CHAPTER 16

EVENT-DRIVEN PROGRAMMING

Objectives

- To describe events, event sources, and event classes (§16.2).
- To define listener classes, register listener objects with the source object, and write the code to handle events (§16.3).
- To define listener classes using inner classes (§16.4).
- To define listener classes using anonymous inner classes (§16.5).
- To explore various coding styles for creating and registering listeners (§16.6).
- To get input from text field upon clicking a button (§16.7).
- To write programs to deal with `WindowEvent` (§16.8).
- To simplify coding for listener classes using listener interface adapters (§16.9).
- To write programs to deal with `MouseEvent` (§16.10).
- To write programs to deal with `KeyEvent` (§16.11).
- To use the `javax.swing.Timer` class to control animations (§16.12).

16.1 Introduction

problem

Suppose you wish to write a GUI program that lets the user enter the loan amount, annual interest rate, and number of years and click the *Compute Loan* button to obtain the monthly payment and total payment, as shown in Figure 16.1(a). How do you accomplish the task? You have to use event-driven programming to write the code to respond to the button-clicking event.

(a) (b) (c) (d)

FIGURE 16.1 (a) The program computes loan payments. (b)–(d) A flag is rising upward.

problem

Suppose you wish to write a program that animates a rising flag, as shown in Figure 16.1(b)–(d). How do you accomplish the task? There are several ways to solve this problem. An effective one is to use a timer in event-driven programming, which is the subject of this chapter.

In event-driven programming, code is executed when an event occurs (e.g., a button click, a mouse movement, or a timer). §14.6, "Example: The `ActionListener` Interface," gave you a taste of event-driven programming. You probably have many questions, such as why a listener class is defined to implement the `ActionListener` interface. This chapter will give you all the answers.

16.2 Event and Event Source

event

When you run a Java GUI program, the program interacts with the user, and the events drive its execution. An *event* can be defined as a signal to the program that something has happened. Events are triggered either by external user actions, such as mouse movements, button clicks, and keystrokes, or by internal program activities, such as a timer. The program can choose to respond to or ignore an event.

fire event
source object

The component that creates an event and fires it is called the *source object* or *source component*. For example, a button is the source object for a button-clicking action event. An event is an instance of an event class. The root class of the event classes is `java.util.EventObject`. The hierarchical relationships of some event classes are shown in Figure 16.2.

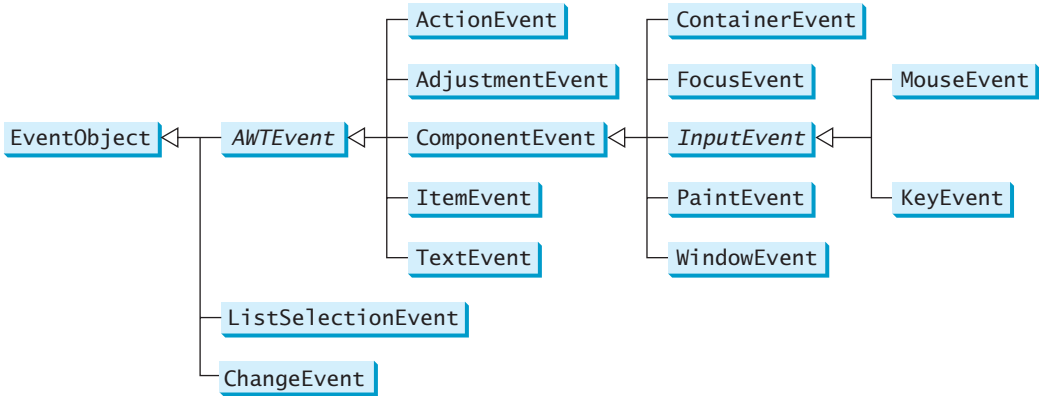


FIGURE 16.2 An event is an object of the `EventObject` class.

An event object contains whatever properties are pertinent to the event. You can identify the source object of an event using the `getSource()` instance method in the `EventObject` class. The subclasses of `EventObject` deal with special types of events, such as action events, window events, component events, mouse events, and key events. Table 16.1 lists external user actions, source objects, and event types fired.

getSource()

TABLE 16.1 User Action, Source Object, and Event Type

User Action	*Source Object*	*Event Type Fired*
Click a button	`JButton`	`ActionEvent`
Press return on a text field	`JTextField`	`ActionEvent`
Select a new item	`JComboBox`	`ItemEvent`, `ActionEvent`
Select item(s)	`JList`	`ListSelectionEvent`
Click a check box	`JCheckBox`	`ItemEvent`, `ActionEvent`
Click a radio button	`JRadioButton`	`ItemEvent`, `ActionEvent`
Select a menu item	`JMenuItem`	`ActionEvent`
Move the scroll bar	`JScrollBar`	`AdjustmentEvent`
Move the scroll bar	`JSlider`	`ChangeEvent`
Window opened, closed, iconified, deiconified, or closing	`Window`	`WindowEvent`
Mouse pressed, released, clicked, entered, or exited	`Component`	`MouseEvent`
Mouse moved or dragged	`Component`	`MouseEvent`
Key released or pressed	`Component`	`KeyEvent`
Component added or removed from the container	`Container`	`ContainerEvent`
Component moved, resized, hidden, or shown	`Component`	`ComponentEvent`
Component gained or lost focus	`Component`	`FocusEvent`

Note

If a component can fire an event, any subclass of the component can fire the same type of event. For example, every GUI component can fire `MouseEvent`, `KeyEvent`, `FocusEvent`, and `ComponentEvent`, since `Component` is the superclass of all GUI components.

Note

All the event classes in Figure 16.2 are included in the `java.awt.event` package except `ListSelectionEvent` and `ChangeEvent`, which are in the `javax.swing.event` package. AWT events were originally designed for AWT components, but many Swing components fire them.

16.3 Listeners, Registrations, and Handling Events

Java uses a delegation-based model for event handling: a source object fires an event, and an object interested in the event handles it. The latter object is called a *listener*. For an object to be a listener for an event on a source object, two things are needed, as shown in Figure 16.3.

listener

ActionEvent/ActionListener

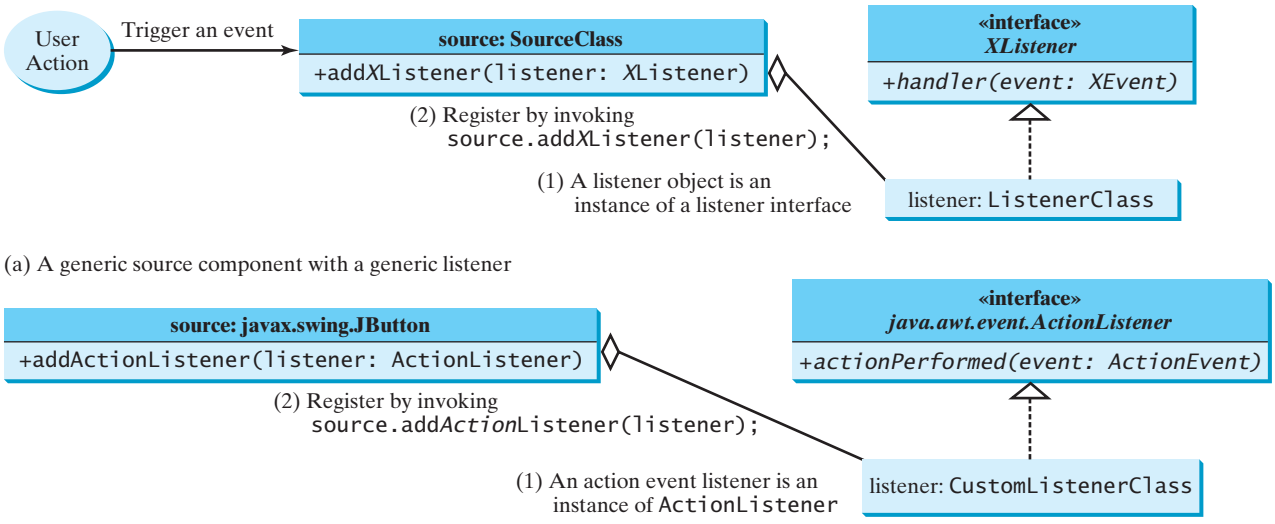


FIGURE 16.3 A listener must be an instance of a listener interface and must be registered with a source component.

listener interface
*X*Listener/*X*Event

1. The listener object must be an instance of the corresponding event-listener interface to ensure that the listener has the correct method for processing the event. Java provides a listener interface for every type of event. The listener interface is usually named `XListener` for `XEvent`, with the exception of `MouseMotionListener`. For example, the corresponding listener interface for `ActionEvent` is `ActionListener`; each listener for `ActionEvent` should implement the `ActionListener` interface. Table 16.2 lists event types, the corresponding listener interfaces, and the methods defined in the listener interfaces. The listener interface contains the method(s), known as the *handler(s)*, for processing the event.

handler

register listener

2. The listener object must be registered by the source object. Registration methods depend on the event type. For `ActionEvent`, the method is `addActionListener`. In general, the method is named `addXListener` for `XEvent`. A source object may fire several types of events. It maintains, for each event, a list of registered listeners and notifies them by invoking the *handler* of the listener object to respond to the event, as shown in Figure 16.4. (Figure 16.4 shows the internal implementation of a source class. You don't have to know how a source class such as `JButton` is implemented in order to use it. Nevertheless, this knowledge will help you to understand the Java event-driven programming framework).

Let's revisit Listing 14.8, HandleEvent.java. Since a `JButton` object fires `ActionEvent`, a listener object for `ActionEvent` must be an instance of `ActionListener`, so the listener class implements `ActionListener` in line 34. The source object invokes `addActionListener(listener)` to register a listener, as follows:

create source object
create listener object
register listener

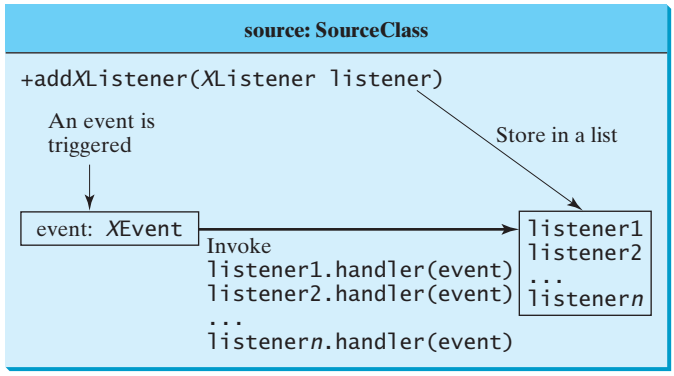
```
JButton jbtOK = new JButton("OK"); // Line 7 in Listing 14.8
ActionListener listener1
   = new OKListenerClass(); // Line 18 in Listing 14.8
jbtOK.addActionListener(listener1); // Line 20 in Listing 14.8
```

When you click the button, the `JButton` object fires an `ActionEvent` and passes it to invoke the listener's `actionPerformed` method to handle the event.

