Module 13

Handling GUI-Generated Events

Objectives

Upon completion of this module, you should be able to:

- Define events and event handling
- Examine the Java SE event model
- Describe GUI behavior
- Determine the user action that originated an event
- Develop event listeners
- Describe concurrency in Swing-based GUIs and describe the features of the `SwingWorker` class

Additional Resources

Additional resources – The following references provide additional information on the topics described in this module:

- Arnold, Gosling, Holmes. *The Java Programming Language, Fourth Edition*. Prentice-Hall. 2005.
- Zakhour, Hommel, Royal, Rabinovitch, Risser, Hoeber. *The Java Tutorial: A Short Course on the Basics, Fourth Edition*. Prentice-Hall. 2006.
- Bates, Sierra. *Head First Java, Second Edition*. O’Reilly Media. 2005.

What Is an Event?

When the user performs an action at the user interface level (clicks a mouse or presses a key), this causes an *event* to be issued. Events are objects that describe what has happened. A number of different types of event classes exist to describe different categories of user action.

Figure 13-1 shows an abstract view of the delegation event model. When a user clicks a GUI button, an event object is created by the JVM and the button sends this event to the event handler for that button by calling the `actionPerformed` method.

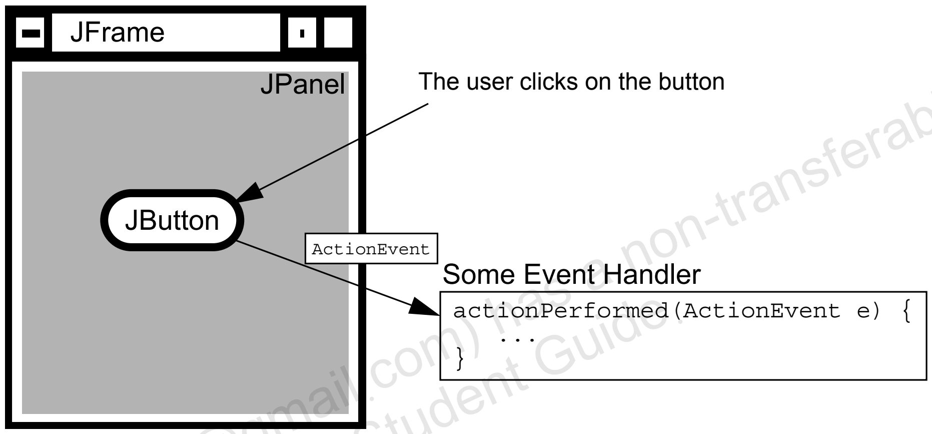
JFrame
JPanel
The user clicks on the button
JButton
ActionEvent
Some Event Handler
actionPerformed(ActionEvent e) {
...
}

Figure 13-1 A User Action Triggers an Event

Event Sources

An *event source* is the generator of an event. For example, a mouse click on a `JButton` component generates an `ActionEvent` instance with the button as the source. The `ActionEvent` instance is an object that contains information about the events that just took place. The `ActionEvent` contains:

- `getActionCommand`, which returns the command name associated with the action
- `getModifiers`, which returns any modifiers held during the action
- `getWhen`, which returns the timestamp when the event occurred
- `paramString`, which returns a string identifying action and the associated command

Event Handlers

An *event handler* is a method that receives an event object, deciphers it, and processes the user's interaction.

Java SE Event Model

This section describes the delegation event model.

Delegation Model

The delegation event model came into existence with JDK version 1.1. With this model, events are sent to the component from which the event originated, but it is up to each component to propagate the event to one or more registered classes, called *listeners*. Listeners contain event handlers that receive and process the event. In this way, the event handler can be in an object separate from the component. Listeners are classes that implement the `EventListener` interface. Figure 13-2 shows the event delegation model.

