

CHAPTER

9

- The Delegation Event Model on page 237
- The Big Picture on page 241
- AWT Adapters on page 251
- Component Events on page 254
- Semantic Events on page 269
- Event Adapters on page 279
- Inner Classes on page 290
- Firing AWT Events from Custom Components on page 298
- Firing Custom Events from Custom Components on page 300
- Dispatching Events and the AWT Event Queue on page 309
- Active Events on page 313
- Inheritance-Based Mechanism on page 315
- Event Handling Design on page 319
- Summary on page 325

The Delegation Event Model (AWT 1.1 and Beyond)



The 1.1 version of the AWT introduced a new delegation-based event model that is vastly improved over the original, inheritance-based model. The delegation event model supports the inheritance-based (original) event model; however, the inheritance-based model will be phased out in a future release.

This chapter discusses the delegation event model. In “Shortcomings of the Inheritance-Based Event Model” on page 229, we discussed some of the drawbacks of the original event model, all of which are addressed by the new model. Applets (and applications) that use the original event model will still continue to work under the new model; however, the old API will be eliminated in a future release. For now, compiler warnings are issued if you are still using the old API, and mixing the two event models in a single component is not supported. We suggest that you migrate your old event handling code to the new model as soon as possible and that you use the new event model for any new development. Guidelines for doing so may be found in Event Handling Design on page 319.

The Delegation Event Model

The delegation event model derives its name from the fact that event handling is delegated from an event source to one or more event listeners.

The premise behind the delegation event model is simple: *components* fire *events*, which can be listened for and acted upon by *listeners*. Listeners are registered with a component by invoking one of a number of `addXYZListener (XYZListener)` methods. After a listener is added to a component, appropriate methods in the listener's interface will be called when the corresponding type of event is fired by the component.

Components, Events and Listeners

The applet shown in Figure 9-1 and listed in Example 9-1 illustrates the fundamental concepts of the delegation event model.

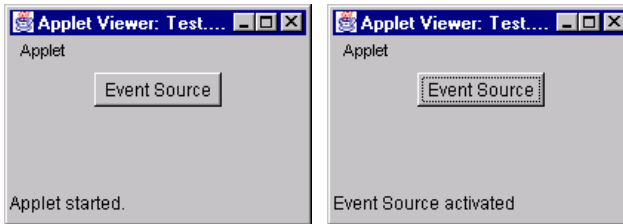


Figure 9-1 A Simple Illustration of the Delegation Event Model

The picture on the left shows the applet in its initial state. The picture on the right was taken after the button was activated. Note the applet's status bar.

The applet instantiates a `ButtonListener` and adds it to the button. `ButtonListener` implements the `ActionListener` interface and updates the applet's status bar when the button is activated.

**Example 9-1** Listening for Action Events

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

public class Test extends Applet {
    public void init() {
        Button eventSource = new Button("Event Source");
        eventSource.addActionListener(new ButtonListener(this));
        add(eventSource);
    }
}

class ButtonListener implements ActionListener {
    private Applet applet;

    public ButtonListener(Applet applet) {
        this.applet = applet;
    }

    public void actionPerformed(ActionEvent event) {
        Button source = (Button)event.getSource();
        applet.showStatus(source.getLabel() + " activated");
    }
}
```

`java.awt.Button` provides registration methods for `ActionListeners`—`addActionListener()` and `removeActionListener()`. When a button is activated, an `ActionEvent` is instantiated and fired to all `ActionListeners` registered with the button at the time the event occurred. The `Button` class fires action events by invoking `actionPerformed()` for every registered `ActionListener()`. The `actionPerformed` method is passed a reference to the `ActionEvent` instantiated by the button at the time the event occurred.

Figure 9-2 shows the sequence of events for firing an event from an event source to a set of listeners. First, the listener must be added to the source as a specific type of listener. Subsequent events of the specified type will cause the source to invoke an *eventHandler* method defined in the listener's interface that takes a single `EventType` argument.

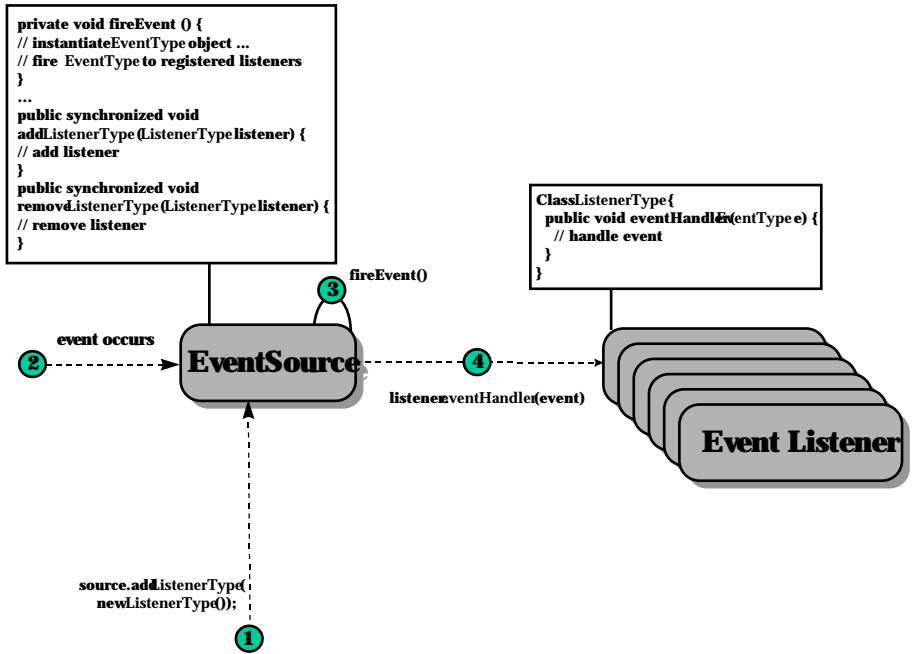


Figure 9-2 Firing an Event to a Set of Listeners
Event sources invoke a method defined by a listener interface, passing an **EventType**.

Event sources implement listener registration methods such as `Button.addActionListener(ActionListener)` and `Button.removeActionListener(ActionListener)`, whereby event listeners of a specified type may register and unregister interest in events fired by the source. When an event occurs, the event source instantiates an appropriate event, which is passed to the set of listeners that are registered with the source at the time the event occurred.

Event listeners are responsible for implementing event handling methods. Event handling methods are passed references to events, which are typically immutable, and contain information about the event and a reference to the event source.



Event listeners are obligated to handle events in a timely manner. Calls made to event listeners are made on the source's thread, so the next listener can't get started until the current listener is done processing the event. One popular technique for event listeners that have long or indefinite event handling tasks is to use an adapter that queues events for delivery on another thread.

AWT TIP ...

The JDK Event Model

The JDK event model revolves around three types of objects:

- *Event Sources* fire events.
- *Event Listeners* handle events.
- *Events* represent an event at the time it occurred.

Event listeners register with an event source by invoking `void EventSource.addListenerType(ListenerType listener)`. Event sources fire events by instantiating an `EventType`, which is passed to `void ListenerType.eventHandler(EventType event)`.

Filtering Events

Under the inheritance-based event model, all events were sent to every component whether the component was interested in the event or not. The delegation event model filters events; events are only delivered to a component if:

- A listener interested in the event type is added to the component, OR
- `Component.enableEvents(long mask)` is invoked, where `mask` represents the events to be delivered.

Regardless of which approach is taken, events associated with either the mask or the type of listener will be fired by the component and either passed along to listeners or made available for overridden event handling methods.

The Big Picture

All of the action, as far as events and listeners are concerned, starts in the `java.util` package. `java.util` comes with a class and an interface—`EventObject` and `EventListener`, respectively, which form the foundation of the delegation event model. `EventObject` is a simple class that does nothing more than keep track of its event source, and `EventListener` is a *tagging*¹

interface, which all listeners extend. `EventListener` and `EventObject` anchor hierarchies of listeners and events, respectively. Figure 9-3 shows the event listeners, all of which extend the `java.util.EventListener` interface.

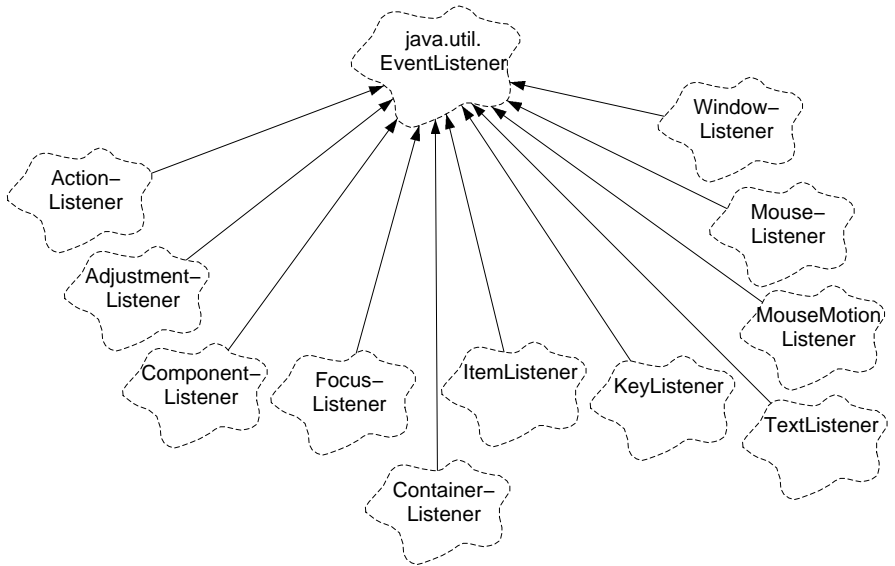


Figure 9-3 Event Listener Interfaces

All of the event listener interfaces shown above reside in the `java.awt.event` package, except for `java.util.EventListener`.

Figure 9-4 shows the hierarchy of AWT event classes.

`java.util.EventObject` maintains a reference to the source of the event,² and the `AWTEvent` class keeps track of the ID of the event and whether or not the event is consumed.³ For the most part, the current version of the AWT has done away with `public` variables—access to the source and ID of an event are provided through public accessor methods on the appropriate class—see “AWT Event Classes (all classes are from the `java.awt.event` package)” on page 254.

1. A tagging interface does not define any methods.
2. Note that the event source is of type `Object`, and not `Component`.
3. Consumed events are not passed along to peers. See “Consuming Input Events” on page 267.

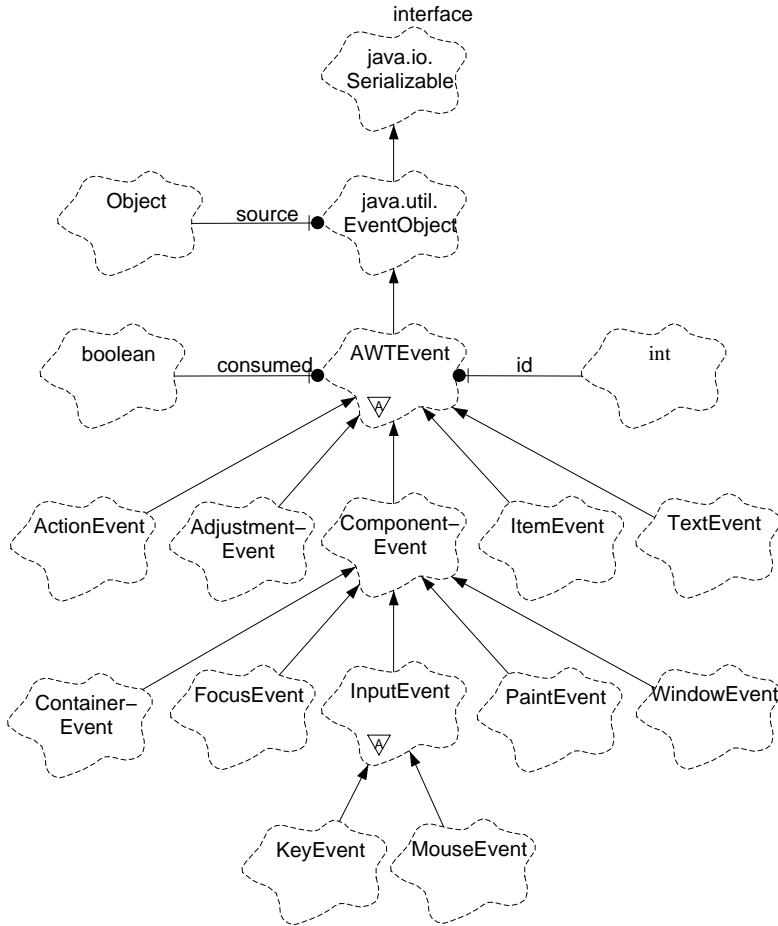


Figure 9-4 AWT Event Class Hierarchy

Events

Table 9-1 lists the public methods for `java.util.EventObject` and its extensions, excluding irrelevant methods such as `toString()`.

The delegation event model provides classes for events instead of representing events by integer constants, thereby eliminating the switch statement that resided in many overridden `handleEvent` methods under the original event model.

However, implementing a class for every possible event would result in an overwhelming number of event classes, so some event classes encompass a number of related events. For instance, `WindowEvent` represents events for activating, deactivating, closing, opening, iconifying, and deiconifying windows. As a result, some of the event classes define `integer` constants for encoding the particular type of event that they represent.⁴ The constants, along with other constants defined by the event classes, are also listed in Table 9-1.

Table 9-1 JDK Event Classes: Public Methods and Constants

Class	Public Methods	Constants
<code>java.util.Event</code>	<code>Object getSource()</code>	
<code>Object</code>		
<code>AWTEvent</code>	<code>int getId()</code>	<code>COMPONENT_EVENT_MASK,</code> <code>FOCUS_EVENT_MASK</code> <code>KEY_EVENT_MASK, MOUSE_EVENT_MASK</code> <code>MOUSE_MOTION_EVENT_MASK</code> <code>WINDOW_EVENT_MASK, ACTION_EVENT_MASK</code> <code>ADJUSTMENT_EVENT_MASK,</code> <code>ITEM_EVENT_MASK</code>
<code>ActionEvent</code>	<code>String getActionCommand()</code> <code>int getModifiers()</code>	<code>SHIFT_MASK, CTRL_MASK, META_MASK,</code> <code>ALT_MASK</code> <code>ACTION_FIRST, ACTION_LAST,</code> <code>ACTION_PERFORMED</code>
<code>AdjustmentEvent</code>	<code>Adjustable getAdjustable()</code> <code>int getValue()</code> <code>int getAdjustmentType()</code>	<code>ADJUSTMENT_FIRST, ADJUSTMENT_LAST</code> <code>ADJUSTMENT_VALUE_CHANGED</code> <code>UNIT_INCREMENT,</code> <code>UNIT_DECREMENT</code> <code>BLOCK_INCREMENT, BLOCK_DECREMENT,</code> <code>TRACK</code>
<code>ComponentEvent</code>	<code>Component getComponent()</code>	<code>COMPONENT_FIRST, COMPONENT_LAST</code> <code>COMPONENT_MOVED, COMPONENT_RESIZED</code> <code>COMPONENT_SHOWN, COMPONENT_HIDDEN</code>
<code>ContainerEvent</code>	<code>Container getContainer()</code> <code>Component getChild()</code>	<code>CONTAINER_FIRST</code> <code>CONTAINER_LAST</code> <code>CONTAINER_ADDED</code> <code>CONTAINER_REMOVED</code>
<code>FocusEvent</code>	<code>boolean isTemporary()</code>	<code>FOCUS_FIRST, FOCUS_LAST</code> <code>FOCUS_GAINED, FOCUS_LOST</code>

4. Note that use of these constants can result in switch statements, but the switch statements will be simple and self-contained.

**Table 9-1 JDK Event Classes: Public Methods and Constants (Continued)**

Class	Public Methods	Constants
InputEvent	boolean isShiftDown() boolean isControlDown() boolean isMetaDown() boolean isAltDown() long getWhen() int getModifiers() void consume() boolean isConsumed()	SHIFT_MASK, CTRL_MASK, META_MASK, ALT_MASK BUTTON1_MASK, BUTTON2_MASK, BUTTON3_MASK
ItemEvent	ItemSelectable getItemSelectable() Object getItem() int getStateChange()	ITEM_FIRST, ITEM_LAST, ITEM_STATE_CHANGED, SELECTED, DESELECTED
KeyEvent	char getKeyChar() int getKeyCode() boolean isActionKey() void setKeyChar(char) void setKeyCode(int) void setModifiers(int) static String getKeyModifiersText(int) static String getKeyText(int)	KEY_FIRST, KEY_LAST KEY_TYPED, KEY_PRESSED, KEY_RELEASED KEY_ACTION_FIRST, KEY_ACTION_LAST HOME, END, PGUP, PGDN, UP, DOWN F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, PRINT_SCREEN, SCROLL_LOCK, CAPS_LOCK, NUM_LOCK, PAUSE, INSERT, ENTER, BACK_SPACE, TAB, ESCAPE, DELETE
MouseEvent	int getClickCount() Point getPoint() int getX() int getY() void translatePoint(int x, int y) boolean isPopupTrigger()	MOUSE_FIRST, MOUSE_LAST MOUSE_CLICKED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_MOVED, MOUSE_ENTERED, MOUSE_EXITED, MOUSE_DRAGGED

Table 9-1 JDK Event Classes: Public Methods and Constants (Continued)

Class	Public Methods	Constants
PaintEvent	Rectangle getUpdateRect() void setUpdateRect(Rectangle)	PAINT_FIRST, PAINT_LAST, PAINT, UPDATE
TextEvent	N/A	TEXT_FIRST TEXT_LAST TEXT_VALUE_CHANGED
WindowEvent	Window getWindow()	WINDOW_FIRST, WINDOW_LAST, WINDOW_ACTIVATED, WINDOW_DEACTIVATED, WINDOW_OPENED, WINDOW_CLOSING, WINDOW_CLOSED, WINDOW_ICONIFIED, WINDOW_DEICONIFIED

Components As Event Sources

Many AWT components are fitted with `addXYZListener (XYZListener)` methods to allow certain types of listeners to register interest in events. For instance, as you can see from Table 9-2, `java.awt.Button` comes with an `addActionListener (ActionListener)` method.

In order to handle a particular event for a given component, simply create a listener and pass it to the component's appropriate `addXYZListener` method, and the events in question will automatically be routed to the listener, meaning one of the methods in Table 9-2 will be invoked. Realize that all of the `java.awt.Component` extensions (`Button`, `Choice`, etc.) inherit the `addXYZListener` methods implemented in the `Component` class. Therefore, buttons, for instance, can support component, focus, key, mouse, and mouse motion listeners in addition to action listeners.

Note that more than one listener of a particular type can be registered with a single component. For instance, a button may have a number of action listeners, all of which listen for action events fired by the button. In such a case, the order in which events are delivered to the registered listeners is undefined. In practice, this is usually not a problem, but there are situations where the order of notification is important. You might think that the order of notification corresponds to the order in which the listeners are added to the component, but that is not necessarily the case. As a matter of fact, we will run across such a situation in "A Rubberband Panel" on page 776.

**Table 9-2 AWT Component Listener Registration Methods**

AWT Class/Interface	Listener Registration Methods
Button	void addActionListener(ActionListener)
Checkbox	void addItemListener(ItemListener)
CheckboxMenuItem	void addItemListener(ItemListener)
Choice	void addItemListener(ItemListener)
Component	void addComponentListener(ComponentListener) void addFocusListener(FocusListener) void addInputMethodListener(InputMethodListener) void addKeyListener(KeyListener) void addMouseListener(MouseListener) void addMouseMotionListener(MouseMotionListener)
Container	addContainerListener(ContainerListener)
List	void addActionListener(ActionListener) void addItemListener(ItemListener)
MenuItem	void addActionListener(ActionListener)
Scrollbar	void addAdjustmentListener(AdjustmentListener)
TextArea	void addTextListener(TextListener)
TextComponent	void addTextListener(TextListener)
TextField	void addTextListener(TextListener) void addActionListener(ActionListener)
Window	void addWindowListener(WindowListener)

AWT TIP

Notification Order for Multiple Listeners Is Undefined

If multiple listeners of a particular type are registered with a single component, the order in which the listeners are notified of events is undefined. Although it might seem that the order of notification should correspond to the order in which the listeners are added to the component, that is not necessarily the case. At any rate, you should not rely upon the observed order of notification for multiple listeners because it may vary from one platform to another.



Multicast Event Sources

All of the event sources listed in Table 9-2 on page 247 are multicast event sources. A multicast event source may have more than one event listener of a given type registered at any point in time. Multicast event sources should implement event listener registration methods that adhere to the following JavaBeans™ design pattern:

