
  g.fillRect(0, 0, dimension.width, dimension.height);
  g.setColor(c.getForeground());
  for (i=0, height=2+fontAscent;
       i<num_lines; i++, height+=fontHeight)
  {
    g.drawString(lines[i], inset, height);
    if (displayAccelerator && i == num_lines - 1)
    {
      String keyText = getAcceleratorString((JToolTip)c);
      if (!keyText.equals(""))
      {
        Font smallFont = new Font(font.getName(),
          font.getStyle(), font.getSize()-2);
        g.setFont(smallFont);
        g.drawString(keyText, fontMetrics.stringWidth(lines[i])
          + accelerator_offset, height);
      }
    }
  }
}

public String getAcceleratorString(JToolTip tip) {
  JComponent comp = tip.getComponent();
  if (comp == null)
    return "";
  KeyStroke[] keys =comp.getRegisteredKeyStrokes();
  String controlKeyStr = "";
  KeyStroke postTip=KeyStroke.getKeyStroke(
    KeyEvent.VK_F1,Event.CTRL_MASK);

  for (int i = 0; i < keys.length; i++)
  {
    // Ignore ToolTipManager postTip action,
    // in swing1.1beta3 and onward
    if (postTip.equals(keys[i]))
      continue;
    char c = (char)keys[i].getKeyCode();
    int mod = keys[i].getModifiers();
    if ( mod == InputEvent.CTRL_MASK )
    {
```

```
        controlKeyStr = "Ctrl+"+(char)keys[i].getKeyCode();
        break;
      }
      else if (mod == InputEvent.ALT_MASK)
      {
        controlKeyStr = "Alt+"+(char)keys[i].getKeyCode();
        break;
      }
    }
    return controlKeyStr;
  }
}
```

Understanding the Code: Multi-line tooltips

Class PlafMacros (first look)

In order to display the individual lines, we need a method to break up the tooltip text. Using `java.util.StringTokenizer` is unacceptable because:

1. It removes adjacent new lines, and we would like to support multiple blank lines.

2. Some platforms, such as Windows NT, use two characters for a line separator and `java.util.StringTokenizer` does not support multi-character dividers.

3. We would like to support text definition in a java resource file. Since the value for each property in a resource file must appear on a single line, we would like to use the conventional "\n" character to denote a new line, regardless of the running platform's line separator.

Our solution is to write our own method, `breakupLines()`. We define two class variables, `LINE_SEPARATOR` and `LINE_SEPARATOR_LEN` to hold the line separator and the character length of the separator. Within this method we first check to see if the text is empty and if so, return a single element array holding an empty string. If not, we loop through the text, one character at a time, and look for a line separator. If we find one, we extract the line of text preceding it and add it to our vector, `data`. We then repeat the process, but change the starting point to begin after the location of the line separator.

If we cannot find a line separator, we normally increment the starting point by one and repeat the process. But to support text defined in a resource file, we now check to see if "\n" appears at the current location. If we see a "\n", we extract the line of text and add it to the vector, `data`. When this process finishes, we add any trailing text to `data`. Finally, we copy this vector of lines into an array and return the array. Note that `PlafMacros` will also be used in `MultiLineLabelUI`, discussed below.

Class MultiLineToolTipUI

The `MultiLineToolTipUI` extends the `ToolTipUI` base class and implements multiline tooltip. We define two class variables and two instance variables.

Class variables:

`MultiLineToolTipUI SINGLETON`: we only need a single instance to handle all tooltips.

`boolean displayAccelerator`: indicates whether we should display the accelerator key, if any, for a tooltip.

Instance variables:

`int accelerator_offset`: defines how much space should be placed between the last line and the accelerator key.

`int inset`: specifies the inset amount between the tooltip window and text.

There is nothing to be done in the constructor. However, we make it `private` to insure only a single instance of this class gets created.

```
public static void initialize() {
  String key = "ToolTipUI";
  Class cls = SINGLETON.getClass();
  String name = cls.getName();
  UIManager.put(key,name);
  UIManager.put(name,cls);
}
```

The `initialize()` method is the main routine you would call at the start of your program to register the new tooltip UI delegate. (Afterwards, you can create tooltips as you normally would, and embed new lines if you want multiline text.) We define the key, "ToolTipUI", and fetch the singleton's class object and class name. We do it this way, rather than hardcoding the name, so that this class can be easily obfuscated (see note). The next two lines register our user defined `ToolTipUI` with the `UIManager`, by storing the class name and class object.

Note: Obfuscation is the process of renaming classes and variables to shorter, more 'meaningless' names, in an effort to make decompiled classes harder to understand. Many tools exist to perform obfuscation for us at compile time.

The methods: `createUI()`, `installUI()`, and `uninstallUI()` override `ComponentUI`'s methods, and initialize the UI once a java component has been constructed. The `createUI()` method returns an instance (the only instance) of this class. The `installUI()` method initializes the tooltip color and font for the specific `JComponent`, if it was not set before hand. It also installs the tooltip border. The `uninstallUI()` method clears the border, so that the border can be garbage collected.

The `setDisplayAcceleratorKey()` method is a class method that the user can call to toggle between displaying and not displaying the accelerator key.

In `getPreferredSize()` we fetch the tooltip font information, along with the tooltip text. We then call `PlafMacros.breakupLines()` and store the individual lines in the array, `lines`. Next, we iterate through each line to determine the maximum width of the entire text. If we're looking at the last line (and `displayAccelerator` is set to `true`), we take into account the accelerator string, if any, to determine the maximum width. Lastly, we determine the height based on the number of lines. `getMinimumSize()` and `getMaximumSize()` simply returns the same results as `getPreferredSize()`.

Like `getPreferredSize()`, the `paint()` method fetches the tooltip font information along with the tooltip text. It then calls `PlafMacros.breakupLines()` and store the individual lines in the array, `lines`. We set the foreground and background colors, then iterate through each line and draw the text. If there is an accelerator key (and `displayAccelerator` is set to `true`), we draw the accelerator key after the last line, and in a smaller font.

So how do we create a multi-line tooltip?

1. import the `jalt` package:

