FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

CHAPTER 22

Event Handling

This chapter examines an important aspect of Java: the event. Event handling is fundamental to Java programming because it is integral to the creation of applets and other types of GUI-based programs. As explained in Chapter 21, applets are event-driven programs that use a graphical user interface to interact with the user. Furthermore, any program that uses a graphical user interface, such as a Java application written for Windows, is event driven. Thus, you cannot write these types of programs without a solid command of event handling. Events are supported by a number of packages, including **java.util**, **java.awt**, and **java.awt.event**.

Most events to which your program will respond are generated when the user interacts with a GUI-based program. These are the types of events examined in this chapter. They are passed to your program in a variety of ways, with the specific method dependent upon the actual event. There are several types of events, including those generated by the mouse, the keyboard, and various GUI controls, such as a push button, scroll bar, or check box.

This chapter begins with an overview of Java's event handling mechanism. It then examines the main event classes and interfaces used by the AWT and develops several examples that demonstrate the fundamentals of event processing. This chapter also explains how to use adapter classes, inner classes, and anonymous inner classes to streamline event handling code. The examples provided in the remainder of this book make frequent use of these techniques.

NOTE *This chapter focuses on events related to GUI-based programs. However, events are also occasionally used for purposes not directly related to GUI-based programs. In all cases, the same basic event handling techniques apply.*

Two Event Handling Mechanisms

Before beginning our discussion of event handling, an important point must be made: The way in which events are handled changed significantly between the original version of Java (1.0) and modern versions of Java, beginning with version 1.1. The 1.0 method of event handling is still supported, but it is not recommended for new programs. Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs and thus is the method employed by programs in this book.

The Delegation Event Model

The modern approach to handling events is based on the *delegation event model,* which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners.* In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the old Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

NOTE *Java also allows you to process events without using the delegation event model. This can be done by extending an AWT component. This technique is discussed at the end of Chapter 24. However, the delegation event model is the preferred design for the reasons just cited.*

The following sections define events and describe the roles of sources and listeners.

Events

In the delegation model, an *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

Event Sources

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

```
public void add Type Listener(Type Listener el)
```

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

```
public void addTypeListener(TypeListener el)
  throws java.util.TooManyListenersException
```

Here, *Type* is the name of the event, and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

```
public void removeTypeListener(TypeListener el)
```

Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. Many other listener interfaces are discussed later in this and other chapters.

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Thus, a discussion of event handling must begin with the event classes. It is important to understand, however, that Java defines several types of events and that not all event classes can be discussed in this chapter. The most widely used events are those defined by the AWT and those defined by Swing. This chapter focuses on the AWT events. (Most of these events also apply to Swing.) Several Swing-specific events are described in Chapter 29, when Swing is covered.

At the root of the Java event class hierarchy is **EventObject**, which is in **java.util**. It is the superclass for all events. Its one constructor is shown here:

```
EventObject(Object src)
```

Here, *src* is the object that generates this event.

EventObject contains two methods: **getSource( )** and **toString( )**. The **getSource( )** method returns the source of the event. Its general form is shown here:

```
Object getSource( )
```

As expected, **toString( )** returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**. It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:

```
int getID( )
```

Additional details about **AWTEvent** are provided at the end of Chapter 24. At this point, it is important to know only that all of the other classes discussed in this section are subclasses of **AWTEvent**.

To summarize:

- **EventObject** is a superclass of all events.
- **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.

The package **java.awt.event** defines many types of events that are generated by various user interface elements. Table 22-1 shows several commonly used event classes and provides a brief description of when they are generated. Commonly used constructors and methods in each class are described in the following sections.

The ActionEvent Class

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or a menu item is selected. The **ActionEvent** class defines four integer constants that can be used to identify any modifiers associated with an action event: **ALT_MASK**, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK**. In addition, there is an integer constant, **ACTION_PERFORMED**, which can be used to identify action events.

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract superclass for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-1 Main Event Classes in **java.awt.event**

ActionEvent has these three constructors:

ActionEvent(Object *src*, int *type*, String *cmd*)
ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)
ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*, and its command string is *cmd*. The argument *modifiers* indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when* parameter specifies when the event occurred.

You can obtain the command name for the invoking **ActionEvent** object by using the **getActionCommand( )** method, shown here:

String getActionCommand( )

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

The **getModifiers( )** method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. Its form is shown here:

int getModifiers( )

The method **getWhen( )** returns the time at which the event took place. This is called the event's *timestamp*. The **getWhen( )** method is shown here:

long getWhen( )

The AdjustmentEvent Class

An **AdjustmentEvent** is generated by a scroll bar. There are five types of adjustment events. The **AdjustmentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

In addition, there is an integer constant, **ADJUSTMENT_VALUE_CHANGED**, that indicates that a change has occurred.

Here is one **AdjustmentEvent** constructor:

AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *data*)

Here, *src* is a reference to the object that generated this event. The *id* specifies the event. The type of the adjustment is specified by *type*, and its associated data is *data*.

The **getAdjustable( )** method returns the object that generated the event. Its form is shown here:

Adjustable getAdjustable( )

The type of the adjustment event may be obtained by the **getAdjustmentType( )** method. It returns one of the constants defined by **AdjustmentEvent**. The general form is shown here:

```
int getAdjustmentType( )
```

The amount of the adjustment can be obtained from the **getValue( )** method, shown here:

```
int getValue( )
```

For example, when a scroll bar is manipulated, this method returns the value represented by that change.

The ComponentEvent Class

A **ComponentEvent** is generated when the size, position, or visibility of a component is changed. There are four types of component events. The **ComponentEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

ComponentEvent has this constructor:

```
ComponentEvent(Component src, int type)
```

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type.*

ComponentEvent is the superclass either directly or indirectly of **ContainerEvent**, **FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**.

The **getComponent( )** method returns the component that generated the event. It is shown here:

```
Component getComponent( )
```

The ContainerEvent Class

A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events. The **ContainerEvent** class defines **int** constants that can be used to identify them: **COMPONENT_ADDED** and **COMPONENT_REMOVED**. They indicate that a component has been added to or removed from the container.

ContainerEvent is a subclass of **ComponentEvent** and has this constructor:

```
ContainerEvent(Component src, int type, Component comp)
```

Here, *src* is a reference to the container that generated this event. The type of the event is specified by *type,* and the component that has been added to or removed from the container is *comp.*

You can obtain a reference to the container that generated this event by using the **getContainer( )** method, shown here:

Container getContainer( )

The **getChild( )** method returns a reference to the component that was added to or removed from the container. Its general form is shown here:

Component getChild( )

The FocusEvent Class

A **FocusEvent** is generated when a component gains or loses input focus. These events are identified by the integer constants **FOCUS_GAINED** and **FOCUS_LOST**.

FocusEvent is a subclass of **ComponentEvent** and has these constructors:

FocusEvent(Component *src*, int *type*)
FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)
FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

Here, *src* is a reference to the component that generated this event. The type of the event is specified by *type.* The argument *temporaryFlag* is set to **true** if the focus event is temporary. Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another user interface operation. For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.)

The other component involved in the focus change, called the *opposite component*, is passed in *other.* Therefore, if a **FOCUS_GAINED** event occurred, *other* will refer to the component that lost focus. Conversely, if a **FOCUS_LOST** event occurred, *other* will refer to the component that gains focus.

You can determine the other component by calling **getOppositeComponent( )**, shown here:

Component getOppositeComponent( )

The opposite component is returned.

The **isTemporary( )** method indicates if this focus change is temporary. Its form is shown here:

boolean isTemporary( )

The method returns **true** if the change is temporary. Otherwise, it returns **false**.

The InputEvent Class

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

InputEvent defines several integer constants that represent any modifiers, such as the control key being pressed, that might be associated with the event. Originally, the **InputEvent** class defined the following eight values to represent the modifiers:

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

However, because of possible conflicts between the modifiers used by keyboard events and mouse events, and other issues, the following extended modifier values were added:

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

When writing new code, it is recommended that you use the new, extended modifiers rather than the original modifiers.

To test if a modifier was pressed at the time an event is generated, use the **isAltDown( )**, **isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )**, and **isShiftDown( )** methods. The forms of these methods are shown here:

```
boolean isAltDown( )
boolean isAltGraphDown( )
boolean isControlDown( )
boolean isMetaDown( )
boolean isShiftDown( )
```

You can obtain a value that contains all of the original modifier flags by calling the **getModifiers( )** method. It is shown here:

```
int getModifiers( )
```

You can obtain the extended modifiers by calling **getModifiersEx( )**, which is shown here:

```
int getModifiersEx( )
```

The ItemEvent Class

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected. (Check boxes and list boxes are described later in this book.) There are two types of item events, which are identified by the following integer constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

In addition, **ItemEvent** defines one integer constant, **ITEM_STATE_CHANGED**, that signifies a change of state.

ItemEvent has this constructor:

```
ItemEvent(ItemSelectable src, int type, Object entry, int state)
```

Here, *src* is a reference to the component that generated this event. For example, this might be a list or choice element. The type of the event is specified by *type.* The specific item that generated the item event is passed in *entry.* The current state of that item is in *state.*

The **getItem( )** method can be used to obtain a reference to the item that generated an event. Its signature is shown here:

```
Object getItem( )
```

The **getItemSelectable( )** method can be used to obtain a reference to the **ItemSelectable** object that generated an event. Its general form is shown here:

```
ItemSelectable getItemSelectable( )
```

Lists and choices are examples of user interface elements that implement the **ItemSelectable** interface.

The **getStateChange( )** method returns the state change (that is, **SELECTED** or **DESELECTED**) for the event. It is shown here:

```
int getStateChange( )
```


The KeyEvent Class

A **KeyEvent** is generated when keyboard input occurs. There are three types of key events, which are identified by these integer constants: **KEY_PRESSED**, **KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing SHIFT does not generate a character.

There are many other integer constants that are defined by **KeyEvent**. For example, **VK_0** through **VK_9** and **VK_A** through **VK_Z** define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such as control, shift, or alt.

KeyEvent is a subclass of **InputEvent**. Here is one of its constructors:

```
KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)
```

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the key was pressed is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in *code*. The character equivalent (if one exists) is passed in *ch*. If no valid character exists, then *ch* contains **CHAR_UNDEFINED**. For **KEY_TYPED** events, *code* will contain **VK_UNDEFINED**.

The **KeyEvent** class defines several methods, but the most commonly used ones are **getKeyChar( )**, which returns the character that was entered, and **getKeyCode( )**, which returns the key code. Their general forms are shown here:

```
char getKeyChar( )
int getKeyCode( )
```

If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

The MouseEvent Class

There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

MouseEvent is a subclass of **InputEvent**. Here is one of its constructors:

MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*)

Here, *src* is a reference to the component that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in *x* and *y*. The click count is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.

Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

int getX( )
int getY( )

Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse. It is shown here:

Point getPoint( )

It returns a **Point** object that contains the X,Y coordinates in its integer members: **x** and **y**.

The **translatePoint( )** method changes the location of the event. Its form is shown here:

void translatePoint(int *x*, int *y*)

Here, the arguments *x* and *y* are added to the coordinates of the event.

The **getClickCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:

int getClickCount( )

The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:

boolean isPopupTrigger( )

Also available is the **getButton( )** method, shown here:

int getButton( )

It returns a value that represents the button that caused the event. The return value will be one of these constants defined by **MouseEvent**:

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

The **NOBUTTON** value indicates that no button was pressed or released.

Java SE 6 added three methods to **MouseEvent** that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:

Point getLocationOnScreen( )

int getXOnScreen( )

int getYOnScreen( )

The **getLocationOnScreen( )** method returns a **Point** object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

The MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent**. Not all mice have wheels. If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

Here is one of the constructors defined by **MouseWheelEvent**:

MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*,
int *x*, int *y*, int *clicks*, boolean *triggersPopup*,
int *scrollHow*, int *amount*, int *count*)

Here, *src* is a reference to the object that generated the event. The type of the event is specified by *type*. The system time at which the mouse event occurred is passed in *when*. The *modifiers* argument indicates which modifiers were pressed when the event occurred. The coordinates of the mouse are passed in *x* and *y*. The number of clicks the wheel has rotated is passed in *clicks*. The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform. The *scrollHow* value must be either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**. The number of units to scroll is passed in *amount*. The *count* parameter indicates the number of rotational units that the wheel moved.

MouseWheelEvent defines methods that give you access to the wheel event. To obtain the number of rotational units, call **getWheelRotation( )**, shown here:

```
int getWheelRotation( )
```

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise.

To obtain the type of scroll, call **getScrollType( )**, shown next:

```
int getScrollType( )
```

It returns either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**.

