4

In this chapter:
- ***Java 1.0 Event Model***
- ***The Event Class***
- ***The Java 1.1 Event Model***

Events

This chapter covers Java's event-driven programming model. Unlike procedural programs, windows-based programs require an event-driven model in which the underlying environment tells your program when something happens. For example, when the user clicks on the mouse, the environment generates an event that it sends to the program. The program must then figure out what the mouse click means and act accordingly.

This chapter covers two different event models, or ways of handling events. In Java 1.0.2 and earlier, events were passed to all components that could possibly have an interest in them. Events themselves were encapsulated in a single `Event` class. Java 1.1 implements a "delegation" model, in which events are distributed only to objects that have been registered to receive the event. While this is somewhat more complex, it is much more efficient and also more flexible, because it allows any object to receive the events generated by a component. In turn, this means that you can separate the user interface itself from the event-handling code.

In the Java 1.1 event model, all event functionality is contained in a new package, `java.awt.event`. Within this package, subclasses of the abstract class `AWTEvent` represent different kinds of events. The package also includes a number of `EventListener` interfaces that are implemented by classes that want to receive different kinds of events; they define the methods that are called when events of the appropriate type occur. A number of adapter classes are also included; they correspond to the `EventListener` interfaces and provide null implementations of the methods in the corresponding listener. The adapter classes aren't essential but provide a convenient shortcut for developers; rather than declaring that your class implements a particular `EventListener` interface, you can declare that your class extends the appropriate adapter.

The old and new event models are incompatible. Although Java 1.1 supports both, you should not use both models in the same program.

4.1 Java 1.0 Event Model

The event model used in versions 1.0 through 1.0.2 of Java is fairly simple. Upon receiving a user-initiated event, like a mouse click, the system generates an instance of the `Event` class and passes it along to the program. The program identifies the event's target (i.e., the component in which the event occurred) and asks that component to handle the event. If the target can't handle this event, an attempt is made to find a component that can, and the process repeats. That is all there is to it. Most of the work takes place behind the scenes; you don't have to worry about identifying potential targets or delivering events, except in a few special circumstances. Most Java programs only need to provide methods that deal with the specific events they care about.

4.1.1 Identifying the Target

All events occur within a Java `Component`. The program decides which component gets the event by starting at the outermost level and working in. In Figure 4-1, assume that the user clicks at the location (156, 70) within the enclosing `Frame`'s coordinate space. This action results in a call to the `Frame`'s `deliverEvent()` method, which determines which component within the frame should receive the event and calls that component's `deliverEvent()` method. In this case, the process continues until it reaches the `Button` labeled Blood, which occupies the rectangular space from (135, 60) to (181, 80). Blood doesn't contain any internal components, so it must be the component for which the event is intended. Therefore, an action event is delivered to Blood, with its coordinates translated to fit within the button's coordinate space—that is, the button receives an action event with the coordinates (21, 10). If the user clicked at the location (47, 96) within the Frame's coordinate space, the `Frame` itself would be the target of the event because there is no other component at this location.

To reach Blood, the event follows the component/container hierarchy shown in Figure 4-2.

4.1.2 Dealing With Events

Once `deliverEvent()` identifies a target, it calls that target's `handleEvent()` method (in this case, the `handleEvent()` method of Blood) to deliver the event for processing. If Blood has not overridden `handleEvent()`, its default implementation would call Blood's `action()` method. If Blood has not overridden `action()`, its default implementation (which is inherited from `Component`) is executed and

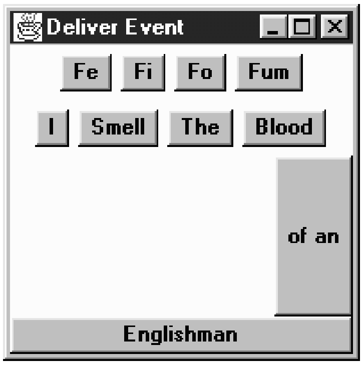

Figure 4–1: deliverEvent

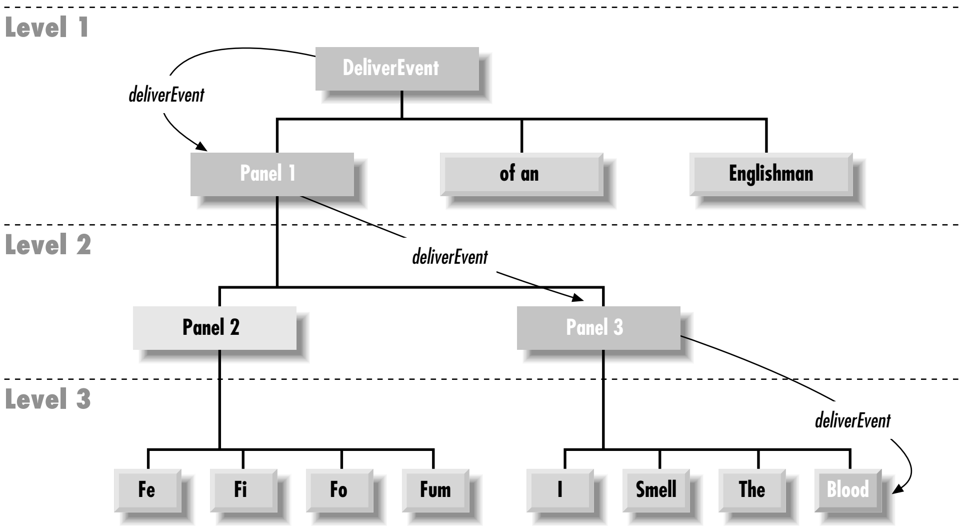

Figure 4–2: deliverEvent screen model

does nothing. For your program to respond to the event, you would have to provide your own implementation of `action()` or `handleEvent()`.

`handleEvent()` plays a particularly important role in the overall scheme. It is really a dispatcher, which looks at the type of event and calls an appropriate method to do the actual work: `action()` for action events, `mouseUp()` for mouse up events, and so on. Table 4-1 shows the event-handler methods you would have to override when using the default `handleEvent()` implementation. If you create your own `handleEvent()`, either to handle an event without a default handler or to process events differently, it is best to leave these naming conventions in place. Whenever you override an event-handler method, it is a good idea to call the overridden method to ensure that you don't lose any functionality. All of the event handler methods return a boolean, which determines whether there is any further event processing; this is described in the next section, "Passing the Buck."

Table 4–1: Event Types and Event Handlers

Event Type	Event Handler
`MOUSE_ENTER`	`mouseEnter()`
`MOUSE_EXIT`	`mouseExit()`
`MOUSE_MOVE`	`mouseMove()`
`MOUSE_DRAG`	`mouseDrag()`
`MOUSE_DOWN`	`mouseDown()`
`MOUSE_UP`	`mouseUp()`
`KEY_PRESS`	`keyDown()`
`KEY_ACTION`	`keyDown()`
`KEY_RELEASE`	`keyUp()`
`KEY_ACTION_RELEASE`	`keyUp()`
`GOT_FOCUS`	`gotFocus()`
`LOST_FOCUS`	`lostFocus()`
`ACTION_EVENT`	`action()`

4.1.3 Passing the Buck

In actuality, `deliverEvent()` does not call `handleEvent()` directly. It calls the `postEvent()` method of the target component. In turn, `postEvent()` manages the calls to `handleEvent()`. `postEvent()` provides this additional level of indirection to monitor the return value of `handleEvent()`. If the event handler returns `true`, the handler has dealt with the event completely. All processing has been completed, and the system can move on to the next event. If the event handler returns `false`, the handler has not completely processed the event, and `postEvent()` will contact the component's `Container` to finish processing the event. Using the screen in Figure 4-1 as the basis, Example 4-1 traces the calls through `deliverEvent()`, `postEvent()`, and `handleEvent()`. The action starts when the user clicks on the Blood button at coordinates (156, 70). In short, Java dives into the depths of the screen's component hierarchy to find the target of the event (by way of the method `deliverEvent()`). Once it locates the target, it tries to find something to deal with the event by working its way back out (by way of `postEvent()`, `handleEvent()`, and the convenience methods). As you can see, there's a lot of overhead, even in this relatively simple example. When we discuss the Java 1.1 event model, you will see that it has much less overhead, primarily because it doesn't need to go looking for a component to process each event.

Example 4–1: The deliverEvent, postEvent, and handleEvent Methods

```
DeliverEvent.deliverEvent (Event e) called
    DeliverEvent.locate (e.x, e.y)
    Finds Panel1
    Translate Event Coordinates for Panel1
    Panel1.deliverEvent (Event e)
        Panel1.locate (e.x, e.y)
        Finds Panel3
        Translate Event Coordinates for Panel3
        Panel3.deliverEvent (Event e)
            Panel3.locate (e.x, e.y)
            Finds Blood
            Translate Event Coordinates for Blood
            Blood.deliverEvent (Event e)
                Blood.postEvent (Event e)
                    Blood.handleEvent (Event e)
                        Blood.mouseDown   (Event e, e.x, e.y)
                            returns false
                        return false
                    Get parent Container Panel3
                    Translate Event Coordinates for Panel3
                    Panel3.postEvent (Event e)
                        Panel3.handleEvent (Event e)
                            Component.mouseDown (Event e, e.x, e.y)
                                returns false
                            return false
                        Get parent Container Panel1
                        Translate Event Coordinates for Panel1
                        Panel1.postEvent (Event e)
                            Panel1.handleEvent (Event e)
                                Component.action (Event e, e.x, e.y)
                                    return false
                                return false
                            Get parent Container DeliverEvent
                            Translate Event Coordinates for DeliverEvent
                            DeliverEvent.postEvent (Event e)
                                DeliverEvent.handleEvent
                                    DeliverEvent.action (Event e, e.x, e.y)
                                        return true
                                    return true
                                return true
                            return true
                        return true
                    return true
                return true
            return true
        return true
    return true
```

4.1.4 Overriding handleEvent()

In many programs, you only need to override convenience methods like `action()` and `mouseUp()`; you usually don't need to override `handleEvent()`, which is the high level event handler that calls the convenience methods. However, convenience methods don't exist for all event types. To act upon an event that doesn't have a convenience method (for example, `LIST_SELECT`), you need to override `handleEvent()` itself. Unfortunately, this presents a problem. Unlike the convenience methods, for which the default versions don't take any action, `handleEvent()` does quite a lot: as we've seen, it's the dispatcher that calls the convenience methods. Therefore, when you override `handleEvent()`, either you should reimplement all the features of the method you are overriding (a very bad idea), or you must make sure that the original `handleEvent()`is still executed to ensure that the remaining events get handled properly. The simplest way for you to do this is for your new `handleEvent()` method to act on any events that it is interested in and return `true` if it has handled those events completely. If the incoming event is not an event that your `handleEvent()` is interested in, you should call `super.handleEvent()` and return its return value. The following code shows how you might override `handleEvent()` to deal with a `LIST_SELECT` event:

```
public boolean handleEvent (Event e) {
    if (e.id == Event.LIST_SELECT) {    // take care of LIST_SELECT
        System.out.println ("Selected item: " + e.arg);
        return true;    // LIST_SELECT handled completely; no further action
    } else {   // make sure we call the overridden method to ensure
               // that other events are handled correctly
        return super.handleEvent (e);
    }
}
```

4.1.5 Basic Event Handlers

The convenience event handlers like `mouseDown()`, `keyUp()`, and `lostFocus()` are all implemented by the `Component` class. The default versions of these methods do nothing and return `false`. Because these methods do nothing by default, when overriding them you do not have to ensure that the overridden method gets called. This simplifies the programming task, since your method only needs to return `false` if it has not completely processed the event. However, if you start to subclass nonstandard components (for example, if someone has created a fancy `AudioButton`, and you're subclassing that, rather than the standard `Button`), you probably should explicitly call the overridden method. For example, if you are overriding `mouseDown()`, you should include a call to `super.mouseDown()`, just as we called `super.handleEvent()` in the previous example. This call is "good housekeeping"; most of the time, your program will work without it. However, your program will break as soon as someone changes the behavior of the `AudioButton` and adds some feature to its `mouseDown()` method. Calling the super class's event handler helps you write "bulletproof" code.

The code below overrides the `mouseDown()` method. I'm assuming that we're extending a standard component, rather than extending some custom component, and can therefore dispense with the call to `super.mouseDown()`.

```
public boolean mouseDown (Event e, int x, int y) {
    System.out.println ("Coordinates: " + x + "-" + y);
    if ((x > 100) || (y < 100))
        return false;     // we're not interested in this event; pass it on
    else                  // we're interested;
        ...               // this is where event-specific processing goes
        return true;      // no further event processing
}
```

Here's a debugging hint: when overriding an event handler, make sure that the parameter types are correct—remember that each convenience method has different parameters. If your overriding method has parameters that don't match the original method, the program will still compile correctly. However, it won't work. Because the parameters don't match, your new method simply overloads the original, rather than overriding it. As a result, your method will never be called.

4.2 The Event Class

An instance of the `Event` class is a platform-independent representation that encapsulates the specifics of an event that happens within the Java 1.0 model. It contains everything you need to know about an event: who, what, when, where, and why the event happened. Note that the `Event` class is not used in the Java 1.1 event model; instead, Java 1.1 has an `AWTEvent` class, with subclasses for different event types.

When an event occurs, you decide whether or not to process the event. If you decide against reacting, the event passes through your program quickly without anything happening. If you decide to handle the event, you must deal with it quickly so the system can process the next event. If handling the event requires a lot of work, you should move the event-handling code into its own thread. That way, the system can process the next event while you go off and process the first. If you do not multithread your event processing, the system becomes slow and unresponsive and could lose events. A slow and unresponsive program frustrates users and may convince them to find another solution for their problems.

4.2.1 Variables

`Event` contains ten instance variables that offer all the specific information for a particular event.