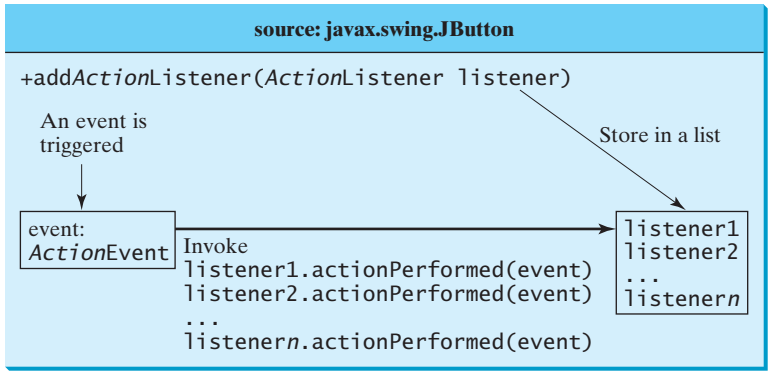
TABLE 16.2 Events, Event Listeners, and Listener Methods

Event Class (Handlers)	*Listener Interface*	*Listener Methods*
`ActionEvent`	`ActionListener`	`actionPerformed(ActionEvent)`
`ItemEvent`	`ItemListener`	`itemStateChanged(ItemEvent)`
`MouseEvent`	`MouseListener`	`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mouseClicked(MouseEvent)`
	`MouseMotionListener`	`mouseDragged(MouseEvent)`
		`mouseMoved(MouseEvent)`
`KeyEvent`	`KeyListener`	`keyPressed(KeyEvent)`
		`keyReleased(KeyEvent)`
		`keyTyped(KeyEvent)`
`WindowEvent`	`WindowListener`	`windowClosing(WindowEvent)`
		`windowOpened(WindowEvent)`
		`windowIconified(WindowEvent)`
		`windowDeiconified(WindowEvent)`
		`windowClosed(WindowEvent)`
		`windowActivated(WindowEvent)`
		`windowDeactivated(WindowEvent)`
`ContainerEvent`	`ContainerListener`	`componentAdded(ContainerEvent)`
		`componentRemoved(ContainerEvent)`
`ComponentEvent`	`ComponentListener`	`componentMoved(ComponentEvent)`
		`componentHidden(ComponentEvent)`
		`componentResized(ComponentEvent)`
		`componentShown(ComponentEvent)`
`FocusEvent`	`FocusListener`	`focusGained(FocusEvent)`
		`focusLost(FocusEvent)`
`AdjustmentEvent`	`AdjustmentListener`	`adjustmentValueChanged(AdjustmentEvent)`
`ChangeEvent`	`ChangeListener`	`stateChanged(ChangeEvent)`
`ListSelectionEvent`	`ListSelectionListener`	`valueChanged(ListSelectionEvent)`

The event object contains information pertinent to the event, which can be obtained using the methods, as shown in Figure 16.5. For example, you can use `e.getSource()` to obtain the source object in order to determine whether it is a button, a check box, or a radio button. For an action event, you can use `e.getWhen()` to obtain the time when the event occurs.



(a) Internal function of a generic source object



(b) Internal function of a `JButton` object

FIGURE 16.4 The source object notifies the listeners of the event by invoking the handler of the listener object.

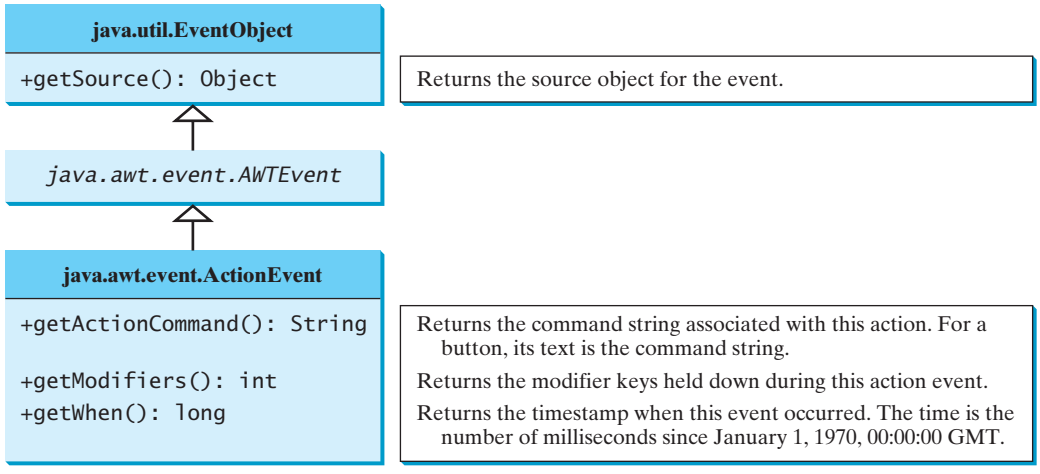


FIGURE 16.5 You can obtain useful information from an event object.

We now write a program that uses two buttons to control the size of a circle, as shown in Figure 16.6.

first version

We will develop this program incrementally. First we write a program in Listing 16.1 that displays the user interface with a circle in the center (line 14) and two buttons in the bottom (line 15).

FIGURE 16.6 The user clicks the *Enlarge* and *Shrink* buttons to enlarge and shrink the size of the circle.

LISTING 16.1 ControlCircle1.java

```
1  import javax.swing.*;
2  import java.awt.*;
3
4  public class ControlCircle1 extends JFrame {
5    private JButton jbtEnlarge = new JButton("Enlarge");                    buttons
6    private JButton jbtShrink = new JButton("Shrink");
7    private CirclePanel canvas = new CirclePanel();                         circle canvas
8
9    public ControlCircle1() {
10     JPanel panel = new JPanel(); // Use the panel to group buttons
11     panel.add(jbtEnlarge);
12     panel.add(jbtShrink);
13
14     this.add(canvas, BorderLayout.CENTER); // Add canvas to center
15     this.add(panel, BorderLayout.SOUTH); // Add buttons to the frame
16   }
17
18   /** Main method */
19   public static void main(String[] args) {
20     JFrame frame = new ControlCircle1();
21     frame.setTitle("ControlCircle1");
22     frame.setLocationRelativeTo(null); // Center the frame
23     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24     frame.setSize(200, 200);
25     frame.setVisible(true);
26   }
27 }
28
29 class CirclePanel extends JPanel {                                         CirclePanel class
30   private int radius = 5; // Default circle radius
31
32   /** Repaint the circle */
33   protected void paintComponent(Graphics g) {                              paint the circle
34     super.paintComponent(g);
35     g.drawOval(getWidth() / 2 - radius, getHeight() / 2 - radius,
36       2 * radius, 2 * radius);
37   }
38 }
```

How do you use the buttons to enlarge or shrink the circle? When the *Enlarge* button is clicked, you want the circle to be repainted with a larger radius. How can you accomplish this? You can expand the program in Listing 16.1 into Listing 16.2 with the following features: second version

1. Define a listener class named `EnlargeListener` that implements `ActionListener` (lines 31–35).
2. Create a listener and register it with `jbtEnlarge` (line 18).

3. Add a method named `enlarge()` in `CirclePanel` to increase the radius, then repaint the panel (lines 41–44).

4. Implement the `actionPerformed` method in `EnlargeListener` to invoke `canvas.enlarge()` (line 33).

5. To make the reference variable `canvas` accessible from the `actionPerformed` method, define `EnlargeListener` as an inner class of the `ControlCircle2` class (lines 31–35). Inner classes are defined inside another class. We will introduce inner classes in the next section.

6. To avoid compile errors, the `CirclePanel` class (lines 37–52) now is also defined as an inner class in `ControlCircle2`,since an old `CirclePanel` class is already defined in Listing 16.1.

LISTING 16.2 ControlCircle2.java

Video Note
Listener and its registration

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  public class ControlCircle2 extends JFrame {
6    private JButton jbtEnlarge = new JButton("Enlarge");
7    private JButton jbtShrink = new JButton("Shrink");
8    private CirclePanel canvas = new CirclePanel();
9
10   public ControlCircle2() {
11     JPanel panel = new JPanel(); // Use the panel to group buttons
12     panel.add(jbtEnlarge);
13     panel.add(jbtShrink);
14
15     this.add(canvas, BorderLayout.CENTER); // Add canvas to center
16     this.add(panel, BorderLayout.SOUTH); // Add buttons to the frame
17
18     jbtEnlarge.addActionListener(new EnlargeListener());
19   }
20
21   /** Main method */
22   public static void main(String[] args) {
23     JFrame frame = new ControlCircle2();
24     frame.setTitle("ControlCircle2");
25     frame.setLocationRelativeTo(null); // Center the frame
26     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27     frame.setSize(200, 200);
28     frame.setVisible(true);
29   }
30
31   class EnlargeListener implements ActionListener { // Inner class
32     public void actionPerformed(ActionEvent e) {
33       canvas.enlarge();
34     }
35   }
36
37   class CirclePanel extends JPanel { // Inner class
38     private int radius = 5; // Default circle radius
39
40     /** Enlarge the circle */
41     public void enlarge() {
42       radius++;
```

create/register listener

listener class

CirclePanel class

enlarge method

```
43        repaint();
44      }
45
46      /** Repaint the circle */
47      protected void paintComponent(Graphics g) {
48        super.paintComponent(g);
49        g.drawOval(getWidth() / 2 - radius, getHeight() / 2 - radius,
50          2 * radius, 2 * radius);
51      }
52    }
53 }
```

Similarly you can add the code for the *Shrink* button to display a smaller circle when the *Shrink* button is clicked.

the Shrink button

16.4 Inner Classes

An *inner class*, or *nested class,* is a class defined within the scope of another class. The code in Figure 16.7(a) defines two separate classes, `Test` and `A`. The code in Figure 16.7(b) defines `A` as an inner class in `Test`.

```
public class Test {
  ...
}

public class A {
  ...
}
```

(a)

```
public class Test {
  ...

  // Inner class
  public class A {
  ...
  }
}
```

(b)

```
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** A method in the outer class */
  public void m() {
    // Do something
  }

  // An inner class
  class InnerClass {
    /** A method in the inner class */
    public void mi() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

(c)

FIGURE 16.7 Inner classes combine dependent classes into the primary class.

The class `InnerClass` defined inside `OuterClass` in Figure 16.7(c) is another example of an inner class. An inner class may be used just like a regular class. Normally, you define a class an inner class if it is used only by its outer class. An inner class has the following features:

- An inner class is compiled into a class named *`OuterClassName$InnerClassName.class`*. For example, the inner class `A` in `Test` is compiled into *`Test$A.class`* in Figure 16.7(b).
- An inner class can reference the data and methods defined in the outer class in which it nests, so you need not pass the reference of an object of the outer class to the constructor of the inner class. For this reason, inner classes can make programs simple and concise.
- An inner class can be defined with a visibility modifier subject to the same visibility rules applied to a member of the class.