If the scroll type is **WHEEL_UNIT_SCROLL**, you can obtain the number of units to scroll by calling **getScrollAmount( )**. It is shown here:

```
int getScrollAmount( )
```

The TextEvent Class

Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program. **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED**.

The one constructor for this class is shown here:

```
TextEvent(Object src, int type)
```

Here, *src* is a reference to the object that generated this event. The type of the event is specified by *type*.

The **TextEvent** object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. For this reason, no methods are discussed here for the **TextEvent** class. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

The WindowEvent Class

There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_GAINED_FOCUS	The window gained input focus.
WINDOW_ICONIFIED	The window was iconified.
WINDOW_LOST_FOCUS	The window lost input focus.
WINDOW_OPENED	The window was opened.
WINDOW_STATE_CHANGED	The state of the window changed.

WindowEvent is a subclass of **ComponentEvent**. It defines several constructors. The first is

WindowEvent(Window *src*, int *type*)

Here, *src* is a reference to the object that generated this event. The type of the event is *type*.

The next three constructors offer more detailed control:

WindowEvent(Window *src*, int *type*, Window *other*)
WindowEvent(Window *src*, int *type*, int *fromState*, int *toState*)
WindowEvent(Window *src*, int *type*, Window *other*, int *fromState*, int *toState*)

Here, *other* specifies the opposite window when a focus or activation event occurs. The *fromState* specifies the prior state of the window, and *toState* specifies the new state that the window will have when a window state change occurs.

A commonly used method in this class is **getWindow( )**. It returns the **Window** object that generated the event. Its general form is shown here:

Window getWindow( )

WindowEvent also defines methods that return the opposite window (when a focus or activation event has occurred), the previous window state, and the current window state. These methods are shown here:

Window getOppositeWindow( )
int getOldState( )
int getNewState( )

Sources of Events

Table 22-2 lists some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, any class derived

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-2 Event Source Examples

from **Component**, such as **Applet**, can generate events. For example, you can receive key and mouse events from an applet. (You may also build your own components that generate events.) In this chapter, we will be handling only mouse and keyboard events, but the following two chapters will be handling events from the sources shown in Table 22-2.

Event Listener Interfaces

As explained, the delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Table 22-3 lists commonly used listener interfaces and provides a brief description of the methods that they define. The following sections examine the specific methods that are contained in each interface.

The ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent *ae*)

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

TABLE 22-3 Commonly Used Event Listener Interfaces

The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent *ce*)
void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)
void focusLost(FocusEvent *fe*)

The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered.

For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface

This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

The MouseWheelListener Interface

This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here:

void mouseWheelMoved(MouseWheelEvent *mwe*)

The TextListener Interface

This interface defines the **textChanged( )** method that is invoked when a change occurs in a text area or text field. Its general form is shown here:

void textChanged(TextEvent *te*)

The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here:

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)

The WindowListener Interface

This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified, the **windowDeiconified( )** method is called. When a window is opened or closed, the **windowOpened( )** or **windowClosed( )** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

void windowActivated(WindowEvent *we*)
void windowClosed(WindowEvent *we*)
void windowClosing(WindowEvent *we*)
void windowDeactivated(WindowEvent *we*)
void windowDeiconified(WindowEvent *we*)
void windowIconified(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

Using the Delegation Event Model

Now that you have learned the theory behind the delegation event model and have had an overview of its various components, it is time to see it in practice. Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it will receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

To see how the delegation model works in practice, we will look at examples that handle two commonly used event generators: the mouse and keyboard.

Handling Mouse Events

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces. (You may also want to implement **MouseWheelListener**, but we won't be doing so, here.) The following applet demonstrates the process. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upper-left corner of the applet display area.

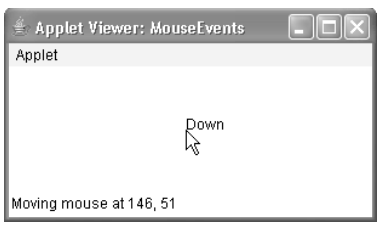
As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint( )** to display output at the point of these occurrences.

```
// Demonstrate the mouse event handlers.
import java.awt.*;
```

```
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="MouseEvents" width=300 height=100>
  </applet>
*/

public class MouseEvents extends Applet
  implements MouseListener, MouseMotionListener {

  String msg = "";
  int mouseX = 0, mouseY = 0; // coordinates of mouse

  public void init() {
     addMouseListener(this);
     addMouseMotionListener(this);
  }

  // Handle mouse clicked.
  public void mouseClicked(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse clicked.";
    repaint();
  }

  // Handle mouse entered.
  public void mouseEntered(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse entered.";
    repaint();
  }

  // Handle mouse exited.
  public void mouseExited(MouseEvent me) {
    // save coordinates
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse exited.";
    repaint();
  }

  // Handle button pressed.
  public void mousePressed(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
  }
```

```
  // Handle button released.
  public void mouseReleased(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "*";
    showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
    repaint();
  }

  // Handle mouse moved.
  public void mouseMoved(MouseEvent me) {
    // show status
    showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
  }

  // Display msg in applet window at current X,Y location.
  public void paint(Graphics g) {
    g.drawString(msg, mouseX, mouseY);
  }
}
```

Sample output from this program is shown here:



Let's look closely at this example. The **MouseEvents** class extends **Applet** and implements both the **MouseListener** and **MouseMotionListener** interfaces. These two interfaces contain methods that receive and process the various types of mouse events. Notice that the applet is both the source and the listener for these events. This works because **Component**, which supplies the **addMouseListener( )** and **addMouseMotionListener( )** methods, is a superclass of **Applet**. Being both the source and the listener for events is a common situation for applets.

Inside **init( )**, the applet registers itself as a listener for mouse events. This is done by using **addMouseListener( )** and **addMouseMotionListener( )**, which, as mentioned, are members of **Component**. They are shown here:

void addMouseListener(MouseListener *ml*)
void addMouseMotionListener(MouseMotionListener *mml*)

Here, *ml* is a reference to the object receiving mouse events, and *mml* is a reference to the object receiving mouse motion events. In this program, the same object is used for both.

The applet then implements all of the methods defined by the **MouseListener** and **MouseMotionListener** interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

Handling Keyboard Events

To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListener** interface.

Before looking at an example, it is useful to review how key events are generated. When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped( )** handler is invoked. Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the keypress and release events. However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed( )** handler.

The following program demonstrates keyboard input. It echoes keystrokes to the applet window and shows the pressed/released status of each key in the status window.

```
// Demonstrate the key event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="SimpleKey" width=300 height=100>
  </applet>
*/

public class SimpleKey extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, Y = 20; // output coordinates

  public void init() {
    addKeyListener(this);
  }

  public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");
  }

  public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
  }

  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
```

```
    repaint();
  }

  // Display keystrokes.
  public void paint(Graphics g) {
    g.drawString(msg, X, Y);
  }
}
```

Sample output is shown here:

If you want to handle the special keys, such as the arrow or function keys, you need to respond to them within the **keyPressed( )** handler. They are not available through **keyTyped( )**. To identify the keys, you use their virtual key codes. For example, the next applet outputs the name of a few of the special keys:

```
// Demonstrate some virtual key codes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
  <applet code="KeyEvents" width=300 height=100>
  </applet>
*/

public class KeyEvents extends Applet
  implements KeyListener {

  String msg = "";
  int X = 10, Y = 20; // output coordinates

  public void init() {
    addKeyListener(this);
  }

  public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");

    int key = ke.getKeyCode();
    switch(key) {
      case KeyEvent.VK_F1:
        msg += "<F1>";
        break;
      case KeyEvent.VK_F2:
```

```
        msg += "<F2>";
        break;
      case KeyEvent.VK_F3:
        msg += "<F3>";
        break;
      case KeyEvent.VK_PAGE_DOWN:
        msg += "<PgDn>";
        break;
      case KeyEvent.VK_PAGE_UP:
        msg += "<PgUp>";
        break;
      case KeyEvent.VK_LEFT:
        msg += "<Left Arrow>";
        break;
      case KeyEvent.VK_RIGHT:
        msg += "<Right Arrow>";
        break;
    }

    repaint();
  }

  public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
  }

  public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
  }

  // Display keystrokes.
  public void paint(Graphics g) {
    g.drawString(msg, X, Y);
  }
}
```

Sample output is shown here:

The procedures shown in the preceding keyboard and mouse event examples can be generalized to any type of event handling, including those events generated by controls. In later chapters, you will see many examples that handle other types of events, but they will all follow the same basic structure as the programs just described.

TABLE 22-4 Commonly Used Listener Interfaces Implemented by Adapter Classes

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Adapter Classes

Java provides a special feature, called an *adapter class*, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.

Table 22-4 lists the commonly used adapter classes in **java.awt.event** and notes the interface that each implements.

The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored. The program has three classes. **AdapterDemo** extends **Applet**. Its **init( )** method creates an instance of **MyMouseAdapter** and registers that object to receive notifications of mouse events. It also creates an instance of **MyMouseMotionAdapter** and registers that object to receive notifications of mouse motion events. Both of the constructors take a reference to the applet as an argument.

MyMouseAdapter extends **MouseAdapter** and overrides the **mouseClicked( )** method. The other mouse events are silently ignored by code inherited from the **MouseAdapter** class. **MyMouseMotionAdapter** extends **MouseMotionAdapter** and overrides the **mouseDragged( )** method. The other mouse motion event is silently ignored by code inherited from the **MouseMotionAdapter** class.

Note that both of the event listener classes save a reference to the applet. This information is provided as an argument to their constructors and is used later to invoke the **showStatus( )** method.

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
```

```
  <applet code="AdapterDemo" width=300 height=100>
  </applet>
*/

public class AdapterDemo extends Applet {
  public void init() {
     addMouseListener(new MyMouseAdapter(this));
     addMouseMotionListener(new MyMouseMotionAdapter(this));
  }
}

class MyMouseAdapter extends MouseAdapter {

  AdapterDemo adapterDemo;
  public MyMouseAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }

  // Handle mouse clicked.
  public void mouseClicked(MouseEvent me) {
    adapterDemo.showStatus("Mouse clicked");
  }
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
  AdapterDemo adapterDemo;
  public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
    this.adapterDemo = adapterDemo;
  }

  // Handle mouse dragged.
  public void mouseDragged(MouseEvent me) {
    adapterDemo.showStatus("Mouse dragged");
  }
}
```

As you can see by looking at the program, not having to implement all of the methods defined by the **MouseMotionListener** and **MouseListener** interfaces saves you a considerable amount of effort and prevents your code from becoming cluttered with empty methods. As an exercise, you might want to try rewriting one of the keyboard input examples shown earlier so that it uses a **KeyAdapter**.

Inner Classes

In Chapter 7, the basics of inner classes were explained. Here you will see why they are important. Recall that an *inner class* is a class defined within another class, or even within an expression. This section illustrates how inner classes can be used to simplify the code when using event adapter classes.

To understand the benefit provided by inner classes, consider the applet shown in the following listing. It *does not* use an inner class. Its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed. There are two

top-level classes in this program. **MousePressedDemo** extends **Applet**, and **MyMouseAdapter** extends **MouseAdapter**. The **init( )** method of **MousePressedDemo** instantiates **MyMouseAdapter** and provides this object as an argument to the **addMouseListener( )** method.

Notice that a reference to the applet is supplied as an argument to the **MyMouseAdapter** constructor. This reference is stored in an instance variable for later use by the **mousePressed( )** method. When the mouse is pressed, it invokes the **showStatus( )** method of the applet through the stored applet reference. In other words, **showStatus( )** is invoked relative to the applet reference stored by **MyMouseAdapter**.