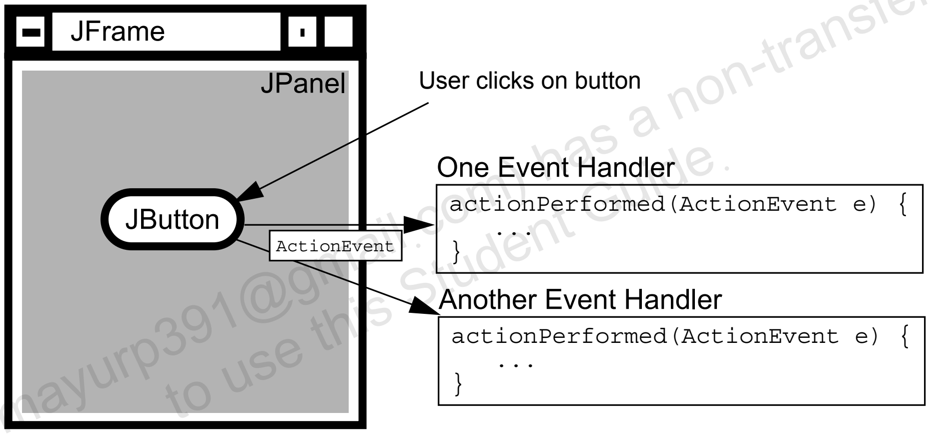

Figure 13-2 Delegation Event Model With Multiple Listeners

Events are objects that are reported only to registered listeners. Every event has a corresponding listener interface that mandates which methods must be defined in a class suited to receiving that type of event. The class that implements the interface defines those methods, and can be registered as a listener.

Events from components that have no registered listeners are not propagated.

A Listener Example

For example, Code 13-1 shows the code for a simple `JFrame` with a single `JButton` on it.

Code 13-1 The `TestButton` Example

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

public class TestButton {
    private JFrame f;
    private JButton b;

  public TestButton() {
    f = new JFrame("Test");
    b = new JButton("Press Me!");
    b.setActionCommand("ButtonPressed");
  }

  public void launchFrame() {
    b.addActionListener(new ButtonHandler());
    f.add(b,BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }

  public static void main(String args[]) {
    TestButton guiApp = new TestButton();
    guiApp.launchFrame();
  }
}
```

The `ButtonHandler` class, shown in Code 13-2, is the handler class to which the event is delegated.

Code 13-2 The `ButtonHandler` Example

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

public class ButtonHandler implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    System.out.println("Action occurred");
    System.out.println("Button’s command is: "
                         + e.getActionCommand());
  }
}
```

Code 13-2 on page 13-5 has the following characteristics:

- The JButton class inherits the addActionListener(ActionListener) method from javax.swing.AbstractButton which is the superclass of JButton.
- The ActionListener interface defines a single method, actionPerformed, which receives an ActionEvent.
- After it is created, a JButton object can have an object registered as a listener for ActionEvents through the addActionListener() method. The registered listener is instantiated from a class that implements the ActionListener interface.
- When the JButton object is clicked, an ActionEvent is sent. The ActionEvent is received through the actionPerformed() method of any ActionListener that is registered on the button through its addActionListener() method.
- The method, getActionCommand(), of the ActionEvent class returns the command name associated with this action. On Line 11, the action command for this button is set to ButtonPressed.

Note – There are other ways to determine why an event has been received. In particular, the method getSource(), documented in the java.util.EventObject base class, is often useful because it enables you to obtain the reference to the object that sent the event.

Events are not handled accidentally. The objects that are scheduled to listen to particular events on a particular GUI component register themselves with that component.

- When an event occurs, only the objects that were registered receive a message that the event occurred.
- The delegation model is good for the distribution of work among objects.

Events do not have to be related to Swing components. This event model provides support for the JavaBeans architecture.

GUI Behavior

This section describes the event categories.

Event Categories

The general mechanism for receiving events from components has been described in the context of a single type of event. Many of the event classes reside in the `java.awt.event` package, but others exist elsewhere in the API. Figure 13-3 shows a UML class hierarchy of GUI event classes.