Instance variables

public Object arg

The `arg` field contains some data regarding the event, to be interpreted by the recipient. For example, if the user presses Return within a `TextField`, an `Event` with an `id` of `ACTION_EVENT` is generated with the `TextField` as the `target` and the string within it as the `arg`. See a description of each specific event to find out what its `arg` means.

public int clickCount

The `clickCount` field allows you to check for double clicking of the mouse. This field is relevant only for `MOUSE_DOWN` events. There is no way to specify the time delta used to determine how quick a double-click needs to be, nor is there a maximum value for `clickCount`. If a user quickly clicks the mouse four times, `clickCount` is four. Only the passage of a system-specific time delta will reset the value so that the next `MOUSE_DOWN` is the first click. The incrementing of `clickCount` does not care which mouse button is pressed.

public Event evt

The `evt` field does not appear to be used anywhere but is available if you wish to pass around a linked list of events. Then your program can handle this event and tell the system to deal with the next one (as demonstrated in the following code), or you can process the entire chain yourself.

```
public boolean mouseDown (Event e, int x, int y) {
    System.out.println ("Coordinates: " + x + "-" + y);
    if (e.evt != null)
        postEvent (e.evt);
    return true;
}
```

public int id

The `id` field of `Event` contains the identifier of the event. The system-generated events are the following `Event` constants:

WINDOW_DESTROY	MOUSE_ENTER
WINDOW_EXPOSE	MOUSE_EXIT
WINDOW_ICONIFY	MOUSE_DRAG
WINDOW_DEICONIFY	SCROLL_LINE_UP

KEY_PRESS	SCROLL_LINE_DOWN
KEY_RELEASE	SCROLL_PAGE_UP
KEY_ACTION	SCROLL_PAGE_DOWN
KEY_ACTION_RELEASE	SCROLL_ABSOLUTE
MOUSE_DOWN	LIST_SELECT
MOUSE_UP	LIST_DESELECT
MOUSE_MOVE	ACTION_EVENT

As a user, you can create your own event types and store your own unique event ID here. In Java 1.0, there is no formal way to prevent conflicts between your events and system events, but using a negative IO is a good ad-hoc method. It is up to you to check all the user events generated in your program in order to avoid conflicts among user events.

public int key

For keyboard-related events, the `key` field contains the integer representation of the keyboard element that caused the event. Constants are available for the keypad keys. To examine `key` as a character, just cast it to a `char`. For nonkeyboard-related events, the value is zero.

pubic int modifiers

The `modifiers` field shows the state of the modifier keys when the event happened. A flag is set for each modifier key pressed by the user when the event happened. Modifier keys are Shift, Control, Alt, and Meta. Since the middle and right mouse key are indicated in a Java event by a modifier key, one reason to use the `modifiers` field is to determine which mouse button triggered an event. See Section 4.2.4 for an example.

public Object target

The `target` field contains a reference to the object that is the cause of the event. For example, if the user selects a button, the button is the target of the event. If the user moves the mouse into a `Frame`, the `Frame` is the target. The `target` indicates where the event happened, not the component that is dealing with it.

public long when

The `when` field contains the time of the event in milliseconds. The following code converts this `long` value to a `Date` to examine its contents:

```
Date d = new Date (e.when);
```

public int x
public int y

The `x` and `y` fields show the coordinates where the event happened. The coordinates are always relative to the top left corner of the target of the event and get translated based on the top left corner of the container as the event gets passed through the containing components. (See the previous Section 4.1.1 for an example of this translation.) It is possible for either or both of these to be outside the coordinate space of the applet (e.g., if user quickly moves the mouse outside the applet).

4.2.2 Constants

Numerous constants are provided with the `Event` class. Several designate which event happened (the why). Others are available to help in determining the function key a user pressed (the what). And yet more are available to make your life easier.

When the system generates an event, it calls a handler method for it. To deal with the event, you have to override the appropriate method. The different event type sections describe which methods you override.

Key constants

These constants are set when a user presses a key. Most of them correspond to function and keypad keys; since such keys are generally used to invoke an action from the program or the system, Java calls them *action keys* and causes them to generate a different `Event` type (`KEY_ACTION`) from regular alphanumeric keys (`KEY_PRESS`).

Table 4-2 shows the constants used to represent keys and the event type that uses each constant. The values, which are all declared `public static final int`, appear in the `key` variable of the event instance. A few keys represent ASCII characters that have string equivalents such as `\n`. Black stars (★) mark the constants that are new in Java 1.1; they can be used with the 1.0 event model, provided that you are running Java 1.1. Java 1.1 events use a different set of key constants defined in the `KeyEvent` class.

Table 4–2: Constants for Keys in Java 1.0

Constant	Event Type	Constant	Event Type
HOME	KEY_ACTION	F9	KEY_ACTION
END	KEY_ACTION	F10	KEY_ACTION
PGUP	KEY_ACTION	F11	KEY_ACTION
PGDN	KEY_ACTION	F12	KEY_ACTION
UP	KEY_ACTION	PRINT_SCREEN★	KEY_ACTION
DOWN	KEY_ACTION	SCROLL_LOCK★	KEY_ACTION
LEFT	KEY_ACTION	CAPS_LOCK★	KEY_ACTION
RIGHT	KEY_ACTION	NUM_LOCK★	KEY_ACTION
F1	KEY_ACTION	PAUSE★	KEY_ACTION

Table 4–2: Constants for Keys in Java 1.0 (continued)

Constant	Event Type	Constant	Event Type
F2	KEY_ACTION	INSERT★	KEY_ACTION
F3	KEY_ACTION	ENTER (\n)★	KEY_PRESS
F4	KEY_ACTION	BACK_SPACE (\b)★	KEY_PRESS
F5	KEY_ACTION	TAB (\t)★	KEY_PRESS
F6	KEY_ACTION	ESCAPE★	KEY_PRESS
F7	KEY_ACTION	DELETE★	KEY_PRESS
F8	KEY_ACTION		

Modifiers

Modifiers are keys like Shift, Control, Alt, or Meta. When a user presses any key or mouse button that generates an `Event`, the `modifiers` field of the `Event` instance is set. You can check whether any modifier key was pressed by ANDing its constant with the `modifiers` field. If multiple modifier keys were down at the time the event occurred, the constants for the different modifiers are ORed together in the field.

```
public static final int ALT_MASK
public static final int CTRL_MASK
public static final int META_MASK
public static final int SHIFT_MASK
```

When reporting a mouse event, the system automatically sets the `modifiers` field. Since Java is advertised as supporting the single-button mouse model, all buttons generate the same mouse events, and the system uses the `modifiers` field to differentiate between mouse buttons. That way, a user with a one- or two-button mouse can simulate a three-button mouse by clicking on his mouse while holding down a modifier key. Table 4-3 lists the mouse modifier keys; an applet in Section 4.2.4 demonstrates how to differentiate between mouse buttons.

Table 4–3: Mouse Button Modifier Keys

Mouse Button	Modifier Key
Left mouse button	None
Middle mouse button	ALT_MASK
Right mouse button	META_MASK

For example, if you have a three-button mouse, and click the right button, Java generates some kind of mouse event with the `META_MASK` set in the `modifiers` field. If you have a one-button mouse, you can generate the same event by clicking the mouse while depressing the Meta key.

NOTE If you have a multibutton mouse and do an Alt+right mouse or Meta+left mouse, the results are platform specific. You should get a mouse event with two masks set.

Key events

The component peers deliver separate key events when a user presses and releases nearly any key. `KEY_ACTION` and `KEY_ACTION_RELEASE` are for the function and arrow keys, while `KEY_PRESS` and `KEY_RELEASE` are for the remaining control and alphanumeric keys.

public static final int KEY_ACTION

The peers deliver the `KEY_ACTION` event when the user presses a function or keypad key. The default `Component.handleEvent()` method calls the `keyDown()` method for this event. If the user holds down the key, this event is generated multiple times. If you are using the 1.1 event model, the interface method `KeyListener.keyPressed()` handles this event.

public static final int KEY_ACTION_RELEASE

The peers deliver the `KEY_ACTION_RELEASE` event when the user releases a function or keypad key. The default `handleEvent()` method for `Component` calls the `keyUp()` method for this event. If you are using the 1.1 event model, the `KeyListener.keyReleased()` interface method handles this event.

public static final int KEY_PRESS

The peers deliver the `KEY_PRESS` event when the user presses an ordinary key. The default `Component.handleEvent()` method calls the `keyDown()` method for this event. Holding down the key causes multiple `KEY_PRESS` events to be generated. If you are using the 1.1 event model, the interface method `KeyListener.keyPressed()` handles this event.

public static final int KEY_RELEASE

The peers deliver `KEY_RELEASE` events when the user releases an ordinary key. The default `handleEvent()` method for `Component` calls the `keyUp()` method for this event. If you are using the 1.1 event model, the interface method `KeyListener.keyReleased()` handles this event.

NOTE If you want to capture arrow and keypad keys under the X Window System, make sure the key codes are set up properly, using the *xmodmap* command.

NOTE Some platforms generate events for the modifier keys by themselves, whereas other platforms require modifier keys to be pressed with another key. For example, on a Windows 95 platform, if Ctrl+A is pressed, you would expect one `KEY_PRESS` and one `KEY_RELEASE`. However, there is a second `KEY_RELEASE` for the Control key. Under Motif, you get only a single `KEY_RELEASE`.

Window events

Window events happen only for components that are children of `Window`. Several of these events are available only on certain platforms. Like other event types, the `id` variable holds the value of the specific event instance.

public static final int WINDOW_DESTROY

The peers deliver the `WINDOW_DESTROY` event whenever the system tells a window to destroy itself. This is usually done when the user selects the window manager's Close or Quit window menu option. By default, `Frame` instances do not deal with this event, and you must remember to catch it yourself. If you are using the 1.1 event model, the `WindowListener.windowClosing()` interface method handles this event.

public static final int WINDOW_EXPOSE

The peers deliver the `WINDOW_EXPOSE` event whenever all or part of a window becomes visible. To find out what part of the window has become uncovered, use the `getClipRect()` method (or `getClipBounds()` in Java version 1.1) of the `Graphics` parameter to the `paint()` method. If you are using the 1.1 event model, the `WindowListener.windowOpening()` interface method most closely corresponds to the handling of this event.

public static final int WINDOW_ICONIFY

The peers deliver the `WINDOW_ICONIFY` event when the user iconifies the window. If you are using the 1.1 event model, the interface method `WindowListener.windowIconified()` handles this event.

public static final int WINDOW_DEICONIFY

The peers deliver the `WINDOW_DEICONIFY` event when the user de-iconifies the window. If you are using the 1.1 event model, the interface method `WindowListener.windowDeiconified()` handles this event.

public static final int WINDOW_MOVED

The `WINDOW_MOVED` event signifies that the user has moved the window. If you are using the 1.1 event model, the `ComponentListener.componentMoved()` interface method handles this event.

Mouse events

The component peers deliver mouse events when a user presses or releases a mouse button. Events are also delivered whenever the mouse moves. In order to be platform independent, Java pretends that all mice have a single button. If you press the second or third button, Java generates a regular mouse event but sets the event's `modifers` field with a flag that indicates which button was pressed. If you press the left button, no `modifiers` flags are set. Pressing the center button sets the `ALT_MASK` flag; pressing the right button sets the `META_MASK` flag. Therefore, you can determine which mouse button was pressed by looking at the `Event.modifiers` attribute. Furthermore, users with a one-button or two-button mouse can generate the same events by pressing a mouse button while holding down the Alt or Meta keys.

NOTE Early releases of Java (1.0.2 and earlier) only propagated mouse events from `Canvas` and `Container` objects. With the 1.1 event model, the events that different components process are better defined.

public static final int MOUSE_DOWN

The peers deliver the `MOUSE_DOWN` event when the user presses any mouse button. This action must occur over a component that passes along the `MOUSE_DOWN` event. The default `Component.handleEvent()` method calls the `mouseDown()` method for this event. If you are using the 1.1 event model, the `MouseListener.mousePressed()` interface method handles this event.

public static final int MOUSE_UP

The peers deliver the `MOUSE_UP` event when the user releases the mouse button. This action must occur over a component that passes along the `MOUSE_UP` event. The default `handleEvent()` method for `Component` calls the `mouseUp()` method for this event. If you are using the 1.1 event model, the interface method `MouseListener.mouseReleased()` handles this event.

public static final int MOUSE_MOVE

The peers deliver the `MOUSE_MOVE` event whenever the user moves the mouse over any part of the applet. This can happen many, many times more than you want to track, so make sure you really want to do something with this event before trying to capture it. (You can also capture `MOUSE_MOVE` events and without losing much, choose to deal with only every third or fourth movement.) The default `handleEvent()` method calls the `mouseMove()` method for the event. If you are using the 1.1 event model, the interface method `MouseMotionListener.mouseMoved()` handles this event.

public static final int MOUSE_DRAG

The peers deliver the `MOUSE_DRAG` event whenever the user moves the mouse over any part of the applet with a mouse button depressed. The default method `handleEvent()` calls the `mouseDrag()` method for the event. If you are using the 1.1 event model, the interface method `MouseMotionListener.mouseDragged()` handles this event.

public static final int MOUSE_ENTER

The peers deliver the `MOUSE_ENTER` event whenever the cursor enters a component. The default `handleEvent()` method calls the `mouseEnter()` method for the event. If you are using the 1.1 event model, the interface method `MouseListener.mouseEntered()` handles this event.

public static final int MOUSE_EXIT

The peers deliver the `MOUSE_EXIT` event whenever the cursor leaves a component. The default `handleEvent()` method calls the `mouseExit()` method for the event. If you are using the 1.1 event model, the interface method `MouseListener.mouseExited()` handles this event.

Scrolling events

The peers deliver scrolling events for the `Scrollbar` component. The objects that have a built-in scrollbar (like `List`, `ScrollPane`, and `TextArea`) do not generate these events. No default methods are called for any of the scrolling events. They must be dealt with in the `handleEvent()` method of the `Container` or a subclass of the `Scrollbar`. You can determine which particular event occurred by checking the `id` variable of the event, and find out the new position of the thumb by looking at the `arg` variable or calling `getValue()` on the scrollbar. See also the description of the `AdjustmentListener` interface later in this chapter.

public static final int SCROLL_LINE_UP

The scrollbar peers deliver the `SCROLL_LINE_UP` event when the user presses the arrow pointing up for the vertical scrollbar or the arrow pointing left for the horizontal scrollbar. This decreases the scrollbar setting by one back toward the minimum value. If you are using the 1.1 event model, the interface method `AdjustmentListener.adjustmentValueChanged()` handles this event.

public static final int SCROLL_LINE_DOWN

The peers deliver the `SCROLL_LINE_DOWN` event when the user presses the arrow pointing down for the vertical scrollbar or the arrow pointing right for the horizontal scrollbar. This increases the scrollbar setting by one toward the maximum value. If you are using the 1.1 event model, the interface method `AdjustmentListener.adjustmentValueChanged()` handles this event.

public static final int SCROLL_PAGE_UP

The peers deliver the `SCROLL_PAGE_UP` event when the user presses the mouse with the cursor in the area between the slider and the decrease arrow. This decreases the scrollbar setting by the paging increment, which defaults to 10, back toward the minimum value. If you are using the 1.1 event model, the interface method `AdjustmentListener.adjustmentValueChanged()` handles this event.

public static final int SCROLL_PAGE_DOWN

The peers deliver the `SCROLL_PAGE_DOWN` event when the user presses the mouse with the cursor in the area between the slider and the increase arrow. This increases the scrollbar setting by the paging increment, which defaults to 10, toward the maximum value. If you are using the 1.1 event model, the interface method `AdjustmentListener.adjustmentValueChanged()` handles this event.

public static final int SCROLL_ABSOLUTE

The peers deliver the `SCROLL_ABSOLUTE` event when the user drags the slider part of the scrollbar. There is no set time period or distance between multiple `SCROLL_ABSOLUTE` events. If you are using the Java version 1.1 event model, the `AdjustmentListener.adjustmentValueChanged()` interface method handles this event.

public static final int SCROLL_BEGIN ★

The `SCROLL_BEGIN` event is not delivered by peers, but you may wish to use it to signify when a user drags the slider at the beginning of a series of `SCROLL_ABSOLUTE` events. `SCROLL_END`, described next, would then be used to signify the end of the series.

public static final int SCROLL_END ★

The `SCROLL_END` event is not delivered by peers, but you may wish to use it to signify when a user drags the slider at the end of a series of `SCROLL_ABSOLUTE` events. `SCROLL_BEGIN`, described previously, would have been used to signify the beginning of the series.