```
// This applet does NOT use an inner class.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="MousePressedDemo" width=200 height=100>
  </applet>
*/

public class MousePressedDemo extends Applet {
  public void init() {
    addMouseListener(new MyMouseAdapter(this));
  }
}

class MyMouseAdapter extends MouseAdapter {
  MousePressedDemo mousePressedDemo;
  public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
    this.mousePressedDemo = mousePressedDemo;
  }
  public void mousePressed(MouseEvent me) {
    mousePressedDemo.showStatus("Mouse Pressed.");
  }
}
```

The following listing shows how the preceding program can be improved by using an inner class. Here, **InnerClassDemo** is a top-level class that extends **Applet**. **MyMouseAdapter** is an inner class that extends **MouseAdapter**. Because **MyMouseAdapter** is defined within the scope of **InnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, the **mousePressed( )** method can call the **showStatus( )** method directly. It no longer needs to do this via a stored reference to the applet. Thus, it is no longer necessary to pass **MyMouseAdapter( )** a reference to the invoking object.

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="InnerClassDemo" width=200 height=100>
  </applet>
*/

public class InnerClassDemo extends Applet {
```

```
  public void init() {
    addMouseListener(new MyMouseAdapter());
  }
  class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent me) {
      showStatus("Mouse Pressed");
    }
  }
}
```

Anonymous Inner Classes

An *anonymous* inner class is one that is not assigned a name. This section illustrates how an anonymous inner class can facilitate the writing of event handlers. Consider the applet shown in the following listing. As before, its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.

```
// Anonymous inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
  <applet code="AnonymousInnerClassDemo" width=200 height=100>
  </applet>
*/

public class AnonymousInnerClassDemo extends Applet {
  public void init() {
    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent me) {
        showStatus("Mouse Pressed");
      }
    });
  }
}
```

There is one top-level class in this program: **AnonymousInnerClassDemo**. The **init( )** method calls the **addMouseListener( )** method. Its argument is an expression that defines and instantiates an anonymous inner class. Let's analyze this expression carefully.

The syntax **new MouseAdapter( ) { ... }** indicates to the compiler that the code between the braces defines an anonymous inner class. Furthermore, that class extends **MouseAdapter**. This new class is not named, but it is automatically instantiated when this expression is executed.

Because this anonymous inner class is defined within the scope of **AnonymousInnerClassDemo**, it has access to all of the variables and methods within the scope of that class. Therefore, it can call the **showStatus( )** method directly.

As just illustrated, both named and anonymous inner classes solve some annoying problems in a simple yet effective way. They also allow you to create more efficient code.

CHAPTER 29

Introducing Swing

In Part II, you saw how to build user interfaces with the AWT classes. Although the AWT is still a crucial part of Java, its component set is no longer widely used to create graphic user interfaces. Today, most programmers use Swing for this purpose. Swing is a set of classes that provides more powerful and flexible GUI components than does the AWT. Simply put, Swing provides the look and feel of the modern Java GUI.

Coverage of Swing is divided between two chapters. This chapter introduces Swing. It begins by describing Swing's core concepts. It then shows the general form of a Swing program, including both applications and applets. It concludes by explaining how painting is accomplished in Swing. The following chapter presents several commonly used Swing components. It is important to understand that the number of classes and interfaces in the Swing packages is quite large, and they can't all be covered in this book. (In fact, full coverage of Swing requires an entire book of its own.) However, these two chapters will give you a basic understanding of this important topic.

NOTE *For a comprehensive introduction to Swing, see my book* Swing: A Beginner's Guide *published by McGraw-Hill/Osborne (2007).*

The Origins of Swing

Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or *peers*. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as *heavyweight*.

The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently on different platforms. This potential variability threatened the overarching philosophy of Java: write once, run anywhere. Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed. Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component is always rectangular and opaque.

Not long after Java's original release, it became apparent that the limitations and restrictions present in the AWT were sufficiently serious that a better approach was needed. The solution was Swing. Introduced in 1997, Swing was included as part of the Java Foundation Classes (JFC). Swing was initially available for use with Java 1.1 as a separate library. However, beginning with Java 1.2, Swing (and the rest of the JFC) was fully integrated into Java.

Swing Is Built on the AWT

Before moving on, it is necessary to make one important point: although Swing eliminates a number of the limitations inherent in the AWT, Swing *does not* replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing. (The AWT is covered in Chapters 23 and 24. Event handling is described in Chapter 22.)

Two Key Swing Features

As just explained, Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing. Each is examined here.

Swing Components Are Lightweight

With very few exceptions, Swing components are *lightweight*. This means that they are written entirely in Java and do not map directly to platform-specific peers. Because lightweight components are rendered using graphics primitives, they can be transparent, which enables nonrectangular shapes. Thus, lightweight components are more efficient and more flexible. Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. This means that each component will work in a consistent manner across all platforms.

Swing Supports a Pluggable Look and Feel

Swing supports a *pluggable look and feel* (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply "plugged in." Once this is done, all components are automatically rendered using that style.

Pluggable look-and-feels offer several important advantages. It is possible to define a look and feel that is consistent across all platforms. Conversely, it is possible to create a look

and feel that acts like a specific platform. For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel. It is also possible to design a custom look and feel. Finally, the look and feel can be changed dynamically at run time.

Java SE 6 provides look-and-feels, such as metal and Motif, that are available to all Swing users. The metal look and feel is also called the *Java look and feel*. It is platform-independent and available in all Java execution environments. It is also the default look and feel. Windows environments also have access to the Windows and Windows Classic look and feel. This book uses the default Java look and feel (metal) because it is platform independent.

The MVC Connection

In general, a visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

No matter what architecture is used to implement a component, it must implicitly contain these three parts. Over the years, one component architecture has proven itself to be exceptionally effective: *Model-View-Controller*, or MVC for short.

The MVC architecture is successful because each piece of the design corresponds to an aspect of a component. In MVC terminology, the *model* corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The *view* determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The *controller* determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the *UI delegate*. For this reason, Swing's approach is called either the *Model-Delegate* architecture or the *Separable Model* architecture. Therefore, although Swing's component architecture is based on MVC, it does not use a classical implementation of it.

Swing's pluggable look and feel is made possible by its Model-Delegate architecture. Because the view (look) and controller (feel) are separate from the model, the look and feel can be changed without affecting how the component is used within a program. Conversely, it is possible to customize the model without affecting the way that the component appears on the screen or responds to user input.

To support the Model-Delegate architecture, most Swing components contain two objects. The first represents the model. The second represents the UI delegate. Models are defined

by interfaces. For example, the model for a button is defined by the **ButtonModel** interface. UI delegates are classes that inherit **ComponentUI**. For example, the UI delegate for a button is **ButtonUI**. Normally, your programs will not interact directly with the UI delegate.

Components and Containers

A Swing GUI consists of two key items: *components* and *containers*. However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

Let's look a bit more closely at components and containers.

Components

In general, Swing components are derived from the **JComponent** class. (The only exceptions to this are the four top-level containers, described in the next section.) **JComponent** provides the functionality that is common to all components. For example, **JComponent** supports the pluggable look and feel. **JComponent** inherits the AWT classes **Container** and **Component**. Thus, a Swing component is built on and compatible with an AWT component.

All of Swing's components are represented by classes defined within the package **javax.swing**. The following table shows the class names for Swing components (including those used as containers).

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayeredPane
JList	JMenu	JMenuBar	JMenuItem
JOptionPane	JPanel	JPasswordField	JPopupMenu
JProgressBar	JRadioButton	JRadioButtonMenuItem	JRootPane
JScrollBar	JScrollPane	JSeparator	JSlider
JSpinner	JSplitPane	JTabbedPane	JTable
JTextArea	JTextField	JTextPane	JTogglebutton
JToolBar	JToolTip	JTree	JViewport
JWindow			

Notice that all component classes begin with the letter **J**. For example, the class for a label is **JLabel**; the class for a push button is **JButton**; and the class for a scroll bar is **JScrollBar**.

Containers

Swing defines two types of containers. The first are top-level containers: **JFrame**, **JApplet**, **JWindow**, and **JDialog**. These containers do not inherit **JComponent**. They do, however, inherit the AWT classes **Component** and **Container**. Unlike Swing's other components, which are lightweight, the top-level containers are heavyweight. This makes the top-level containers a special case in the Swing component library.

As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications is **JFrame**. The one used for applets is **JApplet**.

The second type of containers supported by Swing are lightweight containers. Lightweight containers *do* inherit **JComponent**. An example of a lightweight container is **JPanel**, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Thus, you can use lightweight containers such as **JPanel** to create subgroups of related controls that are contained within an outer container.

The Top-Level Container Panes

Each top-level container defines a set of *panes*. At the top of the hierarchy is an instance of **JRootPane**. **JRootPane** is a lightweight container whose purpose is to manage the other panes. It also helps manage the optional menu bar. The panes that comprise the root pane are called the *glass pane*, the *content pane*, and the *layered pane*.

The glass pane is the top-level pane. It sits above and completely covers all other panes. By default, it is a transparent instance of **JPanel**. The glass pane enables you to manage mouse events that affect the entire container (rather than an individual control) or to paint over any other component, for example. In most cases, you won't need to use the glass pane directly, but it is there if you need it.

The layered pane is an instance of **JLayeredPane**. The layered pane allows components to be given a depth value. This value determines which component overlays another. (Thus, the layered pane lets you specify a Z-order for a component, although this is not something that you will usually need to do.) The layered pane holds the content pane and the (optional) menu bar.

Although the glass pane and the layered panes are integral to the operation of a top-level container and serve important purposes, much of what they provide occurs behind the scene. The pane with which your application will interact the most is the content pane, because this is the pane to which you will add visual components. In other words, when you add a component, such as a button, to a top-level container, you will add it to the content pane. By default, the content pane is an opaque instance of **JPanel**.

The Swing Packages

Swing is a very large subsystem and makes use of many packages. These are the packages used by Swing that are defined by Java SE 6.

javax.swing	javax.swing.border	javax.swing.colorchooser
javax.swing.event	javax.swing.filechooser	javax.swing.plaf
javax.swing.plaf.basic	javax.swing.plaf.metal	javax.swing.plaf.multi
javax.swing.plaf.synth	javax.swing.table	javax.swing.text
javax.swing.text.html	javax.swing.text.html.parser	javax.swing.text.rtf
javax.swing.tree	javax.swing.undo	

The main package is **javax.swing**. This package must be imported into any program that uses Swing. It contains the classes that implement the basic Swing components, such as push buttons, labels, and check boxes.

A Simple Swing Application

Swing programs differ from both the console-based programs and the AWT-based programs shown earlier in this book. For example, they use a different set of components and a different container hierarchy than does the AWT. Swing programs also have special requirements that relate to threading. The best way to understand the structure of a Swing program is to work through an example. There are two types of Java programs in which Swing is typically used. The first is a desktop application. The second is the applet. This section shows how to create a Swing application. The creation of a Swing applet is described later in this chapter.

Although quite short, the following program shows one way to write a Swing application. In the process, it demonstrates several key features of Swing. It uses two Swing components: **JFrame** and **JLabel**. **JFrame** is the top-level container that is commonly used for Swing applications. **JLabel** is the Swing component that creates a label, which is a component that displays information. The label is Swing's simplest component because it is passive. That is, a label does not respond to user input. It just displays output. The program uses a **JFrame** container to hold an instance of a **JLabel**. The label displays a short text message.

```
// A simple Swing application.

import javax.swing.*;

class SwingDemo {

  SwingDemo() {

    // Create a new JFrame container.
    JFrame jfrm = new JFrame("A Simple Swing Application");

    // Give the frame an initial size.
    jfrm.setSize(275, 100);

    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create a text-based label.
```

```
    JLabel jlab = new JLabel(" Swing means powerful GUIs.");

    // Add the label to the content pane.
    jfrm.add(jlab);

    // Display the frame.
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new SwingDemo();
      }
    });
  }
}
```

Swing programs are compiled and run in the same way as other Java applications. Thus, to compile this program, you can use this command line:

```
javac SwingDemo.java
```

To run the program, use this command line:

```
java SwingDemo
```

When the program is run, it will produce the window shown in Figure 29-1.

Because the **SwingDemo** program illustrates several core Swing concepts, we will examine it carefully, line by line. The program begins by importing **javax.swing**. As mentioned, this package contains the components and models defined by Swing. For example, **javax.swing** defines classes that implement labels, buttons, text controls, and menus. It will be included in all programs that use Swing.

Next, the program declares the **SwingDemo** class and a constructor for that class. The constructor is where most of the action of the program occurs. It begins by creating a **JFrame**, using this line of code:

```
JFrame jfrm = new JFrame("A Simple Swing Application");
```

This creates a container called **jfrm** that defines a rectangular window complete with a title bar; close, minimize, maximize, and restore buttons; and a system menu. Thus, it creates a standard, top-level window. The title of the window is passed to the constructor.

FIGURE 29-1 The window produced by the **SwingDemo** program

Next, the window is sized using this statement:

```
jfrm.setSize(275, 100);
```

The **setSize( )** method (which is inherited by **JFrame** from the AWT class **Component**) sets the dimensions of the window, which are specified in pixels. Its general form is shown here:

void setSize(int *width*, int *height*)

In this example, the width of the window is set to 275 and the height is set to 100.

