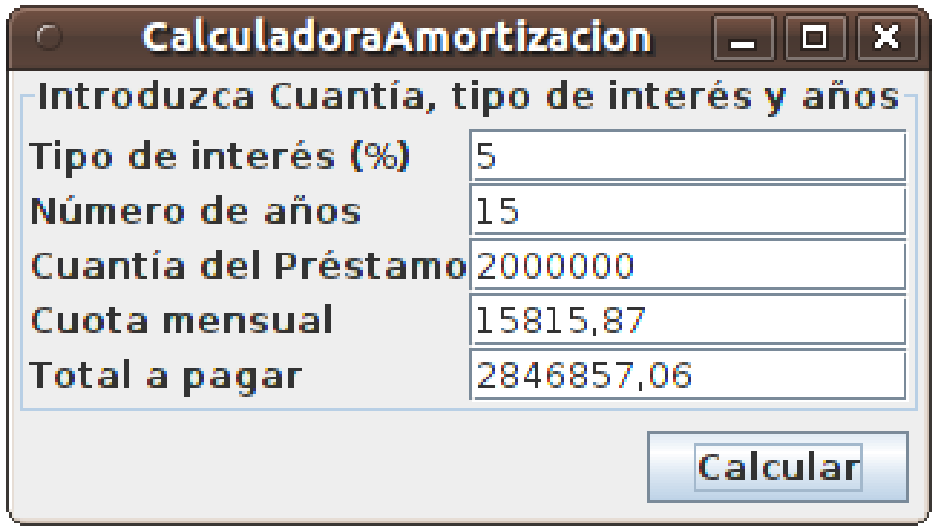
Programación orientada a eventos en Java

F. de Sande

2014–2015

Motivation

- 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



Motivation

Suppose you wish to write a program that animates a rising flag

- 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 lesson

Objectives

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

Procedural vs. Event-Driven Programming

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

Event-driven programming or event-based programming is a programming paradigm in which the flow of the program is determined by events, e.g. sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads

Taste of Event-Driven Programming

GestionaEvento.java

- This program displays a button in the frame
- A message is displayed on the console when a button is clicked

```
Kiko - icarin:~/doc/clases/2011-2012_PAI/transparencias_PAI/document/T4_Events/java
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
sande@icarin:~/doc/clases/2011-2012_PAI/transparencias_PAI/document/T4_Events/java$ java GestionaEvento
Ha clicado OK
Ha clicado OK
Ha clicado OK
```

Events

Definition

- An *event* can be defined as a type of signal to the program indicating 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
- In event-driven programming, code is executed when an event occurs
- The program can choose to respond to or ignore an event
- The component that creates an event and fires it is called the *source object* or *source component*.

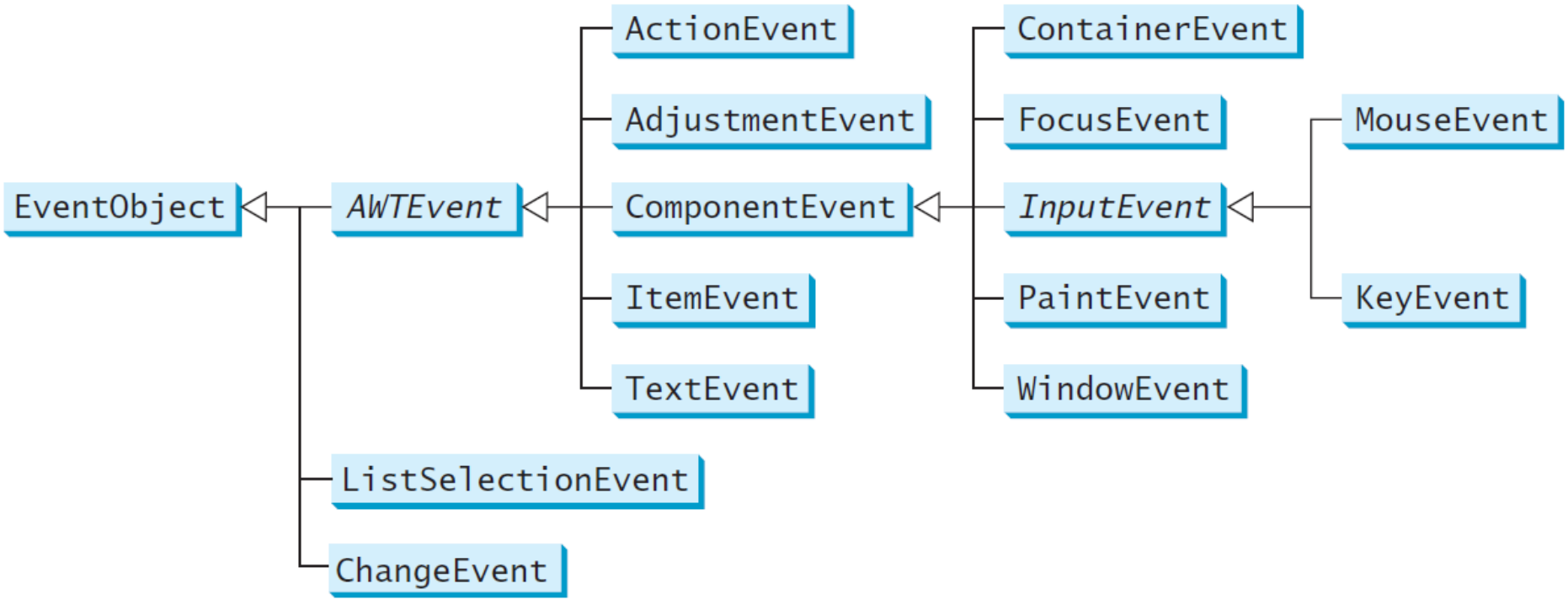
  For example, a button is the source object for a button-clicking action event