List events

Two events specific to the `List` class are passed along by the peers. They signify when the user has selected or deselected a specific choice in the `List`. It is not ordinarily necessary to capture these events, because the peers deliver the `ACTION_EVENT` when the user double-clicks on a specific item in the `List` and it is this `ACTION_EVENT` that triggers something to happen. However, if there is reason to do something when the user has just single-clicked on a choice, these events may be useful. An example of how they would prove useful is if you are displaying a list of filenames with the ability to preview files before loading. Single selection would preview, double-click would load, and deselect would stop previewing.

No default methods are called for any of the list events. They must be dealt with in the `handleEvent()` method of the `Container` of the `List` or a subclass of the `List`. You can determine which particular event occurred by checking the `id` variable of the event.

public static final int LIST_SELECT
The peers deliver the `LIST_SELECT` event when the user selects an item in a `List`. If you are using the 1.1 event model, the interface method `ItemListener.itemStateChanged()` handles this event.

public static final int LIST_DESELECT
The peers deliver the `LIST_DESELECT` event when an item in a `List` has been deselected. This is generated only if the `List` permits multiple selections. If you are using the 1.1 event model, the `ItemListener.itemStateChanged()` interface method handles this event.

Focus events

The peers deliver focus events when a component gains (`GOT_FOCUS`) or loses (`LOST_FOCUS`) the input focus. No default methods are called for the focus events. They must be dealt with in the `handleEvent()` method of the `Container` of the component or a subclass of the component. You can determine which particular event occurred by checking the `id` variable of the event.

NOTE Early releases of Java (1.0.2 and before) did not propagate focus events on all platforms. This is fixed in release 1.1 of Java. Still, you should avoid capturing focus events if you want to write portable 1.0 code.

---

public static final int GOT_FOCUS

The peers deliver the `GOT_FOCUS` event when a component gets the input focus. If you are using the 1.1 event model, the `FocusListener.focusGained()` interface method handles this event.

public static final int LOST_FOCUS

The peers deliver the `LOST_FOCUS` event when a component loses the input focus. If you are using the 1.1 event model, the `FocusListener.focusLost()` interface method handles this event.

FileDialog events

The `FileDialog` events are another set of nonportable events. Ordinarily, the `FileDialog` events are completely dealt with by the system, and you never see them. Refer to Chapter 6, *Containers* for exactly how to work with the `FileDialog` object. If you decide to create a generic `FileDialog` object, you can use these events to indicate file loading and saving. These constants would be used in the `id` variable of the specific event instance:

public static final int LOAD_FILE
public static final int SAVE_FILE

Miscellaneous events

`ACTION_EVENT` is probably the event you deal with most frequently. It is generated when the user performs the desired action for a specific component type (e.g., when a user selects a button or toggles a checkbox). This constant would be found in the `id` variable of the specific event instance.

public static final int ACTION_EVENT

The circumstances that lead to the peers delivering the `ACTION_EVENT` event depend upon the component that is the target of the event and the user's platform. Although the event can be passed along differently on different platforms, users will be accustomed to how the peers work on their specific platforms and will not care that it is different on the other platforms. For example, a Java 1.0 `List` component on a Microsoft Windows platform allows the user to select an item by pressing the first letter of the choice, whereupon the `List` tries to find an item that starts with the letter. The X Window System `List` component does not provide this capability. It works like a normal X `List`, where the user must scroll to locate the item and then select it.

When the `ACTION_EVENT` is generated, the `arg` variable of the specific `Event` instance is set based upon the component type. In Chapters 5–11, which describe Java's GUI components, the description of each component contains an "Events" subsection that describes the value of the event's `arg` field. If you are using the 1.1 event model, the `ActionListener.actionPerformed()` and `ItemListener.itemStateChanged()` interface methods handle this event, depending upon the component type.

4.2.3 Event Methods

Constructors

Ordinarily, the peers deliver all your events for you. However, if you are creating your own components or want to communicate across threads, it may be necessary to create your own events. You can also create your own events to notify your component's container of application-specific occurrences. For example, if you were implementing your own tab sequencing for text fields, you could create a "next text field" event to tell your container to move to the next text field. Once you create the event, you send it through the system using the `Component.postEvent()` method.

public Event (Object target, long when, int id, int x, int y, int key, int modifiers, Object arg)

The first version of the constructor is the most complete and is what the other two call. It initializes all the fields of the `Event` to the parameters passed and sets `clickCount` to 0. See the descriptions of the instance variables Section 4.2.1 for the meanings of the arguments.

public Event (Object target, long when, int id, int x, int y, int key, int modifiers)

The second constructor version calls the first with `arg` set to null.

public Event (Object target, int id, Object arg)

The final version calls the first constructor with the `when`, `x`, `y`, `key`, and `modifiers` parameters set to 0.

Modifier methods

The modifier methods check to see if the different modifier mask values are set. They report the state of each modifier key at the moment an event occurred. It is possible for multiple masks to be set if multiple modifiers are pressed when the event occurs.

There is no `altDown()` method; to check whether the Alt key is pressed you must directly compare the event's `modifiers` against the `Event.ALT_MASK` constant. The `metaDown()` method is helpful when dealing with mouse events to see if the user pressed the right mouse button.

public boolean shiftDown ()

The `shiftDown()` method returns `true` if the Shift key was pressed and `false` otherwise. There is no way to differentiate left and right shift keys.

public boolean controlDown ()

The `controlDown()` method returns `true` if the Control key was pressed and `false` otherwise.

public boolean metaDown ()

The `metaDown()` method returns `true` if the Meta key was pressed and `false` otherwise.

Miscellaneous methods

public void translate (int x, int y)

The `translate()` method translates the `x` and `y` coordinates of the `Event` instance by `x` and `y`. The system does this so that the coordinates of the event are relative to the component receiving the event, rather than the container of the component. The system takes care of all this for you when passing the event through the containment hierarchy (not the object hierarchy), so you do not have to bother with translating them yourself. Figure 4-3 shows how this method would change the location of an event from a container down to an internal component.

protected String paramString ()

When you call the `toString()` method of `Event`, the `paramString()` method is called in turn to build the string to display. In the event you subclass `Event` to add additional information, instead of having to provide a whole new `toString()` method, you need only add the new information to the string already generated by `paramString()`. Assuming the new information is `foo`, this would result in the following method declaration:

```
protected String paramString() {
    return super.paramString() + ",foo=" + foo;
}
```

public String toString ()

The `toString()` method of `Event` returns a string with numerous components. The only variables that will always be in the output will be the event ID and the `x` and `y` coordinates. The others will be present if necessary (i.e., non-null): key (as the integer corresponding to a keyboard event), shift when `shiftDown()` is true; control, when `controlDown()` is true; meta, when `metaDown()` is true; target (if it was a `Component`); and arg (the value depends on the target and ID). `toString()` does not display all pieces of the `Event` information. An event when moving a `Scrollbar` might result in the following:

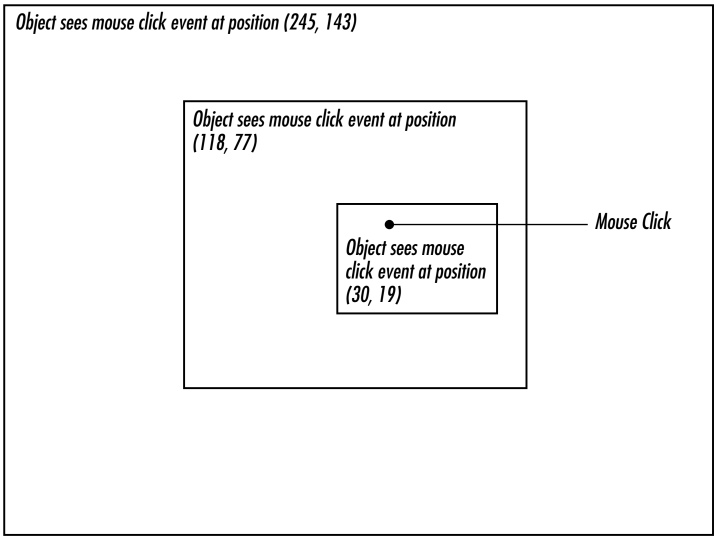
Object sees mouse click event at position (245, 143)
Object sees mouse click event at position (118, 77)
Object sees mouse click event at position (30, 19)
Mouse Click

Figure 4–3: Translating an event's location relative to a component

```
java.awt.Event[id=602,x=374,y=110,target=java.awt.Scrollbar[374,
110,15x50,val=1,vis=true,min=0,max=255,vert],arg=1]
```

4.2.4 Working With Mouse Buttons in Java 1.0

As stated earlier, the `modifiers` component of `Event` can be used to differentiate the different mouse buttons. If the user has a multibutton mouse, the `modifiers` field is set automatically to indicate which button was pressed. If the user does not own a multibutton mouse, he or she can press the mouse button in combination with the Alt or Meta keys to simulate a three-button mouse. Example 4-2 is a sample program called `mouseEvent` that displays the mouse button selected.

Example 4–2: Differentiating Mouse Buttons in Java 1.0

```
import java.awt.*;
import java.applet.*;
public class mouseEvent extends Applet {
    String theString = "Press a Mouse Key";
    public synchronized void setString (String s) {
        theString = s;
    }
    public synchronized String getString () {
        return theString;
    }
    public synchronized void paint (Graphics g) {
        g.drawString (theString, 20, 20);
    }
    public boolean mouseDown (Event e, int x, int y) {
        if (e.modifiers == Event.META_MASK) {
```

Example 4–2: Differentiating Mouse Buttons in Java 1.0 (continued)

```
                setString ("Right Button Pressed");
            } else if (e.modifiers == Event.ALT_MASK) {
                setString ("Middle Button Pressed");
            } else {
                setString ("Left Button Pressed");
            }
            repaint ();
            return true;
        }
        public boolean mouseUp (Event e, int x, int y) {
            setString ("Press a Mouse Key");
            repaint ();
            return true;
        }
    }
```

Unfortunately, this technique does not always work. With certain components on some platforms, the peer captures the mouse event and does not pass it along; for example, on Windows, the display-edit menu of a `TextField` appears when you select the right mouse button. Be cautious about relying on multiple mouse buttons; better yet, if you want to ensure absolute portability, stick to a single button.

4.2.5 Comprehensive Event List

Unfortunately, there are many platform-specific differences in the way event handling works. It's not clear whether these differences are bugs or whether vendors think they are somehow improving their product by introducing portability problems. We hope that as Java matures, different platforms will gradually come into synch. Until that happens, you might want your programs to assume the lowest common denominator. If you are willing to take the risk, you can program for a specific browser or platform, but should be aware of the possibility of changes.

Appendix C, *Platform-Specific Event Handling*, includes a table that shows which components pass along which events by default in the most popular environments. This table was developed using an interactive program called `compList`, which generates a list of supported events for each component. You can find `compList` on this book's Web site, http://www.ora.com/catalog/javawt. If you want to check the behavior of some new platform, or a newer version of one of the platforms in Appendix C, feel free to use `compList`. It does require a little bit of work on your part. You have to click, toggle, type, and mouse over every object. Hopefully, as Java matures, this program will become unnecessary.

4.3 The Java 1.1 Event Model

Now it's time to discuss the new event model that is implemented by the 1.1 release of the JDK. Although this model can seem much more complex (it does have many more pieces), it is really much simpler and more efficient. The new event model does away with the process of searching for components that are interested in an event—`deliverEvent()`, `postEvent()`, `handleEvent()`—and all that. The new model requires objects be registered to receive events. Then, only those objects that are registered are told when the event actually happens.

This new model is called "delegation"; it implements the `Observer-Observable` design pattern with events. It is important in many respects. In addition to being much more efficient, it allows for a much cleaner separation between GUI components and event handling. It is important that any object, not just a `Component`, can receive events. Therefore, you can separate your event-handling code from your GUI code. One set of classes can implement the user interface; another set of classes can respond to the events generated by the interface. This means that if you have designed a good interface, you can reuse it in different applications by changing the event processing. The delegation model is essential to JavaBeans, which allows interaction between Java and other platforms, like OpenDoc or ActiveX. To allow such interaction, it was essential to separate the source of an event from the recipient.*

The delegation model has several other important ramifications. First, event handlers no longer need to worry about whether or not they have completely dealt with an event; they do what they need to, and return. Second, events can be broadcast to multiple recipients; any number of classes can be registered to receive an event. In the old model, broadcasting was possible only in a very limited sense, if at all. An event handler could declare that it hadn't completely processed an event, thus letting its container receive the event when it was done, or an event handler could generate a new event and deliver it to some other component. In any case, developers had to plan how to deliver events to other recipients. In Java 1.1, that's no longer necessary. An event will be delivered to every object that is registered as a listener for that event, regardless of what other objects do with the event. Any listener can mark an event "consumed," so it will be ignored by the peer or (if they care) other listeners.

Finally, the 1.1 event model includes the idea of an event queue. Instead of having to override `handleEvent()` to see all events, you can peek into the system's event queue by using the `EventQueue` class. The details of this class are discussed at the end of this chapter.

* For more information about JavaBeans, see http://splash.javasoft.com/beans/.

In Java 1.1, each component is an event *source* that can generate certain types of events, which are all subclasses of AWTEvent. Objects that are interested in an event are called *listeners*. Each event type corresponds to a listener interface that specifies the methods that are called when the event occurs. To receive an event, an object must implement the appropriate listener interface and must be registered with the event's source, by a call to an "add listener" method of the component that generates the event. Who calls the "add listener" method can vary; it is probably the best design for the component to register any listeners for the events that it generates, but it is also possible for the event handler to register itself, or for some third object to handle registration (for example, one object could call the constructor for a component, then call the constructor for an event handler, then register the event handler as a listener for the component's events).

This sounds complicated, but it really isn't that bad. It will help to think in concrete terms. A TextField object can generate action events, which in Java 1.1 are of the class ActionEvent. Let's say we have an object of class TextActionHandler that is called myHandler that is interested in receiving action events from a text field named inputBuffer. This means that our object must implement the ActionListener interface, and this in turn, means that it must include an actionPerformed() method, which is called when an action event occurs. Now, we have to register our object's interest in action events generated by inputBuffer; to do so, we need a call to inputBuffer.addActionListener(myHandler). This call would probably be made by the object that is creating the TextField but could also be made by our event handler itself. The code might be as simple as this:

```
...
public void init(){
    ...
    inputBuffer = new TextField();
    myHandler = new TextActionHandler();
    inputBuffer.addActionListener(myHandler); // register the handler for the
                                              // buffer's events
    add (inputBuffer);  // add the input buffer to the display
    ...
}
```

Once our object has been registered, myHandler.actionPerformed() will be called whenever a user does anything in the text field that generates an action event, like typing a carriage return. In a way, actionPerformed() is very similar to the action() method of the old event model—except that it is not tied to the Component hierarchy; it is part of an interface that can be implemented by any object that cares about events.