By default, when a top-level window is closed (such as when the user clicks the close box), the window is removed from the screen, but the application is not terminated. While this default behavior is useful in some situations, it is not what is needed for most applications. Instead, you will usually want the entire application to terminate when its top-level window is closed. There are a couple of ways to achieve this. The easiest way is to call **setDefaultCloseOperation( )**, as the program does:

```
jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

After this call executes, closing the window causes the entire application to terminate. The general form of **setDefaultCloseOperation( )** is shown here:

void setDefaultCloseOperation(int *what*)

The value passed in *what* determines what happens when the window is closed. There are several other options in addition to **JFrame.EXIT_ON_CLOSE**. They are shown here:

JFrame.DISPOSE_ON_CLOSE

JFrame.HIDE_ON_CLOSE

JFrame.DO_NOTHING_ON_CLOSE

Their names reflect their actions. These constants are declared in **WindowConstants**, which is an interface declared in **javax.swing** that is implemented by **JFrame**.

The next line of code creates a Swing **JLabel** component:

```
JLabel jlab = new JLabel(" Swing means powerful GUIs.");
```

JLabel is the simplest and easiest-to-use component because it does not accept user input. It simply displays information, which can consist of text, an icon, or a combination of the two. The label created by the program contains only text, which is passed to its constructor.

The next line of code adds the label to the content pane of the frame:

```
jfrm.add(jlab);
```

As explained earlier, all top-level containers have a content pane in which components are stored. Thus, to add a component to a frame, you must add it to the frame's content pane. This is accomplished by calling **add( )** on the **JFrame** reference (**jfrm** in this case). The general form of **add( )** is shown here:

Component add(Component *comp*)

The **add( )** method is inherited by **JFrame** from the AWT class **Container**.

By default, the content pane associated with a **JFrame** uses border layout. The version of **add( )** just shown adds the label to the center location. Other versions of **add( )** enable you to specify one of the border regions. When a component is added to the center, its size is adjusted automatically to fit the size of the center.

Before continuing, an important historical point needs to be made. Prior to JDK 5, when adding a component to the content pane, you could not invoke the **add( )** method directly on a **JFrame** instance. Instead, you needed to call **add( )** on the content pane of the **JFrame** object. The content pane can be obtained by calling **getContentPane( )** on a **JFrame** instance. The **getContentPane( )** method is shown here:

```
Container getContentPane( )
```

It returns a **Container** reference to the content pane. The **add( )** method was then called on that reference to add a component to a content pane. Thus, in the past, you had to use the following statement to add **jlab** to **jfrm**:

```
jfrm.getContentPane().add(jlab); // old-style
```

Here, **getContentPane( )** first obtains a reference to content pane, and then **add( )** adds the component to the container linked to this pane. This same procedure was also required to invoke **remove( )** to remove a component and **setLayout( )** to set the layout manager for the content pane. You will see explicit calls to **getContentPane( )** frequently throughout pre-5.0 code. Today, the use of **getContentPane( )** is no longer necessary. You can simply call **add( )**, **remove( )**, and **setLayout( )** directly on **JFrame** because these methods have been changed so that they operate on the content pane automatically.

The last statement in the **SwingDemo** constructor causes the window to become visible:

```
jfrm.setVisible(true);
```

The **setVisible( )** method is inherited from the AWT **Component** class. If its argument is **true**, the window will be displayed. Otherwise, it will be hidden. By default, a **JFrame** is invisible, so **setVisible(true)** must be called to show it.

Inside **main( )**, a **SwingDemo** object is created, which causes the window and the label to be displayed. Notice that the **SwingDemo** constructor is invoked using these lines of code:

```
SwingUtilities.invokeLater(new Runnable() {
  public void run() {
    new SwingDemo();
  }
});
```

This sequence causes a **SwingDemo** object to be created on the *event dispatching thread* rather than on the main thread of the application. Here's why. In general, Swing programs are event-driven. For example, when a user interacts with a component, an event is generated. An event is passed to the application by calling an event handler defined by the application. However, the handler is executed on the event dispatching thread provided by Swing and not on the main thread of the application. Thus, although event handlers are defined by your program, they are called on a thread that was not created by your program.

To avoid problems (including the potential for deadlock), all Swing GUI components must be created and updated from the event dispatching thread, not the main thread of the application. However, **main( )** is executed on the main thread. Thus, **main( )** cannot directly instantiate a **SwingDemo** object. Instead, it must create a **Runnable** object that executes on the event dispatching thread and have this object create the GUI.

To enable the GUI code to be created on the event dispatching thread, you must use one of two methods that are defined by the **SwingUtilities** class. These methods are **invokeLater( )** and **invokeAndWait( )**. They are shown here:

```
static void invokeLater(Runnable obj)

static void invokeAndWait(Runnable obj)
  throws InterruptedException, InvocationTargetException
```

Here, *obj* is a **Runnable** object that will have its **run( )** method called by the event dispatching thread. The difference between the two methods is that **invokeLater( )** returns immediately, but **invokeAndWait( )** waits until **obj.run( )** returns. You can use one of these methods to call a method that constructs the GUI for your Swing application, or whenever you need to modify the state of the GUI from code not executed by the event dispatching thread. You will normally want to use **invokeLater( )**, as the preceding program does. However, when constructing the initial GUI for an applet, you will need to use **invokeAndWait( )**.

Event Handling

The preceding example showed the basic form of a Swing program, but it left out one important part: event handling. Because **JLabel** does not take input from the user, it does not generate events, so no event handling was needed. However, the other Swing components *do* respond to user input and the events generated by those interactions need to be handled. Events can also be generated in ways not directly related to user input. For example, an event is generated when a timer goes off. Whatever the case, event handling is a large part of any Swing-based application.

The event handling mechanism used by Swing is the same as that used by the AWT. This approach is called the *delegation event model*, and it is described in Chapter 22. In many cases, Swing uses the same events as does the AWT, and these events are packaged in **java.awt.event**. Events specific to Swing are stored in **javax.swing.event**.

Although events are handled in Swing in the same way as they are with the AWT, it is still useful to work through a simple example. The following program handles the event generated by a Swing push button. Sample output is shown in Figure 29-2.

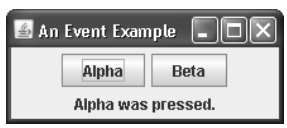


FIGURE 29-2 Output from the **EventDemo** program

```
// Handle an event in a Swing program.

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

class EventDemo {

  JLabel jlab;

  EventDemo() {

    // Create a new JFrame container.
    JFrame jfrm = new JFrame("An Event Example");

    // Specify FlowLayout for the layout manager.
    jfrm.setLayout(new FlowLayout());

    // Give the frame an initial size.
    jfrm.setSize(220, 90);

    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Make two buttons.
    JButton jbtnAlpha = new JButton("Alpha");
    JButton jbtnBeta = new JButton("Beta");

    // Add action listener for Alpha.
    jbtnAlpha.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Alpha was pressed.");
      }
    });

    // Add action listener for Beta.
    jbtnBeta.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        jlab.setText("Beta was pressed.");
      }
    });

    // Add the buttons to the content pane.
    jfrm.add(jbtnAlpha);
    jfrm.add(jbtnBeta);

    // Create a text-based label.
    jlab = new JLabel("Press a button.");

    // Add the label to the content pane.
    jfrm.add(jlab);

    // Display the frame.
```

```
    jfrm.setVisible(true);
  }

  public static void main(String args[]) {
    // Create the frame on the event dispatching thread.
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        new EventDemo();
      }
    });
  }
}
```

First, notice that the program now imports both the **java.awt** and **java.awt.event** packages. The **java.awt** package is needed because it contains the **FlowLayout** class, which supports the standard flow layout manager used to lay out components in a frame. (See Chapter 24 for coverage of layout managers.) The **java.awt.event** package is needed because it defines the **ActionListener** interface and the **ActionEvent** class.

The **EventDemo** constructor begins by creating a **JFrame** called **jfrm**. It then sets the layout manager for the content pane of **jfrm** to **FlowLayout**. Recall that, by default, the content pane uses **BorderLayout** as its layout manager. However, for this example, **FlowLayout** is more convenient. Notice that **FlowLayout** is assigned using this statement:

```
jfrm.setLayout(new FlowLayout());
```

As explained, in the past you had to explicitly call **getContentPane( )** to set the layout manager for the content pane. This requirement was removed as of JDK 5.

After setting the size and default close operation, **EventDemo( )** creates two push buttons, as shown here:

```
JButton jbtnAlpha = new JButton("Alpha");
JButton jbtnBeta = new JButton("Beta");
```

The first button will contain the text "Alpha" and the second will contain the text "Beta." Swing push buttons are instances of **JButton**. **JButton** supplies several constructors. The one used here is

JButton(String *msg*)

The *msg* parameter specifies the string that will be displayed inside the button.

When a push button is pressed, it generates an **ActionEvent**. Thus, **JButton** provides the **addActionListener( )** method, which is used to add an action listener. (**JButton** also provides **removeActionListener( )** to remove a listener, but this method is not used by the program.) As explained in Chapter 22, the **ActionListener** interface defines only one method: **actionPerformed( )**. It is shown again here for your convenience:

void actionPerformed(ActionEvent *ae*)

This method is called when a button is pressed. In other words, it is the event handler that is called when a button press event has occurred.

Next, event listeners for the button's action events are added by the code shown here:

```
// Add action listener for Alpha.
jbtnAlpha.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Alpha was pressed.");
  }
});

// Add action listener for Beta.
jbtnBeta.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("Beta was pressed.");
  }
});
```

Here, anonymous inner classes are used to provide the event handlers for the two buttons. Each time a button is pressed, the string displayed in **jlab** is changed to reflect which button was pressed.

Next, the buttons are added to the content pane of **jfrm**:

```
jfrm.add(jbtnAlpha);
jfrm.add(jbtnBeta);
```

Finally, **jlab** is added to the content pane and window is made visible. When you run the program, each time you press a button, a message is displayed in the label that indicates which button was pressed.

One last point: Remember that all event handlers, such as **actionPerformed( )**, are called on the event dispatching thread. Therefore, an event handler must return quickly in order to avoid slowing down the application. If your application needs to do something time consuming as the result of an event, it must use a separate thread.

Create a Swing Applet

The second type of program that commonly uses Swing is the applet. Swing-based applets are similar to AWT-based applets, but with an important difference: A Swing applet extends **JApplet** rather than **Applet**. **JApplet** is derived from **Applet**. Thus, **JApplet** includes all of the functionality found in **Applet** and adds support for Swing. **JApplet** is a top-level Swing container, which means that it is *not* derived from **JComponent**. Because **JApplet** is a top-level container, it includes the various panes described earlier. This means that all components are added to **JApplet**'s content pane in the same way that components are added to **JFrame**'s content pane.

Swing applets use the same four lifecycle methods as described in Chapter 21: **init( )**, **start( )**, **stop( )**, and **destroy( )**. Of course, you need override only those methods that are needed by your applet. Painting is accomplished differently in Swing than it is in the AWT, and a Swing applet will not normally override the **paint( )** method. (Painting in Swing is described later in this chapter.)

One other point: All interaction with components in a Swing applet must take place on the event dispatching thread, as described in the previous section. This threading issue applies to all Swing programs.

FIGURE 29-3 Output from the example Swing applet

Here is an example of a Swing applet. It provides the same functionality as the previous application, but does so in applet form. Figure 29-3 shows the program when executed by **appletviewer**.

```
// A simple Swing-based applet

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

/*
This HTML can be used to launch the applet:

<object code="MySwingApplet" width=220 height=90>
</object>
*/

public class MySwingApplet extends JApplet {
  JButton jbtnAlpha;
  JButton jbtnBeta;

  JLabel jlab;

  // Initialize the applet.
  public void init() {
    try {
      SwingUtilities.invokeAndWait(new Runnable () {
        public void run() {
          makeGUI(); // initialize the GUI
        }
      });
    } catch(Exception exc) {
      System.out.println("Can't create because of "+ exc);
    }
  }

  // This applet does not need to override start(), stop(),
  // or destroy().