```
import jalt.*;
```

2. Call the `MultiLineToolTipUI` static method to register the `MultiLineToolTipUI` singleton as the delegate for use by all `JToolTips`:

```
MultiLineToolTipUI.initialize();
```

3. Use "\n" in your tooltip string to specify multiple lines:

```
mylabel.setToolTipText(mytext + "\n\ndoubled space\n\ntooltip");
```

Understanding the Code: Multi-line labels

Much of the multi-line label code is modified from various existing methods in the Swing source code. Moreover, the mechanism is very similar to `MultiLineToolTipUI`. Rather than displaying the entire code, we've decided to highlight the interesting sections and suggest that the reader follow through by referencing the included source code. For now, we will just discuss the multi-line support. Afterwards, we will explain how we support vertical text.

Class PlafMacros (revisited)

In `javax.swing.SwingUtilities`, there is a `layoutCompoundLabel()` method that is called by several UI classes whenever a component (possibly containing an icon) needs to be laid out. This method specifies where the label and icon should be laid out by updating three `Rectangle` instances that are passed in as parameters (along with other information). It also returns the text, and possibly clipped if the text string is too long. However, this method assumes a single line of text. It calculates the text height internally, and always clips the text if the text is too big.

We've modified this method to now take into account multi-line text, along with the ability to optionally clip the text. This new method is stored in the `PlafMacros` class under the same method name, `layoutCompoundLabel()`. The first thing we did is pass two additional parameters to the method:

`int textHeight`: rather than having the method internally determine the text height, we pass this value to the method.

`boolean clipIt`: instead of always clipping the text, we now make it an option.

We then extract the text clipping code into a separate method, `getClippedText()`, and optionally call the method if `clipIt` is `true` and the text width is wider than the display width available. Note that if the label has its own `javax.swing.text.View` renderer stored as the client's `html` property, the method ignores the `textheight` parameter, and uses the renderer to size the text.

Another method we've modified is `drawString()`, as defined in `javax.swing.plaf.basic.BasicGraphicsUtils`. This method draws the string, and underlines the first character that matches the mnenomic (if any). We now make `drawString()` return `true` if it did underline the mnenomic character, otherwise it should return `false`. This will insure that the mnenomic character is underlined only once, even if the same character appears on multiple lines in the label.

Class MultiLineLabelUI (horizontal text support)

We will first explain how we implement horizontal multi-line text and gloss over the code supporting vertical text. Afterwards, we will explain vertical text support in detail.

We first define several class variables, which will be used by the user to specify individual (and default) alignment modes and clip modes.

```
public static String MULTI_ALIGNMENT_MODE = "multiAlignment";
public static String BLOCK = "block";
public static String INDIVIDUAL = "individual";

public static String CLIPPED_MODE = "clippedMode";
```

```
  public static String CLIPPED = "clipped";
  public static String NOT_CLIPPED = "notClipped";

  // we only need a single instance to handle all labels.
  protected static MultiLineLabelUI SINGLETON = new MultiLineLabelUI();

  private static String defaultAlignmentMode = INDIVIDUAL;
  private static String defaultClippedMode = CLIPPED;
```

Like MultiLineToolTipUI, MultiLineLabelUI defines a singleton for use by all labels. MultiLineLabelUI supports BLOCK and INDIVIDUAL alignment. This allows us to align all lines in a label together or separately. Figure 27.3 illustrates:

Figure 27.3 INDIVIDUAL and BLOCK alignment
<<file figure27-3.gif>>

The user can specify the default alignment mode of all labels using the MultiLineLabelUI static method setDefaultAlignmentMode(). The user can also specify the alignment mode for each individual label, by assigning a MULTI_ALIGNMENT_MODE client property (using JComponent's putClientProperty() method--refer back to chapter 1).

For example, to set the individual alignment mode, regardless of the default alignment, on a label using MultiLineLablUI, you would do this:

```
  myLabel.putClientProperty(MultiLineLabelUI.MULTI_ALIGNMENT_MODE,
    MultiLineLabelUI.INDIVIDUAL);
```

putClientProperty() fires a PropertyChangeEvent, and the UI's propertyChange() method receives it, and repaints the label accordingly. This method overrides its BasicLabelUI counterpart, and sends all traffic not corresponding to our alignment or clipping (see below) properties up to this class for processing:

