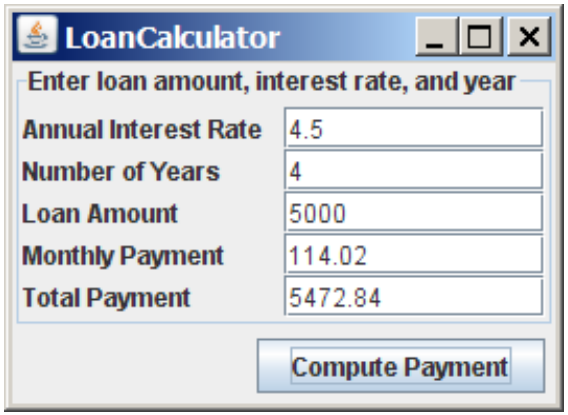
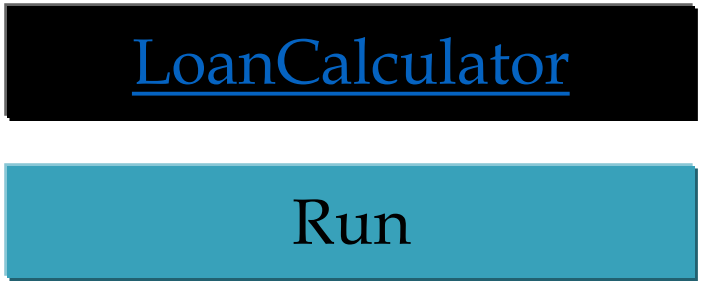
Chapter 16 Event-Driven Programming

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.





Procedural vs. Event-Driven Programming

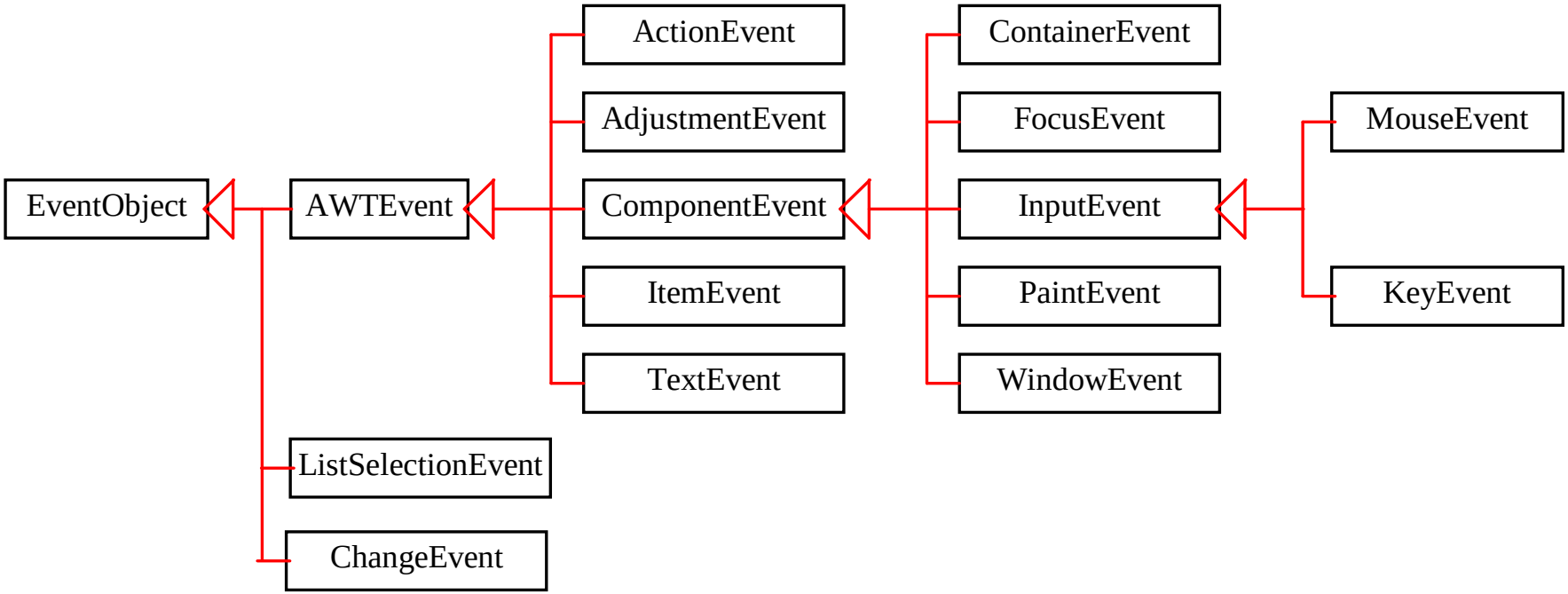
- *Procedural programming* is executed in procedural order.
- In event-driven programming, code is executed upon activation of events.