Event Classes

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 here



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
- 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
- Next table lists external user actions, source objects, and event types generated

User actions, source objects, and event types generated

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`

Listeners, registrations and handling events

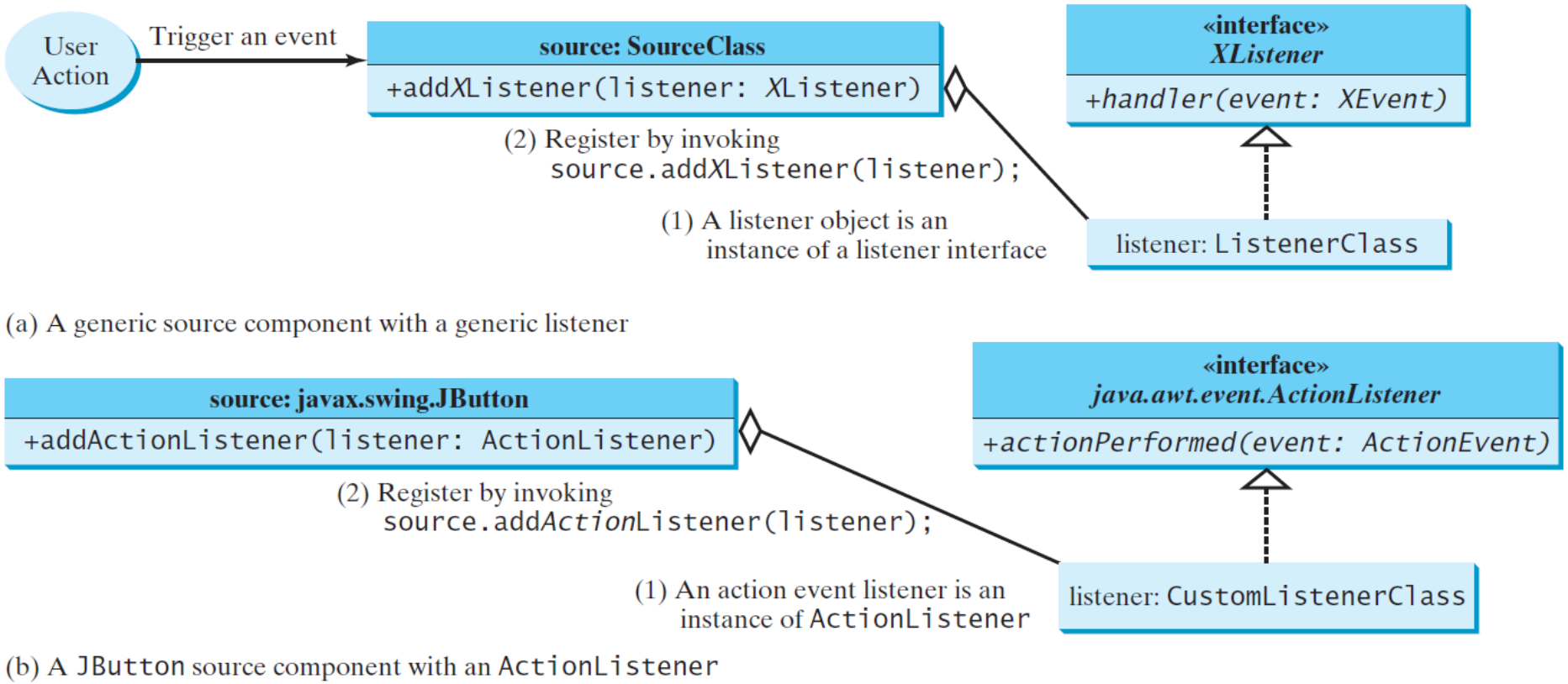
Java uses a delegation-based model for event handling:

(a) a source object fires an event, and
(b) 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
  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
  2. The listener object must be registered by the source object.
     Registration methods depend on the event type

The Delegation Model

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



(a) A generic source component with a generic listener

(b) A JButton source component with an ActionListener

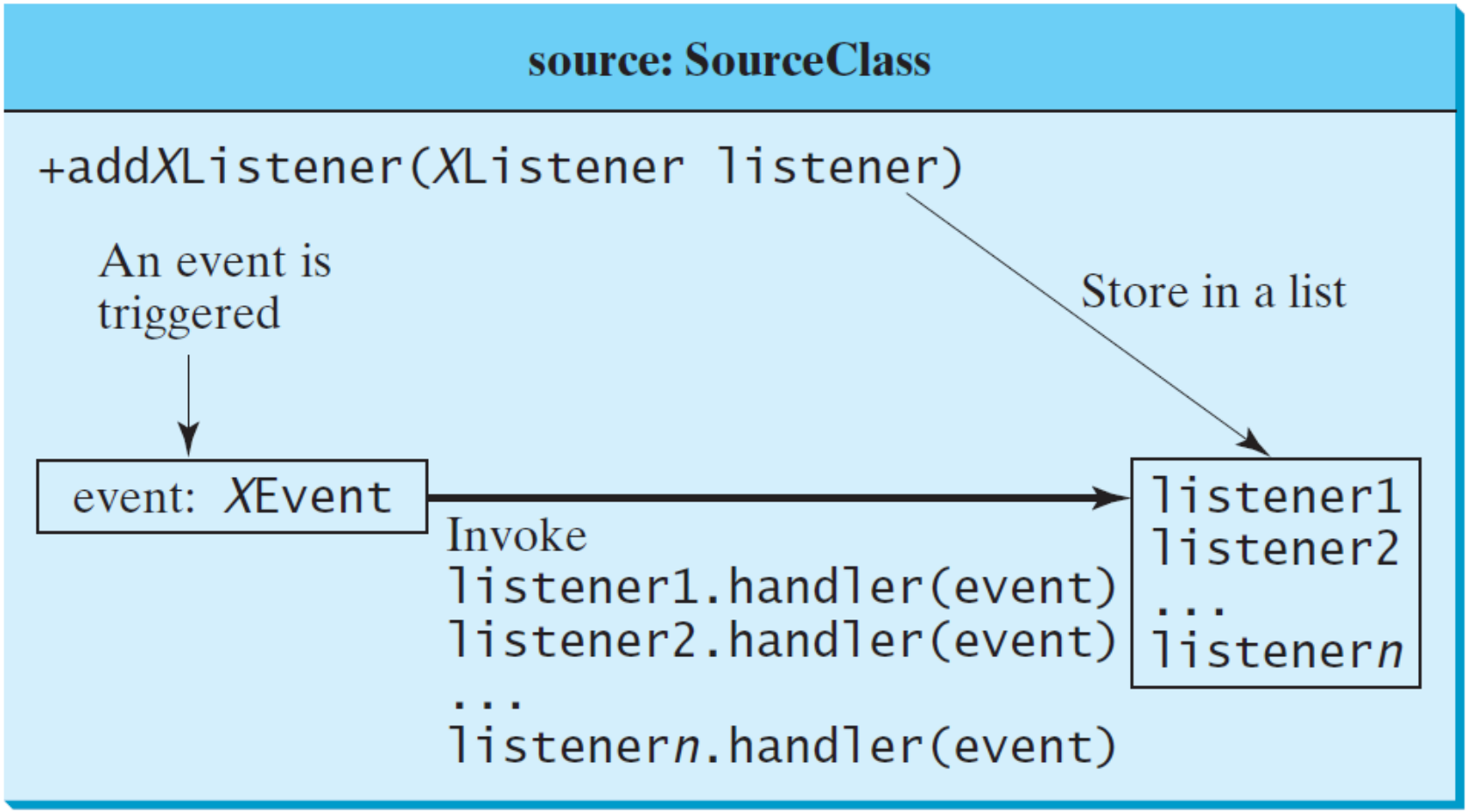
Internal Function of a Source Component

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

Internal Function of a Source Component

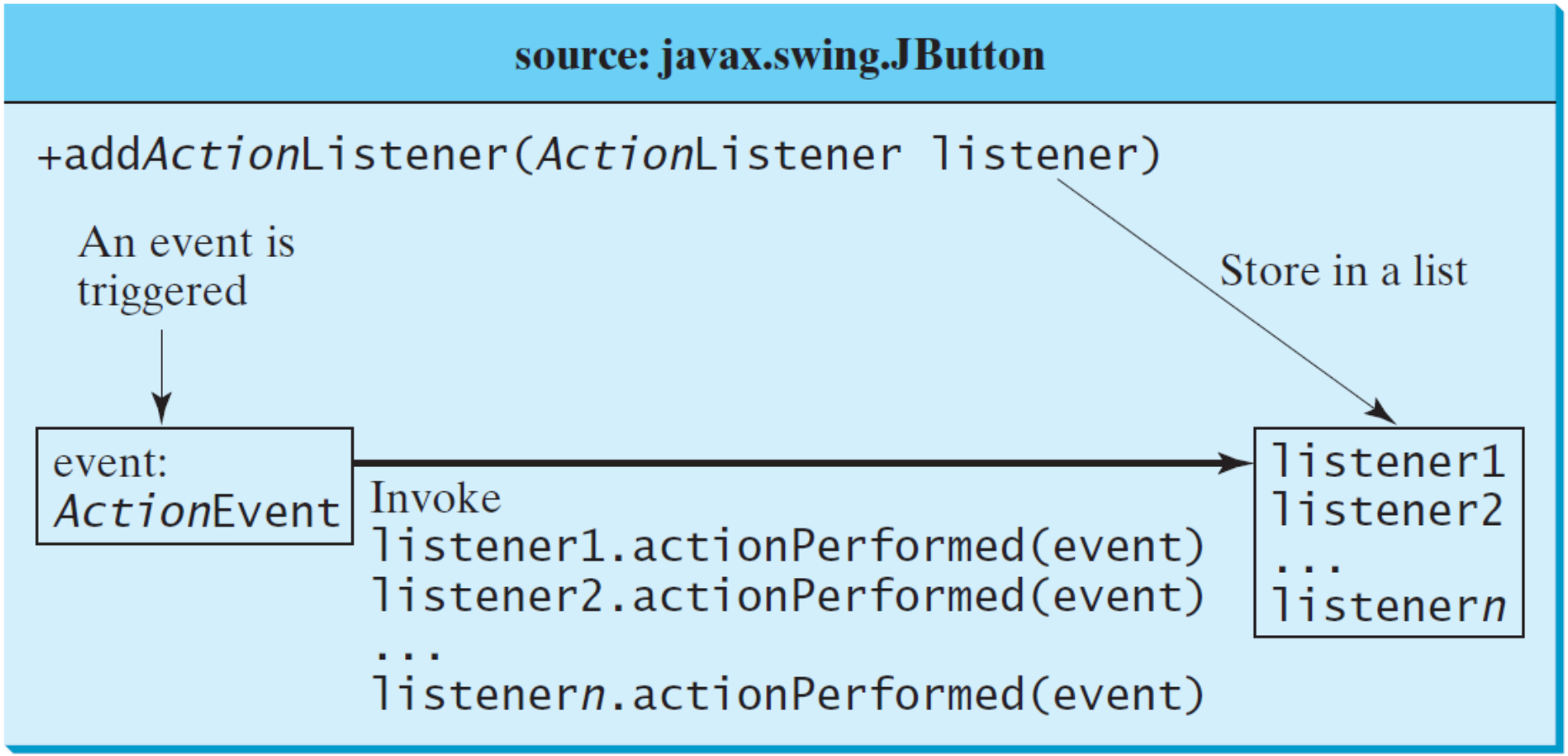
The source object notifies the listeners of the event by invoking the handler of the listener object