```
void addListenerType(ListenerType listener)
void removeListenerType(ListenerType listener)
```

`addListenerType` methods add the specified listener to the set of listeners of type `ListenerType` maintained by the event source. `removeListenerType` methods remove the specified listener from the set of listeners. Both methods should be `synchronized` in order to avoid race conditions. The order of event delivery to registered listeners is defined by the implementation.

Unicast Event Sources

Whenever possible, event sources should be multicast. There are situations, however, where it may be necessary to disallow multiple listeners for a given listener type. For instance, an image button may have a single listener that controls the manner in which the button reacts to mouse events.

Unicast event sources maintain a single event listener for a given type of event. Listener registration methods for a unicast event source are defined by the following JavaBeans design pattern:

```
void addListenerType(ListenerType listener)
    throws java.util.TooManyListenersException
void removeListenerType(ListenerType listener)
```

If the `addListenerType` method is invoked on a unicast event source that already has a listener of `ListenerType` registered, the registered listener should remain unchanged and the event source should throw a `TooManyListenersException`. For the listener associated with a unicast event source to be changed, the `removeListenerType` method must be invoked before the `addListenerType` method is called.

Unicast event sources may also be multicast event sources. For example, the image button mentioned above may have a single listener for reacting to mouse events but may also want to allow multiple listeners that react to action events.



Event Source Interfaces

The `java.awt` package contains two event source interfaces: `Adjustable` and `ItemSelectable`.⁵ `Adjustables` have values that can be set between specified minimum and maximum amounts. They also fire `AdjustmentEvents` whenever their value changes.

An `ItemSelectable` has items, of which zero or more are selectable.

Listeners

The `java.awt.event` package defines eleven interfaces for different types of listeners. Each listener interface defines methods that will be invoked when a specific event occurs. For instance, as you can see from Table 9-3, `java.awt.event.ActionListener` defines a lone method: `actionPerformed()`, which is invoked when an *action* event is fired from a component with which the listener has registered interest.

Table 9-3 also lists the corresponding event constant and convenience methods from the inheritance-based event model to aid you in converting your code to the delegation event model. For example, handling an event under the old event model by checking the `id` field of the event for `ACTION_EVENT` or overriding `action()` gets replaced by an `ActionListener` that has its `actionPerformed` method invoked when an action event is fired.⁶

Table 9-3 `java.awt.event` Listener Interfaces and Methods

Interface	Methods	Corresponding Event/Method from 1.02
<code>ActionListener</code>	<code>void actionPerformed(ActionEvent)</code>	<code>ACTION_EVENT/action()</code>
<code>AdjustmentListener</code>	<code>void adjustmentValueChanged(AdjustmentEvent)</code>	N/A
<code>ComponentListener</code>	<code>void componentHidden(ComponentEvent)</code>	N/A
	<code>void componentMoved(ComponentEvent)</code>	<code>COMPONENT_MOVED</code>
	<code>void componentResized(ComponentEvent)</code>	N/A
	<code>void componentShown(ComponentEvent)</code>	N/A
<code>ContainerListener</code>	<code>void componentAdded(ContainerEvent)</code>	N/A
	<code>void componentRemoved(ContainerEvent)</code>	N/A

5. Event source interfaces are defined in the `java.awt` package, but listener interfaces are defined in `java.awt.event`.

Table 9-3 java.awt.event Listener Interfaces and Methods (Continued)

Interface	Methods	Corresponding Event/Method from 1.02
FocusListener	void focusGained(FocusEvent) void focusLost(FocusEvent)	GOT_FOCUS/gotFocus() LOST_FOCUS/lostFocus()
InputMethodListener	void caretPositionChanged(InputMethodEvent) void inputMethodTextChanged(InputMethodEvent)	
ItemListener	void itemStateChanged(ItemEvent)	LIST_SELECT, LIST_DESELECT
KeyListener	void keyTyped(KeyEvent) void keyPressed(KeyEvent) void keyReleased(KeyEvent)	N/A KEY_PRESS/keyDown() KEY_RELEASE/keyUp()
MouseListener	void mouseClicked(MouseEvent) void mouseEntered(MouseEvent) void mouseExited(MouseEvent) void mousePressed(MouseEvent) void mouseReleased(MouseEvent)	MOUSE_UP/mouseUp() MOUSE_DOWN/mouseDown() MOUSE_UP/mouseUp() MOUSE_ENTER/mouseEnter() MOUSE_EXIT/mouseExit()
MouseMotion-Listener	void mouseDragged(MouseEvent) void mouseMoved(MouseEvent)	MOUSE_DRAG/mouseDrag() MOUSE_MOVE/mouseMove()
TextListener	void textValueChanged(TextEvent)	N/A
WindowListener	void windowActivated(WindowEvent) void windowDeactivated(WindowEvent) void windowClosed(WindowEvent) void windowOpened(WindowEvent) void windowClosing(WindowEvent) void windowIconified(WindowEvent) void windowDeiconified(WindowEvent)	N/A N/A N/A WINDOW_DESTROY WINDOW_EXPOSE WINDOW_ICONIFY WINDOW_DEICONIFY

JavaBeans Design Pattern for Event Handling Methods

All of the methods defined by the listener interfaces in Table 9-3 conform to the JavaBeans design pattern for event handling methods:

```
void eventHandler(EventType evt)
```

The name of the method should reflect the type of event being handled. For instance, the event handling method for the `ActionListener` interface is `actionPerformed()`, indicating that an action of some type has been

6. There are actually a number of different approaches to take when converting event handling code from the old model to the new. See “Event Handling Design” on page 319.



performed. Event handling methods should be passed an event ultimately derived from `java.util.EventObject`. Event handling methods may also throw checked exceptions, in which case the method signature will include a `throws` clause.

Under exceptional circumstances, such as forwarding event notifications to external environments with different conventions, an alternate design pattern for event handling methods may be used:

```
void eventHandler(arbitrary-parameter-list)
```

Methods adhering to the alternate design pattern may take an arbitrary number of arguments of any Java type. Notice that the alternate design pattern specifies only that the method return `void`; the name of the method and the types and number of arguments could be anything. This alternate design pattern makes it difficult for humans and JavaBeans builder tools to identify such methods. As a result, the alternate design pattern should be used only when absolutely necessary.

AWT Adapters

Some of the interfaces listed in Table 9-3 on page 249 require implementing a fair number of methods. Listeners, a good percentage of the time, are only interested in implementing a small subset of the methods defined by listener interfaces. For instance, it is common to implement `WindowListener.windowClosing()` to handle the closing of the window, without any interest in the other six events defined by the `WindowListener` interface.

To free developers from having to implement every method defined in a listener interface, the AWT provides a number of adapter classes that provide no-op implementations of the methods defined in the appropriate interface. For example, `java.awt.event.MouseAdapter` implements all of the methods defined in `java.awt.event.MouseListener` as no-ops. Instead of implementing the `MouseListener` interface, it's preferable to extend the `MouseListener` class.

Example 9-2 lists an applet equipped with a single button. The button has a `ButtonMouseListener` added to it. `ButtonMouseListener` is only interested in mouse enter and exit events; it implements no-op versions of the other methods defined by `java.awt.event.MouseListener`.



Example 9-2 ButtonTest Applet—Implementing the `MouseListener` Interface

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

public class ButtonTest extends Applet {
    public void init() {
        Button button = new Button("Press Me");
        button.addMouseListener(new ButtonMouseListener());
        add(button);
    }
}

class ButtonMouseListener implements MouseListener {
    public void mouseEntered(MouseEvent event) {
        System.out.println("Mouse Entered Button");
    }
    public void mouseExited(MouseEvent event) {
        System.out.println("Mouse Exited Button");
    }
    public void mousePressed (MouseEvent event) { }
    public void mouseClicked (MouseEvent event) { }
    public void mouseReleased(MouseEvent event) { }
}
```

Notice that `MouseListener` defines methods for a mouse click and a mouse release. The `mouseClicked` method is invoked when the mouse button is released immediately following a mouse press. The `mouseReleased` method is invoked when the mouse button is released following a mouse drag.

As you might well imagine, it could become rather tedious to have to implement all the methods associated with one or more listener interfaces if you are only interested in giving meaningful purpose to a small percentage of the methods.

For instance, you might have a selectable object that you'd like to be selected with a mouse press; if so, you'd want to implement `MouseListener` and override `mousePressed()`. Since an interface is being implemented, however, the other four `MouseListener` methods must be implemented, typically as no-ops, in order for the class to be concrete (nonabstract).

Using adapter classes means that instead of implementing an interface and having to code a handful of no-op methods, we can extend a class full of no-ops



and selectively override the methods that we're interested in. Example 9-3 is functionally identical to Example 9-2, except that `ButtonMouseListener` extends the `MouseAdapter` class instead of implementing the `MouseListener` interface.

Example 9-3 Extending `MouseAdapter` Instead of Implementing `MouseListener`

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

public class ButtonTest2 extends Applet {
    public void init() {
        Button button = new Button("Press Me");
        button.addMouseListener(new ButtonMouseListener());
        add(button);
    }
}

class ButtonMouseListener extends MouseAdapter {
    public void mouseEntered(MouseEvent event) {
        System.out.println("Mouse Entered Button");
    }
    public void mouseExited(MouseEvent event) {
        System.out.println("Mouse Exited Button");
    }
}
```



Table 9-4 lists the adapter classes provided by the AWT, along with the listener interfaces they implement. Notice that there are no adapter classes for `ActionListener`, `AdjustmentListener`, `ItemListener`, and `TextListener` because those interfaces define only one method.

Table 9-4 AWT Adapter Classes

Adapter Class	Implements This Interface ...
<code>ComponentAdapter</code>	<code>ComponentListener</code>
<code>ContainerAdapter</code>	<code>ContainerListener</code>
<code>FocusAdapter</code>	<code>FocusListener</code>
<code>KeyAdapter</code>	<code>KeyListener</code>
<code>MouseAdapter</code>	<code>MouseListener</code>
<code>MouseMotionAdapter</code>	<code>MouseMotionListener</code>
<code>WindowAdapter</code>	<code>WindowListener</code>



Component events for showing or hiding a component are fired only when `Component.setVisible()` (or the deprecated `show()/hide()`) is invoked for a component. For instance, if an applet's window is temporarily hidden behind another window, the components contained in the applet will not fire `ComponentEvents` when the applet is hidden and shown.

The `ComponentListener` and `ContainerListener` interfaces are summarized in Table 9-3 on page 249, and the `ComponentEvent` and `ContainerEvent` classes are summarized in Table 9-1 on page 244.

The applet shown in Figure 9-1 has a `ContainerListener` that prints out container events as they occur. Likewise, the button in the applet has a `ComponentListener` that prints out component events. The two `Choice` components either show/hide the button or add/remove the button from its container (the applet).

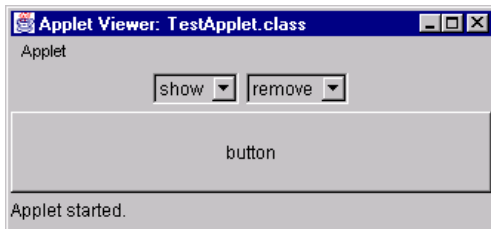
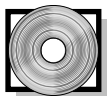


Figure 9-5 Component and Container Events

The applet has a `ContainerListener` added to it, and the button has a `ComponentListener`. Both listeners print events as they occur.

The applet is listed in Example 9-4.



Example 9-4 Handling Component and Container Events

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

public class TestApplet extends Applet {
    public void init() {
        final Button button = new Button("button");
        Choice visible = new Choice(), addRemove = new Choice();
        Panel controls = new Panel();

        visible.add("show");
        visible.add("hide");

        addRemove.add("remove");
        addRemove.add("add");

        controls.add(visible);
        controls.add(addRemove);

        setLayout(new BorderLayout());
        add(button, "Center");
        add(controls, "North");

        button.addComponentListener(new ButtonListener());
        addContainerListener(new AppletListener());

        visible.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                String s =
                    ((Choice)e.getSource()).getSelectedItem();

                if(s.equals("hide")) button.setVisible(false);
                else
                    button.setVisible(true);
            }
        });
        addRemove.addItemListener(new ItemListener() {
            public void itemStateChanged(ItemEvent e) {
                String s =
                    ((Choice)e.getSource()).getSelectedItem();

                if(s.equals("add"))
                    add(button, "Center");
                else
                    remove(button);
            }
        });
    }
}

class ButtonListener implements ComponentListener {
    public void componentResized(ComponentEvent event) {
        Component c = (Component)event.getSource();
        System.out.println("button resized: " + c.getSize());
    }
}
```



```

    }
    public void componentShown(ComponentEvent event) {
        System.out.println("button shown");
    }
    public void componentHidden(ComponentEvent event) {
        System.out.println("button hidden");
    }
    public void componentMoved(ComponentEvent event) {
        Component c = (Component)event.getSource();
        System.out.println("button moved:  " + c.getLocation());
    }
}
class AppletListener implements ContainerListener {
    public void componentAdded(ContainerEvent e) {
        Component c = e.getChild();
        System.out.println("container:  button added");
    }
    public void componentRemoved(ContainerEvent e) {
        Component c = e.getChild();
        System.out.println("container:  button removed");
    }
}

```

When the applet starts, the button is laid out by the applet's layout manager. As a result, the initial output of the applet looks like so:

```

button resized:  java.awt.Dimension[width=325,height=69]
button moved:   java.awt.Point[x=0,y=31]

```

Subsequently, when the button is shown or hidden, the `ButtonListener` is notified. If the button is added to or removed from the applet, the `AppletListener` is notified of the addition or removal.