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



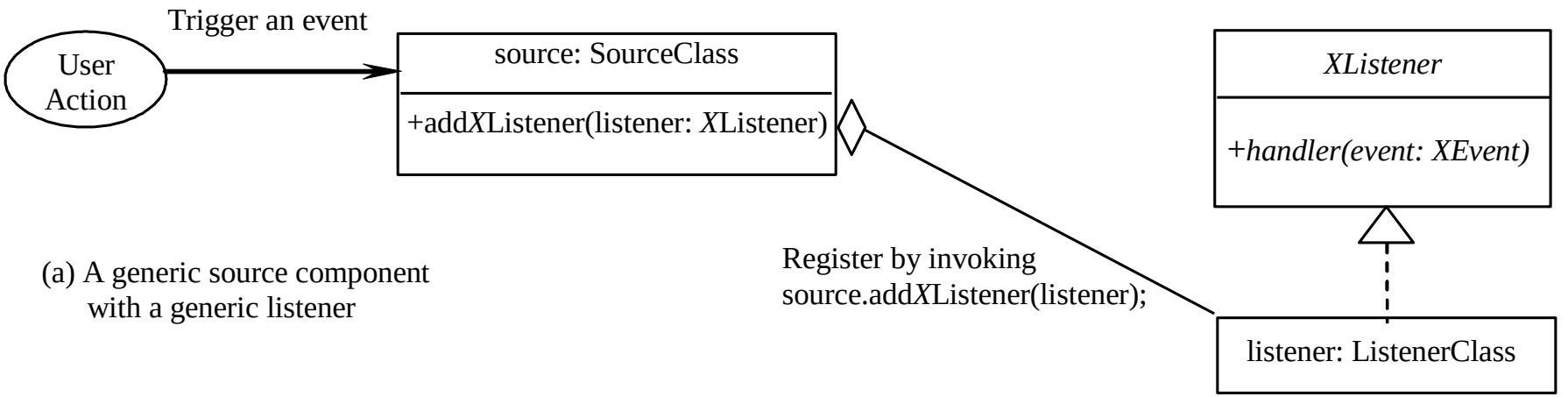
Event Information

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.

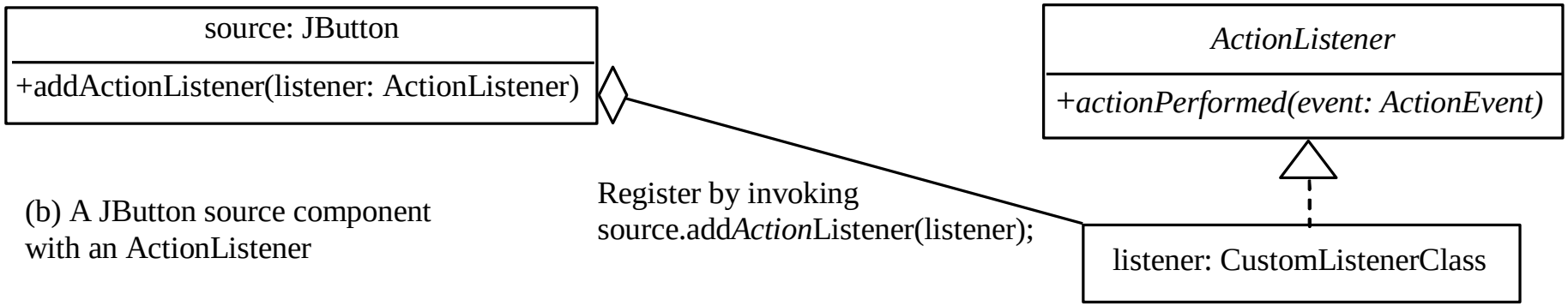
Selected User Actions

User Action	Source Object	Event Type Generated
Click a button	`JButton`	`ActionEvent`
Click a check box	`JCheckBox`	`ItemEvent`, `ActionEvent`
Click a radio button	`JRadioButton`	`ItemEvent`, `ActionEvent`
Press return on a text field	`JTextField`	`ActionEvent`
Select a new item	`JComboBox`	`ItemEvent`, `ActionEvent`
Window opened, closed, etc.	`Window`	`WindowEvent`
Mouse pressed, released, etc.	`Component`	`MouseEvent`
Key released, pressed, etc.	`Component`	`KeyEvent`