```
  public void propertyChange(PropertyChangeEvent e) {
    if (e.getPropertyName().equals(MULTI_ALIGNMENT_MODE) ||
        e.getPropertyName().equals(CLIPPED_MODE)) {
      JLabel label = (JLabel) e.getSource();
      label.repaint();
    } else {
      super.propertyChange(e);
    }
  }
```

Note: BasicLabelUI implements the PropertyChangeListener interface, and when this UI is installed (i.e. when its installUI() method is invoked) BasicLabelUI also attaches itself as a PropertyChangeListener through JComponent's addPropertyChangeListener() method. Since MultiLineLabelUI extends this class, and its installUI() method just calls super.installUI(), this functionality is fully inherited.

MultiLineLabelUI also supports CLIPPED and NOT_CLIPPED text. The user can specify the default clipped mode in setDefaultClippedMode(). The user can also specify the clipping mode for each individual label, by assigning a CLIPPED_MODE property. Figure 27.4 illustrates. The left-most picture shows

a label with enough room to fit the complete text, so clipping would not come into play whether enabled or not. The middle picture shows that same label with its bounds reduced and clipping enabled. The right-most pitcure shows that label with clipping turned off. Each of these labels is in `INDIVIDUAL` alignment mode.

Figure 27.4 `CLIPPED` and `NOT_CLIPPED` clipping modes
<<file figure27-4.gif>>

Note: Clipping occurs when it is determined that assigned text will not fit within a label's bounds. When clipping occurs "..." is appended to the end of the label text. Our `PlafMacros` class provides a way around this, as it is often unacceptable behavior.

The `initialize()` method is similar to `MultiLineToolTipUI`'s `initialize()` method. We call this method to register the new `UI` delegate.

```
  public static void initialize() {
    // don't hardcode the class name, this way we can obfuscate.
    String key = "LabelUI";
    Class cls = SINGLETON.getClass();
    String name = cls.getName();
    UIManager.put(key,name);
    UIManager.put(name,cls);
  }
```

The `getPreferredSize()` method first iterates through each line to determine the maximum width and total number of lines.

```
      FontMetrics fm = label.getToolkit().getFontMetrics(font);
      int fontHeight = fm.getHeight();
      String lines[] = PlafMacros.breakupLines(text);
      int num_lines = lines.length;
      String maxline = "";
      int maxwidth=0;
      boolean clippedMode = isClippedMode(label);

      for (int i=0; i<num_lines; i++) {
        int w = fm.stringWidth(lines[i]);
        if (w > maxwidth) {
          maxline = lines[i];
          maxwidth = w;
        }
      }
```

Then it calls `layoutCL()` which then calls our version of `layoutCompoundLabel()`, passing it the text height (`num_lines*fontHeight`) and the clip mode (`clippedMode`).

```
      layoutCL(label,fm,maxline,icon,viewR,iconR,textR,
               num_lines*fontHeight,clippedMode);
```

If the label has its own renderer stored in the client property, `html`, `layoutCL()` will use this renderer to size and layout the text, ignoring the given text and font height parameters.

The `paint()` method also iterates through each line to determine the maximum width and total number of lines. It then calls `layoutCL()` to get the label (and possibly icon) layout information. The next block of

code does the actual painting.

If the label has its own text renderer, it uses this renderer to do the actual painting. Otherwise, for `BLOCK` alignment, it iterates through each line, clipping each line if necessary. For `INDIVIDUAL` alignment, `layoutCL()` is called again, since each line needs to be realigned horizontally. `layoutCL()` will then return the text, possibly clipped. For each line we then call `paintEnabledText()` or `paintDisabledText()`, depending on the label's enabled state, to actually paint the text. These two routines will then return `true` if the mnemonic character has been underlined for the particular line. This provides us with a mechanism for guaranteeing that future lines will not underline any characters matching the mnemonic.

So how do we create a multi-line label?

1. import the `jalt` package:

```
import jalt.*;
```

2. Call the `MultiLineLabelUI` static method to register the `MultiLineLabelUI` singleton as the delegate for use by all `JLabels`:

```
MultiLineLabelUI.initialize();
```

3. Use "\n" or get your system's line separator string and use it in your label to specify multiple lines:

```
String separator = System.getProperty("line.separator");
JLabel label = new JLabel("some text" + separator +
  "some more text" + separator +
  "last line of text");
```

4. If you want html based text, specify the text using html tags:

```
JLabel label = new JLabel("<html>some text<br>some more text<br>");
```

Class MultiLineLabelUI (vertical text support)

Vertical text support is a bit slower to draw, mainly because each character needs to be rendered individually. However, since most applications require very little amount of vertical text, this is not likely to present any significant performance problems.

We first define a few more class variables.

```
  public static String ORIENTATION_MODE = "orientationMode";
  public static String HORIZONTAL = "horizontal";
  public static String VERTICAL = "vertical";

  public static String VERTICAL_SPACE = "verticalSpace";
  public static String VERTICAL_WIDTH = "verticalWidth";

  private static String DEFAULT_VERTICAL_WIDTH = "W";
  private static String DEFAULT_VERTICAL_SPACE = " ";
```

`ORIENTATION_MODE` is another property used to define the orientation for a specific label. `VERTICAL_SPACE` and `VERTICAL_WIDTH` are properties used to specify the text width of each column and the amount of space between each column (the default is `W` and a blank space, respectively). Ideally, we would use integers to specify the number of points for the width and column spacing. However, in order to limit the number of changes to `layoutCompoundLabel()`, we use character strings instead.

The user can specify the default vertical width and column spacing using the static methods `setDefaultVerticalSpace()` and `setDefaultVerticalWidth()`.

We then define the `getTotalVerticalWidth()` method which returns a string representing the maximum width of the text it will be painting. This string is based on the number of lines of text, multiplied by the column width and column spacing.