Focus Events

At any given time, there is a maximum of one component with keyboard focus per windowing system. When a component has focus, keyboard events are delivered to the component in question.

Components with keyboard focus typically have a prominent appearance that distinguishes them from other components. For instance, buttons that have keyboard focus under Windows draw a dashed rectangle around the inside edge of their border (Windows buttons are interested in receiving focus because they can be activated with the spacebar or accelerators).

AWT components gain focus in one of the following ways:

1. Interact with the component
2. Invoke `Component.focusRequested()`
3. Type TAB and SHIFT-TAB to move the keyboard focus

Focus events are qualified as either `FOCUS_LOST` or `FOCUS_GAINED` events. `FOCUS_LOST` events are further qualified as either temporary or permanent.

Permanent focus events are fired when keyboard focus is deliberately given to a component within a given applet or application. The component that originally had focus fires a `FOCUS_LOST` event, followed by a `FOCUS_GAINED` event being fired by the component that gains the focus, in that order.

Temporary `FOCUS_LOST` events are fired when a component temporarily loses focus. For instance, if a component in an applet has focus and another window is activated, the component will temporarily lose focus, but focus will be restored when the applet/application window is subsequently activated.

Focus events are represented by `java.awt.event.FocusEvent`. A `FocusListener` is registered with the component in question; when a focus change occurs, the listener is notified and is passed an instance of `FocusEvent`.

Figure 9-6 shows an applet with two buttons, both of which have an instance of `ButtonFocusListener` added to them. `ButtonFocusListener` prints information about each focus event it receives.

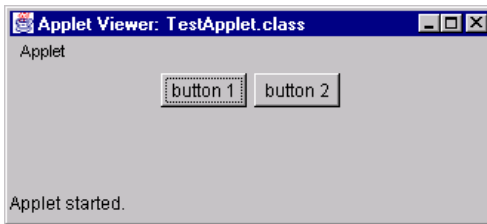
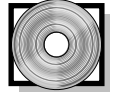


Figure 9-6 Handling Focus Events

Both buttons have a `FocusListener` added to them. The listener prints information about focus events fired by the buttons.

The applet is listed in Example 9-5.

**Example 9-5** Handling Focus Events

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

public class TestApplet extends Applet {
    public void init() {
        Button button = new Button("button 1"),
            button2 = new Button("button 2");

        ButtonFocusListener listener = new ButtonFocusListener();

        button.addFocusListener(listener);
        button2.addFocusListener(listener);

        add(button);
        add(button2);
    }
}

class ButtonFocusListener implements FocusListener {
    public void focusGained(FocusEvent event) {
        report(event);
    }
    public void focusLost(FocusEvent event) {
        report(event);
    }
    private void report(FocusEvent event) {
        Button b = (Button)event.getComponent();

        if(event.getID() == FocusEvent.FOCUS_GAINED)
            System.out.print(b.getLabel() + " gained focus");
        else if(event.getID() == FocusEvent.FOCUS_LOST)
            System.out.print(b.getLabel() + " lost focus");

        if(event.isTemporary())
            System.out.println(": temporary");
        else
            System.out.println();
    }
}
```

When focus is shifted from button 1 to button 2, the applet's output looks like this:

```
button 1 lost focus
button 2 gained focus
```

Subsequently, if another window is activated, button 2 temporarily loses focus:

```
button 2 lost focus: temporary
```



When the applet window is reactivated, button 2 regains focus:

```
button 2 gained focus
```

Key Events

Key events are fired when a key is pressed or released in a component that has keyboard focus. There are three types of key events, all of which are encapsulated in the `java.awt.event.KeyEvent` class. Each type of event is represented by a constant and a corresponding method in the `KeyListener` interface, as listed in Table 9-6.

Table 9-6 Key Events

Event ID	KeyListener Method	Fired When ...
KEY_PRESSED	<code>keyPressed(KeyEvent)</code>	key is pressed
KEY_RELEASED	<code>keyReleased(KeyEvent)</code>	key is released
KEY_TYPED	<code>keyTyped(KeyEvent)</code>	key character is pressed

Figure 9-3 shows an applet that listens for key events fired from a textfield. The `KeyListener` associated with the textfield prints information about each key event that is fired.

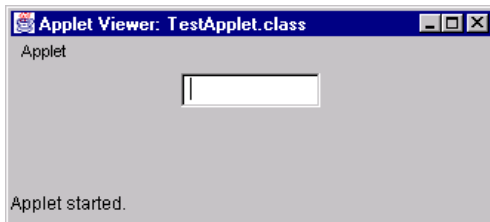
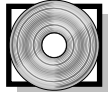


Figure 9-7 Handling Key Events

The textfield has a `KeyListener` that prints information about key events.

The applet shown in Figure 9-7 is listed in Example 9-6.

**Example 9-6** Handling Key Events

```

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class TestApplet extends Applet {
    private TextField tf = new TextField(10);

    public void init() {
        tf.addKeyListener(new TextfieldListener());
        add(tf);
    }
}

class TextfieldListener implements KeyListener {
    public void keyPressed(KeyEvent e) {
        System.out.println("KEY_PRESSED: ");
        report(e);
    }

    public void keyReleased(KeyEvent e) {
        System.out.println("KEY_RELEASED: ");
        report(e);
    }

    public void keyTyped(KeyEvent e) {
        System.out.println("KEY_TYPED: ");
        report(e);
    }

    private void report(KeyEvent e) {
        int keyCode = e.getKeyCode();
        char keyChar = e.getKeyChar();
        String mods = e.getKeyModifiersText(keyCode);
        String txt = e.getKeyText(keyCode);

        if(keyCode != KeyEvent.KEY_UNDEFINED)
            System.out.println("Code: " + keyCode);

        if(keyCode != KeyEvent.CHAR_UNDEFINED)
            System.out.println("Char: " + keyChar);

        System.out.println("Modifiers: " + mods);
        System.out.println("Text: " + txt);

        if(e.isActionKey())
            System.out.println("ACTION");

        System.out.println();
    }
}

```



KEY_PRESSED and **KEY_RELEASED** events are fired when any key is pressed or released in a component that has focus. Each key on the keyboard is represented by a key code; `KeyEvent.getKeyCode()`⁷ can be used to find out which key was pressed or released. Additionally, `KeyEvent.getKeyText()` returns a string representation of the key. For example, if the F1 key is pressed and subsequently released while the textfield in the applet listed in Example 9-6 has focus, the output is as follows:

```
KEY_PRESSED:  
Code: 112  
Modifiers:  
Text: F1  
ACTION
```

```
KEY_RELEASED:  
Code: 112  
Modifiers:  
Text: F1  
ACTION
```

The F1 key has a keycode of 112, and `KeyEvent.getKeyText()` returns the string “F1” when the F1 key is pressed or released.

KEY_TYPED events are fired when a key representing a valid unicode character is pressed. For example, if the ‘g’ key is pressed when the textfield in the applet listed in Example 9-6 has focus, the output is as follows:

```
KEY_PRESSED:  
Code: 71  
Char: g  
Modifiers: Meta+Ctrl+Shift  
Text: G
```

```
KEY_TYPED:  
Code: 0  
Char: g  
Modifiers:  
Text: Unknown keyCode: 0x0
```

```
KEY_RELEASED:  
Code: 71  
Char: g  
Modifiers: Meta+Ctrl+Shift  
Text: G
```

7. `java.awt.event.KeyEvent` methods are listed in Table 9-3 on page 249.



Key events, like mouse events, are input events and therefore may be consumed—see “Consuming Input Events” on page 267. Additionally, `KeyEvent` is the only event class that is not immutable;⁸ `KeyEvent` provides methods for setting the key code, character, and modifiers.

Under the (old) inheritance-based delegation model, key presses other than unicode characters were labeled action events. The `KeyEvent` class provides a corresponding `isAction` method to determine if the key pressed represents a valid unicode character.

Mouse and Mouse Motion Events.

The delegation event model—unlike the inheritance event model—distinguishes between mouse events and mouse motion events by providing two listeners: `MouseListener` and `MouseMotionListener`. Both mouse and mouse motion events are represented by the same event class: `MouseEvent`.

Mouse moved and mouse dragged events are mouse motion events, whereas all other mouse events (enter/exit, pressed/released, and clicked) are simply mouse events. A mouse clicked event is fired when a mouse pressed is immediately followed by a mouse released event (without mouse drags in between).

Mouse events, like key events, are input events and therefore may be consumed—see “Consuming Input Events” on page 267.

The applet listed in Example 9-7 adds a mouse listener and mouse motion listener to the applet itself. The listeners print events as they occur.

Example 9-7 Handling Mouse Events



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

public class TestApplet extends Applet {
    public void init() {
        addMouseListener(new TestMouseListener());
        addMouseMotionListener(new TestMouseMotionListener());
    }
}

class MouseReporter {
    public void report(MouseEvent e) {
        int clickCount = e.getClickCount();
        int mods      = e.getModifiers();
        Point p       = e.getPoint();
        boolean isPopupTrigger = e.isPopupTrigger();
```

8. `PaintEvent` provides a method for setting the update rectangle.



```
String s          = "mouse ";

if((mods & InputEvent.BUTTON3_MASK) != 0)
    s += "button 3";
else if((mods & InputEvent.BUTTON2_MASK) != 0)
    s += "button 2";
else if((mods & InputEvent.BUTTON1_MASK) != 0)
    s += "button 1";
else
    s += "cursor";

switch(e.getID()) {
    case MouseEvent.MOUSE_PRESSED:
        s += " pressed";
        break;
    case MouseEvent.MOUSE_RELEASED:
        s += " released";
        break;
    case MouseEvent.MOUSE_CLICKED:
        s += " clicked";
        break;
    case MouseEvent.MOUSE_MOVED:
        s += " moved";
        break;
    case MouseEvent.MOUSE_ENTERED:
        s += " entered";
        break;
    case MouseEvent.MOUSE_EXITED:
        s += " exited";
        break;
    case MouseEvent.MOUSE_DRAGGED:
        s += " dragged";
        break;
}
System.out.println(s + " at: " + p);
System.out.println(" click count: " + clickCount);
System.out.println(" is popup trigger: " +
                    isPopupTrigger);
System.out.println();
}
}

class TestMouseListener implements MouseListener {
    private MouseReporter reporter = new MouseReporter();

    public void mouseClicked(MouseEvent e) {
        reporter.report(e);
    }
    public void mouseEntered(MouseEvent e) {
        reporter.report(e);
    }
    public void mouseExited(MouseEvent e) {
        reporter.report(e);
    }
}
```



```

    }
    public void mousePressed(MouseEvent e) {
        reporter.report(e);
    }
    public void mouseReleased(MouseEvent e) {
        reporter.report(e);
    }
}
class TestMouseMotionListener implements MouseMotionListener {
    private MouseReporter reporter = new MouseReporter();

    public void mouseDragged(MouseEvent e) {
        reporter.report(e);
    }
    public void mouseMoved(MouseEvent e) {
        reporter.report(e);
    }
}
}

```

The `MouseEvent` class provides methods that can be used to determine the position of the mouse at the time of the event (`getPoint()`, `getX()`, and `getY()`). Additionally, `MouseEvent.getClickCount()` is used to determine if the mouse was double clicked. If the value returned from `getClickCount()` is 2, then the mouse event represents a double click.

Information pertaining to which mouse button was pressed/released or clicked is stored in the `modifiers` field of the event, which is accessible via `InputEvent.getModifiers()`. Since the AWT must be able to deal with mice with differing numbers of buttons, mouse buttons are simulated with modifier keys, as listed in Table 9-7.

Table 9-7 Modifier Keys for Simulating Multiple-Button Mice

Modifier	Button Simulated
NONE	Button 1
ALT	Button 2
META	Button 3

An unfortunate side effect of using modifiers to simulate multiple buttons is that it is not possible to distinguish between a button 1 + modifier click from a button other than button 1. For instance, it is not possible to distinguish between a button 1 + ALT versus button 2.

Window Events

Window events are fired only by instances of `java.awt.Window` (and its extensions) and signify that the window has been activated/deactivated, iconified/deiconified, opened/closed, or is in the process of closing.



Example 9-8 lists an applet that listens for events in an instance of `java.awt.Frame`.



Example 9-8 Handling Window Events

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

public class Test extends Frame {
    public Test() {
        super("WindowListener test");
    }
    public static void main(String args[]) {
        final Frame f = new Test();
        f.setBounds(100,100,250,150);
        f.setVisible(true);
        f.addWindowListener(new TestWindowListener());
    }
}

class TestWindowListener implements WindowListener {
    public void windowActivated(WindowEvent e) {
        System.out.println("window activated");
    }
    public void windowClosed(WindowEvent e) {
        System.out.println("window closed");
        System.exit(0);
    }
    public void windowClosing(WindowEvent e) {
        System.out.println("window closing ...");
        Window w = e.getWindow();
        w.dispose();
    }
    public void windowDeactivated(WindowEvent e) {
        System.out.println("window deactivated");
    }
    public void windowDeiconified(WindowEvent e) {
        System.out.println("window deiconified");
    }
    public void windowOpened(WindowEvent e) {
        System.out.println("window opened");
    }
}
```

The application listed in Example 9-8 invokes `Window.dispose()` to close the window when a window closing event is detected. See “`java.awt.Frame`” on page 579 for more on reacting to window closing events.

Focus and Key Events for Canvases and Panels

Although all components are capable of firing focus and key events, a component must have keyboard focus in order to do so. Canvases and panels, by default, are not typically⁹ interested in obtaining keyboard focus and therefore will not fire



key or focus events. To handle key and/or focus events for a canvas or panel, you must invoke `requestFocus()` on the canvas or panel in question. Once the canvas or panel has keyboard focus, it will fire key and focus events.

Consuming Input Events

There are times when it would be convenient to block an event from making its way to a component's peer. For instance, GUI builders often have a build mode and a test mode. In build mode, it may be desirable to suppress the usual reaction to button clicks or keystrokes and instead bring up a property sheet, while the component should be fully functional in test mode.

The delegation event model allows the consumption of input events (meaning key and mouse events); once consumed, an event is not passed to its native peer. The `InputEvent` class provides a `consume` method, which causes the event in question to be consumed.

Figure 9-8 shows an applet containing a lone button. The button has a mouse listener added to it that consumes mouse pressed events; as a result, the button is not activated when clicked. Additionally, the applet has a key listener attached to it that consumes 'a' key presses.



Figure 9-8 Consuming Input Events

The applet consumes 'a' key presses, and the button consumes mouse pressed events.

The applet is listed in Example 9-9.

9. Whether a component is interested in gaining focus is platform dependent.

**Example 9-9** Consuming Input Events

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

public class ConsumeExample extends Applet {
    public void init() {
        Button button = new Button("Can't Click This");
        button.addMouseListener(new ConsumeButtonListener(this));

        addKeyListener(new ConsumeKeyListener(this));
        requestFocus();

        add(button);
    }
}

class ConsumeButtonListener extends MouseAdapter {
    private Applet applet;

    public ConsumeButtonListener(Applet applet) {
        this.applet = applet;
    }

    public void mousePressed(MouseEvent event) {
        applet.showStatus("Consuming button press");
        event.consume();
    }
}

class ConsumeKeyListener extends KeyAdapter {
    private Applet applet;

    public ConsumeKeyListener(Applet applet) {
        this.applet = applet;
    }

    public void keyPressed(KeyEvent event) {
        char key = event.getKeyChar();

        if(key == 'a') {
            applet.showStatus("Consuming 'a' key");
            event.consume();
        }
    }
}
```

Unlike listeners of most events, listeners of input events are notified of the event before it is passed on to the component's peer. If the event is consumed via `InputEvent.consume()`, the method is not passed on to the component's peer but is still passed on to all registered listeners (of the appropriate type, of course). For a button, for instance, this means that consumed mouse pressed events suppress the usual reaction to mouse presses, but the mouse pressed event is still sent to all of the button's registered mouse listeners.



Paint Events

Paint events are the odd man out as far as AWT events are concerned because they are not handled via the listener model. Paint events are handled internally and wind up resulting in calls to a component's `update` or `paint` methods. To handle paint events, a component's paint method is typically overridden in order to (re)paint the component. See “Rendering Components” on page 443 for more on painting and repainting components.

Semantic Events

The majority of AWT events are component events, such as mouse down, key pressed, etc. The AWT also provides *semantic* events, which are higher-level events, such as an action event. Semantic events do not equate to any single component event but describe an event that may consist of a number of component events. Semantic events, in contrast to component events, are always consumed—semantic events are not passed to a component's peer.