- An inner class can be defined `static`. A `static` inner class can be accessed using the outer class name. A `static` inner class cannot access nonstatic members of the outer class.
- Objects of an inner class are often created in the outer class. But you can also create an object of an inner class from another class. If the inner class is nonstatic, you must first create an instance of the outer class, then use the following syntax to create an object for the inner class:

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

- If the inner class is static, use the following syntax to create an object for it:

```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```

A simple use of inner classes is to combine dependent classes into a primary class. This reduces the number of source files. It also makes class files easy to organize, since they are all named with the primary class as the prefix. For example, rather than creating two source files, **Test.java** and **A.java**, in Figure 16.7(a), you can combine class `A` into class `Test` and create just one source file **Test.java** in Figure 16.7(b). The resulting class files are **Test.class** and **Test$A.class**.

Another practical use of inner classes is to avoid class-naming conflict. Two versions of `CirclePanel` are defined in Listings 16.1 and 16.2. You can define them as inner classes to avoid conflict.

16.5 Anonymous Class Listeners

A listener class is designed specifically to create a listener object for a GUI component (e.g., a button). The listener class will not be shared by other applications and therefore is appropriate to be defined inside the frame class as an inner class.

anonymous inner class

Inner-class listeners can be shortened using anonymous inner classes. An *anonymous inner class* is an inner class without a name. It combines defining an inner class and creating an instance of the class in one step. The inner class in Listing 16.2 can be replaced by an anonymous inner class as shown below.

```
public ControlCircle2() {
  // Omitted

  jbtEnlarge.addActionListener(
    new EnlargeListener());
}

class EnlargeListener
    implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    canvas.enlarge();
  }
}
```

(a) Inner class EnlargeListener

```
public ControlCircle2() {
  // Omitted

  jbtEnlarge.addActionListener(
    new class EnlargeListener
          implements ActionListener() {
      public void
          actionPerformed(ActionEvent e) {
        canvas.enlarge();
      }
    });
}
```

(b) Anonymous inner class

The syntax for an anonymous inner class is as follows:

```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface

  // Other methods if necessary
}
```

Since an anonymous inner class is a special kind of inner class, it is treated like an inner class with the following features:

- An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit `extends` or `implements` clause.
- An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- An anonymous inner class always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is `Object()`.
- An anonymous inner class is compiled into a class named `OuterClassName$n.class`. For example, if the outer class `Test` has two anonymous inner classes, they are compiled into `Test$1.class` and `Test$2.class`.

Listing 16.3 gives an example that handles the events from four buttons, as shown in Figure 16.8.

FIGURE 16.8 The program handles the events from four buttons.

LISTING 16.3 AnonymousListenerDemo.java

Video Note
Anonymous listener

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  public class AnonymousListenerDemo extends JFrame {
5    public AnonymousListenerDemo() {
6      // Create four buttons
7      JButton jbtNew = new JButton("New");
8      JButton jbtOpen = new JButton("Open");
9      JButton jbtSave = new JButton("Save");
10     JButton jbtPrint = new JButton("Print");
11
12     // Create a panel to hold buttons
13     JPanel panel = new JPanel();
14     panel.add(jbtNew);
15     panel.add(jbtOpen);
16     panel.add(jbtSave);
17     panel.add(jbtPrint);
18
19     add(panel);
20
21     // Create and register anonymous inner-class listener
22     jbtNew.addActionListener(
23       new ActionListener() {
24         public void actionPerformed(ActionEvent e) {
25           System.out.println("Process New");
26         }
27       }
28     );
```

anonymous listener
handle event

```
29
30      jbtOpen.addActionListener(
31        new ActionListener() {
32          public void actionPerformed(ActionEvent e) {
33            System.out.println("Process Open");
34          }
35        }
36      );
37
38      jbtSave.addActionListener(
39        new ActionListener() {
40          public void actionPerformed(ActionEvent e) {
41            System.out.println("Process Save");
42          }
43        }
44      );
45
46      jbtPrint.addActionListener(
47        new ActionListener() {
48          public void actionPerformed(ActionEvent e) {
49            System.out.println("Process Print");
50          }
51        }
52      );
53    }
54
55    /** Main method */
56    public static void main(String[] args) {
57      JFrame frame = new AnonymousListenerDemo();
58      frame.setTitle("AnonymousListenerDemo");
59      frame.setLocationRelativeTo(null); // Center the frame
60      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
61      frame.pack();
62      frame.setVisible(true);
63    }
64  }
```

The program creates four listeners using anonymous inner classes (lines 22–52). Without using anonymous inner classes, you would have to create four separate classes. An anonymous listener works the same way as an inner class listener. The program is condensed using an anonymous inner class.

Anonymous inner classes are compiled into `OuterClassName$#.class`, where `#` starts at `1` and is incremented for each anonymous class the compiler encounters. In this example, the anonymous inner class is compiled into `AnonymousListenerDemo$1.class`, `AnonymousListenerDemo$2.class`, `AnonymousListenerDemo$3.class`,and `AnonymousListenerDemo$4.class`.

pack()

Instead of using the `setSize` method to set the size for the frame, the program uses the `pack()` method (line 61), which automatically sizes the frame according to the size of the components placed in it.

16.6 Alternative Ways of Defining Listener Classes

There are many other ways to define the listener classes. For example, you may rewrite Listing 16.3 by creating just one listener, register the listener with the buttons, and let the listener detect the event source—i.e., which button fires the event—as shown in Listing 16.4.

LISTING 16.4 DetectSourceDemo.java

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  public class DetectSourceDemo extends JFrame {
5    // Create four buttons
6    private JButton jbtNew = new JButton("New");
7    private JButton jbtOpen = new JButton("Open");
8    private JButton jbtSave = new JButton("Save");
9    private JButton jbtPrint = new JButton("Print");
10
11   public DetectSourceDemo() {
12     // Create a panel to hold buttons
13     JPanel panel = new JPanel();
14     panel.add(jbtNew);
15     panel.add(jbtOpen);
16     panel.add(jbtSave);
17     panel.add(jbtPrint);
18
19     add(panel);
20
21     // Create a listener
22     ButtonListener listener = new ButtonListener();                    create listener
23
24     // Register listener with buttons
25     jbtNew.addActionListener(listener);                                register listener
26     jbtOpen.addActionListener(listener);
27     jbtSave.addActionListener(listener);
28     jbtPrint.addActionListener(listener);
29   }
30
31   class ButtonListener implements ActionListener {                     listener class
32     public void actionPerformed(ActionEvent e) {                       handle event
33       if (e.getSource() == jbtNew)
34         System.out.println("Process New");
35       else if (e.getSource() == jbtOpen)
36         System.out.println("Process Open");
37       else if (e.getSource() == jbtSave)
38         System.out.println("Process Save");
39       else if (e.getSource() == jbtPrint)
40         System.out.println("Process Print");
41     }
42   }
43
44   /** Main method */
45   public static void main(String[] args) {
46     JFrame frame = new DetectSourceDemo();
47     frame.setTitle("DetectSourceDemo");
48     frame.setLocationRelativeTo(null); // Center the frame
49     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
50     frame.pack();
51     frame.setVisible(true);
52   }
53 }
```

This program defines just one inner listener class (lines 31–42), creates a listener from the class (line 22), and registers it to four buttons (lines 25–28). When a button is clicked, the button fires an `ActionEvent` and invokes the listener's `actionPerformed` method. The `actionPerformed` method checks the source of the event using the `getSource()` method for the event (lines 33, 35, 37, 39) and determines which button fired the event.

You may also rewrite Listing 16.3 by defining the custom frame class that implements `ActionListener`, as shown in Listing 16.5.

LISTING 16.5 FrameAsListenerDemo.java

implement **ActionListener**

register listeners

handle event

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  public class FrameAsListenerDemo extends JFrame
5      implements ActionListener {
6    // Create four buttons
7    private JButton jbtNew = new JButton("New");
8    private JButton jbtOpen = new JButton("Open");
9    private JButton jbtSave = new JButton("Save");
10   private JButton jbtPrint = new JButton("Print");
11
12   public FrameAsListenerDemo() {
13     // Create a panel to hold buttons
14     JPanel panel = new JPanel();
15     panel.add(jbtNew);
16     panel.add(jbtOpen);
17     panel.add(jbtSave);
18     panel.add(jbtPrint);
19
20     add(panel);
21
22     // Register listener with buttons
23     jbtNew.addActionListener(this);
24     jbtOpen.addActionListener(this);
25     jbtSave.addActionListener(this);
26     jbtPrint.addActionListener(this);
27   }
28
29   /** Implement actionPerformed */
30   public void actionPerformed(ActionEvent e) {
31     if (e.getSource() == jbtNew)
32       System.out.println("Process New");
33     else if (e.getSource() == jbtOpen)
34       System.out.println("Process Open");
35     else if (e.getSource() == jbtSave)
36       System.out.println("Process Save");
37     else if (e.getSource() == jbtPrint)
38       System.out.println("Process Print");
39   }
40
41   /** Main method */
42   public static void main(String[] args) {
43     JFrame frame = new FrameAsListenerDemo();
44     frame.setTitle("FrameAsListenerDemo");
45     frame.setLocationRelativeTo(null); // Center the frame
46     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
47     frame.pack();
48     frame.setVisible(true);
49   }
50 }
```

The frame class extends `JFrame` and implements `ActionListener` (line 5). So the class is a listener class for action events. The listener is registered to four buttons (lines 23–26). When a button is clicked, the button fires an `ActionEvent` and invokes the listener's `actionPerformed` method. The `actionPerformed` method checks the source of the event using the `getSource()` method for the event (lines 31, 33, 35, 37) and determines which button fired the event.

This design is not desirable because it places too many responsibilities into one class. It is better to design a listener class that is solely responsible for handling events. This design makes the code easy to read and easy to maintain.

You can define listener classes in many ways. Which way is preferred? Defining listener classes using inner class or anonymous inner class has become a standard for event-handling programming because it generally provides clear, clean, and concise code. So, we will consistently use it in this book.

16.7 Problem: Loan Calculator

Now you can write the program for the loan-calculator problem presented in the introduction of this chapter. Here are the major steps in the program:

1. Create the user interface, as shown in Figure 16.9.

   a. Create a panel of a `GridLayout` with `5` rows and `2` columns. Add labels and text fields into the panel. Set a title "Enter loan amount, interest rate, and years" for the panel.

   b. Create another panel with a `FlowLayout(FlowLayout.RIGHT)` and add a button into the panel.

   c. Add the first panel to the center of the frame and the second panel to the south side of the frame.

2. Process the event.

   Create and register the listener for processing the button-clicking action event. The handler obtains the user input on loan, interest rate, and number of years, computes the monthly and total payments, and displays the values in the text fields.

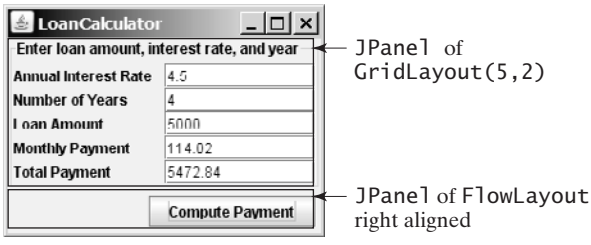


FIGURE 16.9 The program computes loan payments.

The complete program is given in Listing 16.6.