Of course, there are many other kinds of events. Figure 4-4 shows the event hierarchy for Java 1.1. Figure 4-5 shows the different listener interfaces, which are all subinterfaces of EventListener, along with the related adapter classes.

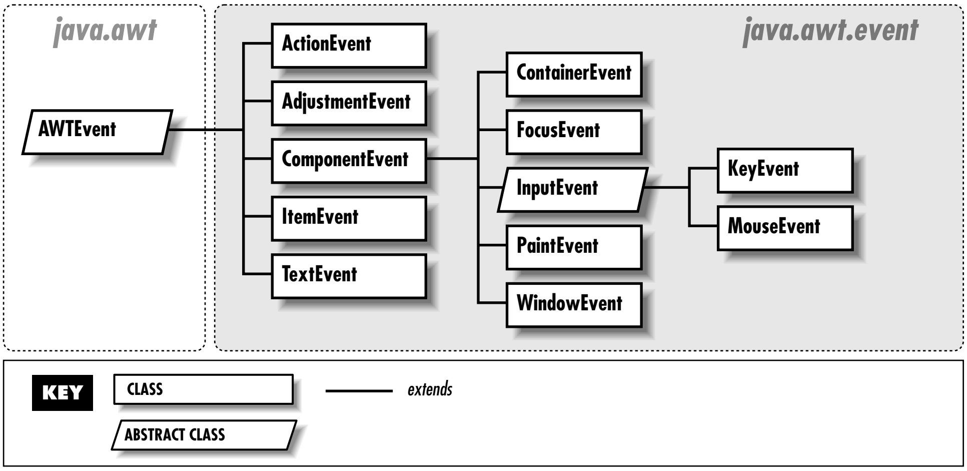

Figure 4–4: AWTEvent class hierarchy

Some of the listener interfaces are constructed to deal with multiple events. For instance, the MouseListener interface declares five methods to handle different kinds of mouse events: mouse down, mouse up, click (both down and up), mouse enter, and mouse exit. Strictly speaking, this means that an object interested in mouse events must implement MouseListener and must therefore implement five methods to deal with all possible mouse actions. This sounds like a waste of the programmer's effort; most of the time, you're only interested in one or two of these events. Why should you have to implement all five methods? Fortunately, you don't. The java.awt.event package also includes a set of *adapter classes*, which are shorthands that make it easier to write event handlers. The adapter class for any listener interface provides a null implementation of all the methods in that interface. For example, the MouseAdapter class provides stub implementations of the methods mouseEntered(), mouseExited(), mousePressed(), mouseReleased(), and mouseClicked(). If you want to write an event-handling class that deals with mouse clicks only, you can declare that your class extends MouseAdapter. It then inherits all five of these methods, and your only programming task is to override the single method you care about: mouseClicked().

A particularly convenient way to use the adapters is to write an anonymous inner class. For example, the following code deals with the MOUSE_PRESSED event without creating a separate listener class:

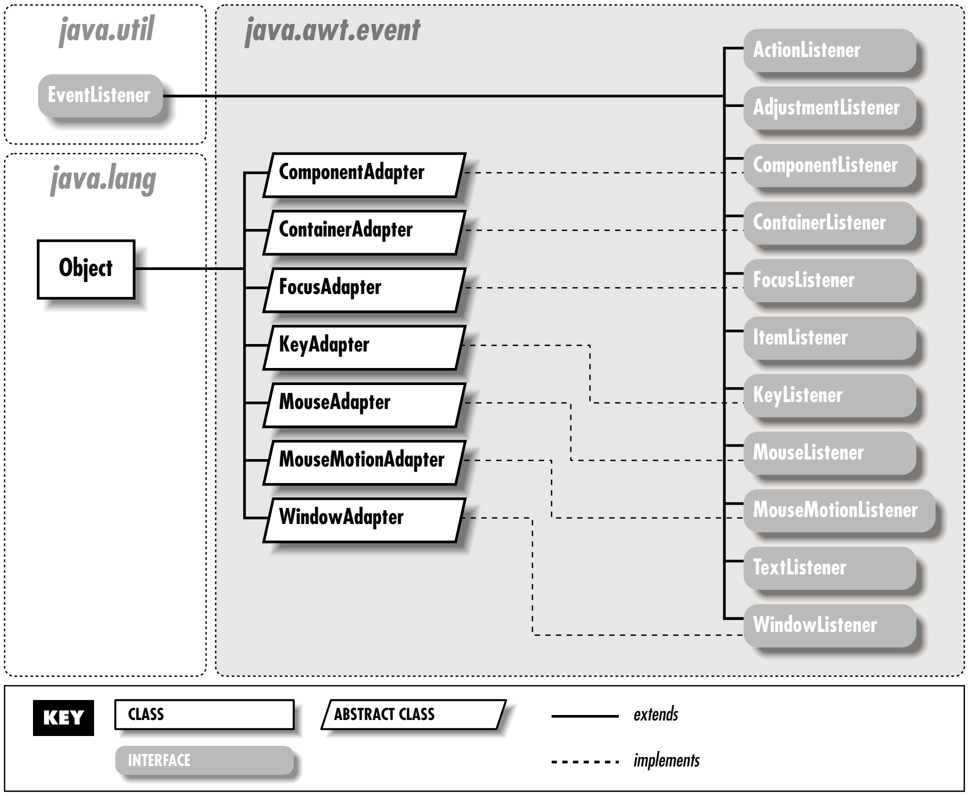

Figure 4–5: AWT EventListener and Adapter class hierarchies

```
addMouseListener (new MouseAdapter()   {
  public void mousePressed (MouseEvent e)  {
    // do what's needed to handle the event
    System.out.println ("Clicked at: " + e.getPoint());
  }
});
```

This code creates a `MouseAdapter`, overrides its `mousePressed()` method, and registers the resulting unnamed object as a listener for mouse events. Its `mousePressed()` method is called when `MOUSE_PRESSED` events occur. You can also use the adapter classes to implement something similar to a callback. For example, you could override `mousePressed()` to call one of your own methods, which would then be called whenever a `MOUSE_PRESSED` event occurs.

There are adapter classes for most of the listener interfaces; the only exceptions are the listener interfaces that contain only one method (for example, there's no `ActionAdapter` to go with `ActionListener`). When the listener interface contains only one method, an adapter class is superfluous. Event handlers may as well implement the listener interface directly, because they will have to override the only method in the interface; creating a dummy class with the interface method stubbed out doesn't accomplish anything. The different adapter classes are discussed with their related `EventListener` interfaces.

With all these adapter classes, listener interfaces, and event classes, it's easy to get confused. Here's a quick summary of the different pieces involved and the roles they play:

- Components generate `AWTEvents` when something happens. Different subclasses of `AWTEvent` represent different kinds of events. For example, mouse events are represented by the `MouseEvent` class. Each component can generate certain subclasses of `AWTEvent`.
- Event handlers are registered to receive events by calls to an "add listener" method in the component that generates the event. There is a different "add listener" method for every kind of `AWTEvent` the component can generate; for example, to declare your interest in a mouse event, you call the component's `addMouseListener()` method.
- Every event type has a corresponding listener interface that defines the methods that are called when that event occurs. To be able to receive events, an event handler must therefore implement the appropriate listener interface. For example, `MouseListener` defines the methods that are called when mouse events occur. If you create a class that calls `addMouseListener()`, that class had better implement the `MouseListener` interface.
- Most event types also have an adapter class. For example, `MouseEvents` have a `MouseAdapter` class. The adapter class implements the corresponding listener interface but provides a `stub` implementation of each method (i.e., the method just returns without taking any action). Adapter classes are shorthand for programs that only need a few of the methods in the listener interface. For example, instead of implementing all five methods of the `MouseListener` interface, a class can extend the `MouseAdapter` class and override the one or two methods that it is interested in.

4.3.1 Using the 1.1 Event Model

Before jumping in and describing all the different pieces in detail, we will look at a simple applet that uses the Java 1.1 event model. Example 4-3 is equivalent to Example 4-2, except that it uses the new event model; when you press a mouse button, it just tells you what button you pressed. Notice how the new class, `mouseEvent11`, separates the user interface from the actual work. The class mouseEvent11 implements a very simple user interface. The class UpDownCatcher handles the events, figures out what to do, and calls some methods in mouseEvent11 to communicate the results. I added a simple interface that is called GetSetString to define the communications between the user interface and the event handler; strictly speaking, this isn't necessary, but it's a good programming practice.

Example 4–3: Handling Mouse Events in Java 1.1

```
// Java 1.1 only
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
interface GetSetString {
    public void setString (String s);
    public String getString ();
}
```

The UpDownCatcher class is responsible for handling events generated by the user interface. It extends MouseAdapter so that it needs to implement only the MouseListener methods that we care about (such as mousePressed() and mouseReleased()).

```
class UpDownCatcher extends MouseAdapter {
    GetSetString gss;
    public UpDownCatcher (GetSetString s) {
        gss = s;
    }
```

The constructor simply saves a reference to the class that is using this handler.

```
    public void mousePressed (MouseEvent e) {
        int mods = e.getModifiers();
        if ((mods & MouseEvent.BUTTON3_MASK) != 0) {
            gss.setString ("Right Button Pressed");
        } else if ((mods & MouseEvent.BUTTON2_MASK) != 0) {
            gss.setString ("Middle Button Pressed");
        } else {
            gss.setString ("Left Button Pressed");
        }
        e.getComponent().repaint();
    }
```

The mousePressed method overrides one of the methods of the MouseAdapter class. The method mousePressed() is called whenever a user presses any mouse button. This method figures out which button on a three-button mouse was pressed and calls the setString() method in the user interface to inform the user of the result.

```
        public void mouseReleased (MouseEvent e) {
            gss.setString ("Press a Mouse Key");
            e.getComponent().repaint();
        }
    }
```

The mouseReleased method overrides another of the methods of the Mouse-Adapter class. When the user releases the mouse button, it calls setString() to restore the user interface to the original message.

```
    public class mouseEvent11 extends Applet implements GetSetString {
        private String theString = "Press a Mouse Key";
        public synchronized void setString (String s) {
            theString = s;
        }
        public synchronized String getString () {
            return theString;
        }
        public synchronized void paint (Graphics g) {
            g.drawString (theString, 20, 20);
        }
        public void init () {
            addMouseListener (new UpDownCatcher(this));
        }
    }
```

mouseEvent11 is a very simple applet that implements our user interface. All it does is draw the desired string on the screen; the event handler tells it what string to draw. The init() method creates an instance of the event handler, which is UpDownCatcher, and registers it as interested in mouse events.

Because the user interface and the event processing are in separate classes, it would be easy to use this user interface for another purpose. You would have to replace only the UpDownCatcher class with something else—perhaps a more complex class that reported when the mouse entered and exited the area.

4.3.2 AWTEvent and Its Children

Under the 1.1 delegation event model, all system events are instances of AWTEvent or its subclasses. The model provides two sets of event types. The first set are fairly raw events, such as those indicating when a component gets focus, a key is pressed, or the mouse is moved. These events exist in ComponentEvent and its subclasses, along with some new events previously available only by overriding non-event-related methods. In addition, higher-level event types (for example, selecting a button) are encapsulated in other subclasses of AWTEvent that are not children of ComponentEvent.

4.3.2.1 AWTEvent

Variables

protected int id ★

The `id` field of `AWTEvent` is protected and is accessible through the `getID()` method. It serves as the identifier of the event type, such as the `ACTION_PERFORMED` type of `ActionEvent` or the `MOUSE_MOVE` type of `Event`. With the delegation event model, it is usually not necessary to look at the event `id` unless you are looking in the event queue; just register the appropriate event listener.

Constants The constants of `AWTEvent` are used in conjunction with the internal method `Component.eventEnabled()`. They are used to help the program determine what style of event handling (true/false-containment or listening-delegation) the program uses and which events a component processes. If you want to process 1.1 events without providing a listener, you need to set the mask for the type of event you want to receive. Look in Chapter 5, *Components,* for more information on the use of these constants:

public final static long ACTION_EVENT_MASK ★
public final static long ADJUSTMENT_EVENT_MASK ★
public final static long COMPONENT_EVENT_MASK ★
public final static long CONTAINER_EVENT_MASK ★
public final static long FOCUS_EVENT_MASK ★
public final static long ITEM_EVENT_MASK ★
public final static long KEY_EVENT_MASK ★
public final static long MOUSE_EVENT_MASK ★
public final static long MOUSE_MOTION_EVENT_MASK ★
public final static long TEXT_EVENT_MASK ★
public final static long WINDOW_EVENT_MASK ★

In addition to the mask constants, the constant `RESERVED_ID_MAX` is the largest event ID reserved for "official" events. You may use ID numbers greater than this value to create your own events, without risk of conflicting with standard events.

public final static long RESERVED_ID_MAX ★

Constructors Since `AWTEvent` is an abstract class, you cannot call the constructors directly. They are automatically called when an instance of a child class is created.

public AWTEvent(Event event) ★

The first constructor creates an `AWTEvent` from the parameters of a 1.0 `Event`. The `event.target` and `event.id` are passed along to the second constructor.

public AWTEvent(Object source, int id) ★

This constructor creates an `AWTEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. It is protected and is accessible through the `getID()` method. With the delegation event model, it is usually not necessary to look at the event `id` unless you are looking in the event queue or in the `processEvent()` method of a component; just register the appropriate event listener.

Methods

public int getID() ★

The `getID()` method returns the `id` from the constructor, thus identifying the event type.

protected void consume() ★

The `consume()` method is called to tell an event that it has been handled. An event that has been marked "consumed" is still delivered to the source component's peer and to all other registered listeners. However, the peer will ignore the event; other listeners may also choose to ignore it, but that's up to them. It isn't possible for a listener to "unconsume" an event that has already been marked "consumed."

Noncomponent events cannot be consumed. Only keyboard and mouse event types can be flagged as consumed. Marking an event "consumed" is useful if you are capturing keyboard input and need to reject a character; if you call `consume()`, the key event never makes it to the peer, and the keystroke isn't displayed. In Java 1.0, you would achieve the same effect by writing an event handler (e.g., `keyDown()`) that returns `true`.

