**Section 6.7  
Looking Back: The Java 1.0 Style of Event Handling**

WHEN JAVA 1.1 WAS RELEASED, it included a large number of changes from Java 1.0. One of the biggest changes was the introduction of an entirely new system for handling events. It is the newer Java 1.1 style of event handling that I have been discussing in this chapter. The newer style is also used in the most recent versions of Java, and it will almost certainly continue to be used in the future. The new style of event handling is more flexible and more efficient than the old style. Unfortunately, it is also more complicated. The big advantage of Java 1.0 event handling is that it isn't necessary to set up listeners for events. Its big disadvantage is that you are not able to set up listeners -- but the disadvantage only becomes apparent in projects that use more than just a few classes and objects. For small projects, it can still make sense to use Java 1.0 style event handling, if you want to go through the trouble of learning a whole new event-handling architecture. It is also true that a few people out there are still using older browsers that support Java 1.0 but not Java 1.1. If you want those people to be able to use your Java programs, you need to write them in Java 1.0.

The features of Java 1.0 that are superceded by newer features of Java 1.1 are said to be deprecated. Deprecated features are still part of the language, but their use is discouraged. The idea is that they might be dropped from Java altogether in some future version (although I don't think that is likely in the near term.) The whole Java 1.0 event-handling architecture is deprecated in Java 1.1 and beyond.

In this section, I'll give an outline of the Java 1.0 style of event handling. For more complete information, see a Java reference or look at Chapters 5 and 6 in the first edition of this on-line text, which should still be available on the Web at <http://math.hws.edu/eck/cs124/javanotes1/index.html>.

One important note: You can use either Java 1.0 or Java 1.1 style event handling. But **you can't mix them in the same applet or program**. For a given project, you have to decide which model you want to use and stick to it.

**The Event Model in Java 1.0**

In Java 1.0 there is just one class of events, java.awt.Event. Every event is generated by a component, which is called the target of the event. The target of an event object, evt, is given by the public instance variable evt.target. Other instance variables carry other information about various types of events. For example, the coordinates of a mouse-related event are given by evt.x and evt.y.

In Java 1.1 style event handling, an event is sent only to objects that are listening for the event. (This is where the Java 1.1 style gets its advantage in efficiency: Events are not processed unless they have been enabled, usually by registering a listener.) In the Java 1.0 style, whenever any event occurs, the system calls a method named handleEvent() in the target component. For many types of events, handleEvent() will in turn call a special purpose event-handling routine such as mouseDown() or keyDown(). Some of these event-handling routines are discussed below. The handleEvent() method might or might not actually handle the event. It returns a boolean value to the system to indicate whether the event was handled. If the event is not handled by the target component, and if that component is contained in some other component -- such as a Panel or Applet -- then the system gives the container component a chance to handle the event by calling the handleEvent() method of the container. This process can continue up a chain of containment until a top-level component such as an applet or frame is reached. If the top-level component doesn't handle the event, then the event is ignored.

This might sound very complicated but what it comes down to in most cases is this: Mouse and keyboard events are handled by the component to which they are targeted (usually a canvas object, if not the applet itself). The top-level applet class does all the other event-handling for the program. Events from buttons, text fields, choice menus, and so on are allowed to filter up to the top level where they are handled by a method in the applet class. (In fact, in any case where this simplified model is not appropriate, you should really be using Java 1.1 style event handling.)

To deal with certain types of events, such as those generated by scroll bars, it is necessary to override the handleEvent() method itself. But for the more common types of events, there are special purpose event-handling methods that you can override.

**Mouse and Keyboard Events in Java 1.0**

As the user moves the mouse and presses buttons on the mouse, several event-handling methods can be called. The most useful mouse-related methods are:

public boolean mouseDown(Event evt, int x, int y) {

. . . // respond to fact that user has pressed mouse button

return true;

}

public boolean mouseUp(Event evt, int x, int y) {

. . . // respond to fact that the user has released mouse button

return true;

}

public boolean mouseDrag(Event evt, int x, int y) {

. . . // respond to fact that mouse has moved, while

// user is holding down a mouse button

return true;

}

public boolean mouseMove(Event evt, int x, int y) {

. . . // respond to fact that mouse has moved, while

// user is NOT holding down a mouse button

return true;

}

Note that these are boolean-valued methods. In almost all cases, you should return true, which indicates that you have handled the event. In the rare cases where you want to let the event be passed on to the next possible handler, return false instead.

If the user clicks the mouse somewhere in the rectangle occupied by a component, the mouseDown() method is called when the user presses the button, and the mouseUp() method when the user releases it. If the user moves the mouse while holding the button down, the computer will call mouseDrag() over and over as the mouse moves. The mouseMove() method is called when the user moves the mouse without holding a button down.