LISTING 16.6 LoanCalculator.java

```
1 import java.awt.*;
2 import java.awt.event.*;
3 import javax.swing.*;
4 import javax.swing.border.TitledBorder;
5
6 public class LoanCalculator extends JFrame {
7   // Create text fields for interest rate,
```

text fields

button

create UI

add to frame

register listener

get input

create loan

set result

```
 8   // year, loan amount, monthly payment, and total payment
 9   private JTextField jtfAnnualInterestRate = new JTextField();
10   private JTextField jtfNumberOfYears = new JTextField();
11   private JTextField jtfLoanAmount = new JTextField();
12   private JTextField jtfMonthlyPayment = new JTextField();
13   private JTextField jtfTotalPayment = new JTextField();
14
15   // Create a Compute Payment button
16   private JButton jbtComputeLoan = new JButton("Compute Payment");
17
18   public LoanCalculator() {
19     // Panel p1 to hold labels and text fields
20     JPanel p1 = new JPanel(new GridLayout(5, 2));
21     p1.add(new JLabel("Annual Interest Rate"));
22     p1.add(jtfAnnualInterestRate);
23     p1.add(new JLabel("Number of Years"));
24     p1.add(jtfNumberOfYears);
25     p1.add(new JLabel("Loan Amount"));
26     p1.add(jtfLoanAmount);
27     p1.add(new JLabel("Monthly Payment"));
28     p1.add(jtfMonthlyPayment);
29     p1.add(new JLabel("Total Payment"));
30     p1.add(jtfTotalPayment);
31     p1.setBorder(new
32       TitledBorder("Enter loan amount, interest rate, and year"));
33
34     // Panel p2 to hold the button
35     JPanel p2 = new JPanel(new FlowLayout(FlowLayout.RIGHT));
36     p2.add(jbtComputeLoan);
37
38     // Add the panels to the frame
39     add(p1, BorderLayout.CENTER);
40     add(p2, BorderLayout.SOUTH);
41
42     // Register listener
43     jbtComputeLoan.addActionListener(new ButtonListener());
44   }
45
46   /** Handle the Compute Payment button */
47   private class ButtonListener implements ActionListener {
48     public void actionPerformed(ActionEvent e) {
49       // Get values from text fields
50       double interest =
51         Double.parseDouble(jtfAnnualInterestRate.getText());
52       int year =
53         Integer.parseInt(jtfNumberOfYears.getText());
54       double loanAmount =
55         Double.parseDouble(jtfLoanAmount.getText());
56
57       // Create a loan object
58       Loan loan = new Loan(interest, year, loanAmount);
59
60       // Display monthly payment and total payment
61       jtfMonthlyPayment.setText(String.format("%.2f",
62         loan.getMonthlyPayment()));
63       jtfTotalPayment.setText(String.format("%.2f",
64         loan.getTotalPayment()));
65     }
66   }
67
```

```
68   public static void main(String[] args) {
69     LoanCalculator frame = new LoanCalculator();
70     frame.pack();
71     frame.setTitle("LoanCalculator");
72     frame.setLocationRelativeTo(null); // Center the frame
73     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
74     frame.setVisible(true);
75   }
76 }
```

The user interface is created in the constructor (lines 18–44). The button is the source of the event. A listener is created and registered with the button (line 43).

The listener class (lines 47–66) implements the `actionPerformed` method. When the button is clicked, the `actionPerformed` method is invoked to get the interest rate (line 51), number of years (line 53), and loan amount (line 55). Invoking `jtfAnnualInterestRate.getText()` returns the string text in the `jtfAnnualInterestRate` text field. The loan is used for computing the loan payments. This class was introduced in Listing 10.2, Loan.java. Invoking `loan.getMonthlyPayment()` returns the monthly payment for the loan. The `String.format` method uses the `printf` like syntax to format a number into a desirable format. Invoking the `setText` method on a text field sets a string value in the text field (line 61).

16.8 Window Events

The preceding sections used action events. Other events can be processed similarly. This section gives an example of handling `WindowEvent`. Any subclass of the `Window` class can fire the following window events: window opened, closing, closed, activated, deactivated, iconified, and deiconified. The program in Listing 16.7 creates a frame, listens to the window events, and displays a message to indicate the occurring event. Figure 16.10 shows a sample run of the program.

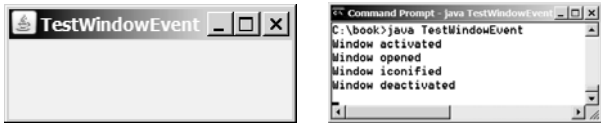


FIGURE 16.10 The window events are displayed on the console when you run the program from the command prompt.

LISTING 16.7 TestWindowEvent.java

```
1  import java.awt.event.*;
2  import javax.swing.JFrame;
3
4  public class TestWindowEvent extends JFrame {
5    public static void main(String[] args) {
6      TestWindowEvent frame = new TestWindowEvent();
7      frame.setSize(220, 80);
8      frame.setLocationRelativeTo(null); // Center the frame
9      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10     frame.setTitle("TestWindowEvent");
11     frame.setVisible(true);
12   }
13
14   public TestWindowEvent() {
15     addWindowListener(new WindowListener() {
16       /**
17        * Handler for window-deiconified event
```

```
18        * Invoked when a window is changed from a minimized
19        * to a normal state.
20        */
21       public void windowDeiconified(WindowEvent event) {
22         System.out.println("Window deiconified");
23       }
24
25       /**
26        * Handler for window-iconified event
27        * Invoked when a window is changed from a normal to a
28        * minimized state. For many platforms, a minimized window
29        * is displayed as the icon specified in the window's
30        * iconImage property.
31        */
32       public void windowIconified(WindowEvent event) {
33         System.out.println("Window iconified");
34       }
35
36       /**
37        * Handler for window-activated event
38        * Invoked when the window is set to be the user's
39        * active window, which means the window (or one of its
40        * subcomponents) will receive keyboard events.
41        */
42       public void windowActivated(WindowEvent event) {
43         System.out.println("Window activated");
44       }
45
46       /**
47        * Handler for window-deactivated event
48        * Invoked when a window is no longer the user's active
49        * window, which means that keyboard events will no longer
50        * be delivered to the window or its subcomponents.
51        */
52       public void windowDeactivated(WindowEvent event) {
53         System.out.println("Window deactivated");
54       }
55
56       /**
57        * Handler for window-opened event
58        * Invoked the first time a window is made visible.
59        */
60       public void windowOpened(WindowEvent event) {
61         System.out.println("Window opened");
62       }
63
64       /**
65        * Handler for window-closing event
66        * Invoked when the user attempts to close the window
67        * from the window's system menu.  If the program does not
68        * explicitly hide or dispose the window while processing
69        * this event, the window-closing operation will be cancelled.
70        */
71       public void windowClosing(WindowEvent event) {
72         System.out.println("Window closing");
73       }
74
75       /**
76        * Handler for window-closed event
77        * Invoked when a window has been closed as the result
```

implement handler

implement handler

implement handler

implement handler

implement handler

```
78         * of calling dispose on the window.
79         */
80        public void windowClosed(WindowEvent event) {
81          System.out.println("Window closed");
82        }
83      });
84    }
85 }
```

implement handler

The `WindowEvent` can be fired by the `Window` class or by any subclass of `Window`. Since `JFrame` is a subclass of `Window`, it can fire `WindowEvent`.

`TestWindowEvent` extends `JFrame` and implements `WindowListener`. The `WindowListener` interface defines several abstract methods (`windowActivated`, `windowClosed`, `windowClosing`, `windowDeactivated`, `windowDeiconified`, `windowIconified`, `windowOpened`) for handling window events when the window is activated, closed, closing, deactivated, deiconified, iconified, or opened.

When a window event, such as activation, occurs, the `windowActivated` method is invoked. Implement the `windowActivated` method with a concrete response if you want the event to be processed.

16.9 Listener Interface Adapters

Because the methods in the `WindowListener` interface are abstract, you must implement all of them even if your program does not care about some of the events. For convenience, Java provides support classes, called *convenience adapters*, which provide default implementations for all the methods in the listener interface. The default implementation is simply an empty body. Java provides convenience listener adapters for every AWT listener interface with multiple handlers. A convenience listener adapter is named *X*Adapter for *X*Listener. For example, `WindowAdapter` is a convenience listener adapter for `WindowListener`. Table 16.3 lists the convenience adapters.

convenience adapter

TABLE 16.3 Convenience Adapters

Adapter	*Interface*
`WindowAdapter`	`WindowListener`
`MouseAdapter`	`MouseListener`
`MouseMotionAdapter`	`MouseMotionListener`
`KeyAdapter`	`KeyListener`
`ContainerAdapter`	`ContainerListener`
`ComponentAdapter`	`ComponentListener`
`FocusAdapter`	`FocusListener`

Using `WindowAdapter`, the preceding example can be simplified as shown in Listing 16.8, if you are interested only in the window-activated event. The `WindowAdapter` class is used to create an anonymous listener instead of `WindowListener` (line 15). The `windowActivated` handler is implemented in line 16.

LISTING 16.8 AdapterDemo.java

```
1 import java.awt.event.*;
2 import javax.swing.JFrame;
3
4 public class AdapterDemo extends JFrame {
```

```
 5   public static void main(String[] args) {
 6     AdapterDemo frame = new AdapterDemo();
 7     frame.setSize(220, 80);
 8     frame.setLocationRelativeTo(null); // Center the frame
 9     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10     frame.setTitle("AdapterDemo");
11     frame.setVisible(true);
12   }
13
14   public AdapterDemo() {
15     addWindowListener(new WindowAdapter() {
16       public void windowActivated(WindowEvent event) {
17         System.out.println("Window activated");
18       }
19     });
20   }
21 }
```

register listener
implement handler

16.10 Mouse Events

A mouse event is fired whenever a mouse is pressed, released, clicked, moved, or dragged on a component. The `MouseEvent` object captures the event, such as the number of clicks associated with it or the location (`x`- and `y`-coordinates) of the mouse, as shown in Figure 16.11.

Since the `MouseEvent` class inherits `InputEvent`, you can use the methods defined in the `InputEvent` class on a `MouseEvent` object.

`Point` class

The `java.awt.Point` class represents a point on a component. The class contains two public variables, `x` and `y`, for coordinates. To create a `Point`, use the following constructor:

```
Point(int x, int y)
```

This constructs a `Point` object with the specified `x`- and `y`-coordinates. Normally, the data fields in a class should be private, but this class has two public data fields.

Java provides two listener interfaces, `MouseListener` and `MouseMotionListener`, to handle mouse events, as shown in Figure 16.12. Implement the `MouseListener` interface to

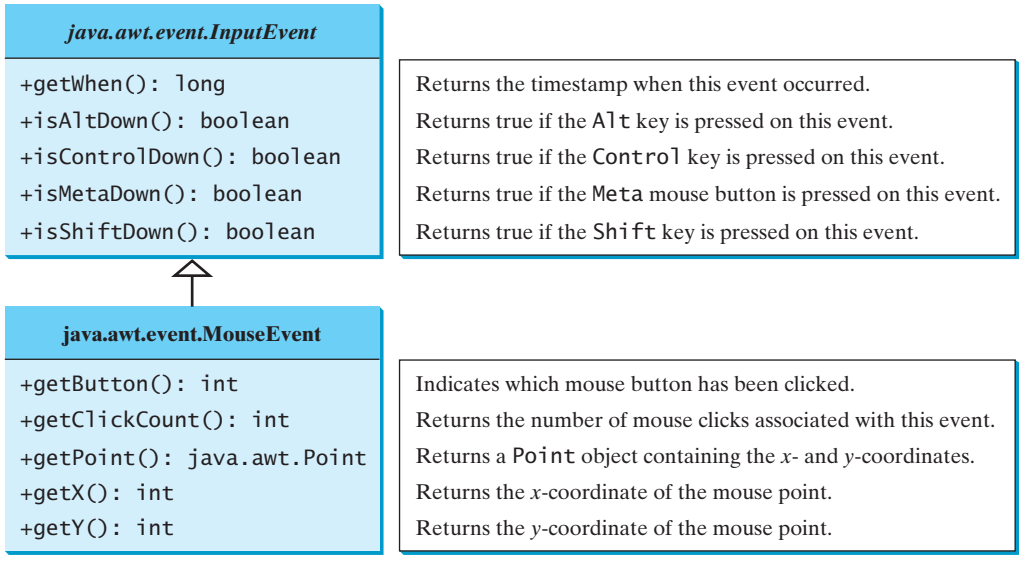


FIGURE 16.11 The `MouseEvent` class encapsulates information for mouse events.