You can assume that an event won't be delivered to the peer until all listeners have had a chance to consume it. However, you should not make any other assumptions about the order in which listeners are called.

protected boolean isConsumed() ★

The `isConsumed()` method returns whether the event has been consumed. If the event has been consumed, either by default or through `consume()`, this method returns `true`; otherwise, it returns `false`.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. Since you are most frequently dealing with children of `AWTEvent`, the children need only to override `paramString()` to add their specific information.

public String toString() ★

The `toString()` method of `AWTEvent` returns a string with the name of the event, specific information about the event, and the source. In the method `MouseAdapter.mouseReleased()`, printing the parameter would result in something like the following:

```
java.awt.event.MouseEvent[MOUSE_RELEASED,(69,107),mods=0,clickCount=1] on panel1
```

4.3.2.2 ComponentEvent

Constants

public final static int COMPONENT_FIRST ★
public final static int COMPONENT_LAST ★

The `COMPONENT_FIRST` and `COMPONENT_LAST` constants hold the endpoints of the range of identifiers for `ComponentEvent` types.

public final static int COMPONENT_HIDDEN ★

The `COMPONENT_HIDDEN` constant identifies component events that occur because a component was hidden. The interface method `ComponentListener.componentHidden()` handles this event.

public final static int COMPONENT_MOVED ★

The `COMPONENT_MOVED` constant identifies component events that occur because a component has moved. The `ComponentListener.componentMoved()` interface method handles this event.

public final static int COMPONENT_RESIZED ★

The `COMPONENT_RESIZED` constant identifies component events that occur because a component has changed size. The interface method `ComponentListener.componentResized()` handles this event.

public final static int COMPONENT_SHOWN ★

The `COMPONENT_SHOWN` constant identifies component events that occur because a component has been shown (i.e., made visible). The interface method `ComponentListener.componentShown()` handles this event.

Constructors

public ComponentEvent(Component source, int id) ★

This constructor creates a `ComponentEvent` with the given `source`; the source is the object generating the event. The `id` field identifies the event type. If system generated, the `id` will be one of the last four constants above. However, nothing stops you from creating your own `id` for your event types.

Methods

public Component getComponent() ★

The `getComponent()` method returns the source of the event—that is, the component initiating the event.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `ComponentEvent` level, `paramString()` adds a string containing the event `id` (if available) and the bounding rectangle for the source (if appropriate). For example:

```
java.awt.event.ComponentEvent[COMPONENT_RESIZED (0, 0, 100x100)] on button0
```

4.3.2.3 ContainerEvent

The `ContainerEvent` class includes events that result from specific container operations.

Constants

public final static int CONTAINER_FIRST ★
public final static int CONTAINER_LAST ★

The `CONTAINER_FIRST` and `CONTAINER_LAST` constants hold the endpoints of the range of identifiers for `ContainerEvent` types.

public final static int COMPONENT_ADDED ★

The `COMPONENT_ADDED` constant identifies container events that occur because a component has been added to the container. The interface method `ContainerListener.componentAdded()` handles this event. Listening for this event is useful if a common listener should be attached to all components added to a container.

public final static int COMPONENT_REMOVED ★

The `COMPONENT_REMOVED` constant identifies container events that occur because a component has been removed from the container. The interface method `ContainerListener.componentRemoved()` handles this event.

Constructors

public ContainerEvent(Container source, int id, Component child) ★

The constructor creates a `ContainerEvent` with the given `source` (the container generating the event), to which the given `child` has been added or removed. The `id` field serves as the identifier of the event type. If system generated, the `id` will be one of the constants described previously. However, nothing stops you from creating your own `id` for your event types.

Methods

public Container getContainer() ★

The `getContainer()` method returns the container that generated the event.

public Component getComponent() ★

The `getComponent()` method returns the component that was added to or removed from the container.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is in turn called to build the string to display. At the `ContainerEvent` level, `paramString()` adds a string containing the event `id` (if available) along with the name of the child.

4.3.2.4 FocusEvent

The `FocusEvent` class contains the events that are generated when a component gets or loses focus. These may be either temporary or permanent focus changes. A temporary focus change is the result of something else happening, like a window appearing in front of you. Once the window is removed, focus is restored. A permanent focus change is usually the result of focus traversal, using the keyboard or the mouse: for example, you clicked in a text field to type in it, or used Tab to move to the next component. More programmatically, permanent focus changes are the result of calls to `Component.requestFocus()`.

Constants

public final static int FOCUS_FIRST ★
public final static int FOCUS_LAST ★

The `FOCUS_FIRST` and `FOCUS_LAST` constants hold the endpoints of the range of identifiers for `FocusEvent` types.

public final static int FOCUS_GAINED ★

The `FOCUS_GAINED` constant identifies focus events that occur because a component gains input focus. The `FocusListener.focusGained()` interface method handles this event.

public final static int FOCUS_LOST ★

The `FOCUS_LOST` constant identifies focus events that occur because a component loses input focus. The `FocusListener.focusLost()` interface method handles this event.

Constructors

public FocusEvent(Component source, int id, boolean temporary) ★

This constructor creates a `FocusEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system generated, the `id` will be one of the two constants described previously. However, nothing stops you from creating your own `id` for your event types. The `temporary` parameter is `true` if this event represents a temporary focus change.

public FocusEvent(Component source, int id) ★

This constructor creates a `FocusEvent` by calling the first constructor with the `temporary` parameter set to `false`; that is, it creates an event for a permanent focus change.

Methods

public boolean isTemporary() ★

The `isTemporary()` method returns `true` if the focus event describes a temporary focus change, `false` if the event describes a permanent focus change. Once set by the constructor, the setting is permanent.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is in turn called to build the string to display. At the `FocusEvent` level, `paramString()` adds a string showing the event `id` (if available) and whether or not it is temporary.

4.3.2.5 WindowEvent

The `WindowEvent` class encapsulates the window-oriented events.

Constants

public final static int WINDOW_FIRST ★
public final static int WINDOW_LAST ★

The `WINDOW_FIRST` and `WINDOW_LAST` constants hold the endpoints of the range of identifiers for `WindowEvent` types.

public final static int WINDOW_ICONIFIED ★

The `WINDOW_ICONIFIED` constant identifies window events that occur because the user iconifies a window. The `WindowListener.windowIconified()` interface method handles this event.

public final static int WINDOW_DEICONIFIED ★

The WINDOW_DEICONIFIED constant identifies window events that occur because the user de-iconifies a window. The interface method WindowListener.windowDeiconified() handles this event.

public final static int WINDOW_OPENED ★

The WINDOW_OPENED constant identifies window events that occur the first time a Frame or Dialog is made visible with show(). The interface method WindowListener.windowOpened() handles this event.

public final static int WINDOW_CLOSING ★

The WINDOW_CLOSING constant identifies window events that occur because the user wants to close a window. This is similar to the familiar event Event.WINDOW_DESTROY dealt with under 1.0 with frames. The WindowListener.windowClosing() interface method handles this event.

public final static int WINDOW_CLOSED ★

The WINDOW_CLOSED constant identifies window events that occur because a Frame or Dialog has finally closed, after hide() or destroy(). This comes after WINDOW_CLOSING, which happens when the user wants the window to close. The WindowListener.windowClosed() interface method handles this event.

NOTE If there is a call to System.exit() in the windowClosing() listener, the window will not be around to call windowClosed(), nor will other listeners know.

public final static int WINDOW_ACTIVATED ★

The WINDOW_ACTIVATED constant identifies window events that occur because the user brings the window to the front, either after showing the window, de-iconifying, or removing whatever was in front. The interface method WindowListener.windowActivated() handles this event.

public final static int WINDOW_DEACTIVATED ★

The WINDOW_DEACTIVATED constant identifies window events that occur because the user makes another window the active window. The interface method WindowListener.windowDeactivated() handles this event.

Constructors

public WindowEvent(Window source, int id) ★

This constructor creates a WindowEvent with the given source; the source is the object generating the event. The id field serves as the identifier of the event type. If system generated, the id will be one of the seven constants described previously. However, nothing stops you from creating your own id for your event types.

Methods

public Window getWindow() ★

The `getWindow()` method returns the `Window` that generated the event.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is in turn called to build the string to display. At the `WindowEvent` level, `paramString()` adds a string containing the event `id` (if available). In a call to `windowClosing()`, printing the parameter would yield:

```
java.awt.event.WindowEvent[WINDOW_CLOSING] on frame0
```

4.3.2.6 PaintEvent

The `PaintEvent` class encapsulates the paint-oriented events. There is no corresponding `PaintListener` class, so you cannot listen for these events. To process them, override the `paint()` and `update()` routines of `Component`. The `PaintEvent` class exists to ensure that events are serialized properly through the event queue.

Constants

public final static int PAINT_FIRST ★
public final static int PAINT_LAST ★

The `PAINT_FIRST` and `PAINT_LAST` constants hold the endpoints of the range of identifiers for `PaintEvent` types.

public final static int PAINT ★

The `PAINT` constant identifies paint events that occur because a component needs to be repainted. Override the `Component.paint()` method to handle this event.

public final static int UPDATE ★

The `UPDATE` constant identifies paint events that occur because a component needs to be updated before painting. This usually refreshes the display. Override the `Component.update()` method to handle this event.

Constructors

public PaintEvent(Component source, int id, Rectangle updateRect) ★

This constructor creates a `PaintEvent` with the given `source`. The source is the object whose display needs to be updated. The `id` field identifies the event type. If system generated, the `id` will be one of the two constants described previously. However, nothing stops you from creating your own `id` for your event types. `updateRect` represents the rectangular area of `source` that needs to be updated.

Methods

public Rectangle getUpdateRect()

The `getUpdateRect()` method returns the rectangular area within the `PaintEvent`'s source component that needs repainting. This area is set by either the constructor or the `setUpdateRect()` method.

public void setUpdateRect(Rectangle updateRect)

The `setUpdateRect()` method changes the area of the `PaintEvent`'s source component that needs repainting.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `PaintEvent` level, `paramString()` adds a string containing the event `id` (if available) along with the area requiring repainting (a clipping rectangle). If you peek in the event queue, one possible result may yield:

```
java.awt.event.PaintEvent[PAINT,updateRect=java.awt.Rectangle[x=0,y=0,
width=192,height=173]] on frame0
```

4.3.2.7 InputEvent

The `InputEvent` class provides the basis for the key and mouse input and movement routines. `KeyEvent` and `MouseEvent` provide the specifics of each.

Constants The constants of `InputEvent` help identify which modifiers are present when an input event occurs, as shown in Example 4-3. To examine the event modifiers and test for the presence of these masks, call `getModifiers()` to get the current set of modifiers.

public final static int ALT_MASK ★
public final static int CTRL_MASK ★
public final static int META_MASK ★
public final static int SHIFT_MASK ★

The first set of `InputEvent` masks are for the different modifier keys on the keyboard. They are often set to indicate which button on a multibutton mouse has been pressed.

public final static int BUTTON1_MASK ★
public final static int BUTTON2_MASK ★
public final static int BUTTON3_MASK ★

The button mask constants are equivalents for the modifier masks, allowing you to write more intelligible code for dealing with button events. `BUTTON2_MASK` is the same as `ALT_MASK`, and `BUTTON3_MASK` is the same as

META_MASK; BUTTON1_MASK currently isn't usable and is never set. For example, if you want to check whether the user pressed the second (middle) mouse button, you can test against BUTTON2_MASK rather than ALT_MASK. Example 4-3 demonstrates how to use these constants.

Constructors InputEvent is an abstract class with no public constructors.

Methods Unlike the Event class, InputEvent has an isAltDown() method to check the ALT_MASK setting.

public boolean isAltDown() ★

The isAltDown() method checks to see if ALT_MASK is set. If so, isAltDown() returns true; otherwise, it returns false.

public boolean isControlDown() ★

The isControlDown() method checks to see if CONTROL_MASK is set. If so, isControlDown() returns true; otherwise, it returns false.

public boolean isMetaDown() ★

The isMetaDown() method checks to see if META_MASK is set. If so, the method isMetaDown() returns true; otherwise, it returns false.

public boolean isShiftDown() ★

The isShiftDown() method checks to see if SHIFT_MASK is set. If so, the method isShiftDown() returns true; otherwise, it returns false.

public int getModifiers() ★

The getModifiers() method returns the current state of the modifier keys. For each modifier key pressed, a different flag is raised in the return argument. To check if a modifier is set, AND the return value with a flag and check for a nonzero value.

```
if ((ie.getModifiers() & MouseEvent.META_MASK) != 0) {
    System.out.println ("Meta is set");
}
```

public long getWhen() ★

The getWhen() method returns the time at which the event occurred. The return value is in milliseconds. Convert the long value to a Date to examine the contents. For example:

```
Date d = new Date (ie.getWhen());
```

public void consume() ★

This class overrides the AWTEvent.consume() method to make it public. Anyone, not just a subclass, can mark an InputEvent as consumed.

public boolean isConsumed() ★

This class overrides the `AWTEvent.isconsumed()` method to make it public. Anyone can find out if an `InputEvent` has been consumed.

4.3.2.8 KeyEvent

The `KeyEvent` class is a subclass of `InputEvent` for dealing with keyboard events. There are two fundamental key actions: key presses and key releases. These are represented by `KEY_PRESSED` and `KEY_RELEASED` events. Of course, it's inconvenient to think in terms of all these individual actions, so Java also keeps track of the "logical" keys you type. These are represented by `KEY_TYPED` events. For every keyboard key pressed, a `KeyEvent.KEY_PRESSED` event occurs; the key that was pressed is identified by one of the virtual keycodes from Table 4-4 and is available through the `getKeyCode()` method. For example, if you type an uppercase A, you will get two `KEY_PRESSED` events, one for shift (`VK_SHIFT`) and one for the "a" (`VK_A`). You will also get two `KeyEvent.KEY_RELEASED` events. However, there will only be one `KeyEvent.KEY_TYPED` event; if you call `getKeyChar()` for the `KEY_TYPED` event, the result will be the Unicode character "A" (type `char`). `KEY_TYPED` events do not happen for action-oriented keys like function keys.

Constants Like the `Event` class, numerous constants help you identify all the keyboard keys. Table 4-4 shows the constants that refer to these keyboard keys. The values are all declared `public static final int`. A few keys represent ASCII characters that have string equivalents like `\n`.

Table 4–4: Key Constants in Java 1.1

VK_ENTER	VK_0	VK_A	VK_F1	VK_ACCEPT
VK_BACK_SPACE	VK_1	VK_B	VK_F2	VK_CONVERT
VK_TAB	VK_2	VK_C	VK_F3	VK_FINAL
VK_CANCEL	VK_3	VK_D	VK_F4	VK_KANA
VK_CLEAR	VK_4	VK_E	VK_F5	VK_KANJI
VK_SHIFT	VK_5	VK_F	VK_F6	VK_MODECHANGE
VK_CONTROL	VK_6	VK_G	VK_F7	VK_NONCONVERT
VK_ALT	VK_7	VK_H	VK_F8	
VK_PAUSE	VK_8	VK_I	VK_F9	
VK_CAPS_LOCK	VK_9	VK_J	VK_F10	
VK_ESCAPE	VK_NUMPAD0	VK_K	VK_F11	
VK_SPACE	VK_NUMPAD1	VK_L	VK_F12	
VK_PAGE_UP	VK_NUMPAD2	VK_M	VK_DELETE	

Table 4–4: Key Constants in Java 1.1 (continued)

VK_PAGE_DOWN	VK_NUMPAD3	VK_N	VK_NUM_LOCK
VK_END	VK_NUMPAD4	VK_O	VK_SCROLL_LOCK
VK_HOME	VK_NUMPAD5	VK_P	VK_PRINTSCREEN
VK_LEFT	VK_NUMPAD6	VK_Q	VK_INSERT
VK_UP	VK_NUMPAD7	VK_R	VK_HELP
VK_RIGHT	VK_NUMPAD8	VK_S	VK_META
VK_DOWN	VK_NUMPAD9	VK_T	VK_BACK_QUOTE
VK_COMMA	VK_MULTIPLY	VK_U	VK_QUOTE
VK_PERIOD	VK_ADD	VK_V	VK_OPEN_BRACKET
VK_SLASH	VK_SEPARATER[a]	VK_W	VK_CLOSE_BRACKET
VK_SEMICOLON	VK_SUBTRACT	VK_X	
VK_EQUALS	VK_DECIMAL	VK_Y	
VK_BACK_SLASH	VK_DIVIDE	VK_Z	

[a] Expect VK_SEPARATOR to be added at some future point. This constant represents the numeric separator key on your keyboard.

public final static int VK_UNDEFINED ★

When a KEY_TYPED event happens, there is no keycode. If you ask for it, the getKeyCode() method returns VK_UNDEFINED.

public final static char CHAR_UNDEFINED ★

For KEY_PRESSED and KEY_RELEASED events that do not have a corresponding Unicode character to display (like Shift), the getKeyChar() method returns CHAR_UNDEFINED.

