SFDC 1019

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

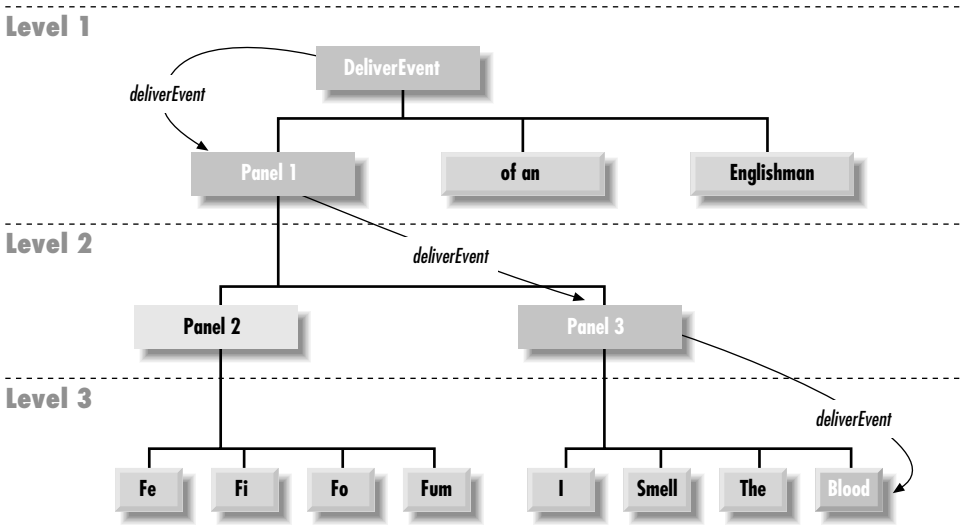

Figure 4–2: deliverEvent screen model

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

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.