```
protected String getTotalVerticalWidth(JLabel label, int numLines) {
  String space = getVerticalSpace(label);
  char[] spaceArr = space.toCharArray();
  int spaceLen = spaceArr.length;
  String width = getVerticalWidth(label);
  char[] widthArr = width.toCharArray();
  int widthLen = widthArr.length;
  char[] totalWidth = new char[spaceLen*numLines+widthLen*(numLines-1)];
  for (int i=0; i<numLines; i++) {
    System.arraycopy(widthArr,0,totalWidth,i*(spaceLen+widthLen),widthLen);
    if (i!=numLines-1) {
      System.arraycopy(spaceArr,
                       0,
                       totalWidth,
                       i*(spaceLen+widthLen)+widthLen,
                        spaceLen);
    }
  }
  return new String(totalWidth);
}
```

In the `getPreferredSize()` method, if the text should be drawn vertically, we call `getTotalVerticalWidth()` to fetch the string representing the maximum width. Next, we call `layoutCL()`, passing this string as the text. For the `textHeight`, we take the number of characters of the maximum line, and multiply it by the font height (`maxline.length()*fontHeight`).

The `paint()` method does the same thing, it calls `getTotalVerticalWidth()` and then calls `layoutCL()` to get the label (and possibly icon) layout information. It then iterates through each line, and draws each character, horizontally centered in the associated column. For `INDIVIDUAL` alignment, it calls `layoutCL()` on each line to re-layout the vertical position of the text.

So how do we create a vertical label?

1. import the `jalt` package:

   ```
   import jalt.*;
   ```

2. Call the `MultiLineLabelUI` static method to register the `MultiLineLabelUI` singleton as the delegate for use by all `JLabels`:

   ```
   MultiLineLabelUI.initialize();
   ```

3. Define your label, using "\n" or the system's line separator string to specify multiple lines and then set the `ORIENTATION_MODE` property to specify vertical text.

   ```
   JLabel label = new JLabel("some text\nsome more text");
   Label.putClientProperty(MultiLineLabelUI.ORIENTATION_MODE,
                           MultiLineLableUI.VERTICAL);
   ```

Running the Code:

An example has been included demonstrating multi-line tooltips and labels, along with vertical labels. Figure 27.1 shows it in action. The menu bar provides choices allowing us to change the alignment of the labels in both the table and the labels on the right-hand side. We can also change the clipping mode of the table's cells, and toggle the display of accelerators in tooltips. Figure 27.2 shows the "Exit" menu item's tooltip displayed with no accelerator (left) and with accelerator (right).

27.2 Swingscape browser

...by Ron Widitz, Sr. Technologist, Zeal, Inc. a modis Solutions Co., rwiditz@modisit.com, http://www.zealinc.com/

This application uses Swing HTML functionality and undo/redo facilities to provide a capable browser (see chapter 11 for an overview of `JEditorPane` and the `javax.swing.undo` package). It demonstrates the use of context-sensitive tooltips with toolbar buttons, unlimited levels of undo, and an HTML 3.2 viewer supporting tables, frames, forms and links. It also provides navigation via dropdown menus, mnemonics, hyperlinks, form submits, toolbar buttons, and text entry.

Figure 27.5 Running the Swingscape example displaying a complex HTML rendering.
<<file figure27-5.gif>>

The Code: Swingscape.java
see \Chapter27\Widitz

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.undo.*;
import javax.swing.text.*;
import javax.swing.text.html.*;
import javax.swing.event.*;
```

```
public class Swingscape extends JFrame implements HyperlinkListener
{
  public static final String TITLE_TEXT = "Swingscape browser";
  public static final String HOME_DEFAULT =
    "http://java.sun.com/products/jfc/";
  protected String m_CurrentURL = HOME_DEFAULT;

  protected JEditorPane m_HtmlPane;
  protected JToolBar m_toolBar;
  protected JButton m_btnBack, m_btnForward, m_btnReload;
  protected JMenuItem m_mnuBack, m_mnuForward;
  protected JTextField m_txtURL;

  protected URLundoManager m_undo = new URLundoManager();

  public Swingscape() {
    super(TITLE_TEXT);

    Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
    if ( dim.width <= 640 ) {
      setSize(dim.width,dim.height);
    }
    else {
      int initWidth = dim.width - dim.width/10;
      int initHeight = dim.height - dim.height/10;
      setSize( initWidth, initHeight );
      setLocation( dim.width/2-initWidth/2,
        dim.height/2-initHeight/2 );
    }

    setJMenuBar(createMenuBar());
    getContentPane().add(m_toolBar, BorderLayout.NORTH);

    m_HtmlPane = new JEditorPane();
    m_HtmlPane.setEditorKit(new HTMLEditorKit());
    m_HtmlPane.setEditable(false);
    m_HtmlPane.addHyperlinkListener(this);

    JScrollPane mScroller = new JScrollPane();
    mScroller.getViewport().add(m_HtmlPane);
    getContentPane().add(mScroller, BorderLayout.CENTER);

    addWindowListener( new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    } );

    m_btnReload.doClick();
    setVisible(true);
  }

  protected JMenuBar createMenuBar() {
    JMenuBar menuBar = new JMenuBar();

    JMenu mFile = new JMenu("File");
    mFile.setMnemonic('f');

    JMenuItem item = new JMenuItem("Reload");
    item.setMnemonic('r');

    Action actionReload = new AbstractAction("Reload") {
```

```
  public void actionPerformed(ActionEvent e) {
    Thread runner = new Thread() {
      public void run() {
        m_HtmlPane.setText("reset doc & reload");
        DisplayPageDirect(m_CurrentURL);
      }
    };
    runner.start();
  }
};

item.addActionListener(actionReload);
mFile.add(item);

item = new JMenuItem("Exit");
item.setMnemonic('x');

item.addActionListener( new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
});

mFile.add(item);
menuBar.add(mFile);

JMenu mGo = new JMenu("Go");
mGo.setMnemonic('g');

m_mnuBack = new JMenuItem("Back");
m_mnuBack.setMnemonic('b');
m_mnuBack.setEnabled(false);

Action actionBack = new AbstractAction("Back") {
  public void actionPerformed(ActionEvent e) {
    Thread runner = new Thread() {
      public void run() {
        try {
          String mDoURL = m_undo.swapURL(m_CurrentURL);
          m_undo.undo();  //URL now in redo
          DisplayPageDirect(mDoURL);
        }
        catch (CannotUndoException exc) {}
        finally {
          updateMenu_Buttons();
        }
      }
    };
    runner.start();
  }
};

m_mnuBack.addActionListener(actionBack);
mGo.add(m_mnuBack);

m_mnuForward = new JMenuItem("Forward");
m_mnuForward.setMnemonic('f');