  // Set up and initialize the GUI.
  private void makeGUI() {
```

```
    // Set the applet to use flow layout.
    setLayout(new FlowLayout());

    // Make two buttons.
    jbtnAlpha = new JButton("Alpha");
    jbtnBeta = new JButton("Beta");

    // Add action listener for Alpha.
    jbtnAlpha.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent le) {
        jlab.setText("Alpha was pressed.");
      }
    });

    // Add action listener for Beta.
    jbtnBeta.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent le) {
        jlab.setText("Beta was pressed.");
      }
    });

    // Add the buttons to the content pane.
    add(jbtnAlpha);
    add(jbtnBeta);

    // Create a text-based label.
    jlab = new JLabel("Press a button.");

    // Add the label to the content pane.
    add(jlab);
  }
}
```

There are two important things to notice about this applet. First, **MySwingApplet** extends **JApplet**. As explained, all Swing-based applets extend **JApplet** rather than **Applet**. Second, the **init( )** method initializes the Swing components on the event dispatching thread by setting up a call to **makeGUI( )**. Notice that this is accomplished through the use of **invokeAndWait( )** rather than **invokeLater( )**. Applets must use **invokeAndWait( )** because the **init( )** method must not return until the entire initialization process has been completed. In essence, the **start( )** method cannot be called until after initialization, which means that the GUI must be fully constructed.

Inside **makeGUI( )**, the two buttons and label are created, and the action listeners are added to the buttons. Finally, the components are added to the content pane. Although this example is quite simple, this same general approach must be used when building any Swing GUI that will be used by an applet.

Painting in Swing

Although the Swing component set is quite powerful, you are not limited to using it because Swing also lets you write directly into the display area of a frame, panel, or one of Swing's other components, such as **JLabel**. Although many (perhaps most) uses of Swing will *not* involve drawing directly to the surface of a component, it is available for those

applications that need this capability. To write output directly to the surface of a component, you will use one or more drawing methods defined by the AWT, such as **drawLine( )** or **drawRect( )**. Thus, most of the techniques and methods described in Chapter 23 also apply to Swing. However, there are also some very important differences, and the process is discussed in detail in this section.

Painting Fundamentals

Swing's approach to painting is built on the original AWT-based mechanism, but Swing's implementation offers more finally grained control. Before examining the specifics of Swing-based painting, it is useful to review the AWT-based mechanism that underlies it.

The AWT class **Component** defines a method called **paint( )** that is used to draw output directly to the surface of a component. For the most part, **paint( )** is not called by your program. (In fact, only in the most unusual cases should it ever be called by your program.) Rather, **paint( )** is called by the run-time system whenever a component must be rendered. This situation can occur for several reasons. For example, the window in which the component is displayed can be overwritten by another window and then uncovered. Or, the window might be minimized and then restored. The **paint( )** method is also called when a program begins running. When writing AWT-based code, an application will override **paint( )** when it needs to write output directly to the surface of the component.

Because **JComponent** inherits **Component**, all Swing's lightweight components inherit the **paint( )** method. However, you *will not* override it to paint directly to the surface of a component. The reason is that Swing uses a bit more sophisticated approach to painting that involves three distinct methods: **paintComponent( )**, **paintBorder( )**, and **paintChildren( )**. These methods paint the indicated portion of a component and divide the painting process into its three distinct, logical actions. In a lightweight component, the original AWT method **paint( )** simply executes calls to these methods, in the order just shown.

To paint to the surface of a Swing component, you will create a subclass of the component and then override its **paintComponent( )** method. This is the method that paints the interior of the component. You will not normally override the other two painting methods. When overriding **paintComponent( )**, the first thing you must do is call **super.paintComponent( )**, so that the superclass portion of the painting process takes place. (The only time this is not required is when you are taking complete, manual control over how a component is displayed.) After that, write the output that you want to display. The **paintComponent( )** method is shown here:

```
protected void paintComponent(Graphics g)
```

The parameter *g* is the graphics context to which output is written.

To cause a component to be painted under program control, call **repaint( )**. It works in Swing just as it does for the AWT. The **repaint( )** method is defined by **Component**. Calling it causes the system to call **paint( )** as soon as it is possible to do so. Because painting is a time-consuming operation, this mechanism allows the run-time system to defer painting momentarily until some higher-priority task has completed, for example. Of course, in Swing the call to **paint( )** results in a call to **paintComponent( )**. Therefore, to output to the surface of a component, your program will store the output until **paintComponent( )** is called. Inside the overridden **paintComponent( )**, you will draw the stored output.

Compute the Paintable Area

When drawing to the surface of a component, you must be careful to restrict your output to the area that is inside the border. Although Swing automatically clips any output that will exceed the boundaries of a component, it is still possible to paint into the border, which will then get overwritten when the border is drawn. To avoid this, you must compute the *paintable area* of the component. This is the area defined by the current size of the component minus the space used by the border. Therefore, before you paint to a component, you must obtain the width of the border and then adjust your drawing accordingly.

To obtain the border width, call **getInsets( )**, shown here:

```
Insets getInsets( )
```

This method is defined by **Container** and overridden by **JComponent**. It returns an **Insets** object that contains the dimensions of the border. The inset values can be obtained by using these fields:

```
int top;
int bottom;
int left;
int right;
```

These values are then used to compute the drawing area given the width and the height of the component. You can obtain the width and height of the component by calling **getWidth( )** and **getHeight( )** on the component. They are shown here:

```
int getWidth( )
int getHeight( )
```

By subtracting the value of the insets, you can compute the usable width and height of the component.

A Paint Example

Here is a program that puts into action the preceding discussion. It creates a class called **PaintPanel** that extends **JPanel**. The program then uses an object of that class to display lines whose endpoints have been generated randomly. Sample output is shown in Figure 10-4.

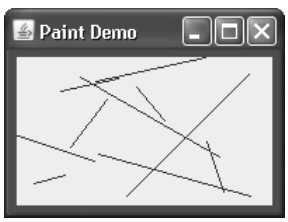


FIGURE 29-4 Sample output from the **PaintPanel** program

```
// Paint lines to a panel.

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

// This class extends JPanel. It overrides
// the paintComponent() method so that random
// lines are plotted in the panel.
class PaintPanel extends JPanel {
  Insets ins; // holds the panel's insets

  Random rand; // used to generate random numbers

  // Construct a panel.
  PaintPanel() {

    // Put a border around the panel.
    setBorder(
      BorderFactory.createLineBorder(Color.RED, 5));

    rand = new Random();
  }

  // Override the paintComponent() method.
  protected void paintComponent(Graphics g) {
    // Always call the superclass method first.
    super.paintComponent(g);

    int x, y, x2, y2;

    // Get the height and width of the component.
    int height = getHeight();
    int width = getWidth();

    // Get the insets.
    ins = getInsets();

    // Draw ten lines whose endpoints are randomly generated.
    for(int i=0; i < 10; i++) {
      // Obtain random coordinates that define
      // the endpoints of each line.
      x = rand.nextInt(width-ins.left);
      y = rand.nextInt(height-ins.bottom);
      x2 = rand.nextInt(width-ins.left);
      y2 = rand.nextInt(height-ins.bottom);

      // Draw the line.
      g.drawLine(x, y, x2, y2);
    }
  }
}
```

```
// Demonstrate painting directly onto a panel.
class PaintDemo {

  JLabel jlab;
  PaintPanel pp;

  PaintDemo() {

    // Create a new JFrame container.
    JFrame jfrm = new JFrame("Paint Demo");

    // Give the frame an initial size.
    jfrm.setSize(200, 150);

    // Terminate the program when the user closes the application.
    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Create the panel that will be painted.
    pp = new PaintPanel();

    // Add the panel to the content pane. Because the default
    // border layout is used, the panel will automatically be
    // sized to fit the center region.
    jfrm.add(pp);

    // Display the frame.
    jfrm.setVisible(true);
  }

   public static void main(String args[]) {
     // Create the frame on the event dispatching thread.
     SwingUtilities.invokeLater(new Runnable() {
       public void run() {
         new PaintDemo();
       }
     });
   }
}
```

Let's examine this program closely. The **PaintPanel** class extends **JPanel**. **JPanel** is one of Swing's lightweight containers, which means that it is a component that can be added to the content pane of a **JFrame**. To handle painting, **PaintPanel** overrides the **paintComponent( )** method. This enables **PaintPanel** to write directly to the surface of the component when painting takes place. The size of the panel is not specified because the program uses the default border layout and the panel is added to the center. This results in the panel being sized to fill the center. If you change the size of the window, the size of the panel will be adjusted accordingly.

Notice that the constructor also specifies a 5-pixel wide, red border. This is accomplished by setting the border by using the **setBorder( )** method, shown here:

void setBorder(Border *border*)

Border is the Swing interface that encapsulates a border. You can obtain a border by calling one of the factory methods defined by the **BorderFactory** class. The one used in the program is **createLineBorder( )**, which creates a simple line border. It is shown here:

static Border createLineBorder(Color *clr*, int *width*)

Here, *clr* specifies the color of the border and *width* specifies its width in pixels.

Inside the override of **paintComponent( )**, notice that it first calls **super.paintComponent( )**. As explained, this is necessary to ensure that the component is properly drawn. Next the width and height of the panel are obtained along with the insets. These values are used to ensure the lines lie within the drawing area of the panel. The drawing area is the overall width and height of a component less the border width. The computations are designed to work with differently sized **PaintPanel**s and borders. To prove this, try changing the size of the window. The lines will still all lie within the borders of the panel.

The **PaintDemo** class creates a **PaintPanel** and then adds the panel to the content pane. When the application is first displayed, the overridden **paintComponent( )** method is called, and the lines are drawn. Each time you resize or hide and restore the window, a new set of lines are drawn. In all cases, the lines fall within the paintable area.

30 CHAPTER

Exploring Swing

The previous chapter described several of the core concepts relating to Swing and showed the general form of both a Swing application and a Swing applet. This chapter continues the discussion of Swing by presenting an overview of several Swing components, such as buttons, check boxes, trees, and tables. The Swing components provide rich functionality and allow a high level of customization. Because of space limitations, it is not possible to describe all of their features and attributes. Rather, the purpose of this overview is to give you a feel for the capabilities of the Swing component set.

The Swing component classes described in this chapter are shown here:

JButton	JCheckBox	JComboBox	JLabel
JList	JRadioButton	JScrollPane	JTabbedPane
JTable	JTextField	JToggleButton	JTree

These components are all lightweight, which means that they are all derived from **JComponent**.

Also discussed is the **ButtonGroup** class, which encapsulates a mutually exclusive set of Swing buttons, and **ImageIcon**, which encapsulates a graphics image. Both are defined by Swing and packaged in **javax.swing**.

One other point: The Swing components are demonstrated in applets because the code for an applet is more compact than it is for a desktop application. However, the same techniques apply to both applets and applications.

JLabel and ImageIcon

JLabel is Swing's easiest-to-use component. It creates a label and was introduced in the preceding chapter. Here, we will look at **JLabel** a bit more closely. **JLabel** can be used to display text and/or an icon. It is a passive component in that it does not respond to user input. **JLabel** defines several constructors. Here are three of them:

JLabel(Icon *icon*)
JLabel(String *str*)
JLabel(String *str*, Icon *icon*, int *align*)

Here, *str* and *icon* are the text and icon used for the label. The *align* argument specifies the horizontal alignment of the text and/or icon within the dimensions of the label. It must be one of the following values: **LEFT**, **RIGHT**, **CENTER**, **LEADING**, or **TRAILING**. These constants are defined in the **SwingConstants** interface, along with several others used by the Swing classes.

Notice that icons are specified by objects of type **Icon**, which is an interface defined by Swing. The easiest way to obtain an icon is to use the **ImageIcon** class. **ImageIcon** implements **Icon** and encapsulates an image. Thus, an object of type **ImageIcon** can be passed as an argument to the **Icon** parameter of **JLabel**'s constructor. There are several ways to provide the image, including reading it from a file or downloading it from a URL. Here is the **ImageIcon** constructor used by the example in this section:

ImageIcon(String *filename*)

It obtains the image in the file named *filename.*

The icon and text associated with the label can be obtained by the following methods:

Icon getIcon( )
String getText( )

The icon and text associated with a label can be set by these methods:

void setIcon(Icon *icon*)
void setText(String *str*)

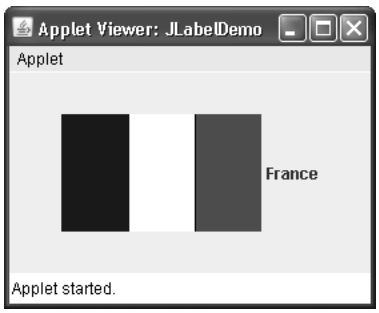
Here, *icon* and *str* are the icon and text, respectively. Therefore, using **setText( )** it is possible to change the text inside a label during program execution.

The following applet illustrates how to create and display a label containing both an icon and a string. It begins by creating an **ImageIcon** object for the file **france.gif**, which depicts the flag for France. This is used as the second argument to the **JLabel** constructor. The first and last arguments for the **JLabel** constructor are the label text and the alignment. Finally, the label is added to the content pane.