Other constants identify what the user did with a key.

public final static int KEY_FIRST ★
public final static int KEY_LAST ★

The KEY_FIRST and KEY_LAST constants hold the endpoints of the range of identifiers for KeyEvent types.

public final static int KEY_PRESSED ★

The KEY_PRESSED constant identifies key events that occur because a keyboard key has been pressed. To differentiate between action and non-action keys, call the isActionKey() method described later. The KeyListener.keyPressed() interface method handles this event.

public final static int KEY_RELEASED ★

The `KEY_RELEASED` constant identifies key events that occur because a keyboard key has been released. The `KeyListener.keyReleased()` interface method handles this event.

public final static int KEY_TYPED ★

The `KEY_TYPED` constant identifies a combination of a key press followed by a key release for a non-action oriented key. The `KeyListener.keyTyped()` interface method handles this event.

Constructors

public KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar) ★

This constructor* creates a `KeyEvent` with the given `source`; the source is the object generating the event. The `id` field identifies the event type. If system-generated, the `id` will be one of the constants above. However, nothing stops you from creating your own `id` for your event types. The `when` parameter represents the time the event happened. The `modifiers` parameter holds the state of the various modifier keys; masks to represent these keys are defined in the `InputEvent` class. Finally, `keyCode` is the virtual key that triggered the event, and `keyChar` is the character that triggered it.

The `KeyEvent` constructor throws the `IllegalArgumentException` run-time exception in two situations. First, if the `id` is `KEY_TYPED` and `keyChar` is `CHAR_UNDEFINED`, it throws an exception because if a key has been typed, it must be associated with a character. Second, if the `id` is `KEY_TYPED` and `keyCode` is not `VK_UNDEFINED`, it throws an exception because typed keys frequently represent combinations of key codes (for example, Shift struck with "a"). It is legal for a `KEY_PRESSED` or `KEY_RELEASED` event to contain both a `keyCode` and a `keyChar`, though it's not clear what such an event would represent.

Methods

public char getKeyChar() ★

The `getKeyChar()` method retrieves the Unicode character associated with the key in this `KeyEvent`. If there is no character, `CHAR_UNDEFINED` is returned.

public void setKeyChar(char KeyChar) ★

The `setKeyChar()` method allows you to change the character for the `KeyEvent`. You could use this method to convert characters to uppercase.

* Beta releases of Java 1.1 have an additional constructor that lacks the `keyChar` parameter. Comments in the code indicate that this constructor will be deleted prior to the 1.1.1 release.

public int getKeyCode() ★

The `getKeyCode()` method retrieves the virtual keycode (i.e., one of the constants in Table 4-4) of this `KeyEvent`.

public void setKeyCode(int keyCode) ★

The `setKeyCode()` method allows you to change the keycode for the `KeyEvent`. Changes you make to the `KeyEvent` are seen by subsequent listeners and the component's peer.

public void setModifiers(int modifiers) ★

The `setModifiers()` method allows you to change the modifier keys associated with a `KeyEvent` to `modifiers`. The parent class `InputEvent` already has a `getModifiers()` method that is inherited. Since this is your own personal copy of the `KeyEvent`, no other listener can find out about the change.

public boolean isActionKey() ★

The `isActionKey()` method allows you to check whether the key associated with the `KeyEvent` is an action key (e.g., function, arrow, keypad) or not (e.g., an alphanumeric key). For action keys, this method returns `true`; otherwise, it returns `false`. For action keys, the `keyChar` field usually has the value `CHAR_UNDEFINED`.

public static String getKeyText (int keyCode) ★

The static `getKeyText()` method returns the localized textual string for `keyCode`. For each nonalphanumeric virtual key, there is a key name (the "key text"); these names can be changed using the AWT properties. Table 4-5 shows the properties used to redefine the key names and the default name for each key.

Table 4–5: Key Text Properties

Property	Default	Property	Default
`AWT.accept`	Accept	`AWT.f8`	F8
`AWT.add`	NumPad +	`AWT.f9`	F9
`AWT.alt`	Alt	`AWT.help`	Help
`AWT.backQuote`	Back Quote	`AWT.home`	Home
`AWT.backSpace`	Backspace	`AWT.insert`	Insert
`AWT.cancel`	Cancel	`AWT.kana`	Kana
`AWT.capsLock`	Caps Lock	`AWT.kanji`	Kanji
`AWT.clear`	Clear	`AWT.left`	Left
`AWT.control`	Control	`AWT.meta`	Meta
`AWT.decimal`	NumPad .	`AWT.modechange`	Mode Change
`AWT.delete`	Delete	`AWT.multiply`	NumPad *
`AWT.divide`	NumPad /	`AWT.noconvert`	No Convert

Table 4–5: Key Text Properties (continued)

Property	Default	Property	Default
`AWT.down`	Down	`AWT.numLock`	Num Lock
`AWT.end`	End	`AWT.numpad`	NumPad
`AWT.enter`	Enter	`AWT.pause`	Pause
`AWT.escape`	Escape	`AWT.pgdn`	Page Down
`AWT.final`	Final	`AWT.pgup`	Page Up
`AWT.f1`	F1	`AWT.printScreen`	Print Screen
`AWT.f10`	F10	`AWT.quote`	Quote
`AWT.f11`	F11	`AWT.right`	Right
`AWT.f12`	F12	`AWT.scrollLock`	Scroll Lock
`AWT.f2`	F2	`AWT.separator`	NumPad ,
`AWT.f3`	F3	`AWT.shift`	Shift
`AWT.f4`	F4	`AWT.space`	Space
`AWT.f5`	F5	`AWT.subtract`	NumPad -
`AWT.f6`	F6	`AWT.tab`	Tab
`AWT.f7`	F7	`AWT.unknown`	Unknown `keyCode`
`AWT.up`	Up		

public static String getKeyModifiersText (int modifiers) ★

The static `getKeyModifiersText()` method returns the localized textual string for `modifiers`. The parameter `modifiers` is a combination of the key masks defined by the `InputEvent` class. As with the keys themselves, each modifier is associated with a textual name. If multiple modifiers are set, they are concatenated with a plus sign (+) separating them. Similar to `getKeyText()`, the strings are localized because for each modifier, an awt property is available to redefine the string. Table 4-6 lists the properties and the default modifier names.

Table 4–6: Key Modifiers Text Properties

Property	Default
`AWT.alt`	Alt
`AWT.control`	Ctrl
`AWT.meta`	Meta
`AWT.shift`	Shift

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `KeyEvent` level, paramString() adds a textual string for the id (if available), the text for the key (if available from getKeyText()), and modifiers (from getKeyModifiersText()). A key press event would result in something like the following:

```
java.awt.event.KeyEvent[KEY_PRESSED,keyCode=118,
F7,modifiers=Ctrl+Shift] on textfield0
```

4.3.2.9 MouseEvent

The MouseEvent class is a subclass of InputEvent for dealing with mouse events.

Constants

public final static int MOUSE_FIRST ★
public final static int MOUSE_LAST ★

The MOUSE_FIRST and MOUSE_LAST constants hold the endpoints of the range of identifiers for MouseEvent types.

public final static int MOUSE_CLICKED ★

The MOUSE_CLICKED constant identifies mouse events that occur when a mouse button is clicked. A mouse click consists of a mouse press and a mouse release. The MouseListener.mouseClicked() interface method handles this event.

public final static int MOUSE_DRAGGED ★

The MOUSE_DRAGGED constant identifies mouse events that occur because the mouse is moved over a component with a mouse button pressed. The interface method MouseMotionListener.mouseDragged() handles this event.

public final static int MOUSE_ENTERED ★

The MOUSE_ENTERED constant identifies mouse events that occur when the mouse first enters a component. The MouseListener.mouseEntered() interface method handles this event.

public final static int MOUSE_EXITED ★

The MOUSE_EXISTED constant identifies mouse events that occur because the mouse leaves a component's space. The MouseListener.mouseExited() interface method handles this event.

public final static int MOUSE_MOVED ★

The MOUSE_MOVED constant identifies mouse events that occur because the mouse is moved without a mouse button down. The interface method MouseMotionListener.mouseMoved() handles this event.

public final static int MOUSE_PRESSED ★

The MOUSE_PRESSED constant identifies mouse events that occur because a mouse button has been pressed. The MouseListener.mousePressed() interface method handles this event.

public final static int MOUSE_RELEASED ★

The `MOUSE_RELEASED` constant identifies mouse events that occur because a mouse button has been released. The `MouseListener.mouseReleased()` interface method handles this event.

Constructors

public MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger) ★

This constructor creates a `MouseEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system-generated, the `id` will be one of the constants described in the previous section. However, nothing stops you from creating your own `id` for your event types. The `when` parameter represents the time the event happened. The `modifiers` parameter holds the state of the various modifier keys, using the masks defined for the `InputEvent` class, and lets you determine which button was pressed. (`x`, `y`) represents the coordinates of the event relative to the origin of `source`, while `clickCount` designates the number of consecutive times the mouse button was pressed within an indeterminate time period. Finally, the `popupTrigger` parameter signifies whether this mouse event should trigger the display of a `PopupMenu`, if one is available. (The `PopupMenu` class is discussed in Chapter 10, *Would You Like to Choose from the Menu?*)

Methods

public int getX() ★

The `getX()` method returns the current x coordinate of the event relative to the source.

public int getY() ★

The `getY()` method returns the current y coordinate of the event relative to the source.

public synchronized Point getPoint() ★

The `getPoint()` method returns the current x and y coordinates of the event relative to the event source.

public synchronized void translatePoint(int x, int y) ★

The `translatePoint()` method translates the x and y coordinates of the `MouseEvent` instance by `x` and `y`. This method functions similarly to the `Event.translate()` method.

public int getClickCount() ★

The `getClickCount()` method retrieves the current `clickCount` setting for the event.

public boolean isPopupTrigger() ★

The `isPopupTrigger()` method retrieves the state of the `popupTrigger` setting for the event. If this method returns `true` and the source of the event has an associated `PopupMenu`, the event should be used to display the menu, as shown in the following code. Since the action the user performs to raise a pop-up menu is platform specific, this method lets you raise a pop-up menu without worrying about what kind of event took place. You only need to call `isPopupTrigger()` and show the menu if it returns `true`.

```
public void processMouseEvent(MouseEvent e) {
    if (e.isPopupTrigger())
        aPopup.show(e.getComponent(), e.getX(), e.getY());
    super.processMouseEvent(e);
}
```

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `MouseEvent` level, a textual string for the `id` (if available) is tacked on to the coordinates, modifiers, and click count. A mouse down event would result in something like the following:

```
java.awt.event.MouseEvent[MOUSE_PRESSED,(5,7),mods=0,clickCount=2] on textfield0
```

4.3.2.10 ActionEvent

The `ActionEvent` class is the first higher-level event class. It encapsulates events that signify that the user is doing something with a component. When the user selects a button, list item, or menu item, or presses the Return key in a text field, an `ActionEvent` passes through the event queue looking for listeners.

Constants

public final static int ACTION_FIRST ★
public final static int ACTION_LAST ★

The `ACTION_FIRST` and `ACTION_LAST` constants hold the endpoints of the range of identifiers for `ActionEvent` types.

public final static int ACTION_PERFORMED ★

The `ACTION_PERFORMED` constant represents when a user activates a component. The `ActionListener.actionPerformed()` interface method handles this event.

public static final int ALT_MASK ★
public static final int CTRL_MASK ★
public static final int META_MASK ★
public static final int SHIFT_MASK ★

Similar to the mouse events, action events have `modifiers`. However, they are not automatically set by the system, so they don't help you see what modifiers were pressed when the event occurred. You may be able to use these constants if you are generating your own action events. To see the value of an action event's modifiers, call `getModifiers()`.

Constructors

public ActionEvent(Object source, int id, String command) ★

This constructor creates an `ActionEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system-generated, the `id` will be `ACTION_PERFORMED`. However, nothing stops you from creating your own `id` for your event types. The `command` parameter is the event's action command. Ideally, the action command should be some locale-independent string identifying the user's action. Most components that generate action events set this field to the selected item's label by default.

public ActionEvent(Object source, int id, String command, int modifiers) ★

This constructor adds `modifiers` to the settings for an `ActionEvent`. This allows you to define action-oriented events that occur only if certain modifier keys are pressed.

Methods

public String getActionCommand() ★

The `getActionCommand()` method retrieves the `command` field from the event. It represents the command associated with the object that triggered the event. The idea behind the action command is to differentiate the command associated with some event from the displayed content of the event source. For example, the action command for a button may be Help. However, what the user sees on the label of the button could be a string localized for the environment of the user. Instead of having your event handler look for 20 or 30 possible labels, you can test whether an event has the action command Help.

public int getModifiers() ★

The `getModifiers()` method returns the state of the modifier keys. For each one set, a different flag is raised in the method's return value. To check if a modifier is set, AND the return value with a flag, and check for a nonzero value.

public String paramString() ★

When you call the `toString()` method of an `AWTEvent`, the `paramString()` method is called in turn to build the string to display. At the `ActionEvent` level, `paramString()` adds a textual string for the event `id` (if available), along with the `command` from the constructor. When the user selects a `Button` with the action command Help, printing the resulting event yields:

```
java.awt.event.ActionEvent[ACTION_PERFORMED,cmd=Help] on button0
```

4.3.2.11 AdjustmentEvent

The `AdjustmentEvent` class is another higher-level event class. It encapsulates events that represent scrollbar motions. When the user moves the slider of a scrollbar or scroll pane, an `AdjustmentEvent` passes through the event queue looking for listeners. Although there is only one type of adjustment event, there are five subtypes represented by constants `UNIT_DECREMENT`, `UNIT_INCREMENT`, and so on.