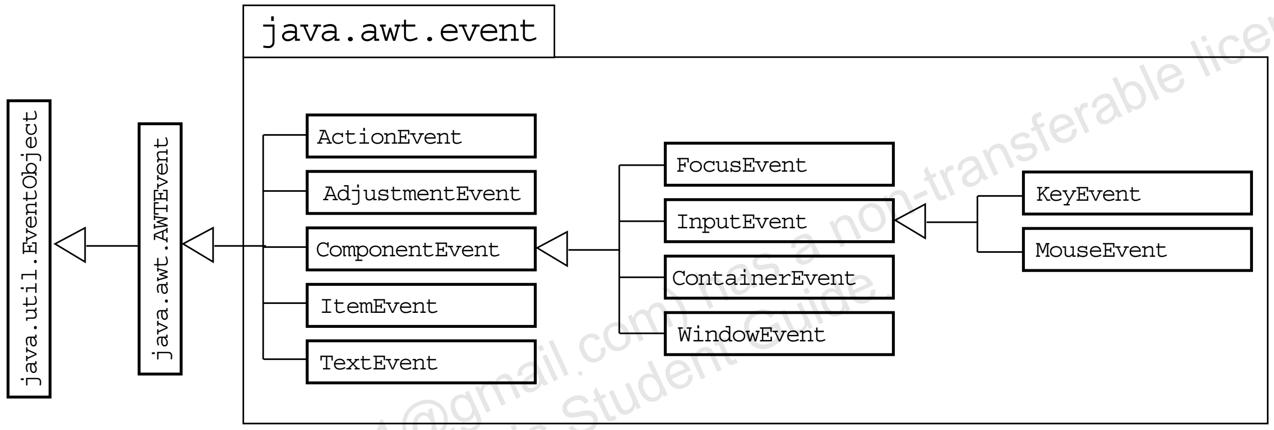

Figure 13-3 Class Hierarchy of GUI Events

For each category of events, an interface has to be implemented by the class of objects that wants to receive the events. That interface demands that one or more methods be defined as well. Those methods are called when particular events arise. Table 13-1 lists these categories and interfaces.

Table 13-1 Method Categories and Interfaces

Category	Interface Name	Methods
Action	`ActionListener`	`actionPerformed(ActionEvent)`
Item	`ItemListener`	`itemStateChanged(ItemEvent)`

Table 13-1 Method Categories and Interfaces (Continued)

Category	Interface Name	Methods

Mouse	`MouseListener`	`mousePressed(MouseEvent)`
		`mouseReleased(MouseEvent)`
		`mouseEntered(MouseEvent)`
		`mouseExited(MouseEvent)`
		`mouseClicked(MouseEvent)`

Mouse motion	`MouseMotionListener`	`mouseDragged(MouseEvent)`
		`mouseMoved(MouseEvent)`

Key	`KeyListener`	`keyPressed(KeyEvent)`
		`keyReleased(KeyEvent)`
		`keyTyped(KeyEvent)`

Focus	`FocusListener`	`focusGained(FocusEvent)`
		`focusLost(FocusEvent)`

Adjustment	`AdjustmentListener`	`adjustmentValueChanged (AdjustmentEvent)`

Component	`ComponentListener`	`componentMoved(ComponentEvent)`
		`componentHidden(ComponentEvent)`
		`componentResized(ComponentEvent)`
		`componentShown(ComponentEvent)`

Window	`WindowListener`	`windowClosing(WindowEvent)`
		`windowOpened(WindowEvent)`
		`windowIconified(WindowEvent)`
		`windowDeiconified(WindowEvent)`
		`windowClosed(WindowEvent)`
		`windowActivated(WindowEvent)`
		`windowDeactivated(WindowEvent)`

Container	`ContainerListener`	`componentAdded(ContainerEvent)`
		`componentRemoved(ContainerEvent)`

Window state	`WindowStateListener`	`windowStateChanged(WindowEvent e)`

Window focus	`WindowFocusListener`	`windowGainedFocus(WindowEvent e)`
		`windowLostFocus(WindowEvent e)`

Mouse wheel	`MouseWheelListener`	`mouseWheelMoved(MouseWheelEvent e)`

Table 13-1 Method Categories and Interfaces (Continued)

Category	Interface Name	Methods
Input methods	`InputMethodListener`	`caretPositionChanged (InputMethodEvent e)` `inputMethodTextChnaged (InputMethodEvent e)`
Hierarchy	`HierarchyListener`	`hierarchyChanged(HierarchyEvent e)`
Hierarchy bounds	`HierarchyBoundsListener`	`ancestorMoved(HierarchyEvent e)` `ancestorResized(HierarchyEvent e)`
AWT	`AWTEventListener`	`eventDispatched(AWTEvent e)`
Text	`TextListener`	`textValueChanged(TextEvent)`

Complex Example

This section examines a more complex Java code software example. It tracks the movement of the mouse when the mouse button is pressed (*mouse dragging*). It also detects mouse movement even when the buttons are not pressed (*mouse moving*).

The events caused by moving the mouse with or without a button pressed can be picked up by objects of a class that implements the `MouseMotionListener` interface. This interface requires two methods, `mouseDragged()` and `mouseMoved()`. Even if you are interested only in the drag movement, you must provide both methods. However, the body of the `mouseMoved()` method can be empty.