```
// Demonstrate JLabel and ImageIcon.
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JLabelDemo" width=250 height=150>
  </applet>
*/

public class JLabelDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
```

```
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    // Create an icon.
    ImageIcon ii = new ImageIcon("france.gif");

    // Create a label.
    JLabel jl = new JLabel("France", ii, JLabel.CENTER);

    // Add the label to the content pane.
    add(jl);
  }
}
```

Output from the label example is shown here:



JTextField

JTextField is the simplest Swing text component. It is also probably its most widely used text component. **JTextField** allows you to edit one line of text. It is derived from **JTextComponent**, which provides the basic functionality common to Swing text components. **JTextField** uses the **Document** interface for its model.

Three of **JTextField**'s constructors are shown here:

JTextField(int *cols*)
JTextField(String *str*, int *cols*)
JTextField(String *str*)

Here, *str* is the string to be initially presented, and *cols* is the number of columns in the text field. If no string is specified, the text field is initially empty. If the number of columns is not specified, the text field is sized to fit the specified string.

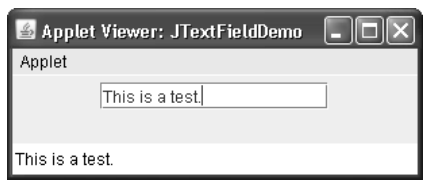
JTextField generates events in response to user interaction. For example, an **ActionEvent** is fired when the user presses ENTER. A **CaretEvent** is fired each time the caret (i.e., the cursor) changes position. (**CaretEvent** is packaged in **javax.swing.event**.) Other events are

also possible. In many cases, your program will not need to handle these events. Instead, you will simply obtain the string currently in the text field when it is needed. To obtain the text currently in the text field, call **getText( )**.

The following example illustrates **JTextField**. It creates a **JTextField** and adds it to the content pane. When the user presses ENTER, an action event is generated. This is handled by displaying the text in the status window.

```
// Demonstrate JTextField.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JTextFieldDemo" width=300 height=50>
  </applet>
*/

public class JTextFieldDemo extends JApplet {
  JTextField jtf;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Add text field to content pane.
    jtf = new JTextField(15);
    add(jtf);
    jtf.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        // Show text when user presses ENTER.
        showStatus(jtf.getText());
      }
    });
  }
}
```

Output from the text field example is shown here:



The Swing Buttons

Swing defines four types of buttons: **JButton**, **JToggleButton**, **JCheckBox**, and **JRadioButton**. All are subclasses of the **AbstractButton** class, which extends **JComponent**. Thus, all buttons share a set of common traits.

AbstractButton contains many methods that allow you to control the behavior of buttons. For example, you can define different icons that are displayed for the button when it is disabled, pressed, or selected. Another icon can be used as a *rollover* icon, which is displayed when the mouse is positioned over a button. The following methods set these icons:

```
void setDisabledIcon(Icon di)
void setPressedIcon(Icon pi)
void setSelectedIcon(Icon si)
void setRolloverIcon(Icon ri)
```

Here, *di*, *pi*, *si*, and *ri* are the icons to be used for the indicated purpose.

The text associated with a button can be read and written via the following methods:

```
String getText( )
void setText(String str)
```

Here, *str* is the text to be associated with the button.

The model used by all buttons is defined by the **ButtonModel** interface. A button generates an action event when it is pressed. Other events are possible. Each of the concrete button classes is examined next.

JButton

The **JButton** class provides the functionality of a push button. You have already seen a simple form of it in the preceding chapter. **JButton** allows an icon, a string, or both to be associated with the push button. Three of its constructors are shown here:

```
JButton(Icon icon)
JButton(String str)
JButton(String str, Icon icon)
```

Here, *str* and *icon* are the string and icon used for the button.

When the button is pressed, an **ActionEvent** is generated. Using the **ActionEvent** object passed to the **actionPerformed( )** method of the registered **ActionListener**, you can obtain the *action command* string associated with the button. By default, this is the string displayed inside the button. However, you can set the action command by calling **setActionCommand( )** on the button. You can obtain the action command by calling **getActionCommand( )** on the event object. It is declared like this:

```
String getActionCommand( )
```

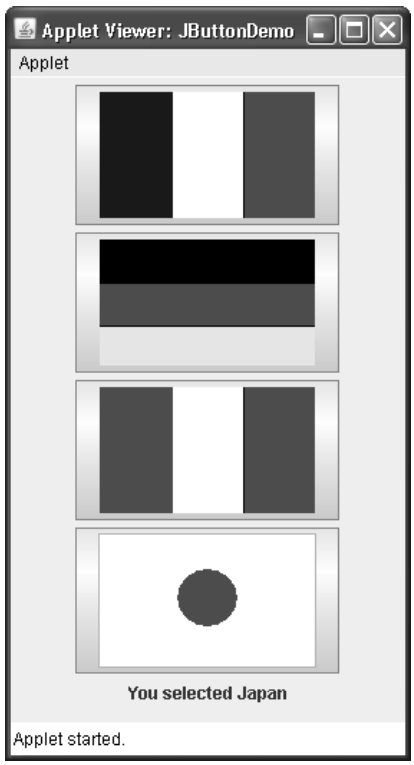
The action command identifies the button. Thus, when using two or more buttons within the same application, the action command gives you an easy way to determine which button was pressed.

In the preceding chapter, you saw an example of a text-based button. The following demonstrates an icon-based button. It displays four push buttons and a label. Each button displays an icon that represents the flag of a country. When a button is pressed, the name of that country is displayed in the label.

```
// Demonstrate an icon-based JButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JButtonDemo" width=250 height=450>
  </applet>
*/

public class JButtonDemo extends JApplet
implements ActionListener {
  JLabel jlab;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Add buttons to content pane.
    ImageIcon france = new ImageIcon("france.gif");
    JButton jb = new JButton(france);
    jb.setActionCommand("France");
    jb.addActionListener(this);
    add(jb);

    ImageIcon germany = new ImageIcon("germany.gif");
    jb = new JButton(germany);
    jb.setActionCommand("Germany");
    jb.addActionListener(this);
    add(jb);

    ImageIcon italy = new ImageIcon("italy.gif");
    jb = new JButton(italy);
    jb.setActionCommand("Italy");
```

```
    jb.addActionListener(this);
    add(jb);

    ImageIcon japan = new ImageIcon("japan.gif");
    jb = new JButton(japan);
    jb.setActionCommand("Japan");
    jb.addActionListener(this);
    add(jb);

    // Create and add the label to content pane.
    jlab = new JLabel("Choose a Flag");
    add(jlab);
  }

  // Handle button events.
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
  }
}
```

Output from the button example is shown here:



JToggleButton

A useful variation on the push button is called a *toggle button*. A toggle button looks just like a push button, but it acts differently because it has two states: pushed and released. That is, when you press a toggle button, it stays pressed rather than popping back up as a regular push button does. When you press the toggle button a second time, it releases (pops up). Therefore, each time a toggle button is pushed, it toggles between its two states.

Toggle buttons are objects of the **JToggleButton** class. **JToggleButton** implements **AbstractButton**. In addition to creating standard toggle buttons, **JToggleButton** is a superclass for two other Swing components that also represent two-state controls. These are **JCheckBox** and **JRadioButton**, which are described later in this chapter. Thus, **JToggleButton** defines the basic functionality of all two-state components.

JToggleButton defines several constructors. The one used by the example in this section is shown here:

JToggleButton(String *str*)

This creates a toggle button that contains the text passed in *str*. By default, the button is in the off position. Other constructors enable you to create toggle buttons that contain images, or images and text.

JToggleButton uses a model defined by a nested class called **JToggleButton.ToggleButtonModel**. Normally, you won't need to interact directly with the model to use a standard toggle button.

Like **JButton**, **JToggleButton** generates an action event each time it is pressed. Unlike **JButton**, however, **JToggleButton** also generates an item event. This event is used by those components that support the concept of selection. When a **JToggleButton** is pressed in, it is selected. When it is popped out, it is deselected.

To handle item events, you must implement the **ItemListener** interface. Recall from Chapter 22, that each time an item event is generated, it is passed to the **itemStateChanged( )** method defined by **ItemListener**. Inside **itemStateChanged( )**, the **getItem( )** method can be called on the **ItemEvent** object to obtain a reference to the **JToggleButton** instance that generated the event. It is shown here:

```
Object getItem( )
```

A reference to the button is returned. You will need to cast this reference to **JToggleButton**.

The easiest way to determine a toggle button's state is by calling the **isSelected( )** method (inherited from **AbstractButton**) on the button that generated the event. It is shown here:

```
boolean isSelected( )
```

It returns **true** if the button is selected and **false** otherwise.

Here is an example that uses a toggle button. Notice how the item listener works. It simply calls **isSelected( )** to determine the button's state.

```
// Demonstrate JToggleButton.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JToggleButtonDemo" width=200 height=80>
  </applet>
*/

public class JToggleButtonDemo extends JApplet {

  JLabel jlab;
  JToggleButton jtbn;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
```

```
  }

  private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Create a label.
    jlab = new JLabel("Button is off.");

    // Make a toggle button.
    jtbn =  new JToggleButton("On/Off");

    // Add an item listener for the toggle button.
    jtbn.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent ie) {
        if(jtbn.isSelected())
          jlab.setText("Button is on.");
        else
          jlab.setText("Button is off.");
      }

    });

    // Add the toggle button and label to the content pane.
    add(jtbn);
    add(jlab);
  }
}
```

The output from the toggle button example is shown here:

Check Boxes

The **JCheckBox** class provides the functionality of a check box. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons, as just described. **JCheckBox** defines several constructors. The one used here is

JCheckBox(String *str*)

It creates a check box that has the text specified by *str* as a label. Other constructors let you specify the initial selection state of the button and specify an icon.

When the user selects or deselects a check box, an **ItemEvent** is generated. You can obtain a reference to the **JCheckBox** that generated the event by calling **getItem( )** on the **ItemEvent** passed to the **itemStateChanged( )** method defined by **ItemListener**. The easiest way to determine the selected state of a check box is to call **isSelected( )** on the **JCheckBox** instance.

In addition to supporting the normal check box operation, **JCheckBox** lets you specify the icons that indicate when a check box is selected, cleared, and rolled-over. We won't be using this capability here, but it is available for use in your own programs.

The following example illustrates check boxes. It displays four check boxes and a label. When the user clicks a check box, an **ItemEvent** is generated. Inside the **itemStateChanged( )** method, **getItem( )** is called to obtain a reference to the **JCheckBox** object that generated the event. Next, a call to **isSelected( )** determines if the box was selected or cleared. The **getText( )** method gets the text for that check box and uses it to set the text inside the label.

```
// Demonstrate JCheckbox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JCheckBoxDemo" width=270 height=50>
  </applet>
*/

public class JCheckBoxDemo extends JApplet
implements ItemListener {
  JLabel jlab;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Add check boxes to the content pane.
    JCheckBox cb = new JCheckBox("C");
    cb.addItemListener(this);
    add(cb);

    cb = new JCheckBox("C++");
```

```
    cb.addItemListener(this);
    add(cb);

    cb = new JCheckBox("Java");
    cb.addItemListener(this);
    add(cb);

    cb = new JCheckBox("Perl");
    cb.addItemListener(this);
    add(cb);

    // Create the label and add it to the content pane.
    jlab = new JLabel("Select languages");
    add(jlab);
  }

  // Handle item events for the check boxes.
  public void itemStateChanged(ItemEvent ie) {
    JCheckBox cb = (JCheckBox)ie.getItem();

    if(cb.isSelected())
      jlab.setText(cb.getText() + " is selected");
    else
      jlab.setText(cb.getText() + " is cleared");
  }
}
```

Output from this example is shown here:

Radio Buttons

Radio buttons are a group of mutually exclusive buttons, in which only one button can be selected at any one time. They are supported by the **JRadioButton** class, which extends **JToggleButton**. **JRadioButton** provides several constructors. The one used in the example is shown here:

JRadioButton(String *str*)

Here, *str* is the label for the button. Other constructors let you specify the initial selection state of the button and specify an icon.

In order for their mutually exclusive nature to be activated, radio buttons must be configured into a group. Only one of the buttons in the group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. A button group is created by the **ButtonGroup** class. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton *ab*)

Here, *ab* is a reference to the button to be added to the group.

A **JRadioButton** generates action events, item events, and change events each time the button selection changes. Most often, it is the action event that is handled, which means

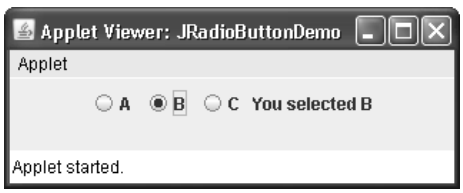
that you will normally implement the **ActionListener** interface. Recall that the only method defined by **ActionListener** is **actionPerformed( )**. Inside this method, you can use a number of different ways to determine which button was selected. First, you can check its action command by calling **getActionCommand( )**. By default, the action command is the same as the button label, but you can set the action command to something else by calling **setActionCommand( )** on the radio button. Second, you can call **getSource( )** on the **ActionEvent** object and check that reference against the buttons. Finally, you can simply check each radio button to find out which one is currently selected by calling **isSelected( )** on each button. Remember, each time an action event occurs, it means that the button being selected has changed and that one and only one button will be selected.

The following example illustrates how to use radio buttons. Three radio buttons are created. The buttons are then added to a button group. As explained, this is necessary to cause their mutually exclusive behavior. Pressing a radio button generates an action event, which is handled by **actionPerformed( )**. Within that handler, the **getActionCommand( )** method gets the text that is associated with the radio button and uses it to set the text within a label.