m_mnuForward.setEnabled(false);

Action actionForward = new AbstractAction("Forward") {
  public void actionPerformed(ActionEvent e) {
    Thread runner = new Thread() {
```

```
      public void run() {
        try {
          m_undo.redo();
          DisplayPageDirect(m_undo.swapURL(m_CurrentURL));
        }
        catch (CannotRedoException exc) {}
        finally {
          updateMenu_Buttons();
        }
      }
    };
    runner.start();
  }
};

m_mnuForward.addActionListener(actionForward);
mGo.add(m_mnuForward);

item = new JMenuItem("Home");
item.setMnemonic('h');

Action actionHome = new AbstractAction("Home") {
  public void actionPerformed(ActionEvent e) {
    Thread runner = new Thread() {
      public void run() {
        Display_RecordUndo(HOME_DEFAULT);
      }
    };
    runner.start();
  }
};

item.addActionListener(actionHome);
mGo.add(item);
menuBar.add(mGo);

m_toolBar = new JToolBar();

m_btnBack = m_toolBar.add(actionBack);
m_btnBack.setEnabled(false);
m_btnBack.setBorderPainted(false);
m_btnBack.setRequestFocusEnabled(false);
m_btnBack.addMouseListener(new PopButtonListener());
m_toolBar.addSeparator();

m_btnForward = m_toolBar.add(actionForward);
m_btnForward.setEnabled(false);
m_btnForward.setBorderPainted(false);
m_btnForward.setRequestFocusEnabled(false);
m_btnForward.addMouseListener(new PopButtonListener());
m_toolBar.addSeparator();

m_btnReload = m_toolBar.add(actionReload);
m_btnReload.setToolTipText(HOME_DEFAULT);
m_btnReload.setBorderPainted(false);
m_btnReload.setRequestFocusEnabled(false);
m_btnReload.addMouseListener(new PopButtonListener());
m_toolBar.addSeparator();

JButton btnHome = m_toolBar.add(actionHome);
btnHome.setToolTipText(HOME_DEFAULT);
btnHome.setBorderPainted(false);
```

```
  btnHome.setRequestFocusEnabled(false);
  btnHome.addMouseListener(new PopButtonListener());
  m_toolBar.addSeparator();

  m_txtURL = new JTextField(HOME_DEFAULT);

  m_txtURL.addActionListener( new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      Thread runner = new Thread() {
        public void run() {
          String m_URL = m_txtURL.getText().trim();
          if ( m_URL.length() > 0 )
          {
            String checkForProtocol =
              m_URL.substring(0,7).toLowerCase();
            if (!checkForProtocol.equals("http://") &&
                !checkForProtocol.startsWith("file:/"))
            {
              if (checkForProtocol.indexOf(':') == 1)
              { // Drive letter?
                m_URL = "file:/"+m_URL;
              }
              else
              { // Assume a website...
                m_URL = "http://"+m_URL;
              }
            }
            Display_RecordUndo(m_URL);
          }
        }
      };
      runner.start();
    }
  });

  m_toolBar.add(m_txtURL);
  m_toolBar.setFloatable(false);

  return menuBar;
}

public void hyperlinkUpdate(HyperlinkEvent e) {
  if ( e.getEventType() == HyperlinkEvent.EventType.ACTIVATED ) {
    final String dest =  e.getURL().toString();
    Thread runner = new Thread() {
      public void run() {
        Display_RecordUndo(dest);
      }
    };
    runner.start();
  }
}

public void updateMenu_Buttons() {
  boolean mDoState = m_undo.canUndo();

  m_mnuBack.setEnabled(mDoState);
  m_btnBack.setEnabled(mDoState);

  if ( mDoState ) {
    m_btnBack.setToolTipText(m_undo.getUndoPresentationName());
  }
```

```
    else {
      m_btnBack.setToolTipText(null);
    }

    mDoState = m_undo.canRedo();

    m_mnuForward.setEnabled(mDoState);
    m_btnForward.setEnabled(mDoState);

    if ( mDoState ) {
      m_btnForward.setToolTipText(m_undo.getRedoPresentationName());
    }
    else {
      m_btnForward.setToolTipText(null);
    }
  }

  public void Display_RecordUndo(String strURL) {
    String mCompareURL = strURL.intern();

    if ( m_CurrentURL != mCompareURL ) {
      m_undo.addURL(m_CurrentURL);
      updateMenu_Buttons();
      DisplayPageDirect(mCompareURL);
    }
  }

  public void DisplayPageDirect(String strURL) {
    m_CurrentURL = strURL;
    m_txtURL.setText(strURL);
    m_btnReload.setToolTipText(strURL);

    try {
      // setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
      // future feature
      m_HtmlPane.setPage(strURL);
    }
    catch (Exception exc) {
      System.out.println("Problem loading URL...");
    }
    /*  setCursor() expected to work properly in
        JDK1.1.8 & JDK1.2.2, bug#4160474
    finally {
      setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
    }
    */
  }

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

  class PopButtonListener extends MouseAdapter {
    public void mouseEntered(MouseEvent e) {
      ( (JButton)e.getSource() ).setBorderPainted(true);
    }

    public void mouseExited(MouseEvent e) {
      ( (JButton)e.getSource() ).setBorderPainted(false);
    }
  }
```

```
class UndoableURL extends AbstractUndoableEdit {
  private String m_URL;

  public UndoableURL(String m_URL) {
    this.m_URL = m_URL;
  }

  public String getPresentationName() {
    return m_URL;
  }
}

class URLundoManager extends CompoundEdit {
  int m_IdxAdd = 0;

  public String getUndoPresentationName() {
    return ((UndoableURL)
      edits.elementAt(m_IdxAdd-1)).getPresentationName();
  }

  public String getRedoPresentationName() {
    return ((UndoableURL)
      edits.elementAt(m_IdxAdd)).getPresentationName();
  }

  public void addURL(String newURL) {
    if ( edits.size()>m_IdxAdd ) {
      edits.setElementAt(new UndoableURL(newURL),m_IdxAdd++);
      for ( int i=m_IdxAdd; i<edits.size(); i++ ) {
        edits.removeElementAt(i);
      }
    }
    else {
      edits.addElement(new UndoableURL(newURL));
      m_IdxAdd++;
    }
  }

  public String swapURL(String newURL) {
    String m_oldURL = getUndoPresentationName();
    edits.setElementAt(new UndoableURL(newURL),m_IdxAdd-1);
    return m_oldURL;
  }

  public synchronized boolean canUndo() {
    if ( m_IdxAdd > 0 ) {
      UndoableURL edit = (UndoableURL)edits.elementAt(m_IdxAdd-1);
      return edit != null && edit.canUndo();
    }
    return false;
  }

  public synchronized boolean canRedo() {
    if ( edits.size()>m_IdxAdd ) {
      UndoableURL edit = (UndoableURL)edits.elementAt(m_IdxAdd);
      return edit != null && edit.canRedo();
    }
    return false;
  }

  public synchronized void undo() throws CannotUndoException {
    ( (UndoableURL)edits.elementAt(--m_IdxAdd) ).undo();
```

```
    }

    public synchronized void redo() throws CannotRedoException {
      ( (UndoableURL)edits.elementAt(m_IdxAdd++) ).redo();
    }
  }
}
```