Constants

public final static int ADJUSTMENT_FIRST ★
public final static int ADJUSTMENT_LAST ★

The `ADJUSTMENT_FIRST` and `ADJUSTMENT_LAST` constants hold the endpoints of the range of identifiers for `AdjustmentEvent` types.

public final static int ADJUSTMENT_VALUE_CHANGED ★

The `ADJUSTMENT_VALUE_CHANGED` constant identifies adjustment events that occur because a user moves the slider of a `Scrollbar` or `ScrollPane`. The `AdjustmentListener.adjustmentValueChanged()` interface method handles this event.

public static final int UNIT_DECREMENT ★

`UNIT_DECREMENT` identifies adjustment events that occur because the user selects the increment arrow.

public static final int UNIT_INCREMENT ★

`UNIT_INCREMENT` identifies adjustment events that occur because the user selects the decrement arrow.

public static final int BLOCK_DECREMENT ★

`BLOCK_DECREMENT` identifies adjustment events that occur because the user selects the block decrement area, between the decrement arrow and the slider.

public static final int BLOCK_INCREMENT ★

`BLOCK_INCREMENT` identifies adjustment events that occur because the user selects the block increment area, between the increment arrow and the slider.

public static final int TRACK ★

TRACK identifies adjustment events that occur because the user selects the slider and drags it. Multiple adjustment events of this subtype usually occur consecutively.

Constructors

public AdjustmentEvent(Adjustable source, int id, int type, int value) ★

This constructor creates an AdjustmentEvent with the given source; the source is the object generating the event. The id field serves as the identifier of the event type. If system-generated, the id of the AdjustmentEvent will be ADJUSTMENT_VALUE_CHANGED. However, nothing stops you from creating your own id for your event types. The type parameter is normally one of the five subtypes, with value being the current setting of the slider, but is not restricted to that.

Methods

public Adjustable getAdjustable() ★

The getAdjustable() method retrieves the Adjustable object associated with this event—that is, the event's source.

public int getAdjustmentType() ★

The getAdjustmentType() method retrieves the type parameter from the constructor. It represents the subtype of the current event and, if system-generated, is one of the following constants: UNIT_DECREMENT, UNIT_INCREMENT, BLOCK_DECREMENT, BLOCK_INCREMENT, or TRACK.

public int getValue() ★

The getValue() method retrieves the value parameter from the constructor. It represents the current setting of the adjustable object.

public String paramString() ★

When you call the toString() method of an AWTEvent, the paramString() method is called to help build the string to display. At the AdjustableEvent level, paramString() adds a textual string for the event id (if available), along with a textual string of the type (if available), and value. For example:

```
java.awt.event.AdjustableEvent[ADJUSTMENT_VALUE_CHANGED,
adjType=TRACK,value=27] on scrollbar0
```

4.3.2.12 ItemEvent

The ItemEvent class is another higher-level event class. It encapsulates events that occur when the user selects a component, like ActionEvent. When the user selects a checkbox, choice, list item, or checkbox menu item, an `ItemEvent` passes through the event queue looking for listeners. Although there is only one type of `ItemEvent`, there are two subtypes represented by the constants `SELECTED` and `DESELECTED`.

Constants

public final static int ITEM_FIRST ★
public final static int ITEM_LAST ★

The `ITEM_FIRST` and `ITEM_LAST` constants hold the endpoints of the range of identifiers for `ItemEvent` types.

public final static int ITEM_STATE_CHANGED ★

The `ITEM_STATE_CHANGED` constant identifies item events that occur because a user selects a component, thus changing its state. The interface method `ItemListener.itemStateChanged()` handles this event.

public static final int SELECTED ★

`SELECTED` indicates that the user selected the item.

public static final int DESELECTED ★

`DESELECTED` indicates that the user deselected the item.

Constructors

public ItemEvent(ItemSelectable source, int id, Object item, int stateChange) ★

This constructor creates a `ItemEvent` with the given `source`; the source is the object generating the event. The `id` field serves as the identifier of the event type. If system-generated, the `id` will be `ITEM_STATE_CHANGE`. However, nothing stops you from creating your own `id` for your event types. The `item` parameter represents the text of the item selected: for a `Checkbox`, this would be its label, for a `Choice` the current selection. For your own events, this parameter could be virtually anything, since its type is `Object`.

Methods

public ItemSelectable getItemSelectable() ★

The `getItemSelectable()` method retrieves the `ItemSelectable` object associated with this event—that is, the event's source.

public Object getItem() ★

The `getItem()` method returns the `item` that was selected. This usually represents some text to help identify the source but could be nearly anything for user-generated events.

public int getStateChange() ★

The getStateChange() method returns the stateChange parameter from the constructor and, if system generated, is either SELECTED or DESELECTED.

public String paramString() ★

When you call the toString() method of an AWTEvent, the paramString() method is called in turn to build the string to display. At the ItemEvent level, paramString() adds a textual string for the event id (if available), along with a textual string indicating the value of stateChange (if available) and item. For example:

```
java.awt.event.ItemEvent[ITEM_STATE_CHANGED,item=Help,
stateChange=SELECTED] on checkbox1
```

4.3.2.13 TextEvent

The TextEvent class is yet another higher-level event class. It encapsulates events that occur when the contents of a TextComponent have changed, although is not required to have a TextComponent source. When the contents change, either programmatically by a call to setText() or because the user typed something, a TextEvent passes through the event queue looking for listeners.

Constants

public final static int TEXT_FIRST ★
public final static int TEXT_LAST ★

The TEXT_FIRST and TEXT_LAST constants hold the endpoints of the range of identifiers for TextEvent types.

public final static int TEXT_VALUE_CHANGED ★

The TEXT_VALUE_CHANGED constant identifies text events that occur because a user changes the contents of a text component. The interface method TextListener.textValueChanged() handles this event.

Constructors

public TextEvent(Object source, int id) ★

This constructor creates a TextEvent with the given source; the source is the object generating the event. The id field identifies the event type. If system-generated, the id will be TEXT_VALUE_CHANGE. However, nothing stops you from creating your own id for your event types.

Method

public String paramString() ★

When you call the toString() method of an AWTEvent, the paramString() method is called in turn to build the string to display. At the TextEvent level, paramString() adds a textual string for the event id (if available).

4.3.3 Event Listener Interfaces and Adapters

Java 1.1 has 11 event listener interfaces, which specify the methods a class must implement to receive different kinds of events. For example, the `ActionListener` interface defines the single method that is called when an `ActionEvent` occurs. These interfaces replace the various event-handling methods of Java 1.0: `action()` is now the `actionPerformed()` method of the `ActionListener` interface, `mouseUp()` is now the `mouseReleased()` method of the `MouseListener` interface, and so on. Most of the listener interfaces have a corresponding adapter class, which is an abstract class that provides a null implementation of all the methods in the interface. (Although an adapter class has no abstract methods, it is declared `abstract` to remind you that it must be subclassed.) Rather than implementing a listener interface directly, you have the option of extending an adapter class and overriding only the methods you care about. (Much more complex adapters are possible, but the adapters supplied with AWT are very simple.) The adapters are available for the listener interfaces with multiple methods. (If there is only one method in the listener interface, there is no need for an adapter.)

This section describes Java 1.1's listener interfaces and adapter classes. It's worth noting here that Java 1.1 does not allow you to modify the original event when you're writing an event handler.

4.3.3.1 ActionListener

The `ActionListener` interface contains the one method that is called when an `ActionEvent` occurs. It has no adapter class. For an object to listen for action events, it is necessary to call the `addActionListener()` method with the class that implements the `ActionListener` interface as the parameter. The method `addActionListener()` is implemented by `Button`, `List`, `MenuItem`, and `TextField` components. Other components don't generate action events.

public abstract void actionPerformed(ActionEvent e) ★

The `actionPerformed()` method is called when a component is selected or activated. Every component is activated differently; for a `List`, activation means that the user has double-clicked on an entry. See the appropriate section for a description of each component.

`actionPerformed()` is the Java 1.1 equivalent of the `action()` method in the 1.0 event model.

4.3.3.2 AdjustmentListener

The `AdjustmentListener` interface contains the one method that is called when an `AdjustmentEvent` occurs. It has no adapter class. For an object to listen for adjustment events, it is necessary to call `addAdjustmentListener()` with the class that implements the `AdjustmentListener` interface as the parameter. The `addAdjustmentListener()` method is implemented by the `Scrollbar` component and the `Adjustable` interface. Other components don't generate adjustment events.

public abstract void adjustmentValueChanged(AdjustmentEvent e) ★

The `adjustmentValueChanged()` method is called when a slider is moved. The `Scrollbar` and `ScrollPane` components have sliders, and generate adjustment events when the sliders are moved. (The `TextArea` and `List` components also have sliders, but do not generate adjustment events.) See the appropriate section for a description of each component.

There is no real equivalent to `adjustmentValueChanged()` in Java 1.0; to work with scrolling events, you had to override the `handleEvent()` method.

4.3.3.3 ComponentListener and ComponentAdapter

The `ComponentListener` interface contains four methods that are called when a `ComponentEvent` occurs; component events are used for general actions on components, like moving or resizing a component. The adapter class corresponding to `ComponentListener` is `ComponentAdapter`. If you care only about one or two of the methods in `ComponentListener`, you can subclass the adapter and override only the methods that you are interested in. For an object to listen for component events, it is necessary to call `Component.addComponentListener()` with the class that implements the interface as the parameter.

public abstract void componentResized(ComponentEvent e) ★

The `componentResized()` method is called when a component is resized (for example, by a call to `Component.setSize()`).

public abstract void componentMoved(ComponentEvent e) ★

The `componentMoved()` method is called when a component is moved (for example, by a call to `Component.setLocation()`).

public abstract void componentShown(ComponentEvent e) ★

The `componentShown()` method is called when a component is shown (for example, by a call to `Component.show()`).

public abstract void componentHidden(ComponentEvent e) ★

The `componentHidden()` method is called when a component is hidden (for example, by a call to `Component.hide()`).

4.3.3.4 ContainerListener and ContainerAdapter

The `ContainerListener` interface contains two methods that are called when a `ContainerEvent` occurs; container events are generated when components are added to or removed from a container. The adapter class for `ContainerListener` is `ContainerAdapter`. If you care only about one of the two methods in `ContainerListener`, you can subclass the adapter and override only the method that you are interested in. For a container to listen for container events, it is necessary to call `Container.addContainerListener()` with the class that implements the interface as the parameter.

public abstract void componentAdded(ContainerEvent e) ★

The `componentAdded()` method is called when a component is added to a container (for example, by a call to `Container.add()`).

public abstract void componentRemoved(ContainerEvent e) ★

The `componentRemoved()` method is called when a component is removed from a container (for example, by a call to `Container.remove()`).

4.3.3.5 FocusListener and FocusAdapter

The `FocusListener` interface has two methods, which are called when a `FocusEvent` occurs. Its adapter class is `FocusAdapter`. If you care only about one of the methods, you can subclass the adapter and override the method you are interested in. For an object to listen for a `FocusEvent`, it is necessary to call the `Component.addFocusListener()` method with the class that implements the `FocusListener` interface as the parameter.

public abstract void focusGained(FocusEvent e) ★

The `focusGained()` method is called when a component receives input focus, usually by the user clicking the mouse in the area of the component.

This method is the Java 1.1 equivalent of `Component.gotFocus()` in the Java 1.0 event model.

public abstract void focusLost(FocusEvent e) ★

The `focusLost()` method is called when a component loses the input focus.

This method is the Java 1.1 equivalent of `Component.lostFocus()` in the Java 1.0 event model.

4.3.3.6 ItemListener

The `ItemListener` interface contains the one method that is called when an `ItemEvent` occurs. It has no adapter class. For an object to listen for an `ItemEvent`, it is necessary to call `addItemListener()` with the class that implements the `ItemListener` interface as the parameter. The `addItemListener()` method is implemented by the `Checkbox`, `CheckboxMenuItem`, `Choice`, and `List` components. Other components don't generate item events.

public abstract void itemStateChanged(ItemEvent e) ★

The `itemStateChanged()` method is called when a component's state is modified. Every component is modified differently; for a `List`, modifying the component means single-clicking on an entry. See the appropriate section for a description of each component.

4.3.3.7 KeyListener and KeyAdapter

The `KeyListener` interface contains three methods that are called when a `KeyEvent` occurs; key events are generated when the user presses or releases keys. The adapter class for `KeyListener` is `KeyAdapter`. If you only care about one or two of the methods in `KeyListener`, you can subclass the adapter and only override the methods that you are interested in. For an object to listen for key events, it is necessary to call `Component.addKeyListener()` with the class that implements the interface as the parameter.

public abstract void keyPressed(KeyEvent e) ★

The `keyPressed()` method is called when a user presses a key. A key press is, literally, just what it says. A key press event is called for every key that is pressed, including keys like Shift and Control. Therefore, a `KEY_PRESSED` event has a virtual key code identifying the physical key that was pressed; but that's not the same as a typed character, which usually consists of several key presses (for example, Shift+A to type an uppercase A). The `keyTyped()` method reports actual characters.

This method is the Java 1.1 equivalent of `Component.keyDown()` in the Java 1.0 event model.

public abstract void keyReleased(KeyEvent e) ★

The `keyReleased()` method is called when a user releases a key. Like the `keyPressed()` method, when dealing with `keyReleased()`, you must think of virtual key codes, not characters.

This method is the Java 1.1 equivalent of `Component.keyUp()` in the Java 1.0 event model.

public abstract void keyTyped(KeyEvent e) ★

The `keyTyped()` method is called when a user types a key. The method `keyTyped()` method reports the actual character typed. Action-oriented keys, like function keys, do not trigger this method being called.

4.3.3.8 MouseListener and MouseAdapter

The `MouseListener` interface contains five methods that are called when a non-motion oriented `MouseEvent` occurs; mouse events are generated when the user presses or releases a mouse button. (Separate classes, `MouseMotionListener` and

MouseMotionAdapter, are used to handle mouse motion events; this means that you can listen for mouse clicks only, without being bothered by thousands of mouse motion events.) The adapter class for MouseListener is MouseAdapter. If you care about only one or two of the methods in MouseListener, you can subclass the adapter and override only the methods that you are interested in. For an object to listen for mouse events, it is necessary to call the method Window.addWindowListener() with the class that implements the interface as the parameter.

public abstract void mouseEntered(MouseEvent e) ★

The mouseEntered() method is called when the mouse first enters the bounding area of the component.

This method is the Java 1.1 equivalent of Component.mouseEnter() in the Java 1.0 event model.

public abstract void mouseExited(MouseEvent e) ★

The mouseExited() method is called when the mouse leaves the bounding area of the component.

This method is the Java 1.1 equivalent of Component.mouseExit() in the Java 1.0 event model.

public abstract void mousePressed(MouseEvent e) ★

The mousePressed() method is called each time the user presses a mouse button within the component's space.

This method is the Java 1.1 equivalent of Component.mouseDown() in the Java 1.0 event model.

public abstract void mouseReleased(MouseEvent e) ★

The mouseReleased() method is called when the user releases the mouse button after a mouse press. The user does not have to be over the original component any more; the original component (i.e., the component in which the mouse was pressed) is the source of the event.