```
// Demonstrate JRadioButton
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JRadioButtonDemo" width=300 height=50>
  </applet>
*/

public class JRadioButtonDemo extends JApplet
implements ActionListener {
  JLabel jlab;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Create radio buttons and add them to content pane.
    JRadioButton b1 = new JRadioButton("A");
    b1.addActionListener(this);
    add(b1);
```

```
    JRadioButton b2 = new JRadioButton("B");
    b2.addActionListener(this);
    add(b2);

    JRadioButton b3 = new JRadioButton("C");
    b3.addActionListener(this);
    add(b3);

    // Define a button group.
    ButtonGroup bg = new ButtonGroup();
    bg.add(b1);
    bg.add(b2);
    bg.add(b3);

    // Create a label and add it to the content pane.
    jlab = new JLabel("Select One");
    add(jlab);
  }

  // Handle button selection.
  public void actionPerformed(ActionEvent ae) {
    jlab.setText("You selected " + ae.getActionCommand());
  }
}
```

Output from the radio button example is shown here:



JTabbedPane

JTabbedPane encapsulates a *tabbed pane*. It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront. Tabbed panes are very common in the modern GUI, and you have no doubt used them many times. Given the complex nature of a tabbed pane, they are surprisingly easy to create and use.

JTabbedPane defines three constructors. We will use its default constructor, which creates an empty control with the tabs positioned across the top of the pane. The other two constructors let you specify the location of the tabs, which can be along any of the four sides. **JTabbedPane** uses the **SingleSelectionModel** model.

Tabs are added by calling **addTab( )**. Here is one of its forms:

void addTab(String *name*, Component *comp*)

Here, *name* is the name for the tab, and *comp* is the component that should be added to the tab. Often, the component added to a tab is a **JPanel** that contains a group of related components. This technique allows a tab to hold a set of components.

The general procedure to use a tabbed pane is outlined here:

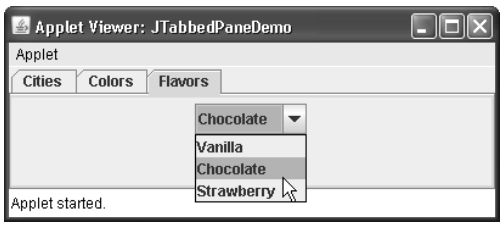
1. Create an instance of **JTabbedPane**.
2. Add each tab by calling **addTab( )**.
3. Add the tabbed pane to the content pane.

The following example illustrates a tabbed pane. The first tab is titled "Cities" and contains four buttons. Each button displays the name of a city. The second tab is titled "Colors" and contains three check boxes. Each check box displays the name of a color. The third tab is titled "Flavors" and contains one combo box. This enables the user to select one of three flavors.

```
// Demonstrate JTabbedPane.
import javax.swing.*;
/*
  <applet code="JTabbedPaneDemo" width=400 height=100>
  </applet>
*/

public class JTabbedPaneDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    JTabbedPane jtp = new JTabbedPane();
    jtp.addTab("Cities", new CitiesPanel());
    jtp.addTab("Colors", new ColorsPanel());
    jtp.addTab("Flavors", new FlavorsPanel());
    add(jtp);
  }
}

// Make the panels that will be added to the tabbed pane.
class CitiesPanel extends JPanel {

  public CitiesPanel() {
    JButton b1 = new JButton("New York");
    add(b1);
    JButton b2 = new JButton("London");
    add(b2);
    JButton b3 = new JButton("Hong Kong");
```

```
    add(b3);
    JButton b4 = new JButton("Tokyo");
    add(b4);
  }
}

class ColorsPanel extends JPanel {

  public ColorsPanel() {
    JCheckBox cb1 = new JCheckBox("Red");
    add(cb1);
    JCheckBox cb2 = new JCheckBox("Green");
    add(cb2);
    JCheckBox cb3 = new JCheckBox("Blue");
    add(cb3);
  }
}

class FlavorsPanel extends JPanel {

  public FlavorsPanel() {
    JComboBox jcb = new JComboBox();
    jcb.addItem("Vanilla");
    jcb.addItem("Chocolate");
    jcb.addItem("Strawberry");
    add(jcb);
  }
}
```

Output from the tabbed pane example is shown in the following three illustrations:



JScrollPane

JScrollPane is a lightweight container that automatically handles the scrolling of another component. The component being scrolled can either be an individual component, such as

a table, or a group of components contained within another lightweight container, such as a **JPanel**. In either case, if the object being scrolled is larger than the viewable area, horizontal and/or vertical scroll bars are automatically provided, and the component can be scrolled through the pane. Because **JScrollPane** automates scrolling, it usually eliminates the need to manage individual scroll bars.

The viewable area of a scroll pane is called the *viewport*. It is a window in which the component being scrolled is displayed. Thus, the viewport displays the visible portion of the component being scrolled. The scroll bars scroll the component through the viewport. In its default behavior, a **JScrollPane** will dynamically add or remove a scroll bar as needed. For example, if the component is taller than the viewport, a vertical scroll bar is added. If the component will completely fit within the viewport, the scroll bars are removed.

JScrollPane defines several constructors. The one used in this chapter is shown here:

JScrollPane(Component *comp*)

The component to be scrolled is specified by *comp*. Scroll bars are automatically displayed when the content of the pane exceeds the dimensions of the viewport.

Here are the steps to follow to use a scroll pane:

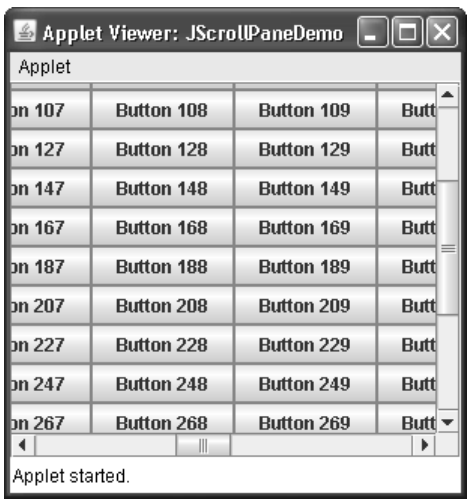
1. Create the component to be scrolled.
2. Create an instance of **JScrollPane**, passing to it the object to scroll.
3. Add the scroll pane to the content pane.

The following example illustrates a scroll pane. First, a **JPanel** object is created, and 400 buttons are added to it, arranged into 20 columns. This panel is then added to a scroll pane, and the scroll pane is added to the content pane. Because the panel is larger than the viewport, vertical and horizontal scroll bars appear automatically. You can use the scroll bars to scroll the buttons into view.

```
// Demonstrate JScrollPane.
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JScrollPaneDemo" width=300 height=250>
  </applet>
*/

public class JScrollPaneDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
```

```
  }

  private void makeGUI() {

    // Add 400 buttons to a panel.
    JPanel jp = new JPanel();
    jp.setLayout(new GridLayout(20, 20));
    int b = 0;
    for(int i = 0; i < 20; i++) {
      for(int j = 0; j < 20; j++) {
        jp.add(new JButton("Button " + b));
        ++b;
      }
    }

    // Create the scroll pane.
    JScrollPane jsp = new JScrollPane(jp);

    // Add the scroll pane to the content pane.
    // Because the default border layout is used,
    // the scroll pane will be added to the center.
    add(jsp, BorderLayout.CENTER);
  }
}
```

Output from the scroll pane example is shown here:



JList

In Swing, the basic list class is called **JList**. It supports the selection of one or more items from a list. Although the list often consists of strings, it is possible to create a list of just about any object that can be displayed. **JList** is so widely used in Java that it is highly unlikely that you have not seen one before.

JList provides several constructors. The one used here is

JList(Object[ ] *items*)

This creates a **JList** that contains the items in the array specified by *items*.

JList is based on two models. The first is **ListModel**. This interface defines how access to the list data is achieved. The second model is the **ListSelectionModel** interface, which defines methods that determine what list item or items are selected.

Although a **JList** will work properly by itself, most of the time you will wrap a **JList** inside a **JScrollPane**. This way, long lists will automatically be scrollable, which simplifies GUI design. It also makes it easy to change the number of entries in a list without having to change the size of the **JList** component.

A **JList** generates a **ListSelectionEvent** when the user makes or changes a selection. This event is also generated when the user deselects an item. It is handled by implementing **ListSelectionListener**. This listener specifies only one method, called **valueChanged( )**, which is shown here:

void valueChanged(ListSelectionEvent *le*)

Here, *le* is a reference to the object that generated the event. Although **ListSelectionEvent** does provide some methods of its own, normally you will interrogate the **JList** object itself to determine what has occurred. Both **ListSelectionEvent** and **ListSelectionListener** are packaged in **javax.swing.event**.

By default, a **JList** allows the user to select multiple ranges of items within the list, but you can change this behavior by calling **setSelectionMode( )**, which is defined by **JList**. It is shown here:

void setSelectionMode(int *mode*)

Here, *mode* specifies the selection mode. It must be one of these values defined by **ListSelectionModel**:

SINGLE_SELECTION

SINGLE_INTERVAL_SELECTION

MULTIPLE_INTERVAL_SELECTION

The default, multiple-interval selection, lets the user select multiple ranges of items within a list. With single-interval selection, the user can select one range of items. With single selection, the user can select only a single item. Of course, a single item can be selected in the other two modes, too. It's just that they also allow a range to be selected.

You can obtain the index of the first item selected, which will also be the index of the only selected item when using single-selection mode, by calling **getSelectedIndex( )**, shown here:

int getSelectedIndex( )

Indexing begins at zero. So, if the first item is selected, this method will return 0. If no item is selected, –1 is returned.

Instead of obtaining the index of a selection, you can obtain the value associated with the selection by calling **getSelectedValue( )**:

Object getSelectedValue( )

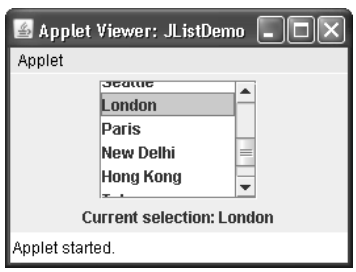
It returns a reference to the first selected value. If no value has been selected, it returns **null**.

The following applet demonstrates a simple **JList**, which holds a list of cities. Each time a city is selected in the list, a **ListSelectionEvent** is generated, which is handled by the **valueChanged( )** method defined by **ListSelectionListener**. It responds by obtaining the index of the selected item and displaying the name of the selected city in a label.