For instance, if you wheel the mouse (mouse moved) into a `java.awt.Button` (mouse entered), press the mouse button (mouse pressed), and release it inside of the button (mouse clicked), the button will fire an action event.

Semantic events are handled in exactly the same manner as component inputs; there are four semantic events in `java.awt.event: ActionEvent`, `AdjustmentEvent`, `ItemEvent`, and `TextEvent` as described in Table 9-8.

Table 9-8 Semantic Events

Semantic Events ...	are fired by ...	when ...
<code>ActionEvent</code>	<code>Button</code>	the button is activated
	<code>List</code>	the item is double-clicked
	<code>MenuItem</code>	the item is selected
	<code>TextField</code>	enter is typed in the field
<code>AdjustmentEvent</code>	<code>Scrollbar</code>	the thumb is moved
<code>ItemEvent</code>	<code>Checkbox</code>	the checkbox is toggled
	<code>CheckboxMenuItem</code>	the menu item is selected
	<code>Choice</code>	an item is selected
	<code>List</code>	an item is selected
<code>TextEvent</code>	<code>TextComponent</code>	the text changes

An `AdjustmentEvent` is fired only by components that can be adjusted—in the current release of the AWT, that would be all classes that implement the `Adjustable` interface, namely, `java.awt.Scrollbar`.¹⁰



Item events are fired by components that have items: `Checkbox`, `CheckboxMenuItem`, `Choice`, and `List`.

Text events are fired by components that have editable text, which means anything that extends `TextComponent`, namely, `TextField` and `TextArea`.

Which components fire action events is not so intuitive—`Button`, `List`, `MenuItem`, and `TextField` all fire action events. Note that `List` and `TextField` fire two different kinds of semantic events: a `List` fires item events when an item in the list is selected, and an action event when an item in the list is double-clicked. A `TextField` fires action events when enter is typed in the field and fires text events whenever its text is modified.

AWT TIP

Have Custom Components Fire Semantic Events When Appropriate

Custom components should fire semantic events when appropriate. The exact definitions of semantic events are intentionally left vague—an action event, for example, means different things depending upon the type of component that fires the event. For instance, image buttons should fire action events in a manner similar to the action events fired by a `java.awt.Button`, while custom components with selectable items should fire item events. Components that are adjustable should fire adjustment events, and components with editable text should fire text events.

Action Events

Figure 9-9 shows an application that handles action events for all of the components that fire them (see Table 9-8 on page 269). Two listeners are implemented: `ActionWindowListener`, which listens for the closing of the window, and `DebugActionListener`, which simply prints out information about action events as they are fired. Instances of `DebugActionListener` are added to each component, and they add an instance of `ActionWindowListener` is added to the frame itself. From there on out, the AWT machinery takes care of business for us.

10. Actually, a `java.awt.package` private class, `ScrollPaneAdjustable` also implements `Adjustable`.

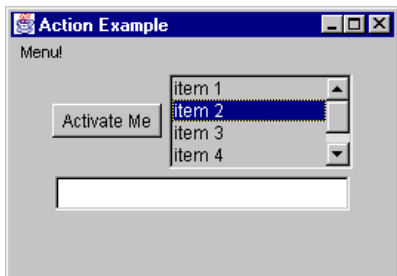
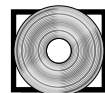


Figure 9-9 Action Events

Action events are generated by double-clicking list items, activating buttons, and entering a carriage return in a textfield or selecting a menu item.

The application is listed in Example 9-10.

Example 9-10 ActionExample2 Application



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

public class ActionExample2 extends Frame {
    private Button    button    = new Button("Activate Me");
    private List      list      = new List();
    private TextField textfield = new TextField(25);
    private MenuItem  menuItem  = new MenuItem("Menu menuItem");

    static public void main(String args[]) {
        ActionExample2 f = new ActionExample2();
        f.setBounds(200,200,200,200);
        f.show();
    }
    public ActionExample2() {
        super("Action Example");
        MenuBar mbar = new MenuBar();
        Menu    menu = new Menu("Menu!");
        menu.add(menuItem);
        mbar.add(menu);
        setMenuBar(mbar);

        list.add("item 1");
        list.add("item 2");
        list.add("item 3");
        list.add("item 4");
        list.add("item 5");
    }
}
```

```

        setLayout(new FlowLayout());
        add(button);
        add(list);
        add(textfield);

        button.addActionListener (new DebugActionListener());
        list.addActionListener (new DebugActionListener());
        textfield.addActionListener(new DebugActionListener());
        menuItem.addActionListener (new DebugActionListener());

        addWindowListener(new ActionWindowListener());
    }
}

class ActionWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent event) {
        Window window = (Window)event.getSource();
        window.dispose();
        System.exit(0);
    }
}

class DebugActionListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        System.out.println("action event in: " +
            event.getActionCommand());
    }
}

```

The `DebugActionListener` in Example 9-10 identifies the component that fired the action event by invoking `ActionEvent.getActionCommand()`. The `getActionCommand` method returns a string; for buttons and menu items, the string is the label of the button or menu item. For lists, the string returned is the item that was double-clicked, whereas for textfields, the string contains the text that resided in the textfield when the event was fired.

Adjustment Events

Adjustment events are fired by classes that implement the `Adjustable` interface. Adjustables, as you can surmise from Table 9-9, are objects that maintain an integer value that can be adjusted between settable minimum and maximum values. Adjustables can have their values adjusted by incrementing or decrementing their unit and block increments or by having their value set directly. The exact meaning of unit and block depends on the object that implements the `Adjustable` interface, but typically a block is defined to be a certain number of units. For example, a text editor would probably define a unit to be a single line of text, and a block would correspond to a page.



The current version of the AWT contains only one public class that implements the `Adjustable` interface—`java.awt.Scrollbar`. Scrollbars can be incremented or decremented by their unit value by activation of the scrollbar arrows. They can also be incremented or decremented by their block value if clicked anywhere in the scrollbar outside of the arrows or the slider (thumb) of the scrollbar. The visible value of a scrollbar corresponds to the width of the slider of the scrollbar and is settable. (See “Applets and Applications” on page 15.)

Table 9-9 Adjustable Interface

Method	Intent
<code>void setMinimum(int)</code>	Sets minimum value
<code>void setMaximum(int)</code>	Sets maximum value
<code>void setUnitIncrement(int)</code>	Sets the unit increment—typically the smallest meaningful increment
<code>void setBlockIncrement(int)</code>	Sets the block increment—typically defined as a number of units
<code>void setVisibleAmount(int)</code>	Sets the length of proportional indicator
<code>void setValue(int)</code>	Sets the current value of adjustable
<code>int getOrientation()</code>	Returns either <code>Adjustable.HORIZONTAL</code> or <code>Adjustable.VERTICAL</code>
<code>int getMinimum()</code>	Returns the minimum value the adjustable can take on
<code>int getMaximum()</code>	Returns the maximum value the adjustable can take on
<code>int getUnitIncrement()</code>	Returns the unit increment
<code>int getBlockIncrement()</code>	Returns the block increment
<code>int getValue()</code>	Returns the current value
<code>int getVisibleAmount()</code>	Returns length of proportional indicator
<code>void addAdjustmentListener(AdjustmentListener)</code>	Adds an adjustment listener to adjustable
<code>void removeAdjustmentListener(AdjustmentListener)</code>	Removes an adjustment listener to adjustable



`java.awt.ScrollPane` contains two adjustables,¹¹ to which you can obtain a reference by invoking `ScrollPane.getHAdjustable()` and `ScrollPane.getVAdjustable()`.

Note that while a scrollpane's adjustables are actually scrollbars, the methods used to access the scrollbars return references to an `Adjustable`, not a `Scrollbar`. The reason for this is twofold. First, `Scrollbar` adds only one public method to the `Adjustable` interface:¹² `setOrientation()`. If `ScrollPane` were to return a reference to a `Scrollbar` instead of an `Adjustable`, the orientation of its scrollbars could be modified, which would be highly undesirable. Also, by returning a reference to an `Adjustable` instead of a `Scrollbar`, `ScrollPane` hides the implementation of its adjustables, which leaves it free to implement its adjustable components using something other than a scrollbar in the future or on different platforms.

The applet pictured in Figure 9-10 contains a scrollbar and a scrollpane and monitors the firing of adjustment events. The applet is listed in Example 9-11. Whenever the slider of a scrollbar is moved, adjustment events are fired, which we listen for and print out in the `adjustmentValueChanged` method for our `DebugAdjustmentListener`.

OO TIP

Hide Implementations of Enclosed Objects by Returning References to Interfaces

It is often the case that objects contain other objects—for instance, a `ScrollPane` contains two scrollbars for scrolling the contents of the scrollpane. Furthermore, it is sometimes desirable to provide accessors to contained objects. In such a case, it is generally preferable to return a reference to an interface that the enclosed objects implement instead of a reference to the actual class of the enclosed object. Doing so hides the actual implementation of the enclosed objects, which affords the enclosing object the freedom to change the actual class of the enclosed objects (as long as the class implements the interface returned).

11. Which may or may not be visible depending upon what is being scrolled and the scrollbar display policy of the scrollpane.
12. Excluding constructors.

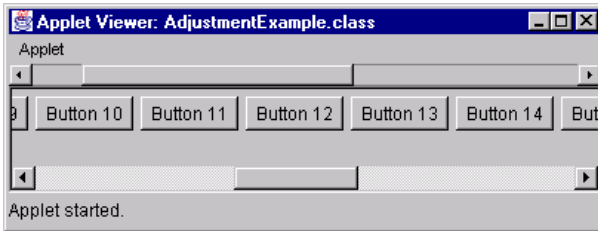
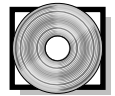


Figure 9-10 Adjustment Events

Adjustment events are fired by the `java.awt.Scrollbar` component, the only adjustable component in the AWT. The bottom scrollbar is part of a scrollpane that scrolls 25 cleverly labeled Buttons. The top scrollbar is a loner.

Example 9-11 AdjustmentExample Applet



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

public class AdjustmentExample extends Applet {
    private ScrollPane scroller = new ScrollPane();
    private Scrollbar sbar = new Scrollbar(Scrollbar.HORIZONTAL);

    public void init() {
        setLayout(new BorderLayout());
        sbar.setValues(0, // value
                     50, // visible
                     0, // minimum
                     100 // maximum
                    );
        sbar.setUnitIncrement (10);
        sbar.setBlockIncrement(20);
        add(sbar, "North");

        scroller.add(new ScrollMe(), 0);
        add(scroller, "Center");

        sbar.addAdjustmentListener(
            new DebugAdjustmentListener());

        scroller.getHAdjustable().addAdjustmentListener(
            new DebugAdjustmentListener());
        scroller.getVAdjustable().addAdjustmentListener(
            new DebugAdjustmentListener());
    }
}
```



```
class ScrollMe extends Panel {
    public ScrollMe() {
        for(int i=0; i < 25; ++i)
            add(new Button("Button " + i));
    }
}
class DebugAdjustmentListener implements AdjustmentListener {
    public void adjustmentValueChanged(AdjustmentEvent event) {
        Object obj = event.getSource();
        System.out.println(obj.toString());
    }
}
```

Item Events

Our next semantic event application deals with item events. An `ItemEvent` is fired any time you select or deselect an item in an item selectable component—see “Item Selectables: Checkboxes, Choices, and Lists” on page 493. As you can see from Table 9-8 on page 269, item events are fired by the AWT components that contain items: `Checkbox`, `CheckboxMenuItem`, `Choice`, and `List`. Figure 9-11 shows our `ItemExample` applet in action; Example 9-12 shows the source for our applet.

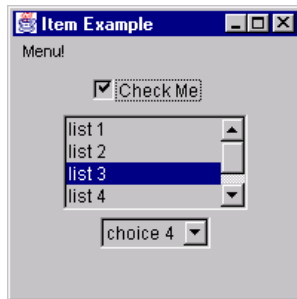
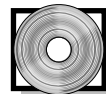


Figure 9-11 Item Events

`ItemEvents` are fired by components that contain items: `Checkbox`, `CheckboxMenuItem`, `Choice`, and `List`.

**Example 9-12** ItemExample Applet

```

import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class ItemExample extends Frame {
    private Checkbox  cbox      = new Checkbox("Check Me");
    private Choice    choice    = new Choice();
    private List      list      = new List();
    private CheckboxMenuItem menuItem =
        new CheckboxMenuItem("Menu menuItem");

    static public void main(String args[]) {
        ItemExample f = new ItemExample();
        f.setBounds(200,200,200,200);
        f.show();
    }
    public ItemExample() {
        super("Item Example");
        MenuBar mbar = new MenuBar();
        Menu menu = new Menu("Menu!");
        menu.add(menuItem);
        mbar.add(menu);
        setMenuBar(mbar);

        list.add("list 1");
        list.add("list 2");
        list.add("list 3");
        list.add("list 4");
        list.add("list 5");

        choice.add("choice 1");
        choice.add("choice 2");
        choice.add("choice 3");
        choice.add("choice 4");
        choice.add("choice 5");

        setLayout(new FlowLayout());
        add(cbox);
        add(list);
        add(choice);

        cbox.addItemListener  (new DebugItemListener());
        list.addItemListener  (new DebugItemListener());
        choice.addItemListener (new DebugItemListener());
        menuItem.addItemListener(new DebugItemListener());

        addWindowListener(new ItemWindowListener());
    }
}

```

```
class ItemWindowListener extends WindowAdapter {
    public void windowClosing(WindowEvent event) {
        Window window = (Window)event.getSource();
        window.dispose();
        System.exit(0);
    }
}

class DebugItemListener implements ItemListener {
    public void itemStateChanged(ItemEvent event) {
        Object obj = event.getSource();
        System.out.println(obj.toString());
    }
}
```

Text Events

Our final semantic event applet deals with text events. A `TextEvent` is fired by components that have editable text—`TextField` and `TextArea`, both of which extend `TextComponent`. A `TextEvent` is fired any time a text component's editable text is modified. You can see our applet in action in Figure 9-12—the corresponding source for the applet is listed in Example 9-13.

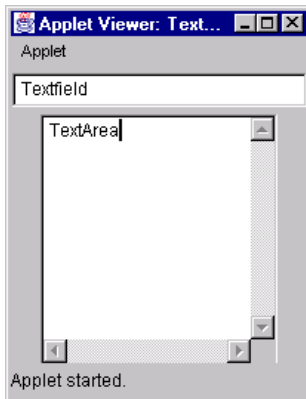
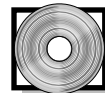


Figure 9-12 Text Events

A `TextEvent` is fired by components that contain editable text: `TextField` and `TextArea` (extensions of `TextComponent`).