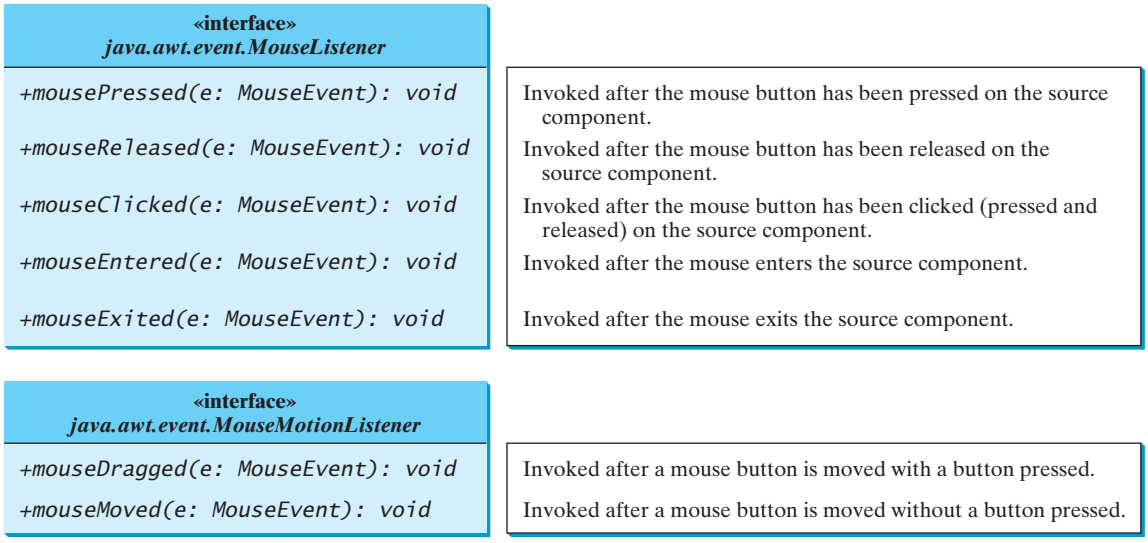


FIGURE 16.12 The `MouseListener` interface handles mouse pressed, released, clicked, entered, and exited events. The `MouseMotionListener` interface handles mouse dragged and moved events.

listen for such actions as pressing, releasing, entering, exiting, or clicking the mouse, and implement the `MouseMotionListener` interface to listen for such actions as dragging or moving the mouse.

16.10.1 Example: Moving a Message on a Panel Using a Mouse

This example writes a program that displays a message in a panel, as shown in Listing 16.9. You can use the mouse to move the message. The message moves as the mouse drags and is always displayed at the mouse point. A sample run of the program is shown in Figure 16.13.

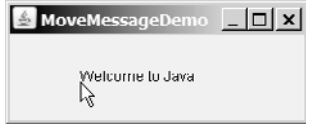


FIGURE 16.13 You can move the message by dragging the mouse.

LISTING 16.9 MoveMessageDemo.java

Video Note
Move message using the mouse

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class MoveMessageDemo extends JFrame {
6    public MoveMessageDemo() {
7      // Create a MovableMessagePanel instance for moving a message
8      MovableMessagePanel p = new MovableMessagePanel
9        ("Welcome to Java");
10
11     // Place the message panel in the frame
12     setLayout(new BorderLayout());
13     add(p);
14   }
15
```

create a panel

```
16   /** Main method */
17   public static void main(String[] args) {
18     MoveMessageDemo frame = new MoveMessageDemo();
19     frame.setTitle("MoveMessageDemo");
20     frame.setSize(200, 100);
21     frame.setLocationRelativeTo(null); // Center the frame
22     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23     frame.setVisible(true);
24   }
25
26   // Inner class: MovableMessagePanel draws a message
27   static class MovableMessagePanel extends JPanel {
28     private String message = "Welcome to Java";
29     private int x = 20;
30     private int y = 20;
31
32     /** Construct a panel to draw string s */
33     public MovableMessagePanel(String s) {
34       message = s;
35       addMouseMotionListener(new MouseMotionAdapter() {
36         /** Handle mouse-dragged event */
37         public void mouseDragged(MouseEvent e) {
38           // Get the new location and repaint the screen
39           x = e.getX();
40           y = e.getY();
41           repaint();
42         }
43       });
44     }
45
46     /** Paint the component */
47     protected void paintComponent(Graphics g) {
48       super.paintComponent(g);
49       g.drawString(message, x, y);
50     }
51   }
52 }
```

inner class

set a new message

anonymous listener

override handler

new location

repaint

paint message

The `MovableMessagePanel` class extends `JPanel` to draw a message (line 27). Additionally, it handles redisplaying the message when the mouse is dragged. This class is defined as an inner class inside the main class because it is used only in this class. Furthermore, the inner class is defined static because it does not reference any instance members of the main class.

The `MouseMotionListener` interface contains two handlers, `mouseMoved` and `mouseDragged`, for handling mouse-motion events. When you move the mouse with the button pressed, the `mouseDragged` method is invoked to repaint the viewing area and display the message at the mouse point. When you move the mouse without pressing the button, the `mouseMoved` method is invoked.

Because the listener is interested only in the mouse-dragged event, the anonymous inner-class listener extends `MouseMotionAdapter` to override the `mouseDragged` method. If the inner class implemented the `MouseMotionListener` interface, you would have to implement all of the handlers, even if your listener did not care about some of the events.

The `mouseDragged` method is invoked when you move the mouse with a button pressed. This method obtains the mouse location using `getX` and `getY` methods (lines 39–40) in the `MouseEvent` class. This becomes the new location for the message. Invoking the `repaint()` method (line 41) causes `paintComponent` to be invoked (line 47), which displays the message in a new location.

16.11 Key Events

Key events enable the use of the keys to control and perform actions or get input from the keyboard. A key event is fired whenever a key is pressed, released, or typed on a component. The `KeyEvent` object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key, as shown in Figure 16.14. Java provides the `KeyListener` interface to handle key events, as shown in Figure 16.15.

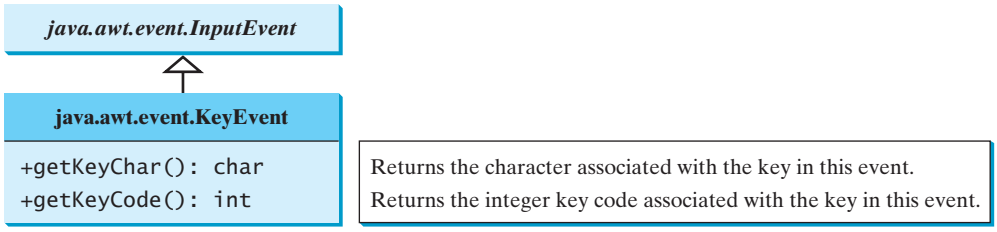


FIGURE 16.14 The `KeyEvent` class encapsulates information about key events.

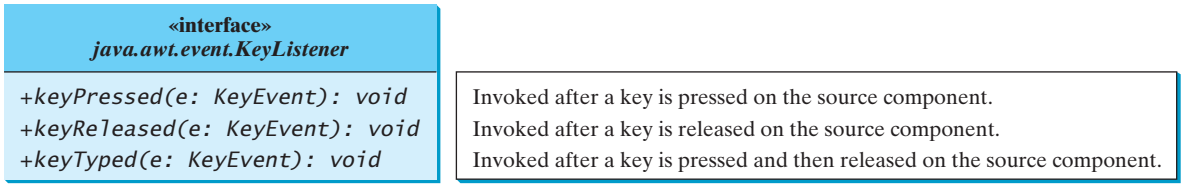


FIGURE 16.15 The `KeyListener` interface handles key pressed, released, and typed events.

The `keyPressed` handler is invoked when a key is pressed, the `keyReleased` handler is invoked when a key is released, and the `keyTyped` handler is invoked when a Unicode character is entered. If a key does not have a Unicode (e.g., function keys, modifier keys, action keys, and control keys), the `keyTyped` handler will be not be invoked.

Every key event has an associated key character or key code that is returned by the `getKeyChar()` or `getKeyCode()` method in `KeyEvent`. The key codes are constants defined in Table 16.4. For a key of the Unicode character, the key code is the same as the Unicode value.

TABLE 16.4 Key Constants

Constant	*Description*	*Constant*	*Description*
`VK_HOME`	The Home key	`VK_SHIFT`	The Shift key
`VK_END`	The End key	`VK_BACK_SPACE`	The Backspace key
`VK_PGUP`	The Page Up key	`VK_CAPS_LOCK`	The Caps Lock key
`VK_PGDN`	The Page Down key	`VK_NUM_LOCK`	The Num Lock key
`VK_UP`	The up-arrow key	`VK_ENTER`	The Enter key
`VK_DOWN`	The down-arrow key	`VK_UNDEFINED`	The keyCode unknown
`VK_LEFT`	The left-arrow key	`VK_F1` to `VK_F12`	The function keys from F1 to F12
`VK_RIGHT`	The right-arrow key		
`VK_ESCAPE`	The Esc key	`VK_0` to `VK_9`	The number keys from 0 to 9
`VK_TAB`	The Tab key	`VK_A` to `VK_Z`	The letter keys from A to Z
`VK_CONTROL`	The Control key		

For the key-pressed and key-released events, `getKeyCode()` returns the value as defined in the table. For the key-typed event, `getKeyCode()` returns `VK_UNDEFINED`, while `getKeyChar()` returns the character entered.

The program in Listing 16.10 displays a user-input character. The user can move the character up, down, left, and right, using the arrow keys `VK_UP`, `VK_DOWN`, `VK_LEFT`, and `VK_RIGHT`. Figure 16.16 contains a sample run of the program.

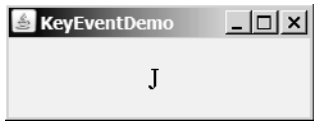


FIGURE 16.16 The program responds to key events by displaying a character and moving it up, down, left, or right.

LISTING 16.10 KeyEventDemo.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class KeyEventDemo extends JFrame {
6    private KeyboardPanel keyboardPanel = new KeyboardPanel();
7
8    /** Initialize UI */
9    public KeyEventDemo() {
10     // Add the keyboard panel to accept and display user input
11     add(keyboardPanel);
12
13     // Set focus
14     keyboardPanel.setFocusable(true);
15   }
16
17   /** Main method */
18   public static void main(String[] args) {
19     KeyEventDemo frame = new KeyEventDemo();
20     frame.setTitle("KeyEventDemo");
21     frame.setSize(300, 300);
22     frame.setLocationRelativeTo(null); // Center the frame
23     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24     frame.setVisible(true);
25   }
26
27   // Inner class: KeyboardPanel for receiving key input
28   static class KeyboardPanel extends JPanel {
29     private int x = 100;
30     private int y = 100;
31     private char keyChar = 'A'; // Default key
32
33     public KeyboardPanel() {
34       addKeyListener(new KeyAdapter() {
35         public void keyPressed(KeyEvent e) {
36           switch (e.getKeyCode()) {
37             case KeyEvent.VK_DOWN: y += 10; break;
38             case KeyEvent.VK_UP: y -= 10; break;
39             case KeyEvent.VK_LEFT: x -= 10; break;
40             case KeyEvent.VK_RIGHT: x += 10; break;
41             default: keyChar = e.getKeyChar();
42           }
```

create a panel

focusable

inner class

register listener
override handler

get the key pressed

```
43
44            repaint();                                                    repaint
45          }
46        });
47      }
48
49      /** Draw the character */
50      protected void paintComponent(Graphics g) {
51        super.paintComponent(g);
52
53        g.setFont(new Font("TimesRoman", Font.PLAIN, 24));
54        g.drawString(String.valueOf(keyChar), x, y);                      redraw character
55      }
56    }
57 }
```

The `KeyboardPanel` class extends `JPanel` to display a character (line 28). This class is defined as an inner class inside the main class, because it is used only in this class. Furthermore, the inner class is defined static, because it does not reference any instance members of the main class.