```
// Demonstrate JList.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

/*
  <applet code="JListDemo" width=200 height=120>
  </applet>
*/

public class JListDemo extends JApplet {
  JList jlst;
  JLabel jlab;
  JScrollPane jscrlp;

  // Create an array of cities.
  String Cities[] = { "New York", "Chicago", "Houston",
                      "Denver", "Los Angeles", "Seattle",
                      "London", "Paris", "New Delhi",
                      "Hong Kong", "Tokyo", "Sydney" };

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());
```

```
    // Create a JList.
    jlst = new JList(Cities);

    // Set the list selection mode to single selection.
    jlst.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    // Add the list to a scroll pane.
    jscrlp = new JScrollPane(jlst);

    // Set the preferred size of the scroll pane.
    jscrlp.setPreferredSize(new Dimension(120, 90));

    // Make a label that displays the selection.
    jlab = new JLabel("Choose a City");

    // Add selection listener for the list.
    jlst.addListSelectionListener(new ListSelectionListener() {
      public void valueChanged(ListSelectionEvent le) {
        // Get the index of the changed item.
        int idx = jlst.getSelectedIndex();

        // Display selection, if item was selected.
        if(idx != -1)
          jlab.setText("Current selection: " + Cities[idx]);
        else // Otherwise, reprompt.
          jlab.setText("Choose a City");

      }
    });

    // Add the list and label to the content pane.
    add(jscrlp);
    add(jlab);
  }
}
```

Output from the list example is shown here:



JComboBox

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class. A combo box normally displays one entry, but it will also display a drop-down list that allows a user to select a different entry. You can also create a combo box

that lets the user enter a selection into the text field. The **JComboBox** constructor used by the example is shown here:

JComboBox(Object[ ] *items*)

Here, *items* is an array that initializes the combo box. Other constructors are available.

JComboBox uses the **ComboBoxModel**. Mutable combo boxes (those whose entries can be changed) use the **MutableComboBoxModel**.

In addition to passing an array of items to be displayed in the drop-down list, items can be dynamically added to the list of choices via the **addItem( )** method, shown here:

void addItem(Object *obj*)

Here, *obj* is the object to be added to the combo box. This method must be used only with mutable combo boxes.

JComboBox generates an action event when the user selects an item from the list. **JComboBox** also generates an item event when the state of selection changes, which occurs when an item is selected or deselected. Thus, changing a selection means that two item events will occur: one for the deselected item and another for the selected item. Often, it is sufficient to simply listen for action events, but both event types are available for your use.

One way to obtain the item selected in the list is to call **getSelectedItem( )** on the combo box. It is shown here:

Object getSelectedItem( )

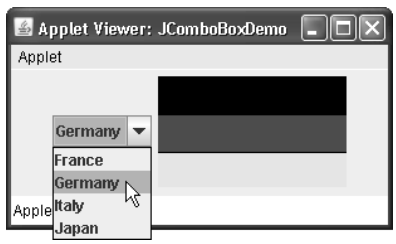
You will need to cast the returned value into the type of object stored in the list.

The following example demonstrates the combo box. The combo box contains entries for "France," "Germany," "Italy," and "Japan." When a country is selected, an icon-based label is updated to display the flag for that country. You can see how little code is required to use this powerful component.

```
// Demonstrate JComboBox.
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
  <applet code="JComboBoxDemo" width=300 height=100>
  </applet>
*/

public class JComboBoxDemo extends JApplet {
  JLabel jlab;
  ImageIcon france, germany, italy, japan;
  JComboBox jcb;

  String flags[] = { "France", "Germany", "Italy", "Japan" };

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
```

```
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    // Change to flow layout.
    setLayout(new FlowLayout());

    // Instantiate a combo box and add it to the content pane.
    jcb = new JComboBox(flags);
    add(jcb);

    // Handle selections.
    jcb.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent ae) {
        String s = (String) jcb.getSelectedItem();
        jlab.setIcon(new ImageIcon(s + ".gif"));
      }
    });

    // Create a label and add it to the content pane.
    jlab = new JLabel(new ImageIcon("france.gif"));
    add(jlab);
  }
}
```

Output from the combo box example is shown here:



Trees

A *tree* is a component that presents a hierarchical view of data. The user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree** class. A sampling of its constructors is shown here:

JTree(Object *obj*[ ])
JTree(Vector<?> *v*)
JTree(TreeNode *tn*)

In the first form, the tree is constructed from the elements in the array *obj*. The second form constructs the tree from the elements of vector *v*. In the third form, the tree whose root node is specified by *tn* specifies the tree.

Although **JTree** is packaged in **javax.swing**, its support classes and interfaces are packaged in **javax.swing.tree**. This is because the number of classes and interfaces needed to support **JTree** is quite large.

JTree relies on two models: **TreeModel** and **TreeSelectionModel**. A **JTree** generates a variety of events, but three relate specifically to trees: **TreeExpansionEvent**, **TreeSelectionEvent**, and **TreeModelEvent**. **TreeExpansionEvent** events occur when a node is expanded or collapsed. A **TreeSelectionEvent** is generated when the user selects or deselects a node within the tree. A **TreeModelEvent** is fired when the data or structure of the tree changes. The listeners for these events are **TreeExpansionListener**, **TreeSelectionListener**, and **TreeModelListener**, respectively. The tree event classes and listener interfaces are packaged in **javax.swing.event**.

The event handled by the sample program shown in this section is **TreeSelectionEvent**. To listen for this event, implement **TreeSelectionListener**. It defines only one method, called **valueChanged( )**, which receives the **TreeSelectionEvent** object. You can obtain the path to the selected object by calling **getPath( )**, shown here, on the event object.

```
TreePath getPath( )
```

It returns a **TreePath** object that describes the path to the changed node. The **TreePath** class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods. In this book, only the **toString( )** method is used. It returns a string that describes the path.

The **TreeNode** interface declares methods that obtain information about a tree node. For example, it is possible to obtain a reference to the parent node or an enumeration of the child nodes. The **MutableTreeNode** interface extends **TreeNode**. It declares methods that can insert and remove child nodes or change the parent node.

The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree. One of its constructors is shown here:

```
DefaultMutableTreeNode(Object obj)
```

Here, *obj* is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children.

To create a hierarchy of tree nodes, the **add( )** method of **DefaultMutableTreeNode** can be used. Its signature is shown here:

```
void add(MutableTreeNode child)
```

Here, *child* is a mutable tree node that is to be added as a child to the current node.

JTree does not provide any scrolling capabilities of its own. Instead, a **JTree** is typically placed within a **JScrollPane**. This way, a large tree can be scrolled through a smaller viewport.

Here are the steps to follow to use a tree:

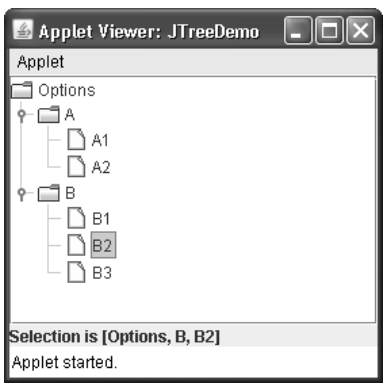
1. Create an instance of **JTree**.
2. Create a **JScrollPane** and specify the tree as the object to be scrolled.
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create a tree and handle selections. The program creates a **DefaultMutableTreeNode** instance labeled "Options." This is the top node of the tree hierarchy. Additional tree nodes are then created, and the **add( )** method is called to connect these nodes to the tree. A reference to the top node in the tree is provided as the argument to the **JTree** constructor. The tree is then provided as the argument to the **JScrollPane** constructor. This scroll pane is then added to the content pane. Next, a label is created and added to the content pane. The tree selection is displayed in this label. To receive selection events from the tree, a **TreeSelectionListener** is registered for the tree. Inside the **valueChanged( )** method, the path to the current selection is obtained and displayed.

```
// Demonstrate JTree.
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
  <applet code="JTreeDemo" width=400 height=200>
  </applet>
*/


public class JTreeDemo extends JApplet {
  JTree tree;
  JLabel jlab;

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    // Create top node of tree.
    DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");

    // Create subtree of "A".
    DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
    top.add(a);
    DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
    a.add(a1);
    DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
    a.add(a2);
```

```
    // Create subtree of "B".
    DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
    top.add(b);
    DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
    b.add(b1);
    DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
    b.add(b2);
    DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
    b.add(b3);

    // Create the tree.
    tree = new JTree(top);

    // Add the tree to a scroll pane.
    JScrollPane jsp = new JScrollPane(tree);

    // Add the scroll pane to the content pane.
    add(jsp);

    // Add the label to the content pane.
    jlab = new JLabel();
    add(jlab, BorderLayout.SOUTH);

    // Handle tree selection events.
    tree.addTreeSelectionListener(new TreeSelectionListener() {
      public void valueChanged(TreeSelectionEvent tse) {
        jlab.setText("Selection is " + tse.getPath());
      }
    });
  }
}
```

Output from the tree example is shown here:



The string presented in the text field describes the path from the top tree node to the selected node.

JTable

JTable is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Depending on its configuration, it is also possible to select a row, column, or cell within the table, and to change the data within a cell. **JTable** is a sophisticated component that offers many more options and features than can be discussed here. (It is perhaps Swing's most complicated component.) However, in its default configuration, **JTable** still offers substantial functionality that is easy to use—especially if you simply want to use the table to present data in a tabular format. The brief overview presented here will give you a general understanding of this powerful component.

Like **JTree**, **JTable** has many classes and interfaces associated with it. These are packaged in **javax.swing.table**.

At its core, **JTable** is conceptually simple. It is a component that consists of one or more columns of information. At the top of each column is a heading. In addition to describing the data in a column, the heading also provides the mechanism by which the user can change the size of a column or change the location of a column within the table. **JTable** does not provide any scrolling capabilities of its own. Instead, you will normally wrap a **JTable** inside a **JScrollPane**.

JTable supplies several constructors. The one used here is

JTable(Object *data*[ ][ ], Object *colHeads*[ ])

Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

JTable relies on three models. The first is the table model, which is defined by the **TableModel** interface. This model defines those things related to displaying data in a two-dimensional format. The second is the table column model, which is represented by **TableColumnModel**. **JTable** is defined in terms of columns, and it is **TableColumnModel** that specifies the characteristics of a column. These two models are packaged in **javax.swing.table**. The third model determines how items are selected, and it is specified by the **ListSelectionModel**, which was described when **JList** was discussed.

A **JTable** can generate several different events. The two most fundamental to a table's operation are **ListSelectionEvent** and **TableModelEvent**. A **ListSelectionEvent** is generated when the user selects something in the table. By default, **JTable** allows you to select one or more complete rows, but you can change this behavior to allow one or more columns, or one or more individual cells to be selected. A **TableModelEvent** is fired when that table's data changes in some way. Handling these events requires a bit more work than it does to handle the events generated by the previously described components and is beyond the scope of this book. However, if you simply want to use **JTable** to display data (as the following example does), then you don't need to handle any events.

Here are the steps required to set up a simple **JTable** that can be used to display data:

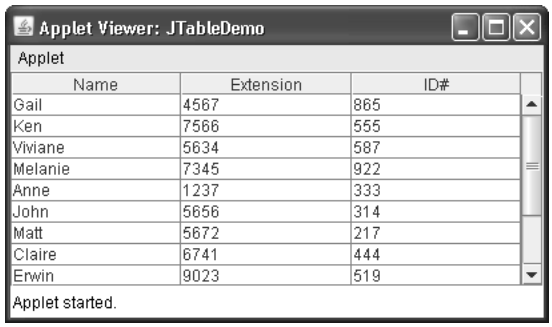
1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

The following example illustrates how to create and use a simple table. A one-dimensional array of strings called **colHeads** is created for the column headings. A two-dimensional array of strings called **data** is created for the table cells. You can see that each element in the array is an array of three strings. These arrays are passed to the **JTable** constructor. The table is added to a scroll pane, and then the scroll pane is added to the content pane. The table displays the data in the **data** array. The default table configuration also allows the contents of a cell to be edited. Changes affect the underlying array, which is **data** in this case.

```
// Demonstrate JTable.
import java.awt.*;
import javax.swing.*;
/*
  <applet code="JTableDemo" width=400 height=200>
  </applet>
*/

public class JTableDemo extends JApplet {

  public void init() {
    try {
      SwingUtilities.invokeAndWait(
        new Runnable() {
          public void run() {
            makeGUI();
          }
        }
      );
    } catch (Exception exc) {
      System.out.println("Can't create because of " + exc);
    }
  }

  private void makeGUI() {

    // Initialize column headings.
    String[] colHeads = { "Name", "Extension", "ID#" };

    // Initialize data.
    Object[][] data = {
      { "Gail", "4567", "865" },
      { "Ken", "7566", "555" },
      { "Viviane", "5634", "587" },
      { "Melanie", "7345", "922" },
      { "Anne", "1237", "333" },
      { "John", "5656", "314" },
      { "Matt", "5672", "217" },
      { "Claire", "6741", "444" },
      { "Erwin", "9023", "519" },
      { "Ellen", "1134", "532" },
      { "Jennifer", "5689", "112" },
      { "Ed", "9030", "133" },
      { "Helen", "6751", "145" }
    };
```

```
    // Create the table.
    JTable table = new JTable(data, colHeads);

    // Add the table to a scroll pane.
    JScrollPane jsp = new JScrollPane(table);

    // Add the scroll pane to the content pane.
    add(jsp);
  }
}
```

Output from this example is shown here:



Continuing Your Exploration of Swing

Swing defines a very large GUI toolkit. It has many more features that you will want to explore on your own. For example, Swing provides toolbars, tooltips, and progress bars. It also provides a complete menu subsystem. Swing's pluggable look and feel lets you substitute another appearance and behavior for an element. You can define your own models for the various components, and you can change the way that cells are edited and rendered when working with tables and trees. The best way to become familiar with Swing's capabilities is to experiment with it.