Understanding the Code

Class Swingscape

This class extends `JFrame` to provide the supporting frame for this example, and implements `HyperlinkListener` to handle URL changes that can occur in the contained editor pane. Two class variables are defined:

`String TITLE_TEXT`: title bar text for the application frame.

`String HOME_DEFAULT`: browser pre-defined homepage.

Instance variables:

`String m_CurrentURL`: the URL being displayed (defaults to `HOME_DEFAULT`).

`JEditorPane m_HtmlPane`: text component for HTML display.

`JToolbar m_toolBar`: browser navigation controls toolbar.

`JButton m_btnBack`: push button to move back to a previously visited URL.

`JButton m_btnForward`: push button to move forward to a previously visited URL.

`JButton m_btnReload`: push button to refresh the current page.

`JMenuItem m_mnuBack`: menu selection to move back to a previously visited URL.

`JMenuItem m_mnuForward`: menu selection to move forward to a previously visited URL.

`JTextField m_txtURL`: URL text entry field.

`URLundoManager m_undo`: an object that manages back/forward operations via undo/redo.

The default constructor for `Swingscape` creates and initializes all GUI components. The frame is initialized by calling `super()` and passing it our `TITLE_TEXT` constant, "Swingscape browser." Its initial size is set to the maximum screen dimensions if the screen width is found to be 640 or less. Otherwise 10% is deducted from each dimension and it is centered via `setLocation()`. A call to `createMenuBar()` is made, which initializes and returns a `JMenuBar`, and initializes the `m_toolBar JToolBar`. The resulting menu bar is then assigned to our frame, and the toolbar is placed in the `NORTH` region of the content pane.

An editor pane is created and set to expect HTML documents in read-only (non-editing) mode. Because `Swingscape` implements the `HyperlinkListener` interface, we pass the editor a `this` reference as the destination of `HyperlinkEvents` (see the `hyperlinkUpdate()` method below):

```
    m_HtmlPane = new JEditorPane();
    m_HtmlPane.setEditorKit(new HTMLEditorKit());
    m_HtmlPane.setEditable(false);
    m_HtmlPane.addHyperlinkListener(this);
```

The editor is then placed in a scroll pane which is placed in the `CENTER` of the content pane. The constructor ends with a programmatic click on the "reload" button which will attempt to load the URL represented by `DEFAULT_HOME`. The reason we perform a programmatic click is because the action handling code for that button is wrapped in a separate thread which will not affect the responsiveness of our interface.

The `createMenuBar()` method creates and populates our dropdown menus, toolbar, and the associated action listeners. They both receive components based on `Action` objects that are responsible for the program's navigational features. The action handling code of each `Action` object is wrapped in a separate thread to ensure that our GUI remains responsive at all times.

Menu item "Reload" from the menu "File" provides a browser page refresh:

```
    Action actionReload = new AbstractAction("Reload") {
      public void actionPerformed(ActionEvent e) {
        Thread runner = new Thread() {
          public void run() {
            m_HtmlPane.setText("reset doc & reload");
            DisplayPageDirect(m_CurrentURL);
          }
        };
        runner.start();
      }
    };
```

The action handling code temporarily sets the editor pane text to "reset doc & reload" and then requests the current `URL` to be loaded.

Menu item "Exit" terminates the `Swingscape`'s execution via `System.exit(0)`. Because we don't place an "Exit" button on the toolbar, we simply create an `ActionListener` as an anonymous inner class to handle this menu item.