Because the program gets input from the keyboard, it listens for `KeyEvent` and extends `KeyAdapter` to handle key input (line 34).

When a key is pressed, the `keyPressed` handler is invoked. The program uses `e.getKeyCode()` to obtain the key code and `e.getKeyChar()` to get the character for the key. When a nonarrow key is pressed, the character is displayed (line 41). When an arrow key is pressed, the character moves in the direction indicated by the arrow key (lines 37–40).

Only a focused component can receive `KeyEvent`. To make a component focusable, set its `isFocusable` property to `true` (line 14). focusable

Every time the component is repainted, a new font is created for the `Graphics` object in line 53. This is not efficient. It is better to create the font once as a data field. efficient?

16.12 Animation Using the `Timer` Class

Not all source objects are GUI components. The `javax.swing.Timer` class is a source component that fires an `ActionEvent` at a predefined rate. Figure 16.17 lists some of the methods in the class.

javax.swing.Timer	
+Timer(delay: int, listener: ActionListener)	Creates a Timer object with a specified delay in milliseconds and an ActionListener.
+addActionListener(listener: ActionListener): void	Adds an ActionListener to the timer.
+start(): void	Starts this timer.
+stop(): void	Stops this timer.
+setDelay(delay: int): void	Sets a new delay value for this timer.

FIGURE 16.17 A `Timer` object fires an `ActionEvent` at a fixed rate.

A `Timer` object serves as the source of an `ActionEvent`. The listeners must be instances of `ActionListener` and registered with a `Timer` object. You create a `Timer` object using its sole constructor with a delay and a listener, where `delay` specifies the number of milliseconds between two action events. You can add additional listeners using the `addActionListener`

method and adjust the `delay` using the `setDelay` method. To start the timer, invoke the `start()` method. To stop the timer, invoke the `stop()` method.

The `Timer` class can be used to control animations. For example, you can use it to display a moving message, as shown in Figure 16.18, with the code in Listing 16.11.

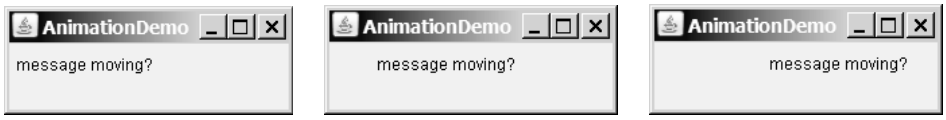


FIGURE 16.18 A message moves in the panel.

LISTING 16.11 AnimationDemo.java

```
1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  public class AnimationDemo extends JFrame {
6    public AnimationDemo() {
7      // Create a MovingMessagePanel for displaying a moving message
8      add(new MovingMessagePanel("message moving?"));
9    }
10
11   /** Main method */
12   public static void main(String[] args) {
13     AnimationDemo frame = new AnimationDemo();
14     frame.setTitle("AnimationDemo");
15     frame.setSize(280, 100);
16     frame.setLocationRelativeTo(null); // Center the frame
17     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18     frame.setVisible(true);
19   }
20
21   // Inner class: Displaying a moving message
22   static class MovingMessagePanel extends JPanel {
23     private String message = "Welcome to Java";
24     private int xCoordinate = 0;
25     private int yCoordinate = 20;
26
27     public MovingMessagePanel(String message) {
28       this.message = message;
29
30       // Create a timer
31       Timer timer = new Timer(1000, new TimerListener());
32       timer.start();
33     }
34
35     /** Paint message */
36     protected void paintComponent(Graphics g) {
37       super.paintComponent(g);
38
39       if (xCoordinate > getWidth()) {
40         xCoordinate = -20;
41       }
42       xCoordinate += 5;
43       g.drawString(message, xCoordinate, yCoordinate);
44     }
```

create panel

set message

create timer
start timer

reset x-coordinate

move message

```
45
46      class TimerListener implements ActionListener {                    listener class
47        /** Handle ActionEvent */
48        public void actionPerformed(ActionEvent e) {                     event handler
49          repaint();                                                     repaint
50        }
51      }
52    }
53  }
```

The `MovingMessagePanel` class extends `JPanel` to display a message (line 22). This class is defined as an inner class inside the main class, because it is used only in this class. Furthermore, the inner class is defined static, because it does not reference any instance members of the main class.

An inner class listener is defined in line 46 to listen for `ActionEvent`. Line 31 creates a `Timer` for the listener. The timer is started in line 32. The timer fires an `ActionEvent` every second, and the listener responds in line 49 to repaint the panel. When a panel is painted, its `x`-coordinate is increased (line 42), so the message is displayed to the right. When the `x`-coordinate exceeds the bound of the panel, it is reset to `-20` (line 40), so the message continues moving from left to right.

In §15.12, "Case Study: The `StillClock` Class," you drew a `StillClock` to show the current time. The clock does not tick after it is displayed. What can you do to make the clock display a new current time every second? The key to making the clock tick is to repaint it every second with a new current time. You can use a timer to control the repainting of the clock with the code in Listing 16.12.

LISTING 16.12 ClockAnimation.java

Video Note
Animate a clock

```
1  import java.awt.event.*;
2  import javax.swing.*;
3
4  public class ClockAnimation extends JFrame {
5    private StillClock clock = new StillClock();                           create a clock
6
7    public ClockAnimation() {
8      add(clock);
9
10     // Create a timer with delay 1000 ms
11     Timer timer = new Timer(1000, new TimerListener());                  create a timer
12     timer.start();                                                       start timer
13   }
14
15   private class TimerListener implements ActionListener {                listener class
16     /** Handle the action event */
17     public void actionPerformed(ActionEvent e) {                         implement handler
18       // Set new time and repaint the clock to display current time
19       clock.setCurrentTime();                                            set new time
20       clock.repaint();                                                   repaint
21     }
22   }
23
24   /** Main method */
25   public static void main(String[] args) {
26     JFrame frame = new ClockAnimation();
27     frame.setTitle("ClockAnimation");
28     frame.setSize(200, 200);
29     frame.setLocationRelativeTo(null); // Center the frame
30     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
31      frame.setVisible(true);
32    }
33 }
```

The program displays a running clock, as shown in Figure 16.19. `ClockAnimation` creates a `StillClock` (line 5). Line 11 creates a `Timer` for a `ClockAnimation`. The timer is started in line 12. The timer fires an `ActionEvent` every second, and the listener responds to set a new time (line 19) and repaint the clock (line 20). The `setCurrentTime()` method defined in `StillClock` sets the current time in the clock.

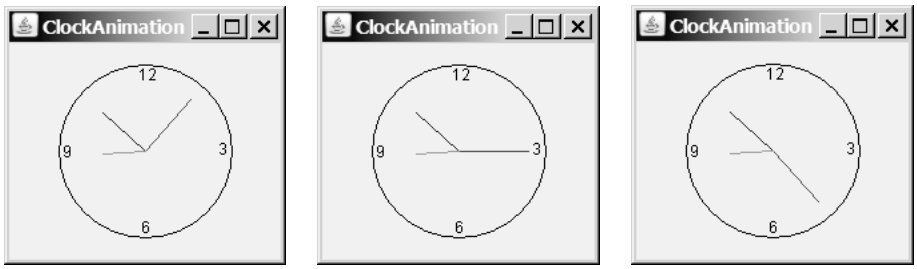


FIGURE 16.19 A live clock is displayed in the panel.

KEY TERMS

anonymous inner class 542
convenience listener adapter 551
event 534
event delegation 535
event handler 559
event listener 535
event-listener interface 536
event object 535
event registration 535
event source (source object) 535
event-driven programming 534
inner class 554

CHAPTER SUMMARY

1. The root class of the event classes is `java.util.EventObject`. The subclasses of `EventObject` deal with special types of events, such as action events, window events, component events, mouse events, and key events. You can identify the source object of an event using the `getSource()` instance method in the `EventObject` class. If a component can fire an event, any subclass of the component can fire the same type of event.

2. The listener object's class must implement the corresponding event-listener interface. Java provides a listener interface for every event class. The listener interface is usually named `XListener` for `XEvent`, with the exception of `MouseMotionListener`. For example, the corresponding listener interface for `ActionEvent` is `ActionListener`; each listener for `ActionEvent` should implement the `ActionListener` interface. The listener interface contains the method(s), known as the *handler(s)*, which process the events.

3. The listener object must be registered by the source object. Registration methods depend on the event type. For `ActionEvent`, the method is `addActionListener`. In general, the method is named `addXListener` for `XEvent`.

4. An *inner class*, or *nested class,* is defined within the scope of another class. An inner class can reference the data and methods defined in the outer class in which it nests, so you need not pass the reference of the outer class to the constructor of the inner class.

5. Convenience adapters are support classes that provide default implementations for all the methods in the listener interface. Java provides convenience listener adapters for every AWT listener interface with multiple handlers. A convenience listener adapter is named *X*Adapter for *X*Listener.

6. A source object may fire several types of events. For each event, the source object maintains a list of registered listeners and notifies them by invoking the *handler* on the listener object to process the event.

7. A mouse event is fired whenever a mouse is clicked, released, moved, or dragged on a component. The mouse-event object captures the event, such as the number of clicks associated with it or the location (`x`- and `y`-coordinates) of the mouse point.

8. Java provides two listener interfaces, `MouseListener` and `MouseMotionListener`, to handle mouse events, implement the `MouseListener` interface to listen for such actions as mouse pressed, released, clicked, entered, or exited, and implement the `MouseMotionListener` interface to listen for such actions as mouse dragged or moved.

9. A `KeyEvent` object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key.

10. The `keyPressed` handler is invoked when a key is pressed, the `keyReleased` handler is invoked when a key is released, and the `keyTyped` handler is invoked when a Unicode character key is entered. If a key does not have a Unicode (e.g., function keys, modifier keys, action keys, and control keys), the `keyTyped` handler will be not be invoked.