**Example 9-13** TextExample Applet

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

public class TextExample extends Applet {
    private TextField textField = new TextField(25);
    private TextArea  textArea  = new TextArea(10, 20);

    public void init() {
        add(textField);
        add(textArea);

        textField.addTextListener(new DebugTextListener());
        textArea.addTextListener(new DebugTextListener());
    }
}

class DebugTextListener implements TextListener {
    public void textValueChanged(TextEvent event) {
        Object obj = event.getSource();
        System.out.println(obj.toString());
    }
}
```

Event Adapters

Event adapters are objects that are interposed between event sources and event listeners. Adapters implement an event listener interface and decouple one or more event sources from one or more event listeners.

Figure 9-13 illustrates the relationships between an event source, adapter, and listener. The adapter class stands in as a listener for a particular event by implementing an appropriate listener interface. When the adapter receives an event, it forwards it to the listener. The adapter may choose to act upon the event either before the event is forwarded to the listener or afterwards. Some adapters may queue events for pending delivery, for instance, whereas others may filter events that are passed on to the listener.

The relationship between adapters and listeners may vary. For instance, Figure 9-13 depicts a delegation relationship between the event adapter and the event listener. The adapter maintains a reference to the listener and invokes listener methods when it receives an event—the adapter *delegates to* the listener. On the other hand, AWT adapters—“AWT Adapters” on page 251—are related to their listeners through inheritance; the adapter *is the superclass of* the listener.

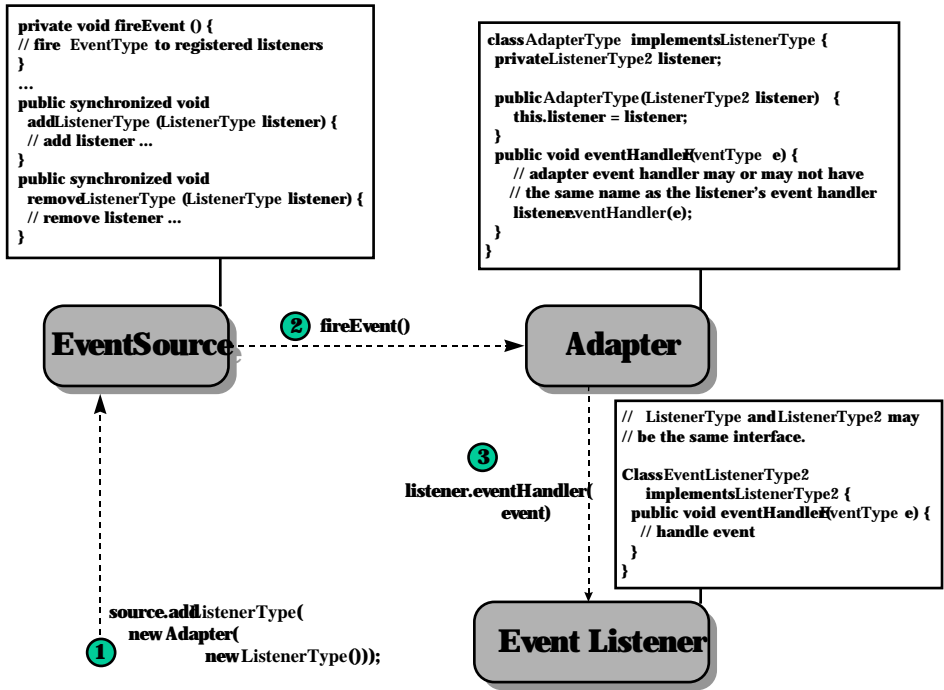


Figure 9-13 Event Adapters

Adapters are interposed between an event source and a listener. Adapters perform services such as filtering, queuing, or routing events.

AWT TIP ...

AWT Adapters Are Their Listener's Superclass

AWT event adapters are related to their listeners through inheritance, unlike the relationship depicted in Figure 9-13.

Listeners extend AWT adapter classes, and therefore the adapter is the superclass of the listener. Instead of manually forwarding events from the adapter to the listener, the listener overrides the AWT adapter's no-op methods.



Handling Events from Multiple Sources Without Adapters

It is not uncommon for a single event listener to register with more than one event source for a single type of event. Since a listener can implement a given listener interface only once, the listener must determine the source of the event and take appropriate action. For example, the `YesNoDialog` class listed in Example 9-14 implements the `ActionListener` interface and registers itself as an `ActionListener` with the two buttons it contains.

Example 9-14 YesNoDialog Without Adapters



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

class YesNoDialog extends Dialog implements ActionListener {
    private Button  yesButton   = new Button("Yes");
    private Button  noButton    = new Button("No");
    private boolean answer;

    public YesNoDialog(String title, String message,
                       boolean isModal) {
        super(new Frame(), title, isModal);

        Panel  buttonPanel = new Panel();
        Panel  labelPanel  = new Panel();

        buttonPanel.add(yesButton);
        buttonPanel.add(noButton);

        labelPanel.add(new Label(message));

        add(labelPanel, "Center");
        add(buttonPanel, "South");

        yesButton.addActionListener(this);
        noButton.addActionListener(this);

        pack();
    }
    public boolean getAnswer() {
        return answer;
    }
    public void actionPerformed(ActionEvent event) {
        Button button = (Button)event.getSource();

        if(button == yesButton)
            yesButtonActivated(event);
        else if(button == noButton)
            noButtonActivated(event);
    }
    public void yesButtonActivated(ActionEvent event) {
        answer = true;
        dispose();
    }
}
```



```
    }  
    public void noButtonActivated(ActionEvent event) {  
        answer = false;  
        dispose();  
    }  
}
```

The dialog implements the `ActionListener` interface and registers itself as an `ActionListener` for both of the dialog's buttons:

```
class YesNoDialog extends Dialog implements ActionListener {  
    ...  
    public YesNoDialog(String title,  
                        String message, boolean isModal) {  
        ...  
        yesButton.addActionListener(this);  
        noButton.addActionListener(this);  
    }  
    ...  
}
```

The dialog's `actionPerformed` method obtains a reference to the event source by invoking `java.util.EventObject.getSource()` and then invokes the appropriate method, depending upon whether the event source is the `yesButton` or the `noButton`.

```
    ...  
    public void actionPerformed(ActionEvent event) {  
        Button button = (Button)event.getSource();  
  
        if(button == yesButton)  
            yesButtonActivated(event);  
        else if(button == noButton)  
            noButtonActivated(event);  
    }  
    public void yesButtonActivated(ActionEvent event) {  
        answer = true;  
        dispose();  
    }  
    public void noButtonActivated(ActionEvent event) {  
        answer = false;  
        dispose();  
    }  
    ...  
}
```

For a simple class such as the dialog listed above, routing events through the `actionPerformed` method is nothing more than a minor inconvenience. However, for more complex listeners, code that routes event handling can become a significant maintenance burden. In addition, implementing a method by



switching off an event type is unnatural from an object-oriented perspective. Object-oriented languages provide polymorphism to eliminate switching off object types. Adapters can provide a similar effect for events.

Figure 9-14 shows a simple applet that displays a `YesNoDialog`. The code for the applet follows the figure.

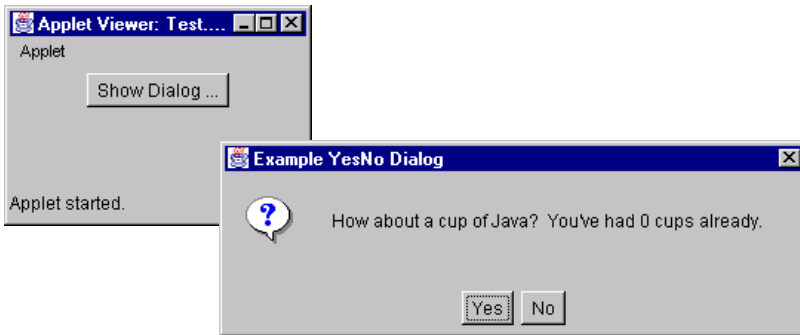


Figure 9-14 `YesNoDialog`
The test applet for the `YesNoDialog` shown above is listed in Example 9-15.

Example 9-15 An Applet That Uses a `YesNoDialog`

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

public class Test extends Applet {
    YesNoDialog dialog = new YesNoDialog("Yes/No Dialog",
                                         "Do you use adapters?",
                                         true); // true means modal

    Button launchButton = new Button("Show Dialog ...");

    public void init() {
        add(launchButton);

        launchButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                Point loc = launchButton.getLocationOnScreen();
                dialog.setLocation(loc.x + 10, loc.y + 10);
            }
        });
    }
}
```





```
        dialog.show();

        if(dialog.getAnswer())
            showStatus("Yes");
        else
            showStatus("No");
    }
}
}
```

Type-Safe Demultiplexing Adapters

Without an adapter, the listener is responsible for invoking an appropriate method based on the identity of the event source; for instance, the `YesNoDialog` in Example 9-14 on page 281 decides whether to invoke `yesButtonActivated` or `noButtonActivated`. Demultiplexing adapters assume that responsibility, resulting in a decoupling between event sources and event listeners.

`YesAdapter` and `NoAdapter`, listed below, decouple the event listener—the dialog—from the event sources, that is, the yes and no buttons.

```
class YesAdapter implements ActionListener {
    YesNoDialog target;

    public YesAdapter(YesNoDialog dialog) {
        target = dialog;
    }
    public void actionPerformed(ActionEvent event) {
        target.yesButtonActivated(event);
    }
}
class NoAdapter implements ActionListener {
    YesNoDialog target;

    public NoAdapter(YesNoDialog dialog) {
        target = dialog;
    }
    public void actionPerformed(ActionEvent event) {
        target.noButtonActivated(event);
    }
}
```

Both adapter classes implement the `ActionListener` interface and are constructed with a reference to the `YesNoDialog`. The `actionPerformed` methods for each adapter class map the event to the appropriate method in the `YesNoDialog` class, thus decoupling the dialog from the buttons.

`YesNoDialog` creates instances of `YesAdapter` and `NoAdapter`, as listed in Example 9-16.

**Example 9-16** YesNoDialog with Type-Safe Adapters

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

class YesNoDialog extends Dialog {
    private Button yesButton    = new Button("Yes");
    private Button noButton     = new Button("No");
    private boolean answer;

    public YesNoDialog(String title, String message,
                       boolean isModal) {
        super(new Frame(), title, isModal);

        Panel buttonPanel = new Panel();
        Panel labelPanel  = new Panel();

        buttonPanel.add(yesButton);
        buttonPanel.add(noButton);

        labelPanel.add(new Label(message));

        add(labelPanel, "Center");
        add(buttonPanel, "South");

        yesButton.addActionListener(new YesAdapter(this));
        noButton.addActionListener(new NoAdapter(this));

        pack();
    }
    public boolean getAnswer() {
        return answer;
    }
    public void yesButtonActivated(ActionEvent event) {
        answer = true;
        dispose();
    }
    public void noButtonActivated(ActionEvent event) {
        answer = false;
        dispose();
    }
}

```

The instances of `YesAdapter` and `NoAdapter` are specified as listeners for the `yes` and `no` buttons. When one of the buttons is activated, an action event is fired to one of the adapters, which is mapped onto the appropriate `YesNoDialog` method. Use of adapters eliminates the event routing code from the previous version of `YesNoDialog`.

Notice that `YesAdapter` and `NoAdapter` are type-safe. The compiler attempts to validate that `yesButtonActivated` and `noButtonActivated` are valid methods for the `YesNoDialog` class. It is noteworthy because our next discussion concerns generic adapters that are not type-safe.

Generic Demultiplexing Adapters

Adapters, by design, require one adapter class per listener *method* invoked, which can result in a high number of adapter classes in practice. For instance, in the previous example `YesNoDialog` employed two types of adapters for two methods that handled the yes/no buttons.

Generic adapters, on the other hand, require one adapter class per listener *class*, which can result in a considerable reduction in the number of adapter classes. Such a reduction is not without penalty; generic adapters are not type-safe. Generic adapters employ the Java reflection API to invoke a method on an object at *runtime*. A single generic adapter can be used to invoke any method on any listener, anytime, anywhere.¹³

The drawback to generic adapters is that using them requires circumventing compile-time checking—type checking is handled entirely at runtime—which is no small transgression in a strongly typed language such as Java.

All other things being equal, it is better to check code at compile time than to defer checking until runtime. However, all other things are not equal in this case. Ultimately the developer is left to choose between a potentially high number of type-safe adapters or a relatively low number of type-unsafe generic adapters.

`YesNoDialog` is modified below to use two `GenericActionAdapters`.

```
// code fragment

class YesNoDialog extends Dialog {
    ...
    public void YesNoDialog(String title, String message) {
        ...
        yesButton.addActionListener(
            new GenericActionAdapter(this,
                                    "yesButtonActivated"));
        noButton.addActionListener(
            new GenericActionAdapter(this,
                                    "noButtonActivated"));
        ...
    }
}
```

13. This was not possible prior to the 1.1 JDK.



The `GenericActionAdapter` constructor takes two arguments: an `Object` that is assumed to be the listener and a `String` representing the *name of the method* to be invoked on the listener. `GenericActionAdapter` uses Java reflection to invoke the named method on the specified listener. In other words, given a reference to a listener and the name of one of the listener's methods, the method is invoked on behalf of the listener. Here's how:

1. The listener's class is obtained via a call to `Object.getClass()`.
2. The `Class` instance is used to obtain a reference to a `Method` representing the listener's method. `Method Class.getMethod(String, Class[])` is passed the name of the method and an array of `Class` instances representing the types of the arguments to the method.
3. Once the method is in hand, it is invoked by calling its `invoke` method.

The `Method` reference is obtained in the `GenericActionAdapter` constructor:

```
public class GenericActionAdapter
    implements ActionListener {
    ...
    public Class[] classTypes = { ActionEvent.class };
    ...

    public GenericActionAdapter(Object listener,
                               String methodName) {
        ...
        method = listener.getClass().getMethod(methodName,
                                                classTypes);
        ...
    }
    ...
}
```

Both event handling methods in the `YesNoDialog` take a single event of type `ActionEvent`; thus, the `classTypes` array passed to `Class.getMethod()`. Once the `Method` is in hand, all that's left is to invoke it on the listener.

```
public void actionPerformed(ActionEvent event) {
    args[0] = event;
    try {
        method.invoke(listener, args);
    }
    ...
}
```

Generic adapters should be careful to handle all possible types of exceptions that can be thrown when the reflection API is used. Table 9-10 lists the types of exceptions that can be thrown from `java.lang.reflect.Method.invoke`.

Table 9-10 Exceptions Thrown from `java.lang.reflect.Method.invoke()`

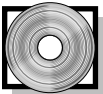
Exception thrown	means ...
<code>IllegalAccessException</code>	Underlying method is inaccessible.
<code>IllegalArgumentException</code>	Either the arguments to the method were invalid or unwrapping conversion failed.
<code>InvocationTargetException</code>	The underlying method threw an exception.
<code>NullPointerException</code>	The specified object is null.

In addition to the exceptions that may be thrown by `Method.invoke`, `GenericActionAdapter` must also check that a reference to the method can be had from the name of the method and the type of the listener. That verification requires catching a `NoSuchMethodException` and a `SecurityException`.

```
public GenericActionAdapter(Object listener,
                            String methodName) {
    this.listener = listener;
    this.methodName = methodName;

    try {
        method =
            listener.getClass().getMethod(methodName, classTypes);
    }
    catch (NoSuchMethodException e) {
        System.out.println("method " +
            methodName + " not found");
    }
    catch (SecurityException e) {
        System.out.println("search for method" + methodName +
            " resulted in a security exception");
    }
}
```

The `GenericActionAdapter` class is listed in its entirety in Example 9-17.

**Example 9-17** `GenericActionAdapter` Class Listing

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

public class GenericActionAdapter implements ActionListener {
    public Object listener;
    public String methodName;
    public Method method;
    public Object[] args = new Object[1];
    public Class[] classTypes = { ActionEvent.class };

    public GenericActionAdapter(Object listener,
                                String methodName) {
```



```

this.listener    = listener;
this.methodName = methodName;
try {
    method =
        listener.getClass().getMethod(methodName, classTypes);
}
catch(NoSuchMethodException e) {
    System.out.println(
        "method " + methodName + " not found");
}
catch(SecurityException e) {
    System.out.println(
        "search for method" + methodName +
        " resulted in a security exception");
}
}
public void actionPerformed(ActionEvent event) {
    args[0] = event;

    try {
        method.invoke(listener, args);
    }
    catch(NullPointerException e) {
        System.out.println("null object, or null method");
    }
    catch(IllegalAccessException e) {
        System.out.println("method " + methodName +
            " cannot be legally accessed");
    }
    catch(IllegalArgumentException e) {
        System.out.println("bad arguments for method " +
            methodName);
    }
    catch(InvocationTargetException e) {
        System.out.println("exception thrown from method" +
            methodName);
    }
}
}
}

```

A final observation about `GenericActionAdapter`: other than declarations, almost all of the code is in the business of handling exceptions. Using the Java reflection mechanisms to circumvent type-safety can be painful because type checking that is normally handled by the compiler at compile time must be manually coded by the developer and handled at runtime. It's not the way you'd want to spend much of your time coding and is motivation for working with Java's type system instead of against it.



AWT TIP ...

Prefer Type Safety

The issue is not whether compile-time checking is preferable to runtime checking. Some object-oriented languages, such as Smalltalk, do not place much stock in compile time checking, whereas other languages, such as C++ and Java do. The issue is: Is circumventing type-safety worth the cost *in Java*? Most of the time, the answer is no. Java is a strongly typed language; as a result, circumventing the type system results in a great deal of manual error checking in addition to cutting across the grain of Java's type system.

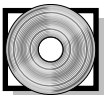
Inner Classes

The 1.1 version of the JDK introduced a handy new construct: inner classes. Our intent here is not to provide an in-depth discussion of inner classes—if you are so inclined, you can find such a discussion from the Java home page (<http://java.sun.com>). Instead, we will show how inner classes can be used for handling events.

Inner classes are pertinent to event handling because they can be used to simplify the relationship between event sources and listeners. In Java, a class can be a top-level class or an inner class. Top-level classes are members of a package and were the only type of classes in the 1.0 JDK. Inner classes are classes that can be defined within a block or as part of a new expression.

ThreeDButton

To start off, we'll implement a simple custom component—`ThreeDButton`, shown in Figure 9-15; our exciting applet is listed in Example 9-18. Note, of course, that our `ThreeDButton` has no event handling associated with it and, therefore, is always drawn raised. As we go along, we'll add event handling behavior to illustrate various concepts.



Example 9-18 `ThreeDButtonTest` Applet

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

public class ThreeDButtonTest extends Applet {
    public void init() {
        add(new ThreeDButton());
    }
}
```

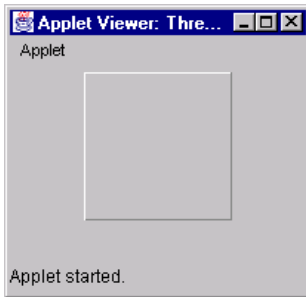


Figure 9-15 ThreeDButton—A Simple Custom Component With No Event Handling.

```
class ThreeDButton extends Canvas {
    static public int BORDER_INSET = 0, BORDER_RAISED = 1;
    private int state = BORDER_RAISED;

    public void paint(Graphics g) {
        paintBorderRaised();
    }
    public Dimension getPreferredSize() {
        return new Dimension(100,100);
    }
    public int getState() {
        return state;
    }
    public void paintBorderRaised() {
        Graphics g = getGraphics();

        try {
            g.setColor(Color.lightGray);
            g.draw3DRect(0,0,
                getSize().width-1,getSize().height-1, true);
            state = BORDER_RAISED;
        }
        finally {
            g.dispose();
        }
    }
    public void paintBorderInset() {
        Graphics g = getGraphics();

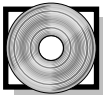
        try {
            g.setColor(Color.lightGray);
            g.draw3DRect(0,0,
                getSize().width-1,getSize().height-1, false);
            state = BORDER_INSET;
        }
    }
}
```



```
    }  
    finally {  
        g.dispose();  
    }  
}
```

Encapsulating Event Handling in a Separate Listener Class

Example 9-19 introduces a new class, `ThreeDButtonListener`, that handles mouse and mouse motion events for a `ThreeDButton`, similar to the manner in which such events are handled for `java.awt.Button`.