The "Go" menu has three menu items each with corresponding toolbar buttons. Menu items "Back" and "Forward" are implemented with similar code. The "Back" menu item provides a way of revisiting previous `URL`s in reverse order. It accomplishes this by moving the current `URL` into a redoable state while requesting the undoable `URL`:

```
    Action actionBack = new AbstractAction("Back") {
      public void actionPerformed(ActionEvent e) {
        Thread runner = new Thread() {
          public void run() {
            try {
              String mDoURL = m_undo.swapURL(m_CurrentURL);
              m_undo.undo();  // URL now in redo
              DisplayPageDirect(mDoURL);
            }
            catch (CannotUndoException exc) {}
            finally {
              updateMenu_Buttons();
            }
          }
        };
        runner.start();
      }
    };
```

The `swapURL()` method (see below) grabs the undo/"Back" `URL String` while placing the current one on the undo stack. The `undo()` method is called to move this `URL` to a redo/"Forward" state. The `URL` corresponding to undo/"Back" is requested using our custom `DisplayPageDirect()` method. Since `undo()` can throw an exception, it is wrapped within a try/catch block. The `updateMenu_Buttons()` call is placed in a `finally` clause so that no matter what happens, the state of the toolbar buttons and menu items will be updated according to the undo/redo state of our `URLundoManager` instance, `m_undo` (see below).

Menu item "Forward" reverses any undo/"Back"s by displaying URLs in the order they were originally visited. The relevant code moves the current URL into an undoable state and then the redoable URL is requested:

```
m_undo.redo();
DisplayPageDirect(m_undo.swapURL(m_CurrentURL));
```

Initially, the redo/"Forward" URL is shifted to a temporary undo state via `redo()`. Our `swapURL()` method is then used to grab the former redo/"Forward" URL while placing the currently displayed URL on the undo stack.

Menu item "Home" displays the URL obtained via the `HOME_DEFAULT String` representation. The `Display_RecordUndo()` method (see below) is called to provide undo/redo functionality for this URL change as well.

For a more modernized feel, each toolbar button is created with: no painted border, no keyboard focus changing abilities, and a custom `PopButtonListener` to toggle between border painting modes when the mouse moves in and out of its bounds (see below).

A text field is added to the toolbar for display and entry of destination URLs. Its associated `ActionListener` is constructed as an anonymous inner class. It trims white space, checks for http and the local file protocols, and attempts to select an appropriate protocol, if none is typed, before attempting to load the resulting URL:

```
String checkForProtocol =
  m_URL.substring(0,7).toLowerCase();
if (!checkForProtocol.equals("http://") &&
    !checkForProtocol.startsWith("file:/"))
{
  if (checkForProtocol.indexOf(':') == 1)
  { // Drive letter?
    m_URL = "file:/"+m_URL;
  }
  else
  { // Assume a website...
    m_URL = "http://"+m_URL;
  }
}
Display_RecordUndo(m_URL);
```

Initially, any leading or trailing white spaces are trimmed away using the `String trim()` method. If anything remains, the first seven characters are forced to be lower case for case-insensitive text comparison. If a protocol of "http" or "file" is present, we assume that we can proceed with an attempt to display the URL. If neither protocol is found, a quick check is made for a colon that may represent the second character of a drive specification (e.g. `c:` or `d:` in Windows/DOS-based systems). If this is found we attach "file:/" as a prefix. The URL is then requested via `Display_RecordUndo()` which also places the current URL in an undo/"Back" state.

Note: Operating system specific code might be added to do a better job recognizing filepaths. This functionality is beyond the scope of this example, but would be expected of a commercial implementation.

The `createMenuBar()` method ends by setting the toolbar's floatable property to `false` and returning the menu bar. Both items are dealt with accordingly in the constructor, as discussed above.

Method `hyperlinkUpdate()` is called whenever an HTML link is selected from our `m_HtmlPane` editor

pane. The referenced URL is requested if the event proves to be an ACTIVATED link:

```
if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
  Display_RecordUndo(e.getURL().toString());
}
```

Method updateMenu_Buttons() checks the undoable and redoable states via URLundoManager methods, and sets the menu and toolbar button states accordingly. The tooltips are changed to the current undo and redo URLs, or are temporarily disabled if no undo or redo URLs exist.

Method Display_RecordUndo() efficiently stores a single URL String. The intern() method is called so unique String pool references will be used (this can save memory and provides faster String comparisons). It then checks if the given URL is different than the current URL. If so, it gets added to the undo list, menus/toolbar buttons are updated, and the URL is then requested for display:

```
String mCompareURL = strURL.intern();

if ( m_CurrentURL != mCompareURL ) {
  m_undo.addURL(m_CurrentURL);
  updateMenu_Buttons();
  DisplayPageDirect(mCompareURL);
}
```

As we have seen, all Swingscape URL requests are forwarded to DisplayPageDirect() whose job it is to attempt to load the destination page in our m_HtmlPane editor pane. The URL String parameter becomes the current page and the it is shown in the toolbar's text field. The Reload button's tooltip text is updated to indicate this URL is the one that will be refreshed when pressed.

```
public void DisplayPageDirect(String strURL) {
  m_CurrentURL = strURL;
  m_txtURL.setText(strURL);
  m_btnReload.setToolTipText(strURL);
```

This method assigns a wait cursor to our editor pane as an attempt is made to load the requested URL via m_HtmlPane.setPage(strURL). A finally clause serves to reset the cursor to the default cursor whether the page was successfully loaded or not. (This functionality is commented out--see note below.)