This method is the Java 1.1 equivalent of Component.mouseUp() in the Java 1.0 event model.

public abstract void mouseClicked(MouseEvent e) ★

The mouseClicked() method is called once each time the user clicks a mouse button; that is, once for each mouse press/mouse release combination.

4.3.3.9 MouseMotionListener and MouseMotionAdapter

The MouseMotionListener interface contains two methods that are called when a motion-oriented MouseEvent occurs; mouse motion events are generated when the user moves the mouse, whether or not a button is pressed. (Separate classes, MouseListener and MouseAdapter, are used to handle mouse clicks and entering/exiting components. This makes it easy to ignore mouse motion events, which are very frequent and can hurt performance. You should listen only for mouse motion events if you specifically need them.) MouseMotionAdapter is the adapter class for MouseMotionListener. If you care about only one of the methods in MouseMotionListener, you can subclass the adapter and override only the method that you are interested in. For an object to listen for mouse motion events, it is necessary to call Component.addMouseMotionListener() with the class that implements the interface as the parameter.

public abstract void mouseMoved(MouseEvent e) ★

The mouseMoved() method is called every time the mouse moves within the bounding area of the component, and no mouse button is pressed.

This method is the Java 1.1 equivalent of Component.mouseMove() in the Java 1.0 event model.

public abstract void mouseDragged(MouseEvent e) ★

The mouseDragged() method is called every time the mouse moves while a mouse button is pressed. The source of the MouseEvent is the component that was under the mouse when it was first pressed.

This method is the Java 1.1 equivalent of Component.mouseDrag() in the Java 1.0 event model.

4.3.3.10 TextListener

The TextListener interface contains the one method that is called when a TextEvent occurs. It has no adapter class. For an object to listen for a TextEvent, it is necessary to call addTextListener() with the class that implements the TextListener interface as the parameter. The addTextListener() method is implemented by the TextComponent class, and thus the TextField and TextArea components. Other components don't generate text events.

public abstract void textValueChanged(TextEvent e) ★

The textValueChanged() method is called when a text component's contents are modified, either by the user (by a keystroke) or programmatically (by the setText() method).

4.3.3.11 WindowListener and WindowAdapter

The WindowListener interface contains seven methods that are called when a WindowEvent occurs; window events are generated when something changes the visibility or status of a window. The adapter class for WindowListener is WindowAdapter.

If you care about only one or two of the methods in `WindowListener`, you can subclass the adapter and override only the methods that you are interested in. For an object to listen for window events, it is necessary to call the method `Window.addWindowListener()` or `Dialog.addWindowListener()` with the class that implements the interface as the parameter.

public abstract void windowOpened(WindowEvent e) ★
: The `windowOpened()` method is called when a `Window` is first opened.

public abstract void windowClosing(WindowEvent e) ★
: The `windowClosing()` method is triggered whenever the user tries to close the `Window`.

public abstract void windowClosed(WindowEvent e) ★
: The `windowClosed()` method is called after the `Window` has been closed.

public abstract void windowIconified(WindowEvent e) ★
: The `windowIconified()` method is called whenever a user iconifies a `Window`.

public abstract void windowDeiconified(WindowEvent e) ★
: The `windowDeiconified()` method is called when the user deiconifies the `Window`.

public abstract void windowActivated(WindowEvent e) ★
: The `windowActivated()` method is called whenever a `Window` is brought to the front.

public abstract void windowDeactivated(WindowEvent e) ★
: The `windowDeactivated()` method is called when the `Window` is sent away from the front, either through iconification, closing, or another window becoming active.

4.3.4 AWTEventMulticaster

The `AWTEventMulticaster` class is used by AWT to manage the listener queues for the different events, and for sending events to all interested listeners when they occur (multicasting). Ordinarily, you have no need to work with this class or know about its existence. However, if you wish to create your own components that have their own set of listeners, you can use the class instead of implementing your own event-delivery system. See "Constructor methods" in this section for more on how to use the `AWTEventMulticaster`.

`AWTEventMulticaster` looks like a strange beast, and to some extent, it is. It contains methods to add and remove every possible kind of listener and implements all of the listener interfaces (11 as of Java 1.1). Because it implements all the listener interfaces, you can pass an event multicaster as an argument wherever you expect any kind of listener. However, unlike a class you might implement to listen for a specific kind of event, the multicaster includes machinery for maintaining chains of listeners. This explains the rather odd signatures for the `add()` and `remove()` methods. Let's look at one in particular:

```
public static ActionListener add(ActionListener first, ActionListener second)
```

This method takes two `ActionListeners` and returns another `ActionListener`. The returned listener is actually an event multicaster that contains the two listeners given as arguments in a linked list. However, because it implements the `ActionListener` interface, it is just as much an `ActionListener` as any class you might write; the fact that it contains two (or more) listeners inside it is irrelevant. Furthermore, both arguments can also be event multicasters, containing arbitrarily long chains of action listeners; in this case, the returned listener combines the two chains. Most often, you will use add to add a single listener to a chain that you're building, like this:

```
actionListenerChain=AWTEventMulticaster.add(actionListenerChain,
                                            newActionListener);
```

`actionListenerChain` is an `ActionListener`—but it is also a multicaster holding a chain of action listeners. To start a chain, use `null` for the first argument. You rarely need to call the `AWTEventMulticaster` constructor. `add()` is a static method, so you can use it with either argument set to `null` to start the chain.

Now that you can maintain chains of listeners, how do you use them? Simple; just deliver your event to the appropriate method in the chain. The multicaster takes care of sending the event to all the listeners it contains:

```
actionListenerChain.actionPerformed(new ActionEvent(...));
```

Variables

protected EventListener a; ★
protected EventListener b; ★

The `a` and `b` event listeners each consist of a chain of `EventListeners`.

Constructor methods

protected AWTEventMulticaster(EventListener a, EventListener b) ★

The constructor is protected. It creates an `AWTEventMulticaster` instance from the two chains of listeners. An instance is automatically created for you when you add your second listener by calling an `add()` method.

Listener methods

These methods implement all of the listener interfaces. Rather than repeating all the descriptions, the methods are just listed.

public void actionPerformed(ActionEvent e) ★
public void adjustmentValueChanged(AdjustmentEvent e) ★
public void componentAdded(ContainerEvent e) ★
public void componentHidden(ComponentEvent e) ★
public void componentMoved(ComponentEvent e) ★
public void componentRemoved(ContainerEvent e) ★
public void componentResized(ComponentEvent e) ★
public void componentShown(ComponentEvent e) ★
public void focusGained(FocusEvent e) ★
public void focusLost(FocusEvent e) ★
public void itemStateChanged(ItemEvent e) ★
public void keyPressed(KeyEvent e) ★
public void keyReleased(KeyEvent e) ★
public void keyTyped(KeyEvent e) ★
public void mouseClicked(MouseEvent e) ★
public void mouseDragged(MouseEvent e) ★
public void mouseEntered(MouseEvent e) ★
public void mouseExited(MouseEvent e) ★
public void mouseMoved(MouseEvent e) ★
public void mousePressed(MouseEvent e) ★
public void mouseReleased(MouseEvent e) ★
public void textValueChanged(TextEvent e) ★
public void windowActivated(WindowEvent e) ★
public void windowClosed(WindowEvent e) ★
public void windowClosing(WindowEvent e) ★
public void windowDeactivated(WindowEvent e) ★
public void windowDeiconified(WindowEvent e) ★
public void windowIconified(WindowEvent e) ★
public void windowOpened(WindowEvent e) ★

These methods broadcast the event given as an argument to all the listeners.

Support methods

There is an `add()` method for every listener interface. Again, I've listed them with a single description.

public static ActionListener add(ActionListener first, ActionListener second) ★
public static AdjustmentListener add(AdjustmentListener first, AdjustmentListener second) ★

public static ComponentListener add(ComponentListener first, ComponentListener second) ★
public static ContainerListener add(ContainerListener first, ContainerListener second) ★
public static FocusListener add(FocusListener first, FocusListener second) ★
public static ItemListener add(ItemListener first, ItemListener second) ★
public static KeyListener add(KeyListener first, KeyListener second)
public static MouseListener add(MouseListener first, MouseListener second) ★
public static MouseMotionListener add(MouseMotionListener first, MouseMotionListener second) ★
public static TextListener add(TextListener first, TextListener second) ★
public static WindowListener add(WindowListener first, WindowListener second) ★

These methods combine the listener sets together; they are called by the "add listener" methods of the various components. Usually, the `first` parameter is the initial listener chain, and the `second` parameter is the listener to add. However, nothing forces that. The combined set of listeners is returned.

protected static EventListener addInternal(EventListener first, EventListener second) ★

The `addInternal()` method is a support routine for the various `add()` methods. The combined set of listeners is returned.

Again, there are `remove()` methods for every listener type, and I've economized on the descriptions.

public static ComponentListener remove(ComponentListener list, ComponentListener oldListener) ★
public static ContainerListener remove(ContainerListener list, ContainerListener oldListener) ★
public static FocusListener remove(FocusListener list, FocusListener oldListener) ★
public static KeyListener remove(KeyListener list, KeyListener oldListener) ★
public static MouseMotionListener remove(MouseMotionListener list, MouseMotionListener oldListener) ★
public static MouseListener remove(MouseListener list, MouseListener oldListener) ★
public static WindowListener remove(WindowListener list, WindowListener oldListener) ★
public static ActionListener remove(ActionListener list, ActionListener oldListener) ★
public static ItemListener remove(ItemListener list, ItemListener oldListener) ★
public static AdjustmentListener remove(AdjustmentListener list, AdjustmentListener oldListener) ★
public static TextListener remove(TextListener list, TextListener oldListener) ★

These methods remove `oldListener` from the list of listeners, `list`. They are called by the "remove listener" methods of the different components. If `oldListener` is not found in the `list`, nothing happens. All these methods return the new list of listeners.

protected static EventListener removeInternal(EventListener list, EventListener oldListener) ★

The `removeInternal()` method is a support routine for the various `remove()` methods. It removes `oldListener` from the list of listeners, `list`. Nothing happens if `oldListener` is not found in the `list`. The new set of listeners is returned.

protected EventListener remove(EventListener oldListener) ★

This `remove()` method removes `oldListener` from the `AWTEventMulticaster`. It is a support routine for `removeInternal()`.

protected void saveInternal(ObjectOutputStream s, String k) throws IOException ★

The `saveInternal()` method is a support method for serialization.

4.3.4.1 Using an event multicaster

Example 4-4 shows how to use `AWTEventMulticaster` to create a component that generates `ItemEvents`. The `AWTEventMulticaster` is used in the `addItemListener()` and `removeItemListener()` methods. When it comes time to generate the event in `processEvent()`, the `itemStateChanged()` method is called to notify anyone who might be interested. The item event is generated when a mouse button is clicked; we just count the number of clicks to determine whether an item was selected or deselected. Since we do not have any mouse listeners, we need to enable mouse events with `enableEvents()` in the constructor, as shown in the following example.

Example 4–4: Using an AWTEventMulticaster

```
// Java 1.1 only
import java.awt.*;
import java.awt.event.*;
class ItemEventComponent extends Component implements ItemSelectable {
    boolean selected;
    int i = 0;
    ItemListener itemListener = null;
    ItemEventComponent () {
        enableEvents (AWTEvent.MOUSE_EVENT_MASK);
    }
    public Object[] getSelectedObjects() {
        Object o[] = new Object[1];
        o[0] = new Integer (i);
        return o;
    }
    public void addItemListener (ItemListener l) {
        itemListener = AWTEventMulticaster.add (itemListener, l);
    }
    public void removeItemListener (ItemListener l) {
        itemListener = AWTEventMulticaster.remove (itemListener, l);
    }
    public void processEvent (AWTEvent e) {
```

Example 4–4: Using an AWTEventMulticaster (continued)

```
            if (e.getID() == MouseEvent.MOUSE_PRESSED) {
                if (itemListener != null) {
                    selected = !selected;
                    i++;
                    itemListener.itemStateChanged (
                        new ItemEvent (this, ItemEvent.ITEM_STATE_CHANGED,
                            getSelectedObjects(),
                            (selected?ItemEvent.SELECTED:ItemEvent.DESELECTED)));
                }
            }
        }
    }

    public class ItemFrame extends Frame implements ItemListener {
        ItemFrame () {
            super ("Listening In");
            ItemEventComponent c = new ItemEventComponent ();
            add (c, "Center");
            c.addItemListener (this);
            c.setBackground (SystemColor.control);
            setSize (200, 200);
        }
        public void itemStateChanged (ItemEvent e) {
            Object[] o = e.getItemSelectable().getSelectedObjects();
            Integer i = (Integer)o[0];
            System.out.println (i);
        }
        public static void main (String args[]) {
            ItemFrame f = new ItemFrame();
            f.show();
        }
    }
```

The `ItemFrame` displays just an `ItemEventComponent` and listens for its item events.

The `EventQueue` class lets you manage Java 1.1 events directly. You don't usually need to manage events yourself; the system takes care of event delivery behind the scene. However, should you need to, you can acquire the system's event queue by calling `Toolkit.getSystemEventQueue()`, peek into the event queue by calling `peekEvent()`, or post new events by calling `postEvent()`. All of these operations may be restricted by the `SecurityManager`. You should not remove the events from the queue (i.e., don't call `getNextEvent()`) unless you really mean to.

Constructors

public EventQueue() ★

This constructor creates an `EventQueue` for those rare times when you need to manage your own queue of events. More frequently, you just work with the system event queue acquired through the `Toolkit`.

Methods

public synchronized AWTEvent peekEvent() ★

The `peekEvent()` method looks into the event queue and returns the first event, without removing that event. If you modify the event, your modifications are reflected in the event still on the queue. The returned object is an instance of `AWTEvent`. If the queue is empty, `peekEvent()` returns `null`.

public synchronized AWTEvent peekEvent(int id) ★

This `peekEvent()` method looks into the event queue for the first event of the specified type. `id` is one of the integer constants from an `AWTEvent` subclass or an integer constant of your own. If there are no events of the appropriate type on the queue, `peekEvent()` returns `null`.

Note that a few of the `AWTEvent` classes have both event types and subtypes; `peekEvent()` checks event types only and ignores the subtype. For example, to find an `ItemEvent`, you would call `peekEvent(ITEM_STATE_CHANGED)`. However, a call to `peekEvent(SELECTED)` would return `null`, since `SELECTED` identifies an `ItemEvent` subtype.

public synchronized void postEvent(AWTEvent theEvent) ★

This version of `postEvent()` puts a new style (Java1.1) event on the event queue.

public synchronized AWTEvent getNextEvent() throws InterruptedException ★

The `getNextEvent()` method removes an event from the queue. If the queue is empty, the call waits. The object returned is the item taken from the queue; it is either an `Event` or an `AWTEvent`. If the method call is interrupted, the method `getNextEvent()` throws an `InterruptedException`.