To pick up other mouse events, including mouse clicking, you must implement the `MouseListener` interface. This interface includes several events, including `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased`, and `mouseClicked`.

When mouse or keyboard events occur, information about the position of the mouse or the key that was pressed is available in the event that it generated. In Code 13-2 on page 13-5 on event handling, there was a separate class named `ButtonHandler` that handled events. In Code 13-3, events are handled within the class named `TwoListener`.

Code 13-3 The `TwoListener` Example

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

public class TwoListener
       implements MouseMotionListener, MouseListener {
  private JFrame f;
  private JTextField tf;

  public TwoListener() {
    f = new JFrame("Two listeners example");
    tf = new JTextField(30);
  }

  public void launchFrame() {
    JLabel label = new JLabel("Click and drag the mouse");
    // Add components to the frame
    f.add(label, BorderLayout.NORTH);
    f.add(tf, BorderLayout.SOUTH);
    // Add this object as a listener
    f.addMouseMotionListener(this);
    f.addMouseListener(this);
    // Size the frame and make it visible
    f.setSize(300, 200);
    f.setVisible(true);
  }

  // These are MouseMotionListener events
  public void mouseDragged(MouseEvent e) {
    String s =  "Mouse dragging:  X = " + e.getX()
                + " Y = " + e.getY();
    tf.setText(s);
  }

  public void mouseEntered(MouseEvent e) {
    String s = "The mouse entered";
    tf.setText(s);
  }

  public void mouseExited(MouseEvent e) {
    String s = "The mouse has left the building";
    tf.setText(s);
  }

  // Unused MouseMotionListener method.
  // All methods of a listener must be present in the
  // class even if they are not used.
48    public void mouseMoved(MouseEvent e) { }
49
50    // Unused MouseListener methods.
51    public void mousePressed(MouseEvent e) { }
52    public void mouseClicked(MouseEvent e) { }
53    public void mouseReleased(MouseEvent e) { }
54
55    public static void main(String args[]) {
56      TwoListener two = new TwoListener();
57      two.launchFrame();
58    }
59  }
```

A number of points shown in Code 13-3 on page 13-9 are described in the following paragraphs.

Implementing Multiple Interfaces

The class is declared in Lines 5 and 6 using the following:

```
implements MouseMotionListener, MouseListener
```

You can declare multiple interfaces by using comma separation.

Listening to Multiple Sources

If you issue the following method calls in Lines 20 and 21

```
f.addMouseListener(this);
f.addMouseMotionListener(this);
```

both types of events cause methods to be called in the `TwoListener` class. An object can *listen* to as many event sources as required. The object's class needs to implement only the required interfaces.

Obtaining Details About the Event

The event arguments with which handler methods, such as `mouseDragged()`, are called contain potentially important information about the original event. To determine the details of what information is available for each category of event, check the appropriate class documentation in the `java.awt.event` package.

Multiple Listeners

The AWT event listening framework permits the attachment of multiple listeners to the same component. In general, if you choose to write a program that performs multiple actions based on a single event, include code for that behavior in your handler method. However, sometimes a program's design requires multiple, unrelated parts of the same program to react to the same event. This might happen if, for example, a context-sensitive help system is added to an existing program.

The listener mechanism enables you to call an `addXxxListener()` method as many times as needed, and you can specify as many different listeners as your design requires. All registered listeners have their handler methods called when the event occurs.

Note – The order in which the handler methods are called is undefined. Generally, if the order of invocation matters, then the handlers are not unrelated. In this case, register only the first listener and have that listener call others directly.

Developing Event Listeners

In this section, you will learn about the design and implementation choices for implementing event listeners.

Event Adapters

It is a lot of work to implement all of the methods in each of the listener interfaces, particularly the `MouseListener` interface and `WindowListener` interface.

For example, the `MouseListener` interface declares the following methods:

```
public void mouseClicked(MouseEvent event)
public void mouseEntered(MouseEvent event)
public void mouseExited(MouseEvent event)
public void mousePressed(MouseEvent event)
public void mouseReleased(MouseEvent event)
```

As a convenience, the Java programming language provides adapter classes that implement each interface containing more than one method. The methods in these adapter classes are empty.

You can extend an adapter class and override only those methods that you need, as shown in Code 13-4.

Code 13-4 The `MouseClickHandler` Example

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

public class MouseClickHandler extends MouseAdapter {

  // We just need the mouseClick handler, so we use
  // an adapter to avoid having to write all the
  // event handler methods

  public void mouseClicked(MouseEvent e) {
    // Do stuff with the mouse click...
  }
}
```

Event Handling Using Inner Classes

Lines 26 and 12–18 in Code 13-5 show how to create event handlers as inner classes. Using inner classes for event handlers gives you access to the private data of the outer class (Line 16).