11. You can use the `Timer` class to control Java animations. A timer fires an `ActionEvent` at a fixed rate. The listener updates the painting to simulate an animation.

Review Questions

Sections 16.2–16.3

16.1 Can a button fire a `WindowEvent`? Can a button fire a `MouseEvent`? Can a button fire an `ActionEvent`?

16.2 Why must a listener be an instance of an appropriate listener interface? Explain how to register a listener object and how to implement a listener interface.

16.3 Can a source have multiple listeners? Can a listener listen on multiple sources? Can a source be a listener for itself?

16.4 How do you implement a method defined in the listener interface? Do you need to implement all the methods defined in the listener interface?

Sections 16.4–16.9

16.5 Can an inner class be used in a class other than the class in which it nests?

16.6 Can the modifiers `public`, `private`, and `static` be used on inner classes?

16.7 If class `A` is an inner class in class `B`, what is the .class file for `A`? If class B contains two anonymous inner classes, what are the .class file names for these two classes?

16.8 What is wrong in the following code?

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

public class Test extends JFrame {
  public Test() {
    JButton jbtOK = new JButton("OK");
    add(jbtOK);
  }

  private class Listener
      implements ActionListener {
    public void actionPerform
        (ActionEvent e) {
      System.out.println
        (jbtOK.getActionCommand());
    }
  }

  /** Main method omitted */
}
```

(a)

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

public class Test extends JFrame {
  public Test() {
    JButton jbtOK = new JButton("OK");
    add(jbtOK);
    jbtOK.addActionListener(
      new ActionListener() {
        public void actionPerformed
            (ActionEvent e) {
          System.out.println
            (jbtOK.getActionCommand());
        }
    }  // Something missing here
  }

  /** Main method omitted */
}
```

(b)

16.9 What is the difference between the `setSize(width, height)` method and the `pack()` method in `JFrame`?

Sections 16.10–16.11

16.10 What method do you use to get the source of an event? What method do you use to get the timestamp for an action event, a mouse event, or a key event? What method do you use to get the mouse-point position for a mouse event? What method do you use to get the key character for a key event?

16.11 What is the listener interface for mouse pressed, released, clicked, entered, and exited? What is the listener interface for mouse moved and dragged?

16.12 Does every key in the keyboard have a Unicode? Is a key code in the `KeyEvent` class equivalent to a Unicode?

16.13 Is the `keyPressed` handler invoked after a key is pressed? Is the `keyReleased` handler invoked after a key is released? Is the `keyTyped` handler invoked after *any* key is typed?

Section 16.12

16.14 How do you create a timer? How do you start a timer? How do you stop a timer?

16.15 Does the `Timer` class have a no-arg constructor? Can you add multiple listeners to a timer?

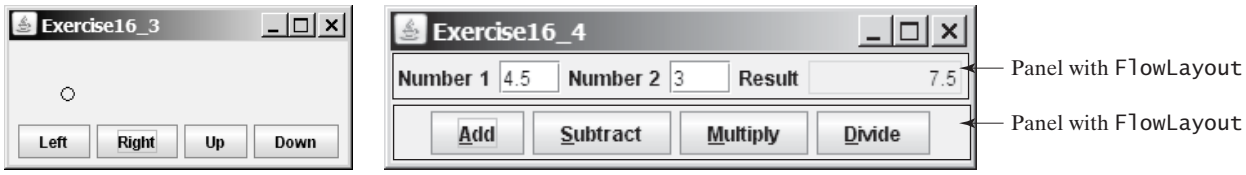
Programming Exercises

Sections 16.2–16.9

16.1 (*Finding which button has been clicked on the console*) Add the code to Exercise 12.1 that will display a message on the console indicating which button has been clicked.

16.2 (*Using `ComponentEvent`*) Any GUI component can fire a `ComponentEvent`. The `ComponentListener` defines the `componentMoved`, `componentResized`, `componentShown`, and `componentHidden` methods for processing component events. Write a test program to demonstrate `ComponentEvent`.

16.3* (*Moving the ball*) Write a program that moves the ball in a panel. You should define a panel class for displaying the ball and provide the methods for moving the button left, right, up, and down, as shown in Figure 16.20(a).



(a) (b)

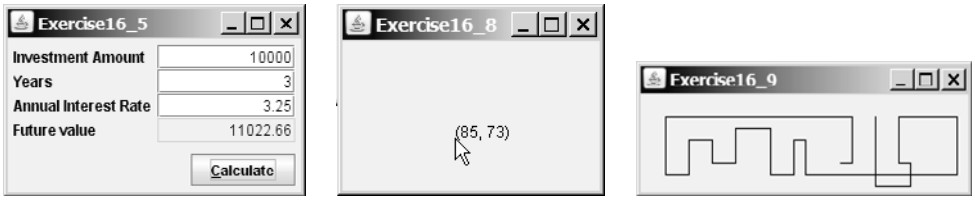
FIGURE 16.20 (a) Exercise 16.3 displays which button is clicked on a message panel. (b) The program performs addition, subtraction, multiplication, and division on double numbers.

16.4* (*Creating a simple calculator*) Write a program to perform add, subtract, multiply, and divide operations (see Figure 16.20(b)).

16.5* (*Creating an investment-value calculator*) Write a program that calculates the future value of an investment at a given interest rate for a specified number of years. The formula for the calculation is as follows:

$$\texttt{futureValue} = \texttt{investmentAmount} * (1 + \texttt{monthlyInterestRate})^{\texttt{years}*12}$$

Use text fields for interest rate, investment amount, and years. Display the future amount in a text field when the user clicks the *Calculate* button, as shown in Figure 16.21(a).



(a) (b) (c)

FIGURE 16.21 (a) The user enters the investment amount, years, and interest rate to compute future value. (b) Exercise 16.8 displays the mouse position. (c) Exercise 16.9 uses the arrow keys to draw the lines.

Section 16.10

16.6** (*Alternating two messages*) Write a program to rotate with a mouse click two messages displayed on a panel, "Java is fun" and "Java is powerful".

16.7* (*Setting background color using a mouse*) Write a program that displays the background color of a panel as black when the mouse is pressed and as white when the mouse is released.

16.8* (*Displaying the mouse position*) Write two programs, such that one displays the mouse position when the mouse is clicked (see Figure 16.21(b)) and the other displays the mouse position when the mouse is pressed and ceases to display it when the mouse is released.

Section 16.11

16.9* (*Drawing lines using the arrow keys*) Write a program that draws line segments using the arrow keys. The line starts from the center of the frame and draws

toward east, north, west, or south when the right-arrow key, up-arrow key, left-arrow key, or down-arrow key is clicked, as shown in Figure 16.21(c).

16.10** (*Entering and displaying a string*) Write a program that receives a string from the keyboard and displays it on a panel. The *Enter* key signals the end of a string. Whenever a new string is entered, it is displayed on the panel.

16.11* (*Displaying a character*) Write a program to get a character input from the keyboard and display the character where the mouse points.

Section 16.12

16.12** (*Displaying a running fan*) Listing 15.4, DrawArcs.java, displays a motionless fan. Write a program that displays a running fan.

16.13** (*Slide show*) Twenty-five slides are stored as image files (slide0.jpg, slide1.jpg, . . . , slide24.jpg) in the image directory downloadable along with the source code in the book. The size of each image is 800 × 600. Write a Java application that automatically displays the slides repeatedly. Each slide is shown for a second. The slides are displayed in order. When the last slide finishes, the first slide is redisplayed, and so on.

(*Hint*: Place a label in the frame and set a slide as an image icon in the label.)

16.14** (*Raising flag*) Write a Java program that animates raising a flag, as shown in Figure 16.1. (See §15.11, "Displaying Images," on how to display images.)

Video Note
Animate a rising flag

16.15** (*Racing car*) Write a Java program that simulates car racing, as shown in Figure 16.22(a). The car moves from left to right. When it hits the right end, it restarts from the left and continues the same process. You can use a timer to control animation. Redraw the car with a new base coordinates (x, y), as shown in Figure 16.22(b).

FIGURE 16.22 (a) Exercise 16.15 displays a moving car. (b) You can redraw a car with a new base point.

16.16* (*Displaying a flashing label*) Write a program that displays a flashing label.

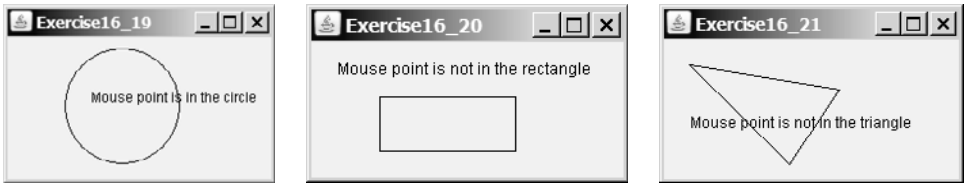
(*Hint*: To make the label flash, you need to repaint the panel alternately with the label and without it (blank screen) at a fixed rate. Use a `boolean` variable to control the alternation.)

16.17* (*Controlling a moving label*) Modify Listing 16.11, AnimationDemo.java, to control a moving label using the mouse. The label freezes when the mouse is pressed, and moves again when the button is released.

Comprehensive

16.18* (*Moving a circle using keys*) Write a program that moves a circle up, down, left, or right using the arrow keys.

16.19** (*Geometry: inside a circle?*) Write a program that draws a fixed circle centered at (100, 60) with radius 50. Whenever a mouse is moved, display the message indicating whether the mouse point is inside the circle, as shown in Figure 16.23(a).



(a) (b) (c)

FIGURE 16.23 Detect whether a point is inside a circle, a rectangle, or a triangle.

16.20** (*Geometry: inside a rectangle?*) Write a program that draws a fixed rectangle centered at (100, 60) with width 100 and height 40. Whenever a mouse is moved, display the message indicating whether the mouse point is inside the rectangle, as shown in Figure 16.23(b). To detect whether a point is inside a rectangle, use the `MyRectangle2D` class defined in Exercise 10.12.

16.21** (*Geometry: inside a triangle?*) Write a program that draws a fixed triangle with three vertices at (20, 20), (100, 100), and (140, 40). Whenever a mouse is moved, display the message indicating whether the mouse point is inside the triangle, as shown in Figure 16.23(c). To detect whether a point is inside a triangle, use the `Triangle2D` class defined in Exercise 10.13.

16.22*** (*Game: bean-machine animation*) Write a program that animates a bean machine introduced in Exercise 15.24. The animation terminates after ten balls are dropped, as shown in Figure 16.24.

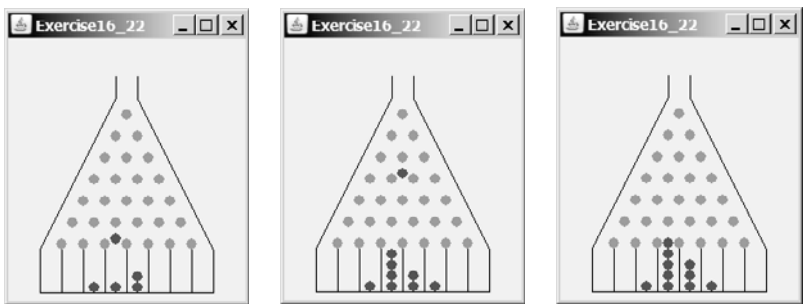


FIGURE 16.24 The balls are dropped to the bean machine.