(a) Internal function of a generic source object

Internal Function of a Source Component

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



(b) Internal function of a JButton object

The Delegation Model: Example

GestionaEvento.java

- Since a `JButton` object fires `ActionEvent`, a listener object for `ActionEvent` must be an instance of `ActionListener`, so the listener class implements `ActionListener`
- The source object invokes `addActionListener(listener)` to register a listener, as follows:

```
JButton jboton = new JButton("OK");
ActionListener oyente = new OKoyente();
jboton.addActionListener(oyente);
```

- 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

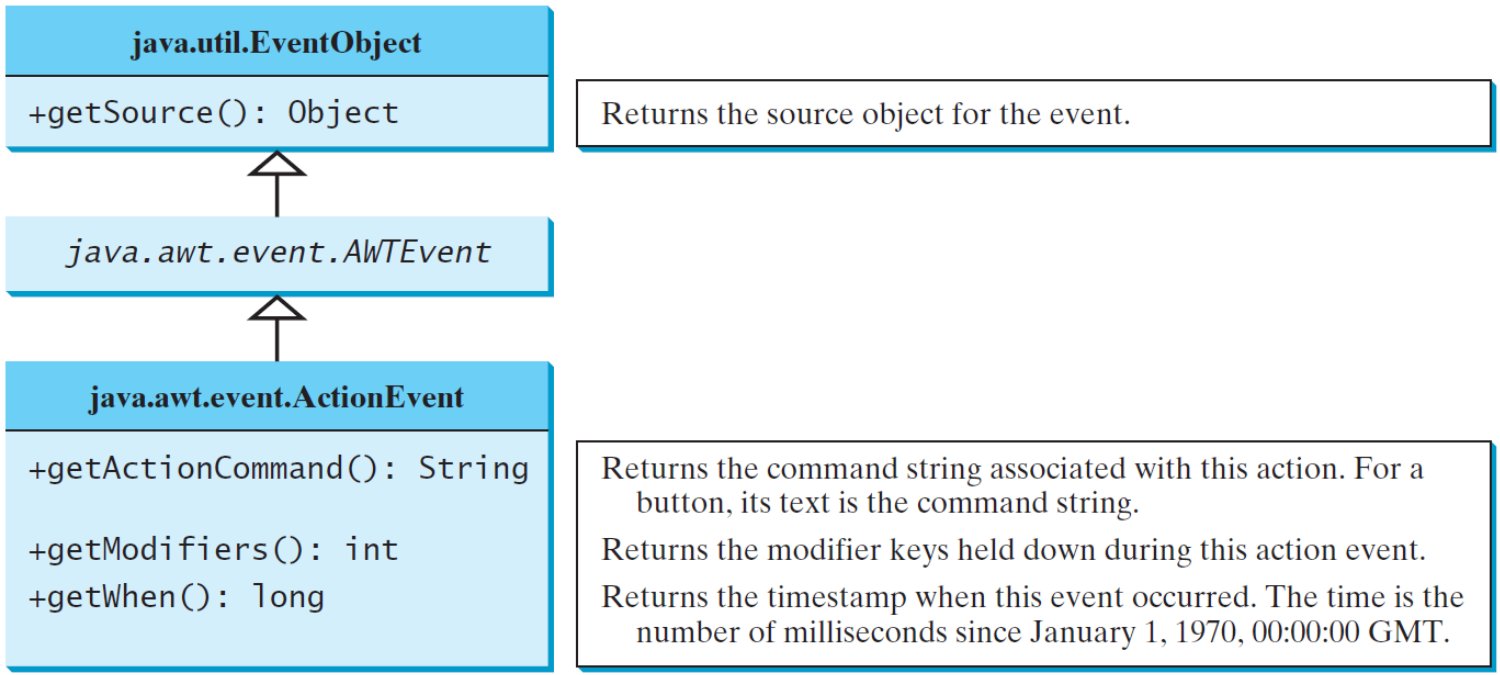
Selected Event Handlers

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)`

java.awt.event.ActionEvent

The event object contains information pertinent to the event, which can be obtained using the methods