Example 9-19 `ThreeDButtonListener` Class Listing

```
class ThreeDButtonListener extends MouseAdapter  
    implements MouseMotionListener {  
    public void mousePressed(MouseEvent event) {  
        ThreeDButton button = (ThreeDButton)event.getSource();  
        button.paintBorderInset();  
    }  
    public void mouseClicked(MouseEvent event) {  
        ThreeDButton button = (ThreeDButton)event.getSource();  
        button.paintBorderRaised();  
    }  
    public void mouseReleased(MouseEvent event) {  
        ThreeDButton button = (ThreeDButton)event.getSource();  
        button.paintBorderRaised();  
    }  
    public void mouseDragged(MouseEvent event) {  
        ThreeDButton button = (ThreeDButton)event.getSource();  
  
        if(button.contains(event.getX(), event.getY())) {  
            if(button.getState() == ThreeDButton.BORDER_RAISED)  
                button.paintBorderInset();  
        }  
        else {  
            if(button.getState() == ThreeDButton.BORDER_INSET)  
                button.paintBorderRaised();  
        }  
    }  
    public void mouseMoved(MouseEvent event) {  
    }  
}
```

There are three things to notice about our event handling class:

First, every method (except `mouseMoved()`) in `ThreeDButtonListener` must, by invoking `java.util.EventObject.getSource()`, obtain a reference to the `ThreeDButton` that fired the event because each method manipulates the button in some fashion. An alternative implementation would be to have `ThreeDButtonListener` maintain a reference to the `ThreeDButton` with



which it is associated, as we've done in Example 9-20. Regardless of which approach we take, since `ThreeDButtonListener` is a top-level class, we must somehow associate the listener to the button on whose behalf it is handling events.

Example 9-20 `ThreeDButton` Associated with `ThreeDButtonListener`

```
class ThreeDButtonListener extends MouseAdapter
    implements MouseMotionListener {
    private ThreeDButton button;
    public ThreeDButtonListener(ThreeDButton button) {
        this.button = button;
    }
    public void mousePressed (MouseEvent event) {
        button.paintBorderInset();
    }
    public void mouseClicked (MouseEvent event) {
        button.paintBorderRaised();
    }
    public void mouseReleased(MouseEvent event) {
        button.paintBorderRaised();
    }
    public void mouseDragged(MouseEvent event) {
        if(button.contains(event.getX(), event.getY())) {
            if(button.getState() == ThreeDButton.BORDER_RAISED)
                button.paintBorderInset();
        }
        else {
            if(button.getState() == ThreeDButton.BORDER_INSET)
                button.paintBorderRaised();
        }
    }
    public void mouseMoved(MouseEvent event) {
    }
}
```

Second, notice the curious (but bug-free) implementation of `ThreeDButtonListener.mouseMoved()`. The reason for the no-op implementation is due to the fact that `ThreeDButtonListener` implements the `MouseMotionListener` interface and therefore must implement every method defined in the interface in order to be a concrete (nonabstract) class. This, by the way, is the lesser of two evils: Had `ThreeDButtonListener` extended `MouseMotionAdapter` and implemented `MouseListener`, we would have had two no-op methods to implement: `mouseEntered()` and `mouseExited()`.

Listening to Yourself

Third, `ThreeDButtonListener` really has no implementation other than manipulating the `ThreeDButton` with which it is associated. Perhaps such code would be better off placed in the `ThreeDButton` class itself, as we've done in

Example 9-21. Now, however, we have three no-op methods because `ThreeDButton` already extends `Canvas`, and, therefore, we must implement both the `MouseListener` and `MouseMotionListener` interfaces instead of extending an adapter class.

Example 9-21 `ThreeDButton` Listening to Itself

```
class ThreeDButton extends Canvas
    implements MouseListener, MouseMotionListener {
    static public int BORDER_INSET = 0, BORDER_RAISED = 1;
    int state = BORDER_RAISED;
    ...
    public void mousePressed (MouseEvent event) {
        paintBorderInset();
    }
    public void mouseClicked (MouseEvent event) {
        paintBorderRaised();
    }
    public void mouseReleased(MouseEvent event) {
        paintBorderRaised();
    }
    public void mouseDragged(MouseEvent event) {
        if(contains(event.getX(), event.getY())) {
            if(state == ThreeDButton.BORDER_RAISED)
                paintBorderInset();
        }
        else {
            if(state == ThreeDButton.BORDER_INSET)
                paintBorderRaised();
        }
    }
    public void mouseEntered(MouseEvent event) { }
    public void mouseExited (MouseEvent event) { }
    public void mouseMoved (MouseEvent event) { }
}
```

Although it might make sense to have `ThreeDButton` handle its own events instead of giving that responsibility to another object (something discussed later in this chapter), we're stuck either implementing no-op methods or associating a separate listener class with the component that fires events.

Named Inner Classes

Enter inner classes. Example 9-22 shows an implementation of `ThreeDButton` that defines two inner classes: `ThreeDButtonMouseListener` and `ThreeDButtonMouseMotionListener`. Since inner classes have direct access to the variables and methods defined in their enclosing class, there is no need to maintain an association between the listener classes and the button. Also, since the event handling is encapsulated in separate classes, each class can extend an



adapter class instead of implementing a listener and having to implement no-op methods to satisfy the requirement that concrete classes must implement all methods of the interfaces they implement.

Example 9-22 ThreeDButton with Inner Classes for Event Handling

```
class ThreeDButton extends Canvas {
    static public int BORDER_INSET = 0, BORDER_RAISED = 1;
    int state = BORDER_RAISED;

    public ThreeDButton() {
        addMouseListener (new ThreeDButtonMouseListener());
        addMouseMotionListener(
            new ThreeDButtonMouseMotionListener());
    }
    class ThreeDButtonMouseListener extends MouseAdapter {
        public void mousePressed (MouseEvent event) {
            paintBorderInset();
        }
        public void mouseClicked (MouseEvent event) {
            paintBorderRaised();
        }
        public void mouseReleased(MouseEvent event) {
            paintBorderRaised();
        }
    }
    class ThreeDButtonMouseMotionListener
        extends MouseMotionAdapter {
        public void mouseDragged(MouseEvent event) {
            if(contains(event.getX(), event.getY())) {
                if(state == ThreeDButton.BORDER_RAISED)
                    paintBorderInset();
            }
            else {
                if(state == ThreeDButton.BORDER_INSET)
                    paintBorderRaised();
            }
        }
    }
    ...
}
```

Anonymous Inner Classes

Notice that with our inner class implementation, `ThreeDButtonMouseListener` and `ThreeDButtonMouseMotionListener` are very unwieldy names for our event handling classes and really add no value, except to fulfill the requirement of a class name. Inner classes provide some syntactic sugar for eliminating such names: anonymous classes.

Example 9-23 is a rewrite of Example 9-22; it uses anonymous (as opposed to named) inner classes for handling mouse events. Anonymous inner classes are defined by inserting a class body after a `new` expression—the anonymous inner class is constructed on the fly and either implements or extends the interface or class, respectively, named in the `new` expression.

Example 9-23 ThreeDButton with Anonymous Inner Classes for Event Handling

```
class ThreeDButton extends Canvas {
    static public int BORDER_INSET = 0, BORDER_RAISED = 1;
    int state = BORDER_RAISED;

    public ThreeDButton() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed (MouseEvent event) {
                paintBorderInset();
            }
            public void mouseClicked (MouseEvent event) {
                paintBorderRaised();
            }
            public void mouseReleased(MouseEvent event) {
                paintBorderRaised();
            }
        });
        addMouseMotionListener(new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent event) {
                if(contains(event.getX(), event.getY())) {
                    if(state == ThreeDButton.BORDER_RAISED)
                        paintBorderInset();
                }
                else {
                    if(state == ThreeDButton.BORDER_INSET)
                        paintBorderRaised();
                }
            }
        });
    }
    ...
}
```

Modifying Default Event Handling Behavior

At this point, the astute reader may observe that we've fused the event handling for `ThreeDButton` to the component class, and therefore `ThreeDButton` must be subclassed in order to modify its event handling algorithm, which was one of the major drawbacks of the old event model. Example 9-24 remedies this situation by allowing a `ThreeDButton` to be constructed with alternative listeners. Note that we've made the nondefault constructor public, but obviously you are free to change the scope of the constructor to suite your needs.¹⁴

14. In this case, it may actually make sense to allow only subclasses to change the default listeners.



Example 9-24 ThreeDButton with Interchangeable Default Event Handling

```
class ThreeDButton extends Canvas {
    static public int BORDER_INSET = 0, BORDER_RAISED = 1;
    int state = BORDER_RAISED;

    public ThreeDButton() {
        this(new MouseAdapter() {
            public void mousePressed (MouseEvent event) {
                paintBorderInset();
            }
            public void mouseClicked (MouseEvent event) {
                paintBorderRaised();
            }
            public void mouseReleased(MouseEvent event) {
                paintBorderRaised();
            }
        }
        ),
        new MouseMotionAdapter() {
            public void mouseDragged(MouseEvent event) {
                if(contains(event.getX(), event.getY())) {
                    if(state == ThreeDButton.BORDER_RAISED)
                        paintBorderInset();
                }
                else {
                    if(state == ThreeDButton.BORDER_INSET)
                        paintBorderRaised();
                }
            }
        }
    );
}

public ThreeDButton(MouseListener ml,
                    MouseMotionListener mml){
    addMouseListener      (ml);
    addMouseMotionListener(mml);
}
...
}
```

Now we've placed the default implementation for handling events in the component that fires the events without having to implement any no-op methods or associate a separate listener with our component—all through the magic of inner classes. Additionally, the default event handling may be overridden by clients of the class. Again, this is not always a recommended approach, but in this case it is a good design.

The last observation that we'll make concerning inner classes is that they provide a mechanism similar to callbacks (function pointers) in C, or blocks in Smalltalk. The difference is that Java uses objects to provide such a mechanism, whereas C uses pointers to functions and Smalltalk uses untyped blocks.

Firing AWT Events from Custom Components

A good percentage of AWT components fire some sort of semantic event. Likewise, many custom components perform one or more actions that could be construed as semantic events. Equipping a custom component to fire semantic events is a fairly common and, thankfully, straightforward task that can be enumerated as follows:

1. Have the component implement a listener interface, if appropriate.
2. Add an appropriate listener member to the class.
3. Implement `addXXXListener()` and `removeXXXListener()`.
4. Implement `processXXXEvent()`.

Example 9-25 lists the `ThreeDButton` class modified to fire action events under the same circumstances that `java.awt.Button` does.

Example 9-25 ThreeDButton That Fires Action Events

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

public class ThreeDButtonTest extends Applet
    implements ActionListener {
    public void init() {
        ThreeDButton button = new ThreeDButton();
        button.addActionListener(this);
        add(button);
    }
    public void actionPerformed(ActionEvent event) {
        System.out.println(event.getActionCommand());
    }
}

class ThreeDButton extends Canvas {
    ...
    private ActionListener actionListener = null;
    ...

    public void addActionListener(ActionListener l) {
        actionListener =
            AWTEventMulticaster.add(actionListener, l);
    }
    public void removeActionListener(ActionListener l) {
        actionListener =
            AWTEventMulticaster.remove(actionListener, l);
    }
    public void processActionEvent() {
        if(actionListener != null) {
            actionListener.actionPerformed(
```



```

        new ActionEvent(this, ActionEvent.ACTION_PERFORMED,
                        "3DButton Action"));
    }
}

```

An `ActionListener` reference is added to `ThreeDButton`, and `addActionListener()` and `removeActionListener()` are implemented to delegate to the `AWTEventMulticaster`. The `AWTEventMulticaster` implements an efficient event dispatching mechanism; we could have kept track of a vector (for instance) of action listeners that we manage ourselves, but the `AWTEventMulticaster` takes care of that headache for us in a manner that is more efficient than a vector implementation. Whenever a listener is added or removed via the `AWTEventMulticaster`, the multicaster returns a reference to a listener that is the first listener in the chain of listeners (of the appropriate type) that are currently registered with the component in question.

We have also implemented `processActionEvent()`, which invokes `actionPerformed()` on our action listener. Invoking `actionPerformed()` on the lone `ActionListener` results in `actionPerformed()` being invoked for every action listener currently registered with our button.

`ThreeDButton.processActionEvent()` constructs an `ActionEvent`, signifying that the button is the source of the event, the type of action event is `ACTION_PERFORMED`, and the action command (a `String` representing the action) is `"3DButton Action"`.

`ThreeDButtonTest` implements `ActionListener` and registers itself as an action listener with our button. In addition, it implements `actionPerformed()` and prints out the action command of the button that was the source of the event. It is important to realize that `ThreeDButtonTest` registers itself as an action listener and reacts to action events in exactly the same manner as it would for any AWT component that fires action events.

AWTEventMulticaster Limitations

If you were paying close attention to the above discussion, perhaps you found it somewhat curious that we can invoke `actionPerformed()` on the single `ActionListener` returned by the `AWTEventMulticaster` methods `add()` and `remove()`, and somehow have `actionPerformed()` invoked for every action listener currently registered with our component. This process works because the listener returned by the `AWTEventMulticaster` `add` and `remove` methods is actually an instance of `AWTEventMulticaster`!



`AWTEventMulticaster` implements every listener interface defined in the AWT and overrides all of the associated methods to turn around and invoke the same method for all currently registered listeners. So, for instance, when we invoke `AWTEventMulticaster.add()`, in Example 9-25, the `ActionListener` returned is actually an instance of `AWTEventMulticaster`. When we subsequently invoke `actionPerformed()` on the listener, `AWTEventMulticaster.actionPerformed()` is called; this method runs through the list of currently registered action listeners and invokes `actionPerformed()` for each one.

As a result of all this object-oriented tomfoolery, `AWTEventMulticaster` is only useful for the set of events defined by the AWT—if you wish to fire custom events, you must resort to maintaining a list of listeners yourself and ensure that the listeners are notified when your custom events are fired. This, of course, is a segue into our next section.

Firing Custom Events from Custom Components

Firing standard AWT events from custom components is relatively straightforward, as we've seen above. However, implementing custom events and firing them from a custom component is a little trickier. We'll outline the steps involved and then, for the sake of illustration, develop a custom component that fires custom events.

A Not-So-Contrived Scenario

Let's say that you've been tasked with developing a tree control similar to what you'd find in the Windows 95 Explorer application. For those of you unfamiliar with the Explorer, take a look at Figure 9-16. (A future version of the AWT will provide such a component.)

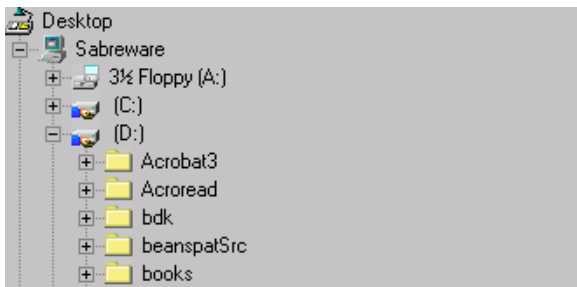


Figure 9-16 The Windows 95 Explorer Tree Control.



Being the good object-oriented developer that you are, you decide to encapsulate the functionality of the plus-minus buttons associated with each node in its own custom component. Furthermore, the plus-minus custom component will fire events whenever it is expanded or contracted, which will serve as notification to any registered listeners that the node has been either expanded or contracted.

If you take a look at the AWT events and listeners, you will find that no event/listener pairs correspond to expansion and contraction.¹⁵ Therefore, a custom event class will need to be implemented in addition to the custom component. So, we have three tasks to accomplish:

- Implement a plus-minus custom component.
- Implement an event representing Expand/Contract events.
- Have the plus-minus component fire events implemented in the previous step when appropriate.

We should note that an alternative approach would be for the plus-minus component to simply fire item events. The `ItemEvent` class defines two constants, `ItemEvent.SELECTED` and `ItemEvent.DESELECTED`, that are used to signify the type of state change for the component in question. We could have used values for the state change other than the selection/deselection values to signify expansion/contraction. However, our intent here is to illustrate firing custom events, so we've chosen the alternative route of implementing a custom event class.

Steps Involved

Of course, the steps outlined above are specific to our plus-minus custom component and a bit too high level, so let's take a look at what's involved in general for developing a custom component that fires custom events:

1. Develop a custom event class.
 2. Develop a listener interface.
 - `void XXXChanged(XXXEvent)`
 3. Develop an interface for registering listeners.
 - `public void addXXXListener(XXXListener).`
 - `public void removeXXXListener(XXXListener).`
15. A 1.1 beta version of the AWT specified `ItemEvents` as encompassing expand/contract in addition to select/deselect, but it was thought that expand/contract was not general enough for `ItemEvent` and the feature was pulled.



- `public void processXXXEvent(XXXEvent). // fire event to listeners`
4. Develop a custom component that fires custom events.
 - Implements interface for registering listeners developed in previous step.

Of course, carrying out the steps listed above is easier said than done, so we will implement the plus-minus custom component in order to illustrate how it's done. Note that each step listed above is actually fairly straightforward to implement, but getting your arms wrapped around the task as a whole can take some practice.

Develop a Custom Event Class

In “Item Events” on page 276, we took a look at the `ItemEvent` class and noted that having a `Checkbox` fire an item event seemed to be somewhat of a stretch because a `Checkbox` has but one item (itself). The AWT provides one event/listener pair for components that have selectable items, whether they have one item or more than one item. We'll take the same approach with our expandable event and have it encompass components that have one expandable item, or components that may potentially have more than one expandable item.

Following the naming convention established by the AWT events, we'll name our custom event class `ItemExpandEvent`. `ItemExpandEvent` public methods are listed in Table 9-11.

Table 9-11 `ItemExpandEvent` Public Methods

Method	Description
<code>ItemExpandable getItemExpandable()</code>	Returns source of the event, which must be an <code>ItemExpandable</code>
<code>Object getItem()</code>	Returns the item that was expanded or contracted
<code>int getExpandState()</code>	Returns <code>ItemExpandState.EXPANDED</code> or <code>ItemExpandState.CONTRACTED</code>

The implementation of `ItemExpandEvent` is shown in Example 9-26.

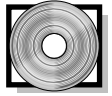
**Example 9-26** ItemExpandEvent Custom Event