Note: setCursor() logic intermittently functions on Windows™ platforms but Sunsoft claims to have a fix according to Bug Parade#4160474 included in JDK1.1.8 and JDK1.2.2 releases.

Method main() provides the entry point for the application and calls the Swingscape constructor to get things rolling.

Class PopButtonListener

This class extends MouseAdapter to provide popup/hidden button border support for the toolbar. Method mouseEntered() shows the button's border when the mouse enters its bounds, making the buttons appear to popup in a rollover fashion. Method mouseExited() turns off the button border when the mouse moves out of its bounds.

Class UndoableURL

This class extends AbstractUndoableEdit to provide the undoable/redoable URL objects (which correspond to "Back"/"Forward" actions, respectively). URLundoManager (see below) collectively organizes these objects. One instance variable is declared:

String m_URL: "Back" or "Forward" String URL including protocol and any trailing arguments.

The constructor of the `UndoableURL()` initializes this variable as the given `String` parameter. Method `getPresentationName()` simply returns this `String` when requested by our `URLundoManager`.

Class URLundoManager

This class extends `CompoundEdit` to provide the mechanism behind the browser's undo/redo ("Back"/"Forward") functionality. As we learned in chapter 19, `CompoundEdit` contains a protected `Vector` called `edits` used to store its `UndoableEdits`. We also learned that the `UndoManager` class extends `CompoundEdit` to provide a convenient mechanism for undoing and redoing single edits.

Our `URLundoManager` class functions similar to `UndoManager`, but streamlines its methods for the browser's needs while providing additional control over undo/redo states. The most significant difference is that we support an intermediate state between undo and redo. For instance, when an undo is invoked, the associated `UndoableURL` will not automatically move to the redo state. It will only move to the redo state after it has been displayed and a new `URL` is entered, or another undo is performed. This will become clear as we explain the methods below. Additionaly, because `URLundoManager` performs its own management of its `edits Vector`, whenever a new edit is added all redoable edits need to be removed.

One instance variable is declared:

`int m_IdxAdd`: offset into the `edits Vector` used to determine where undo's end and redo's begin. It always points to the position where we would next add an undoable; this would also be the current redoable (i.e. "Forward") if there are any. `m_IdxAdd-1` will point to the most recent undo (assuming `m_IdxAdd` is greater than `0`).

Method `getUndoPresentationName()` retrieves the `String URL` of the most recent undoable `UndoableURL` by calling its `getPresentationName()` method:

```
return ((UndoableURL)
  edits.elementAt(m_IdxAdd-1)).getPresentationName();
```

Method `getRedoPresentationName()` retrieves the `String URL` from the current redoable. It works identical to `getUndoPresentationName()` except it uses `m_IdxAdd` as its offset into the `edits Vector`.

Method `addURL()` creates a new `UndoableURL` and adds it to the `edits Vector`. Initially, it checks if any redos exist:

```
if ( edits.size()>m_IdxAdd ) {
```

If so, it stores the new edit in the most recent redoable position and eliminates any remaining redoables:

```
  edits.setElementAt(new UndoableURL(newURL),m_IdxAdd++);
  for ( int i=edits.size()-1; i>m_IdxAdd-1; i-- ) {
    edits.removeElementAt(i);
  }
}
```

If there aren't any redoables, the edit is simply added to the end of the `edits Vector`, and the current offset is incremented:

```
edits.addElement(new UndoableURL(newURL));
m_IdxAdd++;
```

Method `swapURL()` returns the most recent `UndoableURL`'s `URL` and replaces it with a new one based on the given `String URL` parameter:

```
public String swapURL(String newURL) {
  String m_oldURL = getUndoPresentationName();
  edits.setElementAt(new UndoableURL(newURL),m_IdxAdd-1);
  return m_oldURL;
}
```

This method helps provide an intermediate state between undo and redo. The current URL is neither undoable nor redoable until it is replaced.

Method `canUndo()` determines whether any undoables exist. This is used to visually set the "Back" menu item and button states. It checks whether the undoable offset corresponds to a valid edit and then whether that edit is undoable:

```
if (m_IdxAdd > 0) {
  UndoableURL edit = (UndoableURL)edits.elementAt(m_IdxAdd-1);
  return edit != null && edit.canUndo();
}
return false;
```

Method `canRedo()` determines whether any redoables exist. This is used to visually set the "Forward" menu item and button states, and works very similar to `canUndo()`.

Method `undo()` executes the most recent `UndoableURL`'s internal housekeeping (by calling its inherited `undo()` method). Before using this method, we are expected to call the `swapURL()` method so the current URL will be placed in a redo/"Forward" state. Method `redo()` works similarly, however, proper browser redo/"Forward" behavior necessitates calling `swapURL()` after this method so that the current URL will be moved to an undo state.

Running the Code

The default homepage is Sun's JFC product webpage. It demonstrates HTML tables, forms, graphics and even an animated Java mascot--Duke is swinging near the bottom of the page. Choose some links or enter a different URL in the text field. You will see the "Back" button become bold and its tooltip will reflect the last page visited. If you press "Back" or choose Go/"Back" from the menu, you'll see that the "Forward" button becomes enabled (as expected). Notice how the menu item, button states and tooltips reflect the currently allowed options.

Bug Alert! The HTMLEditorKit is not completely matured yet. Expect occasional exceptions. See bug #4180751 at the JDC Bug Parade for the most common exception we experienced. It is a good idea to run this example and test whether this bug has been fixed in your version of Java. The exception message has the following format:

Exception occurred during event dispatching:

Javax.swing.text.StateInvariantError: infinite loop in formatting

---