The Delegation Model



(a) A generic source component with a generic listener



(b) A JButton source component with an ActionListener

The Delegation Model: Example

```
class OKListener extends ActionListner {
     void actionPerformed(event e) {
          // some codes such as
          // System.exit(0);
     }
}
JButton jbt = new JButton("OK");
ActionListener listener = new OKListener();
jbt.addActionListener(listener);
```

Simplified Method - 1

```
JButton jbt = new JButton("OK");
jbt.addActionListener(
  new ActionListener() {
    void actionPerformed(Event t) {
      System.exit(0);
    }
  }
);
```

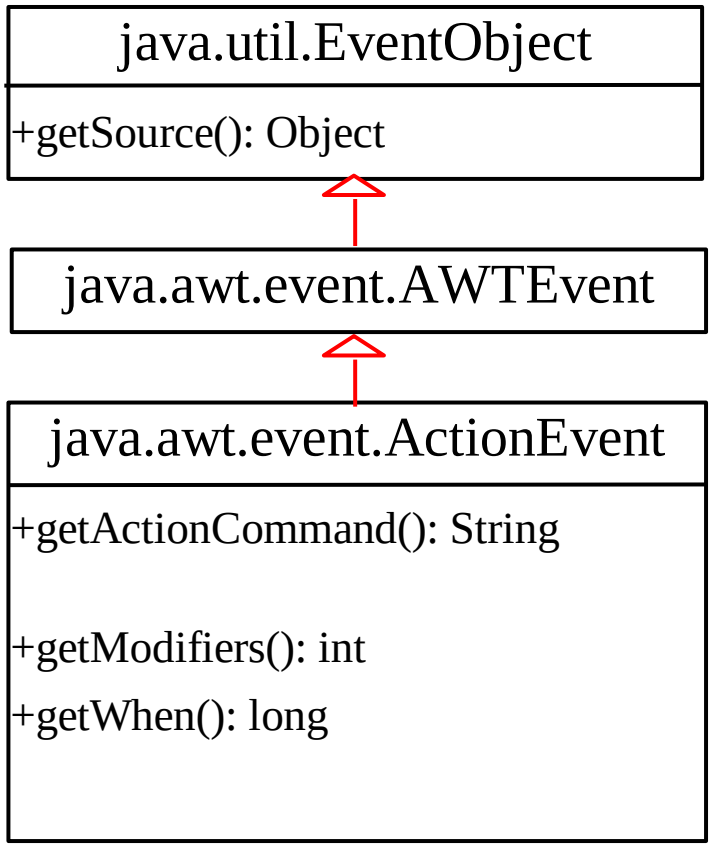
Simplified Method - 2

```
JButton jbt = new JButton("OK");
jbt.addActionListener(
   new ActionListener() {
     void actionPerformed(Event t) {
        jbtActionPerformed();
     }
   }
);
void jbtActionPerformed() {
    System.exit(0);
}
```

Selected Event Handlers

Event Class	Listener Interface	Listener Methods (Handlers)
ActionEvent	ActionListener	actionPerformed(ActionEvent)
ItemEvent	ItemListener	itemStateChanged(ItemEvent)
WindowEvent	WindowListener	windowClosing(WindowEvent)
		windowOpened(WindowEvent)
		windowIconified(WindowEvent)
		windowDeiconified(WindowEvent)
		windowClosed(WindowEvent)
		windowActivated(WindowEvent)
		windowDeactivated(WindowEvent)
ContainerEvent	ContainerListener	componentAdded(ContainerEvent)
		componentRemoved(ContainerEvent)
MouseEvent	MouseListener	mousePressed(MouseEvent)
		mouseReleased(MouseEvent)
		mouseClicked(MouseEvent)
		mouseExited(MouseEvent)
		mouseEntered(MouseEvent)
KeyEvent	KeyListener	keyPressed(KeyEvent)
		keyReleased(KeyEvent)
		keyTypeed(KeyEvent)

java.awt.event.ActionEvent



Returns the object on which the event initially occurred.