Code 13-5 The `TestInner` Example

```
1   import java.awt.*;
2   import java.awt.event.*;
3   import javax.swing.*;
4   public class TestInner {
5     private JFrame f;
6     private JTextField tf;
7
8     public TestInner() {
9       f = new JFrame("Inner classes example");
10      tf = new JTextField(30);
11    }
12
13    class MyMouseMotionListener extends MouseMotionAdapter {
14        public void mouseDragged(MouseEvent e) {
15          String s = "Mouse dragging:  X = "+ e.getX()
16                      + " Y = " + e.getY();
17          tf.setText(s);
18        }
19    }
20
21    public void launchFrame() {
22      JLabel label = new JLabel("Click and drag the mouse");
23      // Add components to the frame
24      f.add(label, BorderLayout.NORTH);
25      f.add(tf, BorderLayout.SOUTH);
26      // Add a listener that uses an Inner class
27      f.addMouseMotionListener(new MyMouseMotionListener());
28      f.addMouseListener(new MouseClickHandler());
29      // Size the frame and make it visible
30      f.setSize(300, 200);
31      f.setVisible(true);
32    }
33
34    public static void main(String args[]) {
35      TestInner obj = new TestInner();
36      obj.launchFrame();
37    }
38  }
```

Event Handling Using Anonymous Classes

You can include an entire class definition within the scope of an expression. This approach defines what is called an *anonymous* inner class and creates the instance all at one time. Anonymous inner classes are often used in event handling, Code 13-6 shows an example.

Code 13-6 The `TestAnonymous` Example

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

public class TestAnonymous {
  private JFrame f;
  private JTextField tf;

  public TestAnonymous() {
    f = new JFrame("Anonymous classes example");
    tf = new JTextField(30);
  }

  public void launchFrame() {
    JLabel label = new JLabel("Click and drag the mouse");
    // Add components to the frame
    f.add(label, BorderLayout.NORTH);
    f.add(tf, BorderLayout.SOUTH);
    // Add a listener that uses an anonymous class
    f.addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        String s = "Mouse dragging:  X = "+ e.getX()
                   + " Y = " + e.getY();
        tf.setText(s);
      }
    }); // <- note the closing parenthesis
    f.addMouseListener(new MouseClickHandler()); // Not shown
    // Size the frame and make it visible
    f.setSize(300, 200);
    f.setVisible(true);
  }

  public static void main(String args[]) {
    TestAnonymous obj = new TestAnonymous();
    obj.launchFrame();
  }
}
```

Note – The compilation of an anonymous class generates a file, such as `TestAnonymous$1.class`.

Concurrency in Swing

Applications that contain a GUI require several threads for handling the GUI efficiently.

- Threads responsible for executing the application code are called current threads.
- Threads responsible for handling the events generated by various components are called event dispatch threads.
- Threads responsible for executing lengthy tasks are called worker threads. Some examples of these lengthy tasks include waiting for some shared resource, waiting for user input, blocking for network or disk I/O, performing either CPU or memory-intensive calculations. These tasks can be executed in the background without affecting the performance of the GUI.

You can use instances of the `SwingWorker` class to represent these worker threads. The `SwingWorker` class extends the `Object` class and implements the `RunnableFuture` interface.

The `SwingWorker` class provides the following utility methods:

- For communication and coordination between worker thread tasks and tasks on other threads, the `SwingWorker` class provides properties such as `progress` and `state` to support inter-thread communication.
- To execute simple background tasks, the `doInBackground` method can be used for running the tasks background.
- To execute tasks that have intermediate results, the results are published in a GUI using the publish and process methods.
- To cancel the background threads, they can be canceled using the `cancel` method.

Note – A full description of the use of the `SwingWorker` class is outside the scope of this module. For more information, see the following URL: `http://java.sun.com/docs/books/tutorial/uiswing/concurrency/index.html`.