Chapter 16 Event-Driven Programming
Motivations

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. How do you accomplish the task? You have to use event-driven programming to write the code to respond to the button-clicking event.

![LoanCalculator](image)
Motivations

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 solutions to this problem. An effective way to solve it is to use a timer in event-driven programming, which is the subject of this chapter.
Objectives

- To get a taste of event-driven programming (§16.1).
- 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 listener classes (§16.6).
- To develop a GUI application for a loan calculator (§16.7).
- To write programs to deal with MouseEvents (§16.8).
- To simplify coding for listener classes using listener interface adapters (§16.9).
- To write programs to deal with KeyEvents (§16.10).
- To use the javax.swing.Timer class to control animations (§16.11).
Procedural vs. Event-Driven Programming

- *Procedural programming* is executed in procedural order.

- In event-driven programming, code is executed upon activation of events.
Taste of Event-Driven Programming

The example displays a button in the frame. A message is displayed on the console when a button is clicked.
Handling GUI Events

Source object (e.g., button)
Listener object contains a method for processing the event.
public class HandleEvent extends JFrame {
    public HandleEvent() {
        ...
        OKListenerClass listener1 = new OKListenerClass();
        jbtOK.addActionListener(listener1);
        ...
    }

    public static void main(String[] args) {
        ...
    }
}

class OKListenerClass implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("OK button clicked");
    }
}
public class HandleEvent extends JFrame {
    public HandleEvent() {
        ...
        OKListenerClass listener1 = new OKListenerClass();
        jbtOK.addActionListener(listener1);
        ...
    }

    public static void main(String[] args) {
        ...
    }
}

class OKListenerClass implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("OK button clicked");
    }
}
public class HandleEvent extends JFrame {
    public HandleEvent() {
        ...
        OKListenerClass listener1 = new OKListenerClass();
        jbtOK.addActionListener(listener1);
        ...
    }

    public static void main(String[] args) {
        ...
    }
}

class OKListenerClass implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.out.println("OK button clicked");
    }
}
Events

- An *event* can be defined as a type of signal to the program that something has happened.

- The event is generated by external user actions such as mouse movements, mouse clicks, and keystrokes, or by the operating system, such as a timer.
Event Classes

EventObject

AWTEvent

ComponentEvent

AdjustmentEvent

ActionEvent

InputEvent

ContainerEvent

FocusEvent

WindowEvent

KeyEvent

MouseMove

ChangeEvent

ItemEvent

PaintEvent

ListSelectionEvent

ChangeEvent

TextEvent

FocusEvent
Event Information

An event object contains whatever properties are pertinent to the event. You can identify the source object of the event using the `getSource()` instance method in the `EventObject` class. The subclasses of `EventObject` deal with special types of events, such as button actions, window events, component events, mouse movements, and keystrokes. Table 15.1 lists external user actions, source objects, and event types generated.
# Selected User Actions

<table>
<thead>
<tr>
<th>User Action</th>
<th>Source Object</th>
<th>Event Type Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click a button</td>
<td>JButton</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Click a check box</td>
<td>JCheckBox</td>
<td>ItemEvent, ActionEvent</td>
</tr>
<tr>
<td>Click a radio button</td>
<td>JRadioButton</td>
<td>ItemEvent, ActionEvent</td>
</tr>
<tr>
<td>Press return on a text field</td>
<td>JTextField</td>
<td>ActionEvent</td>
</tr>
<tr>
<td>Select a new item</td>
<td>JComboBox</td>
<td>ItemEvent, ActionEvent</td>
</tr>
<tr>
<td>Window opened, closed, etc.</td>
<td>Window</td>
<td>WindowEvent</td>
</tr>
<tr>
<td>Mouse pressed, released, etc.</td>
<td>Component</td>
<td>MouseEvent</td>
</tr>
<tr>
<td>Key released, pressed, etc.</td>
<td>Component</td>
<td>KeyEvent</td>
</tr>
</tbody>
</table>
The Delegation Model

(a) A generic source component with a generic listener

User Action

source: SourceClass

+addXListener(listener: XListener)

XListener

+handler(event: XEvent)

listener: ListenerClass

User Action

Register by invoking source.addXListener(listener);

(b) A JButton source component with an ActionListener

source: JButton

+addActionListener(listener: ActionListener)

ActionListener

+actionPerformed(event: ActionEvent)

listener: CustomListenerClass

Register by invoking source.addActionListener(listener);
Internal Function of a Source Component

(a) Internal function of a generic source object

source: SourceClass

+addXListener(XListener listener)