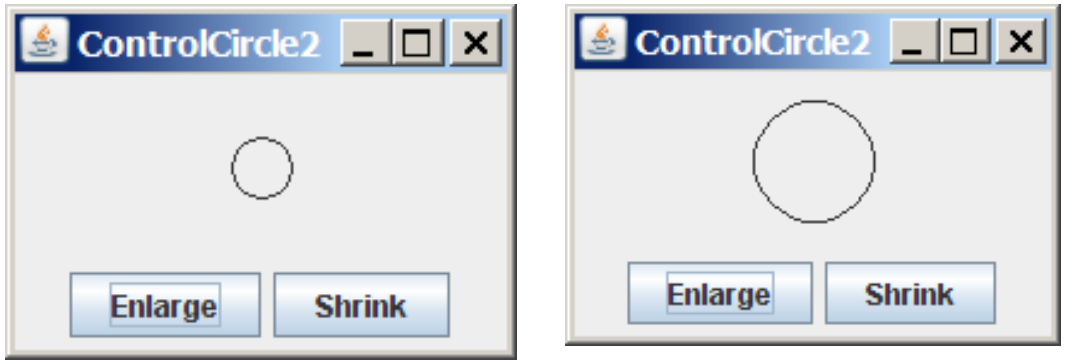
Returns the command string associated with this action. For a button, its text is the command string.

Returns the modifier keys held down during this action event.

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.

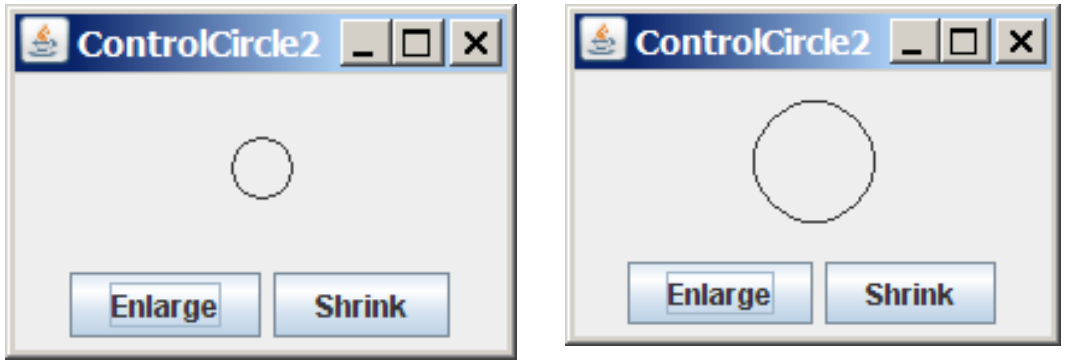


ControlCircle1

Run

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.



ControlCircle2 Run

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.

ShowInnerClass

Inner Classes, cont.

```
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)

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$*n*.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:

```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface
  // Other methods if necessary
}
```

AnonymousListenerDemo

Run

Alternative Ways of Defining Listener Classes

There are many other ways to define the listener classes. For example, you may rewrite Listing 6.3 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.

DetectSourceDemo

Run

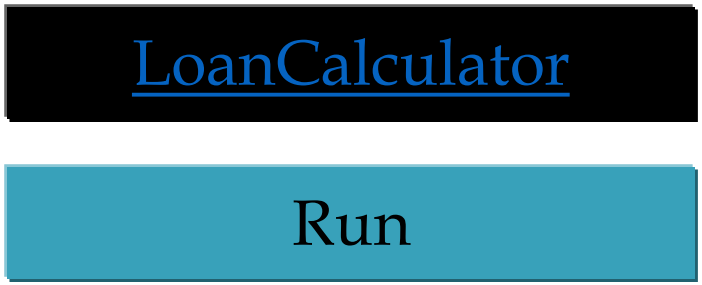
Alternative Ways of Defining Listener Classes

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

FrameAsListenerDemo

Run

Problem: Loan Calculator



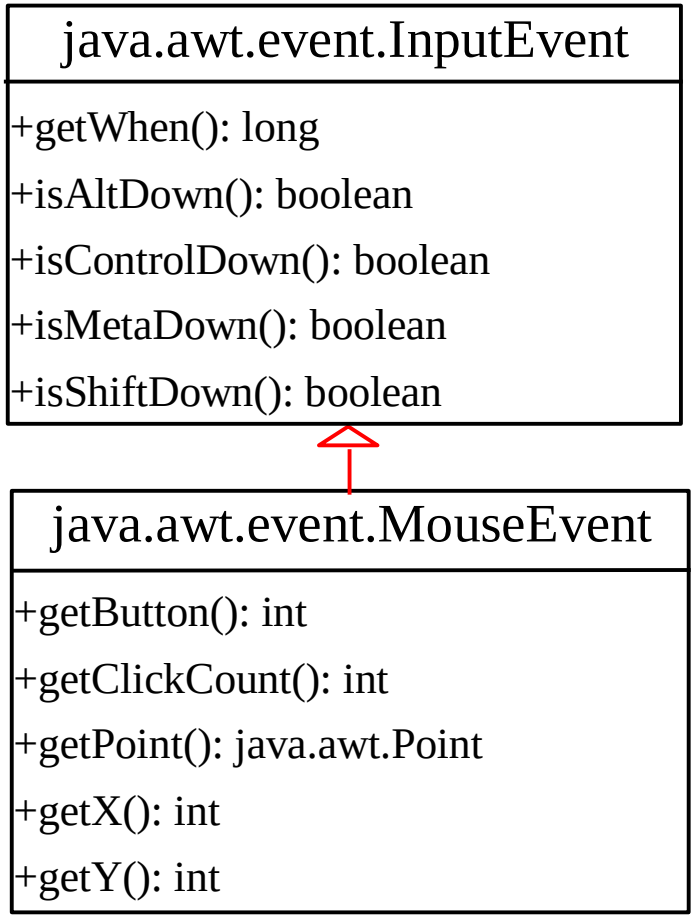
Example: Handling Window Events

- Objective: Demonstrate handling the window events. Any subclass of the Window class can generate the following window events: window opened, closing, closed, activated, deactivated, iconified, and deiconified. This program creates a frame, listens to the window events, and displays a message to indicate the occurring event.

TestWindowEvent

Run

MouseEvent



Returns the timestamp when this event occurred.

Returns whether or not the Alt modifier is down on this event.

Returns whether or not the Control modifier is down on this event.

Returns whether or not the Meta modifier is down on this event

Returns whether or not the Shift modifier is down on this event.

Indicates which mouse button has been clicked.

Returns the number of mouse clicks associated with this event.

Returns a Point object containing the x and y coordinates.

Returns the x-coordinate of the mouse point.

Returns the y-coordinate of the mouse point.

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.event.MouseListener	
+mousePressed(e: MouseEvent): void	Invoked when the mouse button has been pressed on the source component.
+mouseReleased(e: MouseEvent): void	Invoked when the mouse button has been released on the source component.
+mouseClicked(e: MouseEvent): void	Invoked when the mouse button has been clicked (pressed and released) on the source component.
+mouseEntered(e: MouseEvent): void	Invoked when the mouse enters the source component.
+mouseExited(e: MouseEvent): void	Invoked when the mouse exits the source component.

java.awt.event.MouseMotionListener	
+mouseDragged(e: MouseEvent): void	Invoked when a mouse button is moved with a button pressed.
+mouseMoved(e: MouseEvent): void	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.

MoveMessageDemo

Run

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:

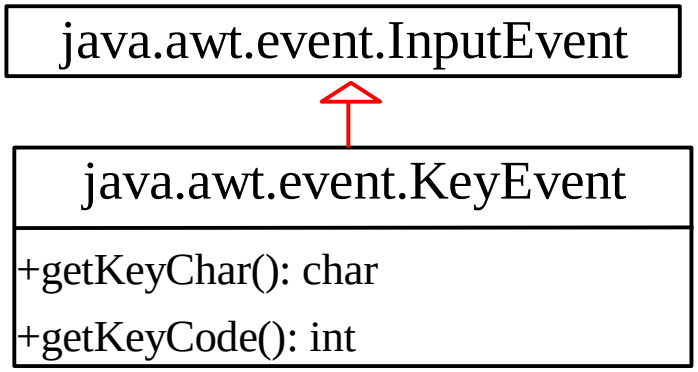
  `getKeyChar()` method

  `getKeyCode()` method

- Keys:

Home	`VK_HOME`
End	`VK_END`
Page Up	`VK_PGUP`
Page Down	`VK_PGDN`
etc...	

The `KeyEvent` Class, cont.

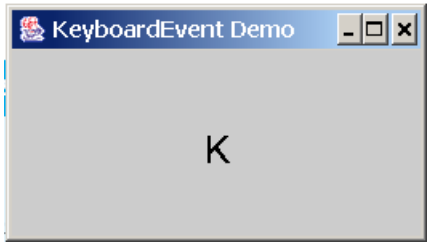


Returns the character associated with the key in this event.

Returns the integer keyCode associated with the key in this event.

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

Run

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)	Creates a Timer 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.

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

AnimationDemo

Run

Clock Animation

In Chapter 14, 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.

ClockAnimation

Run