```
import java.awt.AWTEvent;

public class ItemExpandEvent extends AWTEvent {
    public static final int EXPANDED = 1;
    public static final int CONTRACTED = 2;

    Object item;
    int expandState;

    public ItemExpandEvent(ItemExpandable source, Object item,
                           int expandState) {
        super(source, -1);
        this.item = item;
        this.expandState = expandState;
    }
    public ItemExpandable getItemExpandable() {
        return (ItemExpandable)source;
    }
    public Object getItem() {
        return item;
    }
    public int getExpandState() {
        return expandState;
    }
    public String paramString() {
        String s = null;
        switch(expandState) {
            case EXPANDED: s += "EXPANDED"; break;
            case CONTRACTED: s += "CONTRACTED"; break;
            default: s += "unknown expand state";
        }
        return super.paramString() + "[expanded=" + s + " ]";
    }
}
```



`ItemExpandEvent` extends `AWTEvent` and as a result inherits the ability to keep track of the source of the event. The source of the event must be an implementation of the `ItemExpandable` interface, as you can see from the `getItemExpandable()` implementation.

`ItemExpandEvent` keeps track of the individual item (within the `ItemExpandable`) that was expanded or contracted and also provides access to the source of the event (the `ItemExpandable`).

It is interesting to note that not only will our plus-minus component implement `ItemExpandable`, but in all likelihood the tree control in which it resides would also implement `ItemExpandable`. The difference between the plus-minus



component and the tree control is that the plus-minus component has but one expandable item (itself), whereas the tree control has potentially many expandable items (plus-minus or other `ItemExpandable` extensions).

`ItemExpandEvent` provides an accessor method, `getExpandState()` that returns either `ItemExpandEvent.EXPANDED` or `ItemExpandEvent.CONTRACTED`. Note that the expand state must be specified (along with the `ItemExpandable` and specific object being expanded or contracted) at construction time.

Finally, note that `ItemExpandEvent` provides a `paramString` method that will print out fascinating information whenever `toString()` is invoked on the event.

Develop a Listener Interface

The next thing that we need is an interface for listening to components that fire events of type `ItemExpandEvent`. The `ItemExpandListener` implementation is shown in Example 9-27.



Example 9-27 `ItemExpandListener` Interface

```
import java.util.EventListener;

public interface ItemExpandListener extends EventListener {
    void itemExpandStateChanged(ItemExpandEvent e);
}
```

As is typically the case, our listener interface is very simple. As is also typically the case, the listener interface extends the `java.util.EventListener` interface. Whenever the expansion state of an item in an `ItemExpandable` changes, the `itemExpandStateChanged` method of its listeners is invoked.

Define an Interface for Registering Listeners

Our plus-minus custom component will implement the `ItemExpandable` interface, which allows item expand listeners to register interest in item expand events and to obtain a reference to an array of all currently expanded items. The `ItemExpandable` interface is shown in Example 9-28.



Example 9-28 `ItemExpandable` Interface

```
public interface ItemExpandable {
    public Object[] getExpandedItems    ();
    public void     addExpandListener   (ItemExpandListener l);
    public void     removeExpandListener(ItemExpandListener l);
}
```



In addition to enabling listeners to register and unregister, the `ItemExpandListener` interface also provides for returning an array of objects that are currently expanded. When our plus-minus component is expanded, it will return an array with itself as the lone item in the array; when our plus-minus component is contracted, it will return `null` from `getExpandedItems()`. Other components that implement the `ItemExpandable` interface may contain many expandable objects and therefore may potentially return an array with any number of objects from the `getExpandedItems` method.

Develop a Custom Component That Fires Custom Events

Our final step involves developing the custom component(s) that fire the custom events. In our case, that would be our plus-minus component, which we'll name `PlusMinus`. The `PlusMinus` class extends `Component`¹⁶ and implements the `ItemExpandable` interface defined above:

```
public class PlusMinus extends Component
    implements ItemExpandable {
    protected boolean expanded = false;
    private Object[] expandedItems = new Object[1];
    private Vector listeners = new Vector();
    ...
    public Object[] getExpandedItems() {
        if(expanded) {
            expandedItems[0] = this;
            return expandedItems;
        }
        else
            return null;
    }
    public void addExpandListener(ItemExpandListener l) {
        listeners.addElement(l);
    }
    public void removeExpandListener(ItemExpandListener l) {
        listeners.removeElement(l);
    }
    ...
}
```

If expanded, the `getExpandedItems` method sets the first element of the array to the current instance of `PlusMinus`; otherwise, it returns `null` to signify that it is not expanded. Of course, callers of `getExpandedItems()` must live with the inconvenience of having to dig the instance of `PlusMinus` out of the first slot in the array, which is the price they pay for the generality of the `ItemExpandable` interface.

16. It's a lightweight component! See "Lightweight Components" on page 651.

Notice that we're not using the `AWTEventMulticaster` to broadcast events to listeners as we did when we fired AWT events from custom components in "Firing AWT Events from Custom Components" on page 298. As discussed in "AWTEventMulticaster Limitations" on page 299, the `AWTEventMulticaster` can only be used for the standard events that come with the AWT, and our event is a custom event. As a result, we've taken the easy way out and have kept track of a vector of listeners. Finally, we have `PlusMinus.processExpandEvent()`, which broadcasts expand events to registered listeners:

```
protected synchronized void processExpandEvent(
                                ItemExpandEvent event) {
    Enumeration e = listeners.elements();

    while(e.hasMoreElements()) {
        ItemExpandListener l = (ItemExpandListener)
                                e.nextElement();
        l.itemExpandStateChanged(event);
    }
}
```

`processExpandEvent()` simply cycles through the currently registered listeners and invokes `itemExpandStateChanged()` for each.

The `PlusMinus` constructor invokes `addMouseListener()` with an inner class version of `MouseAdapter` to handle mouse pressed events. Whenever a mouse pressed event occurs, the expansion state of the instance of `PlusMinus` is toggled and the component is repainted.

```
public PlusMinus() {
    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent event) {
            if(expanded) contract();
            else expand ();
            repaint();
        }
    });
}
```

Example 9-29 shows the implementation of the `PlusMinus` class in its entirety.

Example 9-29 PlusMinus Class Listing



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

public class PlusMinus extends Component
                                implements ItemExpandable {
    protected boolean expanded = false;
    private Object[] expandedItems = new Object[1];
```



```

private Vector listeners = new Vector();

public PlusMinus() {
    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent event) {
            if(expanded) contract();
            else expand ();

            repaint();
        }
    });
}

public void paint(Graphics g) {
    drawBorder(g);
    drawPlusOrMinus(g);
}

public void expand() {
    ItemExpandEvent event =
        new ItemExpandEvent(this, // we are event source
                           this, // we are item
                           ItemExpandEvent.EXPANDED);

    expanded = true;
    processExpandEvent(event);
}

public void contract() {
    ItemExpandEvent event =
        new ItemExpandEvent(this, // we are event source
                           this, // we are item
                           ItemExpandEvent.CONTRACTED);

    expanded = false;
    processExpandEvent(event);
}

public Dimension getPreferredSize() {
    return new Dimension(11,11);
}

public Object[] getExpandedItems() {
    if(expanded) {
        expandedItems[0] = this;
        return expandedItems;
    }
    else
        return null;
}

public void addExpandListener(ItemExpandListener l) {
    listeners.addElement(l);
}

public void removeExpandListener(ItemExpandListener l) {
    listeners.removeElement(l);
}

protected synchronized void processExpandEvent(
    ItemExpandEvent event) {
    Enumeration e = listeners.elements();

```



```
        while(e.hasMoreElements()) {
            ItemExpandListener l = (ItemExpandListener)
                e.nextElement();
            l.itemExpandStateChanged(event);
        }
    }
    private void drawBorder(Graphics g) {
        Dimension size = getSize();
        g.setColor(Color.darkGray.brighter());
        g.drawRect(0,0,size.width-1,size.height-1);
    }
    private void drawPlusOrMinus(Graphics g) {
        Dimension size = getSize();

        if(expanded) drawMinusSign(g, size, Color.black);
        else         drawPlusSign (g, size, Color.black);
    }
    private void drawMinusSign(Graphics g, Dimension size,
        Color color) {
        g.setColor(color);
        g.drawLine(2,size.height/2,size.width-3,size.height/2);
    }
    private void drawPlusSign(Graphics g, Dimension size,
        Color color) {
        g.setColor(color);
        g.drawLine(size.width/2,2,size.width/2,size.height-3);
        g.drawLine(2,size.height/2,size.width-3,size.height/2);
    }
}
}
```

Finally, we need an applet with which to test our custom component. This turns out to be trivial—the applet is shown in action in Figure 9-17. Our applet implements the `ItemExpandListener` interface, creates an instance of `PlusMinus`, and registers itself as being interested in expand events. The applet is listed in Example 9-30.



Example 9-30 PlusMinusTest Applet

```
import java.applet.Applet;
import java.awt.*;

public class PlusMinusTest extends Applet
    implements ItemExpandListener {
    public void init() {
        PlusMinus pm = new PlusMinus();
        pm.addExpandListener(this);
        add(pm);
    }
}
```

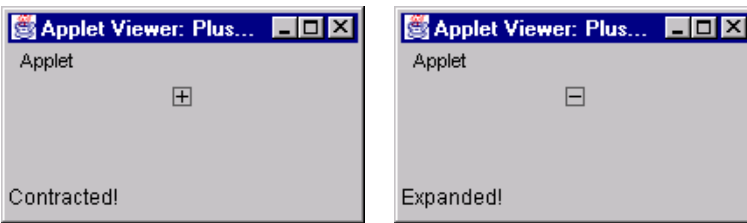


Figure 9-17 PlusMinusTest Applet

The picture on the left shows the PlusMinus custom component contracted; on the right, it's expanded.

```
public void itemExpandStateChanged(ItemExpandEvent event) {
    ItemExpandable ie = event.getItemExpandable();
    Object[] items = ie.getExpandedItems();

    if(items != null) showStatus("Expanded!");
    else showStatus("Contracted!");
}
}
```

Whenever the plus minus component is expanded or contracted, it fires an `ItemExpandEvent`, which the applet is listening for.

Dispatching Events and the AWT Event Queue

There are times when it is desirable to create an event and pass it to a specific component for handling. For instance, some applications may wish to record a sequence of events and play them back at a later time. Essentially, there are two ways to send an event to a component: invoke `Component.dispatchEvent()` or use an event queue.

`Component.dispatchEvent(AWTEvent)` processes the event it is passed in the exact same manner as events that are initiated by user gestures. Figure 9-18 shows an applet that contains two buttons; when the button on the right is activated, mouse pressed and mouse released events are delivered to the button on the left. The mouse events cause the left-hand button to activate and fire an action event.

Example 9-31 lists the applet.

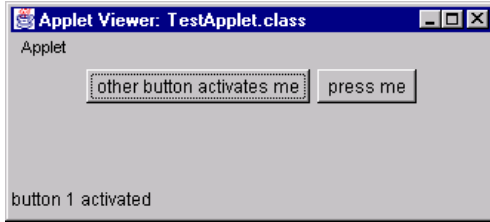
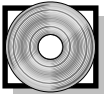


Figure 9-18 Dispatching Events Directly to a Component
When the button on the right is activated, mouse pressed and released events are dispatched to the button on the left, causing it to fire an action event.



Example 9-31 Dispatching Events Directly to a Component

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

public class TestApplet extends Applet {
    public void init() {
        final Button
            button1 = new Button("other button activates me"),
            button2 = new Button("press me");

        add(button1);
        add(button2);

        button1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showStatus("button 1 activated");
            }
        });

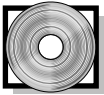
        button2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                MouseEvent press = new MouseEvent(
                    button1,
                    MouseEvent.MOUSE_PRESSED,
                    System.currentTimeMillis(),
                    0, 5, 5, 1, false);

                MouseEvent release = new MouseEvent(
                    button1,
                    MouseEvent.MOUSE_RELEASED,
                    System.currentTimeMillis(),
                    0, 5, 5, 1, false);
```




At any time, there exists only one system event queue, which can be accessed by invoking `Toolkit.getSystemEventQueue()`.¹⁸ Additionally, events may be posted to a local event queue. Example 9-32 lists another version of the applet listed in Example 9-31 on page 310.

Example 9-32 Dispatching Events to a Component Through an Event Queue



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

public class TestApplet extends Applet {
    public void init() {
        final Button
            button1 = new Button("other button activates me"),
            button2 = new Button("press me");

        final EventQueue queue = new EventQueue();

        add(button1);
        add(button2);

        button1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showStatus("button 1 activated");
            }
        });

        button2.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                MouseEvent press = new MouseEvent(
                    button1,
                    MouseEvent.MOUSE_PRESSED,
                    System.currentTimeMillis(),
                    0, 5, 5, 1, false);

                MouseEvent release = new MouseEvent(
                    button1,
                    MouseEvent.MOUSE_RELEASED,
                    System.currentTimeMillis(),
                    0, 5, 5, 1, false);

                queue.postEvent(press);

                try {
                    Thread.currentThread().sleep(50);
                }
                catch (InterruptedException ex) {
```

18. Subject to security restrictions.



```

        ex.printStackTrace();
    }

    queue.postEvent(release);
    });
}
}

```

Instead of dispatching the event directly to the left-hand button, an event queue is instantiated, and the mouse events are posted to the queue. Whenever an event queue is instantiated, an event dispatch thread is created and events are delivered on the dispatch thread. On the other hand, events that are dispatched directly to a component are not delivered on a separate thread.

Active Events

Prior to the 1.2 AWT, events posted to an event queue had to have either a `Component` or a `MenuComponent` specified as the source of the event in order for the event to be dispatched. When an event is dispatched by an event queue, either `Component.dispatchEvent()` or `MenuComponent.dispatchEvent()` is invoked for the source of the event, depending upon the type of the event source.

The 1.2 AWT introduces a new type of event—an active event. Active events are represented by the `java.awt.ActiveEvent` interface, which defines a lone method: `void dispatch()`. Active events are dispatched by invoking the event's `dispatch` method, instead of invoking a method provided by the event source. As a result, active events allow noncomponents to arrange for a certain behavior to occur on a different thread by posting the event to an event queue.

The applet shown in Figure 9-19 contains a button that posts an active event to an event queue. When the button is activated, an `ActiveEvent` is instantiated along with an `EventQueue`, and the event is posted to the queue.