16.23*** (*Geometry: closest pair of points*) Write a program that lets the user click on the panel to dynamically create points. Initially, the panel is empty. When a panel has two or more points, highlight the pair of closest points. Whenever a new point is created, a new pair of closest points is highlighted. Display the points using small circles and highlight the points using filled circles, as shown in Figure 16.25(a)–(c).

(*Hint*: store the points in an `ArrayList`.)

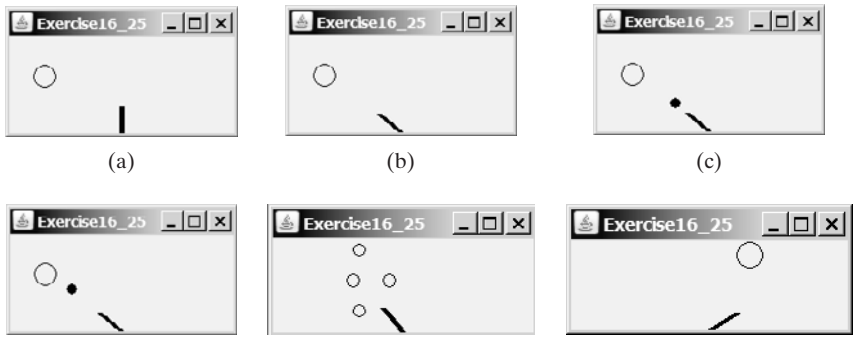
16.24* (*Controlling a clock*) Modify Listing 16.12 ClockAnimation.java to add two methods `start()` and `stop()` to start and stop the clock. Write a program

(a) (b) (c) (d)

FIGURE 16.25 Exercise 16.23 allows the user to create new points with a mouse click and highlights the pair of the closest points. Exercise 16.24 allows the user to start and stop a clock.

that lets the user control the clock with the *Start* and *Stop* buttons, as shown in Figure 16.25(d).

16.25* (*Game: hitting balloons*) Write a program that displays a balloon in a random position in a panel (Figure 16.26(a)). Use the left- and right-arrow keys to point the gun left or right to aim at the balloon (Figure 16.26(b)). Press the up-arrow key to fire a small ball from the gun (Figure 16.26(c)). Once the ball hits the balloon, the debris is displayed (Figure 16.26(e)) and a new balloon is displayed in a random location (Figure 16.26(f)). If the ball misses the balloon, the ball disappears once it hits the boundary of the panel. You can then press the up-arrow key to fire another ball. Whenever you press the left- or the right-arrow key, the gun turns 5 degrees left or right. (Instructors may modify the game as follows: 1. display the number of the balloons destroyed; 2. display a countdown timer (e.g., 60 seconds) and terminate the game once the time expires; 3. allow the balloon to rise dynamically.)



(a) (b) (c) (d) (e) (f)

FIGURE 16.26 (a) A balloon is displayed in a random location. (b) Press the left-/right-arrow keys to aim the balloon. (c) Press the up-arrow key to fire a ball. (d) The ball moves straight toward the balloon. (e) The ball hits the balloon. (f) A new balloon is displayed in a random position.

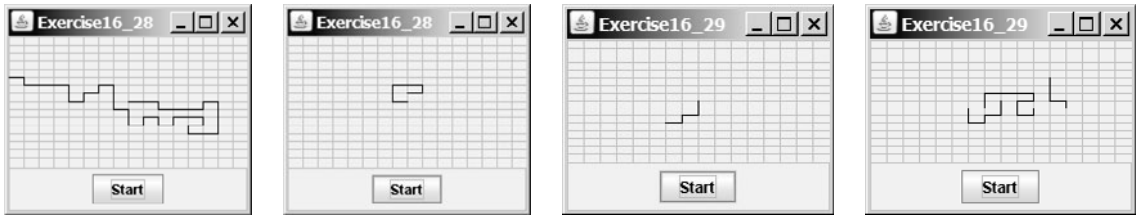
16.26** (*Moving a circle using mouse*) Write a program that displays a circle with radius 10 pixels. You can point the mouse inside the circle and drag (i.e., move with mouse pressed) the circle wherever the mouse goes, as shown in Figure 16.27(a)–(b).

16.27*** (*Game: eye-hand coordination*) Write a program that displays a circle of radius 10 pixels filled with a random color at a random location on a panel, as shown in Figure 16.27(c). When you click the circle, it is gone and a new random-color circle is displayed at another random location. After twenty circles are clicked, display the time spent in the panel, as shown in Figure 16.27(d).

(a) (b) (c) (d)

FIGURE 16.27 (a)–(b) You can point, drag, and move the circle. (c) When you click a circle, a new circle is displayed at a random location. (d) After 20 circles are clicked, the time spent in the panel is displayed.

16.28* (*Simulation: self-avoiding random walk*) A self-avoiding walk in a lattice is a path from one point to another which does not visit the same point twice. Self-avoiding walks have applications in physics, chemistry, and mathematics. They can be used to model chainlike entities such as solvents and polymers. Write a program that displays a random path that starts from the center and ends at a point on the boundary, as shown in Figure 16.28(a), or ends at a dead-end point (i.e., surrounded by four points that are already visited), as shown in Figure 16.28(b). Assume the size of the lattice is **16** by **16**.



(a) (b) (c) (d)

FIGURE 16.28 (a) A path ends at a boundary point. (b) A path ends at dead-end point. (c)–(d) Animation shows the progress of a path step by step.

16.29* (*Animation: self-avoiding random walk*) Revise the preceding exercise to display the walk step by step in an animation, as shown in Figure 16.28(c)–(d).

16.30** (*Simulation: self-avoiding random walk*) Write a simulation program to show that the chance of getting dead-end paths increases as the grid size increases. Your program simulates lattices with size from 10 to 80. For each lattice size, simulate a self-avoiding random walk 10000 times and display the probability of the dead-end paths, as shown in the following sample output:

```
For a lattice of size 10, the probability of dead-end paths is 10.6%
For a lattice of size 11, the probability of dead-end paths is 14.0%
...
For a lattice of size 80, the probability of dead-end paths is 99.5%
```

16.31* (*Geometry: displaying an n-sided regular polygon*) Exercise 15.25 created the `RegularPolygonPanel` for displaying an n-sided regular polygon. Write a program that displays a regular polygon and uses two buttons named *+1* and *−1* to increase or decrease the size of the polygon, as shown in Figure 16.29(a)–(b).

16.32** (*Geometry: adding and removing points*) Write a program that lets the user click on the panel to dynamically create and remove points. When the user right-click the mouse, a point is created and displayed at the mouse point, and

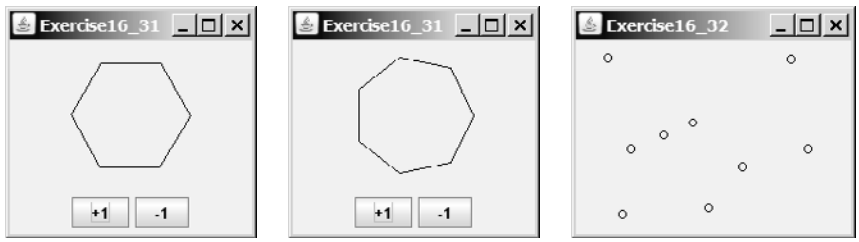


FIGURE 16.29 Clicking the *+1* or *−1* button increases or decreases the number of sides of a regular polygon in Exercise 16.31. Exercise 16.32 allows the user to create/remove points dynamically.

the user can remove a point by pointing to it and left-clicking the mouse, as shown in Figure 16.29(c).

16.33** (*Geometry: palindrome*) Write a program that animates a palindrome swing, as shown in Figure 16.30. Press the up-arrow key to increase the speed and the down-arrow key to decrease it. Press the *S* key to stop animation and the *R* key to resume.

FIGURE 16.30 Exercise 16.33 animates a palindrome swing.

16.34** (*Game: hangman*) Write a program that animates a hangman game swing, as shown in Figure 16.31. Press the up-arrow key to increase the speed and the down-arrow key to decrease it. Press the *S* key to stop animation and the *R* key to resume.

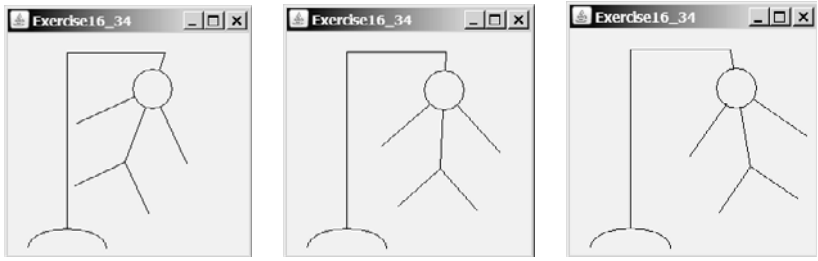
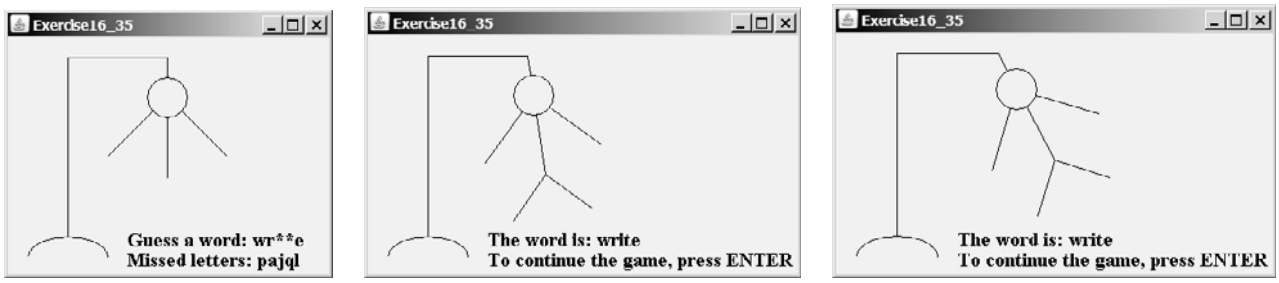


FIGURE 16.31 Exercise 16.34 animates a hangman game.

16.35*** (*Game: hangman*) Exercise 9.31 presents a console version of the popular hangman game. Write a GUI program that lets a user play the game. The user guesses a word by entering one letter at a time, as shown in Figure 16.32(a). If the user misses seven times, a hanging man swings, as shown in Figure 16.32(b)–(c). Once a word is finished, the user can press the *Enter* key to continue to guess another word.



(a) (b) (c)

FIGURE 16.32 Exercise 16.35 develops a complete hangman game.

16.36* (*Flipping coins*) Write a program that displays head (`H`) or tail (`T`) for each of nine coins, as shown in Figure 16.33. When a cell is clicked, the coin is flipped. A cell is a `JLable`. Write a custom cell class that extends `JLable` with the mouse listener for handling the clicks. When the program starts, all cells initially display `H`.

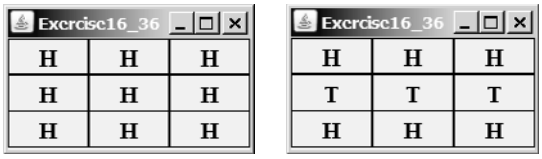


FIGURE 16.33 Exercise 16.36 enables the user to click a cell to flip a coin.

This page intentionally left blank