

Java, Java, Java
Object-Oriented Problem Solving
Third Edition

R. Morelli and R. Walde
Trinity College
Hartford, CT

June 25, 2017

13.5 The Java Event Model

As we saw in Chapter 4, whatever happens while the computer is running is classified as an event. Every keystroke and mouse click, every time a disk is inserted into a disk drive, an event is generated. The handling of events are an important element of GUI programming. Therefore, before we begin discussing how to design GUIs, it will be useful to review the main concepts of Java's **event model**.

When a Java program is running, events generated by the hardware are passed up through the operating system (and through the browser, for applets) to the program. Those events that belong to the program must be handled by the program (refer to Fig. 4.18 in Chapter 4). For example, if you click your browser's menu bar, that event will be handled by the browser itself. If you click a button contained in the Java program, that event should be handled by the program.

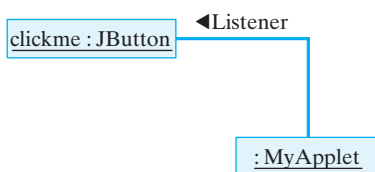
In Java, whenever something happens within a GUI component, an event object is generated and passed to the *event listener* that has been registered to handle that component's events. You've seen numerous examples of this process in earlier chapters, but we've included a simple example to serve as a reminder.

Suppose you create a `JButton` in a GUI as follows:

```
private JButton clickme = new JButton("ClickMe");
```

Whenever the user clicks the `JButton`, an `ActionEvent` is generated. In order to handle these events, the GUI must register the `JButton` with a listener object that listens for action events. This can be done in an applet's `init()` method or in an application's constructor method, as in this example:

```
public MyGUI() {
    // Add clickme to the GUI and assign it a listener
    add(clickme);
    clickme.addActionListener(this);
}
```



In this case, we have designated the GUI itself (`this`) as an `ActionListener` for `clickme` (Fig. 13.5). A **listener** is any object that implements a *listener interface*, which is one of the interfaces derived from `java.util.EventListener`. An `ActionListener` is an object that listens for and receives `ActionEvents`.

Figure 13.5: The GUI listens for action events on the `JButton`.

In order to complete the event-handling code, the GUI must implement the `ActionListener` interface. As Figure 13.6 shows, implementing an interface is a matter of declaring the interface in the class heading and implementing the methods contained in the interface, in this case the `actionPerformed()` method.

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

public class MyGUI extends JFrame
    implements ActionListener {
    private JButton clickme = new JButton("ClickMe");

    public MyGUI() {
        // Add clickme to the GUI and assign it a listener
        getContentPane().add(clickme);
        clickme.addActionListener(this);
        setSize(200,200);
        setVisible(true);
    } // init()
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == clickme) {
            clickme.setText(clickme.getText()+"*");
        }
    } // actionPerformed()
    public static void main(String args[]) {
        MyGUI gui = new MyGUI();
    }
} // MyGUI
```

Figure 13.6: A simple GUI application that handles action events on a `JButton`.

Now that we have implemented the code in Figure 13.6, whenever the user clicks `clickme`, that action is encapsulated within an `ActionEvent` object and passed to the `actionPerformed()` method. This method contains Java code that will handle the user's action in an appropriate way. For this example, it modifies the button's label by appending an asterisk to it each time it is clicked. Figure 13.7 depicts the sequence of actions and events that occur when the the user clicks a button.

Figure 13.7: A UML depiction of the sequence of actions and events that take place when a button is clicked. The vertical lines represent time lines, with time running from top to bottom. The arrows between lines represent messages passing between objects.

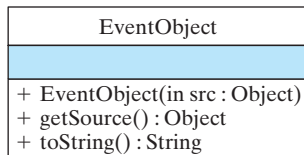
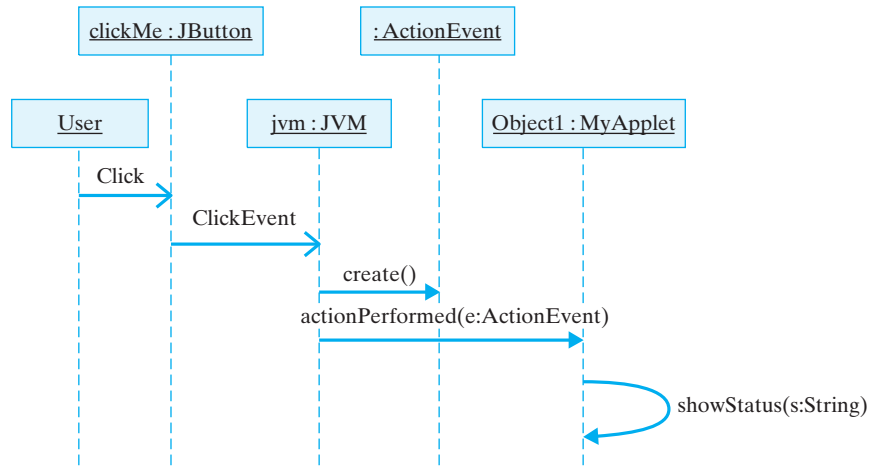


Figure 13.8: An EventObject. The `getSource()` method is used to get the object that caused the event.

The methods used to handle the `ActionEvent` are derived from the `java.util.EventObject` class, the root class for all events (Fig. 13.8). Our example (Fig. 13.6) uses the `getSource()` method to get a reference to the object that generated the event. To see what information is contained in an event object, we can use the `toString()` method to print a string representation of the event that was generated. Here's what it displays:

```
java.awt.event.ActionEvent[ACTION_PERFORMED,cmd=ClickMe]
on javax.swing.JButton[,58,5,83x27,
layout=javax.swing.OverlayLayout]
```

As you can see, the event generated was an `ACTION_PERFORMED` event, in response to the `ClickMe` command. The source of the event was the `JButton`.

13.5.1 Event Classes

Although the event model is the same for both AWT and Swing classes, the Swing package introduces many additional events. Table 13.1 lists the events that are generated by both AWT and Swing components. You already have worked with some of these. We have written GUIs that handled `ActionEvents` for `JButtons` and `JTextFields` in preceding chapters.

In viewing Table 13.1, it's important to remember that the classes listed there are arranged in a hierarchy. This will affect the events that a particular object can generate. For example, a `JButton` is a `JComponent` (Fig. 13.2), so in addition to generating `ActionEvents` when the user clicks on it, it can also generate `MouseEvents` when the user moves the mouse over it. Similarly, because a `JTextField` is also a `JComponent`, it can generate `KeyEvents` as well as `ActionEvents`.

Note that the more generic events, such as those that involve moving, focusing, or resizing a component, are associated with the more generic components. For example, the `JComponent` class contains methods that are used to manage `ComponentEvents`. Because they are subclasses of `JComponent`, `JButtons` and `JTextFields` can also use these meth-

TABLE 13.1 Java's AWTEvents for each Component type (Original source: David Flanagan, *Java in a Nutshell*, 2d ed., O'Reilly Associates, 1997. Modified for Swing components.)

Components	Events	Description
Button, JButton	ActionEvent	User clicked button
CheckBox, JCheckBox	ItemEvent	User toggled a checkbox
CheckboxMenuItem, JCheckboxMenuItem	ItemEvent	User toggled a checkbox
Choice, JPopupMenu	ItemEvent	User selected a choice
Component, JComponent	ComponentEvent	Component was moved or resized
	FocusEvent	Component acquired or lost focus
	KeyEvent	User typed a key
	MouseEvent	User manipulated the mouse
Container, JContainer	ContainerEvent	Component added/removed from container
List, JList	ActionEvent	User double-clicked a list item
	ItemEvent	User clicked a list item
Menu, JMenu	ActionEvent	User selected menu item
Scrollbar, JScrollbar	AdjustmentEvent	User moved scrollbar
TextComponent, JTextComponent	TextEvent	User edited text
TextField, JTextField	ActionEvent	User typed Enter key
Window, JWindow	WindowEvent	User manipulated window

ods. Defining the more generic methods in the `JComponent` superclass is another example of the effective use of inheritance.



JAVA EFFECTIVE DESIGN **Inheritance.** The higher a method is defined in the inheritance hierarchy, the broader is its use.

Table 13.2 lists events that are new with the Swing classes. Some of the events apply to new components. For example, `JTable` and `JTree` do not have AWT counterparts. Other events provide Swing components with capabilities that are not available in their AWT counterparts. For example, a `CaretEvent` allows the programmer to have control over mouse clicks that occur within a text component.

TABLE 13.2 Some of the events that are defined in the Swing library.

Component	Events	Description
JPopupMenu	PopupMenuEvent	User selected a choice
JComponent	AncestorEvent	An event occurred in an ancestor
JList	ListSelectionEvent	User double-clicked a list item
	ListDataEvent	List's contents were changed
JMenu	MenuEvent	User selected menu item
JTextComponent	CaretEvent	Mouse clicked in text
	UndoableEditEvent	An undoable edit has occurred
JTable	TableModelEvent	Items added/removed from table
	TableColumnModelEvent	A table column was moved
JTree	TreeModelEvent	Items added/removed from tree
	TreeSelectionEvent	User selected a tree node
	TreeExpansionEvent	User expanded or collapsed a tree node
JWindow	WindowEvent	User manipulated window

Tables 13.1 and 13.2 provide only a brief summary of these classes and Swing components. For further details you should consult the JDK online documentation at



<http://java.sun.com/j2se/1.5.0/docs/api/>

SELF-STUDY EXERCISES

EXERCISE 13.2 Is it possible to register a component with more than one listener?

EXERCISE 13.3 Is it possible for a component to have two different kinds of listeners?

13.6 CASE STUDY: Designing a Basic GUI

What elements make up a basic user interface? If you think about all of the various interfaces you've encountered—and don't just limit yourself to computers—they all have the following elements:

- Some way to provide help/guidance to the user.
- Some way to allow input of information.
- Some way to allow output of information.
- Some way to control the interaction between the user and the device.

Think about the interface on a beverage machine. Printed text on the machine will tell you what choices you have, where to put your money, and what to do if something goes wrong. The coin slot is used to input money. There's often some kind of display to tell you how much money you've inserted. And there's usually a bunch of buttons and levers that let you control the interaction with the machine.

These same kinds of elements make up the basic computer interface. Designing a Graphical User Interface is primarily a process of choosing components that can effectively perform the tasks of input, output, control, and guidance.



JAVA EFFECTIVE DESIGN **User Interface.** A user interface must effectively perform the tasks of input, output, control, and guidance.

In the programs we designed in the earlier chapters, we used two different kinds of interfaces. In the *command-line* interface, we used printed prompts to inform the user, typed commands for data entry and user control, and printed output to report results. Our GUI interfaces used `JLabels` to guide and prompt the user, `JTextFields` and `JTextAreas` as basic input and output devices, and either `JButtons` or `JTextFields` for user control.

Let's begin by building a basic GUI in the form of a Java application. To keep the example as close as possible to the GUIs we've already used, we will build it out of the following Swing components: `JLabel`, `JTextField`, `JTextArea`, and `JButton`.