An event is triggered

event: XEvent

Invoke
listener1.handler(event)
listener2.handler(event)
...
listenerN.handler(event)

Keep it a list

(b) Internal function of a JButton object

source: JButton

+addActionListener(ClickListener listener)

An event is triggered

event: ActionEvent

Invoke
listener1.actionPerformed(event)
listener2.actionPerformed(event)
...
listenerN.actionPerformed(event)

Keep it a list
The Delegation Model: Example

JButton jbt = new JButton("OK");
ActionListener listener = new OKListener();
jbt.addActionListener(listener);
## Selected Event Handlers

<table>
<thead>
<tr>
<th>Event Class</th>
<th>Listener Interface</th>
<th>Listener Methods (Handlers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionEvent</td>
<td>ActionListener</td>
<td><code>actionPerformed(ActionEvent)</code></td>
</tr>
<tr>
<td>ItemEvent</td>
<td>ItemListener</td>
<td><code>itemStateChanged(ItemEvent)</code></td>
</tr>
<tr>
<td>WindowEvent</td>
<td>WindowListener</td>
<td><code>windowClosing(WindowEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>windowOpened(WindowEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>windowIconified(WindowEvent)</code></td>
</tr>
<tr>
<td>ContainerEvent</td>
<td>ContainerListener</td>
<td><code>windowDeiconified(WindowEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>windowClosed(WindowEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>windowActivated(WindowEvent)</code></td>
</tr>
<tr>
<td>MouseEvent</td>
<td>MouseListener</td>
<td><code>windowDeactivated(WindowEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>componentAdded(ContainerEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>componentRemoved(ContainerEvent)</code></td>
</tr>
<tr>
<td>KeyEvent</td>
<td>KeyListener</td>
<td><code>mousePressed(MouseEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>mouseReleased(MouseEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>mouseClicked(MouseEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>mouseExited(MouseEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>mouseEntered(MouseEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>keyPressed(KeyEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>keyReleased(KeyEvent)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>keyTyped(KeyEvent)</code></td>
</tr>
</tbody>
</table>
java.awt.event.ActionEvent

java.util.EventObject

+getSource(): Object

Returns the object on which the event initially occurred.

java.awt.event.AWTEvent

java.awt.event.ActionEvent

+getActionCommand(): String

Returns the command string associated with this action. For a button, its text is the command string.

+getModifiers(): int

Returns the modifier keys held down during this action event.

+getWhen(): long

Returns the timestamp when this event occurred. The time is the number of milliseconds since January 1, 1970, 00:00:00 GMT.
Example: First Version for ControlCircle (no listeners)

Now let us consider to write a program that uses two buttons to control the size of a circle.
Example: Second Version for ControlCircle (with listener for Enlarge)

Now let us consider to write a program that uses two buttons to control the size of a circle.
Inner Class Listeners

A listener class is designed specifically to create a listener object for a GUI component (e.g., a button). It will not be shared by other applications. So, it is appropriate to define the listener class inside the frame class as an inner class.
Inner Classes

Inner class: A class is a member of another class.

Advantages: In some applications, you can use an inner class to make programs simple.

An inner class can reference the data and methods defined in the outer class in which it nests, so you do not need to pass the reference of the outer class to the constructor of the inner class.
Inner Classes, cont.

(a) 
```
public class Test {
    ...
}

public class A {
    ...
}
```

(b) 
```
public class Test {
    ...

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

(c) 
```
// 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();
        }
    }
}
```
Inner Classes (cont.)

- Inner classes can make programs simple and concise.
- An inner class supports the work of its containing outer class and is compiled into a class named \( OuterClassName\$InnerClassName.class \).

For example, the inner class \( InnerClass \) in \( OuterClass \) is compiled into \( OuterClass\$InnerClass.class \).
Inner Classes (cont.)

- An inner class can be declared public, protected, or private subject to the same visibility rules applied to a member of the class.

- An inner class can be declared 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.
Anonymous Inner Classes

- 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$nn.class`. For example, if the outer class `Test` has two anonymous inner classes, these two classes are compiled into `Test$1.class` and `Test$2.class`. 
Anonymous Inner Classes (cont.)

Inner class listeners can be shortened using anonymous inner classes. An anonymous inner class is an inner class without a name. It combines declaring an inner class and creating an instance of the class in one step. An anonymous inner class is declared as follows:

```java
new SuperClassName/InterfaceName() {
    // Implement or override methods in superclass or interface
    // Other methods if necessary
}
```
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.
Alternative Ways of Defining Listener Classes

You may also define the custom frame class that implements `ActionListener`.

FrameAsListenerDemo Run
Problem: Loan Calculator

LoanCalculator

Run
MouseEvent

**java.awt.event.MouseEvent**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+getButton(): int</td>
<td>Indicates which mouse button has been clicked.</td>
</tr>
<tr>
<td>+getClickCount(): int</td>
<td>Returns the number of mouse clicks associated with this event.</td>
</tr>
<tr>
<td>+getPoint(): java.awt.Point</td>
<td>Returns a Point object containing the x and y coordinates.</td>
</tr>
<tr>
<td>+getX(): int</td>
<td>Returns the x-coordinate of the mouse point.</td>
</tr>
<tr>
<td>+GetY(): int</td>
<td>Returns the y-coordinate of the mouse point.</td>
</tr>
</tbody>
</table>

**java.awt.event.InputEvent**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+getWhen(): long</td>
<td>Returns the timestamp when this event occurred.</td>
</tr>
<tr>
<td>+isAltDown(): boolean</td>
<td>Returns whether or not the Alt modifier is down on this event.</td>
</tr>
<tr>
<td>+isControlDown(): boolean</td>
<td>Returns whether or not the Control modifier is down on this event.</td>
</tr>
<tr>
<td>+isMetaDown(): boolean</td>
<td>Returns whether or not the Meta modifier is down on this event.</td>
</tr>
<tr>
<td>+isShiftDown(): boolean</td>
<td>Returns whether or not the Shift modifier is down on this event.</td>
</tr>
</tbody>
</table>
Handling Mouse Events

- Java provides two listener interfaces, `MouseListener` and `MouseMotionListener`, to handle mouse events.

- The `MouseListener` listens for actions such as when the mouse is pressed, released, entered, exited, or clicked.

- The `MouseMotionListener` listens for actions such as dragging or moving the mouse.
Handling Mouse Events

```
java.awt.eventMouseListener
+ mousePressed(e: MouseEvent): void
+ mouseReleased(e: MouseEvent): void
+ mouseClicked(e: MouseEvent): void
+ mouseEntered(e: MouseEvent): void
+ mouseExited(e: MouseEvent): void
```

Invoked when the mouse button has been pressed on the source component.
Invoked when the mouse button has been released on the source component.
Invoked when the mouse button has been clicked (pressed and released) on the source component.
Invoked when the mouse enters the source component.
Invoked when the mouse exits the source component.

```
java.awt.event.MouseMotionListener
+ mouseDragged(e: MouseEvent): void
+ mouseMoved(e: MouseEvent): void
```

Invoked when a mouse button is moved with a button pressed.
Invoked when a mouse button is moved without a button pressed.
Example: Moving Message Using Mouse

Objective: Create a program to display a message in a panel. You can use the mouse to move the message. The message moves as the mouse drags and is always displayed at the mouse point.
Handling Keyboard Events

To process a keyboard event, use the following handlers in the `KeyListener` interface:

- `keyPressed(KeyEvent e)`
  Called when a key is pressed.

- `keyReleased(KeyEvent e)`
  Called when a key is released.

- `keyTyped(KeyEvent e)`
  Called when a key is pressed and then released.
The `KeyEvent` Class

Methods:

- `getKeyChar()` method
- `getKeyCode()` method

Keys:

- Home `VK_HOME`
- End `VK_END`
- Page Up `VK_PGUP`
- Page Down `VK_PGDN`
- etc...
### The `KeyEvent` Class, cont.

- **java.awt.event.KeyEvent**
  - `getKeyChar()`: char
  - `getKeyCode()`: int

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getKeyChar()</code></td>
<td>Returns the character associated with the key in this event.</td>
</tr>
<tr>
<td><code>getKeyCode()</code></td>
<td>Returns the integer keyCode associated with the key in this event.</td>
</tr>
</tbody>
</table>
Example: Keyboard Events Demo

Objective: Display a user-input character. The user can also move the character up, down, left, and right using the arrow keys.

[keyEventDemo]
The Timer Class

Some non-GUI components can fire events. The `javax.swing.Timer` class is a source component that fires an ActionEvent at a predefined rate.

```
javax.swing.Timer
+Timer(delay: int, listener: ActionListener)
+addActionListener(listener: ActionListener): void
+start(): void +stop(): void +setDelay(delay: int): void
```

Creates a Timer with a specified delay in milliseconds and an ActionListener.
Adds an ActionListener to the timer.
Starts this timer.
Stops this timer.
Sets a new delay value for this timer.

The Timer class can be used to control animations. For example, you can use it to display a moving message.
Clock Animation

In Chapter 13, 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 how to repaint the clock.