In all of these methods, the parameters x and y give the horizontal and vertical position of the mouse, in coordinates appropriate to the component. The parameter evt carries full information about the event that caused the method to be called. (The x and y coordinates have been pulled out of this Event object for your convenience.) You can check whether the user was holding down the shift key, the control key, or the Meta key when the event occured by calling the boolean-valued methods evt.shiftDown(), evt.controlDown(), and evt.metaDown(). Holding down the Meta key is equivalent to pressing the right mouse button, so evt.metaDown() also returns true if the right mouse button is down. (For some reason, there is no method for testing whether the Alt key -- or middle mouse button -- is down. Instead you have to test "if ((evt.modifiers & Event.ALT\_MASK) != 0)"!)

The applet at the bottom of this page uses Java 1.0 style handling of mouse events. If you are interested, you can find the source code for this applet in the file [TrackLines.java](http://math.hws.edu/eck/cs124/javanotes3/source/TrackLines.java).

As for keyboard events in Java 1.0, every time the user presses a key on the keyboard, two events are generated: one when the user presses the key and one when the user releases the key. A component can be programmed to respond to these events by overriding the keyDown() and keyUp() methods, which are defined as follows:

public boolean keyDown(Event evt, int key) {

. . . // respond to the fact that a key has been pressed

return true;

}

public boolean keyUp(Event evt, int key) {

. . . // respond to the fact that a key has been released

return true;

}

The key parameter in these methods tells you which key was pressed by the user. You might be surprised to see that this parameter has type int rather than char. This is because characters aren't the only things that the user can type! The user can also press "action keys" such as the arrow keys and the function keys F1, F2, etc.

If the user actually types a character, then the key parameter tells you which character was typed. Because key is of type int, you have to use a type-cast to discover which character was typed:

char typedChar = (char)key;

If the user pressed one of the action keys, then the value of the key parameter will be a special constant value that specifies which key was pressed. The value will be one of the following predefined constants: Event.UP, Event.DOWN, Event.LEFT, Event.RIGHT, Event.HOME, Event.END, Event.PGUP, Event.PGDN, or Event.F1 through Event.F12. The first four of these, which are probably the most useful, correspond to the up, down, left, and right arrow keys.

When dealing with keyboard events, there is the complication of having to worry about the input focus. A component that processes keyboard events should keep track of whether it has the input focus, so that it can change its appearance when it has the focus. A focus event is sent to a component whenever it gains or loses the input focus. In Java 1.0, this means calling the event-handling methods "public boolean gotFocus(Event evt, Object what)" and "public boolean lostFocus(Event evt, Object what)". You can override these methods in a component that needs to respond to focus events. In general, both of the parameters to these methods can be ignored.

**Action Events in Java 1.0**

Besides mouse and keyboard events, there are the events that are generated when the user interacts with graphical interface components such as buttons, menus, and text boxes. (The user actually causes these other events with the mouse or keyboard, but they are translated by the system into more meaningful terms.) For many of these events, the system calls the action() method, which takes the form

public boolean action(Event evt, Object arg) {

. . . // respond to the action event

return true; // or return false, if action not handled

}

To deal with action events, you should override the action() method in your subclass of Applet. The method should handle all the action events from any component contained in the applet. When you write an action() method, you know exactly which components might generate action events for it to handle, so you can write it to handle just those components.

The second parameter in the action() method, arg, contains some information relevant to the event. What type of information arg contains depends on the type of object that generated the event. The first parameter, evt, has an instance variable, evt.target, which tells which component first received the event. For an action event, this is also the component that generated the event in the first place. This means that an action() method often looks something like this:

public boolean action(Event evt, Object arg) {

if (evt.target == button1) {

// handle a click on button1

return true;

}

else if (evt.target == button2) {

// handle a click on button2

return true;

}

else if (evt.target == colorChoice) {

// handle a selection from Choice object colorChoice

return true;

}

.

. // handle other possible targets

.

else

return super.action(evt,arg);

}

(In simple cases, when there are only a few components to worry about, you might not even have to check evt.target; the second parameter, arg, might contain all the information you need.)

Here are the details about the action events that can be generated by various types of components:

* A **Button** generates an action event when the user clicks on the button. The arg parameter of action() is a string that is equal to the button's label. (If you want to use this parameter, you can type cast arg to type String.)
* A **Choice** generates an action event when the user selects one of the items in the Choice menu. The arg parameter is the text of the item that the user selected.
* A **TextField** generates an action event if the user presses return while typing in the input box. The arg parameter is a string that gives the contents of the box.
* A **Checkbox** generates an action event when the user clicks on it. The arg is an object of type Boolean that gives the new state of the box. (An object of type Boolean contains a value belonging to the primitive type boolean. A boolean value is not an object. "Wrapping" it inside a Boolean object makes it possible to treat it as an object. To get the boolean value contained in arg, you could say "boolean newState = ((Boolean)arg).booleanValue();". It's generally much easier to use the Checkbox's getState() method to find out its state.)

Although this brief overview of events in Java 1.0 does not give you enough information to write complex applications using the Java 1.0 event model, that's not something you should be doing anyway. There is probably enough information on this page for you to use Java 1.0 event handling in simple applets.