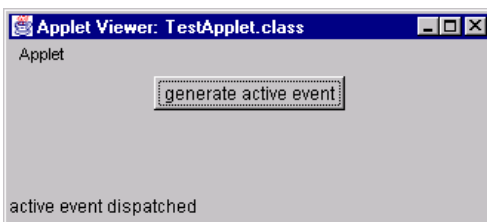
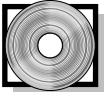


Figure 9-19 Active Events

Activating the button contained in the applet causes an `ActiveEvent` to be fired.



The applet is listed in Example 9-33.



Example 9-33 Using ActiveEvents

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

public class TestApplet extends Applet {
    public void init() {
        final Button
            button1 = new Button("generate active event");

        add(button1);

        button1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                AnActiveEvent ae = new AnActiveEvent();
                EventQueue queue = new EventQueue();

                queue.postEvent(ae);
            }
        });
    }
}

class AnActiveEvent extends AWTEvent implements ActiveEvent {
    public AnActiveEvent() {
        super(new Object(), AWTEvent.RESERVED_ID_MAX + 1);
    }
    public void dispatch() {
        System.out.println("active event dispatched");
    }
}
```

Event queues expect events posted to them to be `AWTEvent`s, so `AnActiveEvent` extends `AWTEvent` and implements the `ActiveEvent` interface.

Notice that the source of the event is specified as simply a new object; the source does not have to be a component or menu component. When the event queue sees that the event is an instance of `ActiveEvent`, it simply invokes the event's `dispatch` method instead of calling a method on the event source. The result is that each activation of the button results in an active event being posted to an event queue. In turn, the event queue processes the event by invoking `ActiveEvent.dispatch`, which is overridden in `AnActiveEvent` to simply print out a string.



Inheritance-Based Mechanism

Under certain circumstances, an inheritance-based event handling mechanism *may* be preferable to delegation. For instance, if a class provides some fundamental functionality that is tied to a particular event, an inheritance-based approach might possibly be preferred. The JDK documentation for the event model stresses that the inheritance-based mechanism is to be used sparingly, if at all.

The JDK delegation event model provides an inheritance-based alternative to using delegation for event handling. The inheritance-based mechanism is based upon the original JDK event model and suffers from its shortcomings; as a result, we reiterate the documentation warning: *the inheritance-based mechanism should be used sparingly, if at all.*

Using the inheritance-based mechanism involves overriding one of the following `java.awt.Component` methods:

```
void processEvent(AWTEvent event)
void processComponentEvent(ComponentEvent event)
void processFocusEvent(FocusEvent event)
void processKeyEvent(KeyEvent event)
void processMouseEvent(MouseEvent event)
void processMouseMotionEvent(MouseEvent event)
```

For every occurrence of an AWT event, `Component.processEvent()` is invoked. `processEvent()` determines the type of event that occurred and then invokes an appropriate `Component.process...Event` method. The `process...Event` method then fires the event to a set of listeners of the appropriate listener type.

For instance, when the mouse enters an AWT component, `Component.processEvent` is invoked. `processEvent()` is passed an `AWTEvent`, which is used to determine the type of the event. `processEvent()` invokes `Component.processMouseEvent()`, passing along the `AWTEvent`, which is cast to a `MouseEvent`. Finally, `processMouseEvent` invokes the `mouseEntered` method for every `MouseListener` that is registered with the component.

AWT TIP*Use Inheritance-Based Event Handling Sparingly*

The documentation that comes with the AWT recommends that you use the inheritance-based mechanism sparingly. As we'll see later on in this chapter, the inheritance-based mechanism can be simulated by listening to yourself, in which case you get nearly all of the benefits associated with the inheritance-based mechanism, while avoiding all of its drawbacks.

Example 9-34 lists the implementation of a button that draws a 3D rectangle around the border of the button when the mouse enters and draws a flat border when the mouse exits, using the inheritance-based feature of the new event model. Under the original event model, we may have chosen to override `mouseenter()` and `mouseexit()` in order to get the job done. With the new event model, we override `processMouseEvent()`, which must determine the actual type of mouse event and act accordingly.

Just as with the old (inheritance-based) event model, you can choose between overriding a kitchen-sink method, whereby all events are funneled through (`processEvent()`—the equivalent of `handleEvent()`), or you can override one of a number of convenience methods. Example 9-34 overrides `processMouseEvent()`, and Example 9-35 overrides `processEvent()`.

Note that you must call `enableEvents(long mask)` and specify the type(s) of event(s) that you'd like to be fired from the component (or alternatively add an appropriate event listener to the component), or your overridden methods will never be invoked. This is the one major difference between the inheritance-based event model and the delegation-based model. Figure 9-20 shows our highlight button test applet in action.

**Example 9-34** Inheritance-Based Event Handling: Overriding `processMouseEvent()`

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

public class HighlightButtonTest extends Applet {
    public void init() {
        HighlightButton leftButton = new HighlightButton();
        HighlightButton rightButton = new HighlightButton();

        add(leftButton);
        add(rightButton);
    }
}
```



Figure 9-20 Using the Inheritance-Based Event Mechanism Uses inheritance-based event handling to draw a 3D border around a rectangle when the mouse enters (the mouse is in the rectangle on the left). Inheritance-based event handling is supported in the delegation-based event model.

```

    }
}

class HighlightButton extends Canvas {
    public HighlightButton() {
        // enableEvents() is a protected method, so we can
        // only call it in extensions of Component. If we
        // added a MouseListener, it would undam the flow of
        // mouse events from the component
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
    }
    public void paint(Graphics g) {
        paintBorder();
    }
    public Dimension getPreferredSize() {
        return new Dimension(100,100);
    }
    public void paintBorder() {
        Graphics g = getGraphics();
        g.setColor(Color.gray);
        g.drawRect(0,0,getSize().width-1,getSize().height-1);
    }
    public void highlight() {
        Graphics g = getGraphics();
        g.setColor(Color.lightGray);
        g.draw3DRect(0,0,getSize().width,getSize().height, true);
    }
    public void unhighlight() {
        paintBorder();
    }
}

```



```
public void processMouseEvent(MouseEvent event) {
    if(event.getID() == MouseEvent.MOUSE_ENTERED) {
        HighlightButton canvas =
            (HighlightButton)event.getSource();
        canvas.highlight();
    }
    else if(event.getID() == MouseEvent.MOUSE_EXITED) {
        HighlightButton canvas =
            (HighlightButton)event.getSource();
        canvas.unhighlight();
    }
}
}
```

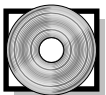
AWT TIP ...

Not All Events Are Delivered to Components

A common mistake when using the inheritance-based mechanism under the delegation event model is neglecting to call `enableEvents(long)`. Under the inheritance-based event model, all events were delivered to a component, whether the component was interested in the event or not. The delegation event model is much more selective, delivering events only to components that indicate interest in the particular type of event by calling `enableEvent(long)` or adding a listener to the component. Therefore, if you must use the inheritance-based mechanism, don't forget to call `enableEvents()` for events you are interested in.

Example 9-35 illustrates overriding `processEvent()` instead of a convenience method. Note that we are careful to call `super.processEvent()` after we are finished processing the events we are interested in because, just like `handleEvent()` from the old event model, `processEvent()` dispatches events to convenience methods. If we had, for instance, also overridden `processKeyEvent()` in Example 9-35 and neglected to invoke `super.processEvent()`, our overridden `processKeyEvent()` would never be invoked.

Example 9-35 Inheritance-Based Event Handling: Overriding `processEvent()`



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

public class HighlightButtonTest2 extends Applet {
    public void init() {
        HighlightButton leftButton = new HighlightButton();
```



```

        HighlightButton rightButton = new HighlightButton();

        add(leftButton);
        add(rightButton);
    }
}

class HighlightButton2 extends Canvas {
    public HighlightButton2() {
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
    }
    public void paint(Graphics g) {
        paintBorder();
    }
    public Dimension getPreferredSize() {
        return new Dimension(100,100);
    }
    public void paintBorder() {
        Graphics g = getGraphics();
        g.setColor(Color.gray);
        g.drawRect(0,0,getSize().width-1,getSize().height-1);
    }
    public void highlight() {
        Graphics g = getGraphics();
        g.setColor(Color.lightGray);
        g.draw3DRect(0,0,getSize().width,getSize().height, true);
    }
    public void unhighlight() {
        paintBorder();
    }
    public void processEvent(AWTEvent event) {
        if(event.getID() == MouseEvent.MOUSE_ENTERED) {
            HighlightButton canvas =
                (HighlightButton)event.getSource();
            canvas.highlight();
        }
        else if(event.getID() == MouseEvent.MOUSE_EXITED) {
            HighlightButton canvas =
                (HighlightButton)event.getSource();
            canvas.unhighlight();
        }
        super.processEvent(event); // must do
    }
}

```

Event Handling Design

Although the new event model is much improved over the old model, you may find yourself pining for the simplicity of the old model where you essentially had only one choice for handling events, namely, extending a component and



embedding event handling code in the extension.¹⁹ The new model, coupled with inner classes, allows for much more flexibility than the old event model. For instance, when handling events with the delegation-based event model you can:

- Use the old event model
- Use the inheritance-based mechanism provided with the new model
- Have components listen to themselves
- Encapsulate the event handling code in a separate listener class

In addition to the approaches outlined above, you can also employ inner classes when handling events. In spite of all the choices you face when implementing event handling, deciding which approach to use is actually easier than it appears. As we shall soon discover, in nearly every case, the best approaches are the last two—the first two approaches are rarely (if ever) recommended. Let's take a look at each option and then discuss whether or not to use inner classes.

Using the Inheritance-Based Event Model

This option is viable because the new event model supports the old model. However, at some future time, the old model will cease to be supported, so it's a good idea to bite the bullet and revamp your event handling code sooner than later. Employing the old event model winds up being double work because you will eventually have to change it. Therefore, using the old event model is not recommended.

In nearly every case, listening to yourself or delegating to a separate event handling class is preferable to using the inheritance-based mechanism because the inheritance-based mechanism in the new event model suffers from the same deficiencies as the old event model. See "Shortcomings of the Inheritance-Based Event Model" on page 229 for a discussion of the old event model's deficiencies.

Listening to Yourself

You know the age-old question: "Are you crazy if you talk to yourself?" and the age-old reply: "Only if you answer." Perhaps you would be considered merely eccentric if you only listen to yourself. When handling events for a component, having the component register itself as a listener is a valid, and often recommended, approach.

19. Actually, you could manually delegate event handling to another object, but it's a lot of work and is quite error-prone.



When basic event handling for a component is not likely to change regardless of the manner in which the component is extended, it makes perfect sense to embed the event handling in the component by having the component listen to itself. Components listen to themselves by implementing a particular listener interface (or extending an adapter class) and then adding themselves as a listener:

```
public class SomeComponent extends Canvas
                               implements MouseAdapter{
    public SomeComponent() {
        addMouseListener(this); // listens to its own mouse events
    }
    public void mousePressed(MouseEvent event) {
        // React to mouse pressed
    }
    public void mouseEntered(MouseEvent event) { }
    public void mouseExited (MouseEvent event) { }
    public void mouseClicked(MouseEvent event) { }
    public void mouseReleased(MouseEvent event) { }
}
```

Of course, there are drawbacks to listening to yourself. First, you are almost certain to have to implement no-op event handling methods because you cannot extend an adapter class (components already extend either `Component` or an extension of `Component`). However, as we've seen in "ThreeDButton with Inner Classes for Event Handling" on page 295, we can employ inner classes to eliminate this drawback. In fact, our `SomeComponent` class can be rewritten as:

```
public class SomeComponent extends Canvas
    public SomeComponent() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent event) {
                // React to mouse pressed
            }
        });
    }
    ...
}
```

Another drawback to listening to yourself is that it involves hardcoding event handling in the component that fires the events. However, as we saw in "ThreeDButton with Interchangeable Default Event Handling" on page 297, this drawback is easily obviated by providing a mechanism to replace the default listener(s). Also, you can decide whether you'd like to make the overriding mechanism available to all classes or whether you'd like to restrict such meddling to an extension of the component, by making the appropriate methods either public or protected, respectively.



There are a number of strategies to employ if you'd like to be able to swap default listeners at runtime. In “ThreeDButton with Interchangeable Default Event Handling” on page 297, we provided a constructor that took a listener argument for specifying an alternate listener. However, that approach allows the listener to be set at construction time only. For a more flexible approach, you can provide setters and getters for the suspect listener:

```
public class IHaveADefaultEventHandlerThatIsReplaceable
    extends Component {
    private MouseListener listener;

    public IHaveADefaultEventHandlerThatIsReplaceable() {
        addMouseListener(listener = new MouseAdapter() {
            public void mousePressed(MouseEvent event) {
                // default event handling for mouse pressed
            }
        });
    }

    public void setMouseListener(MouseListener newListener) {
        if(listener != null)
            remove(listener);

        addMouseListener(listener = newListener);
    }
    ...
}
```

Notice that although a default mouse listener is added to the component in its constructor, it can be swapped out at any time by invocation of `setMouseListener()`.

In short, basic event handling that is not likely to change (or will rarely change) is a good candidate for listening to yourself. Whether you also provide a mechanism for changing the default event handling behavior depends on just how stable you believe the event handling for the component in question happens to be.

Encapsulating Event Handling in a Separate Class

Encapsulating event handling in a separate class is preferable when you have default event handling for a basic component but you are fairly certain that the event handling will be modified, either at runtime or by extensions of the component. Of course, you're probably thinking that that's exactly what we just illustrated in the last section, by employing the “listen to yourself pattern.” However, encapsulating the event handling in a separate class is a more flexible approach because, unlike listening to yourself, it provides a top-level base (event handling) class that others may extend to suit their needs.



The drawback to this approach is that you must implement a separate class, and you must somehow associate the event handling class with the component that fires the appropriate events. However, for event handling that is a likely candidate for change, this drawback is a small price to pay.

In general, then, it is a good idea to encapsulate event handling in a separate class when you can envision the event handling in question being modified for your component, especially when the default event handling is a candidate for subclassing.

Employing Inner Classes

Typically, the decision to use inner classes is most pertinent when you are listening to yourself, as we've seen with our `ThreeDButton` event handling discussion. Realize that inner classes are basically syntactic sugar. Syntactic sugar, in spite of the derogatory connotations usually associated with the term, can often be an exceedingly good thing.

We recommend that you use inner classes as long as their use does not require you to adopt an approach that is unsatisfactory for the given circumstance. For instance, if you are quite certain that some particular event handling functionality is likely to change for a given component, then using inner classes to fuse the event handling into the component class, without providing a mechanism for modifying the event handling in question, would be a pretty bonehead design decision.

Named Inner Classes vs. Anonymous Inner Classes

Of course, if you are going to use inner classes, then the next thing to consider is whether or not you want to give the class a name. The solution is quite straightforward: if you have components that will share the listener, then give the class a name. If the class has a name, then you can instantiate one listener for multiple components, as the following pseudocode illustrates:

```
Choice c1, c2, c3;
MyComponentListener listener = new MyComponentListener();
...
c1.add(listener);
c2.add(listener);
c3.add(listener);
...

class MyComponentListener extends ItemListener {
    // handle item state changes for all three choices
}
```



On the other hand, if the event handling is very specific to a particular component, an anonymous inner class is more convenient:

```
Button addButton = new Button("Add ...");

addButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        // Implementation specific to the add button only
    }
});
```

Propagating Events to Containers

We'll conclude our discussion of the new event model by making some observations about propagating events to a component's container. Note that automatic propagation is flat out not possible with the new event model. While some may argue that such a mechanism is indispensable, we'll go out on a limb here by voting for its exclusion in future versions of the AWT. For one thing, as we discuss in "Shortcomings of the Inheritance-Based Event Model" on page 229, mixing automatic propagation with the component hierarchy can result in subtle bugs, at least as it was implemented in the old model. For another thing, we believe that the best approach is to have objects interested in events fired by a particular component to explicitly register interest in those events. Surely, there are cases where one could argue that automatic propagation is a must-have, but such situations, in our humble opinion, are few and far between. Finally, relying on automatic propagation of events, instead of explicitly expressing interest in a set of events for a given component, often obfuscates code to such a degree that the inconvenience of explicitly registering interest is well worth it.



Summary

This chapter has covered the delegation-based event model that supersedes the inheritance-based model that came with the original AWT. Although the original event model was sufficient for applets (and applications) with simple event handling needs, it did not scale well for a number of reasons (see “Shortcomings of the Inheritance-Based Event Model” on page 229). The new event model addresses nearly all of the drawbacks of the original event model while maintaining compatibility, at least for the time being, with the old model.

We’ve explored many of the aspects of the new event model in this chapter: events, components, and listeners, along with semantic events, consuming events, firing standard and custom events from custom components, etc.

While the new event model is greatly improved over the original model, the developer is now faced with a number of choices with regard to implementing event handling. We’ve provided some guidelines for different approaches in “Event Handling Design” on page 319; these guidelines will help you move from the old event model to the new.