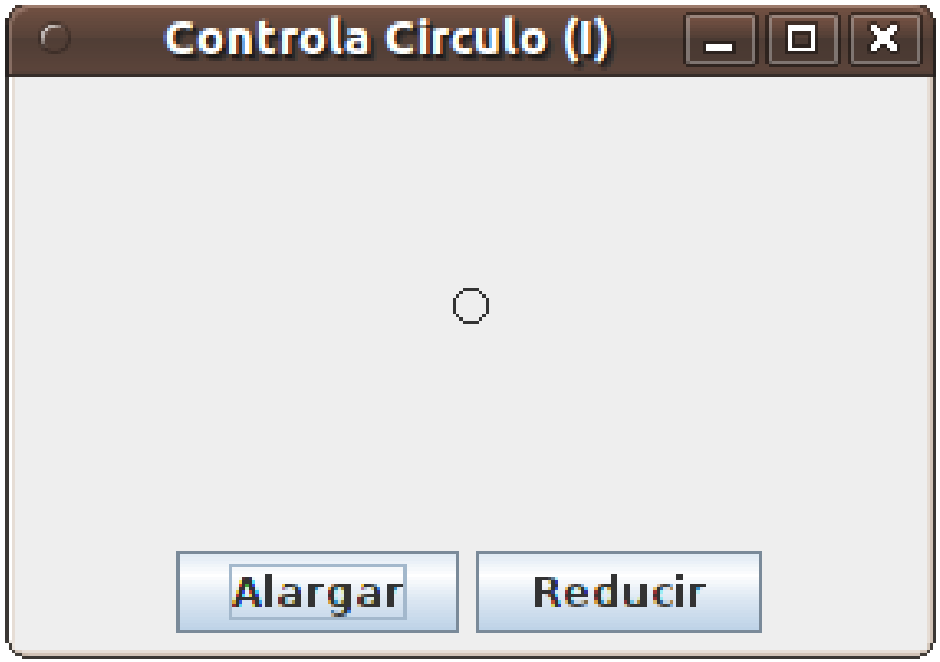
- For example, you can use `e.getSource()` to obtain the source object in order to determine whether it is a button, a check box, or a radio button
- For an action event, you can use `e.getWhen()` to obtain the time when the event occurs

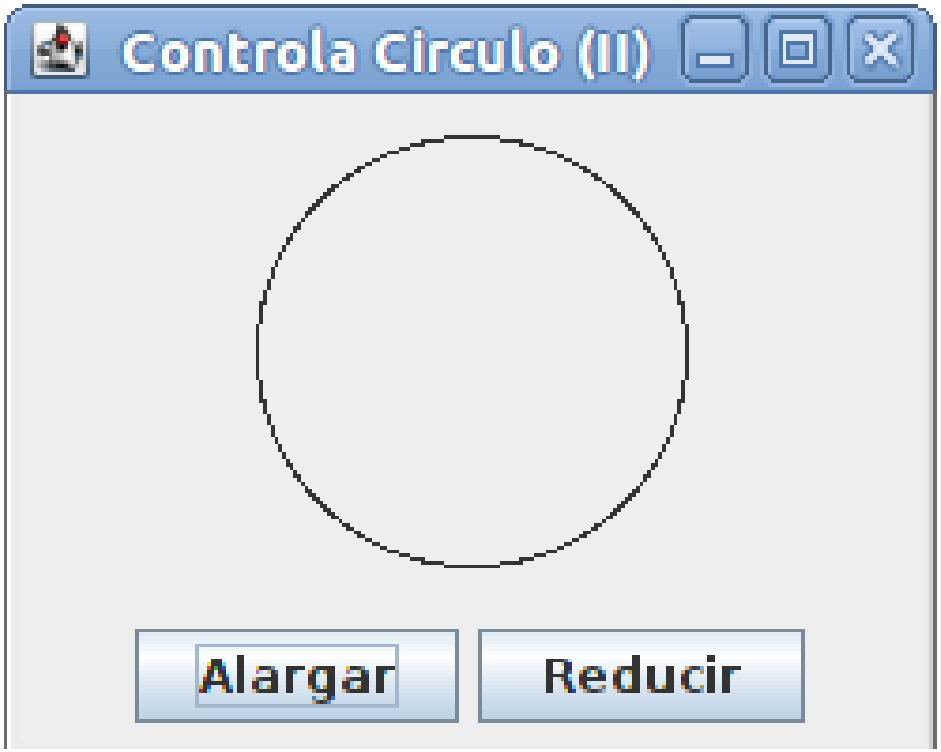


Example: `ControlaCirculo1.java`. A program that uses two buttons to control the size of a circle

First Version (no listeners)

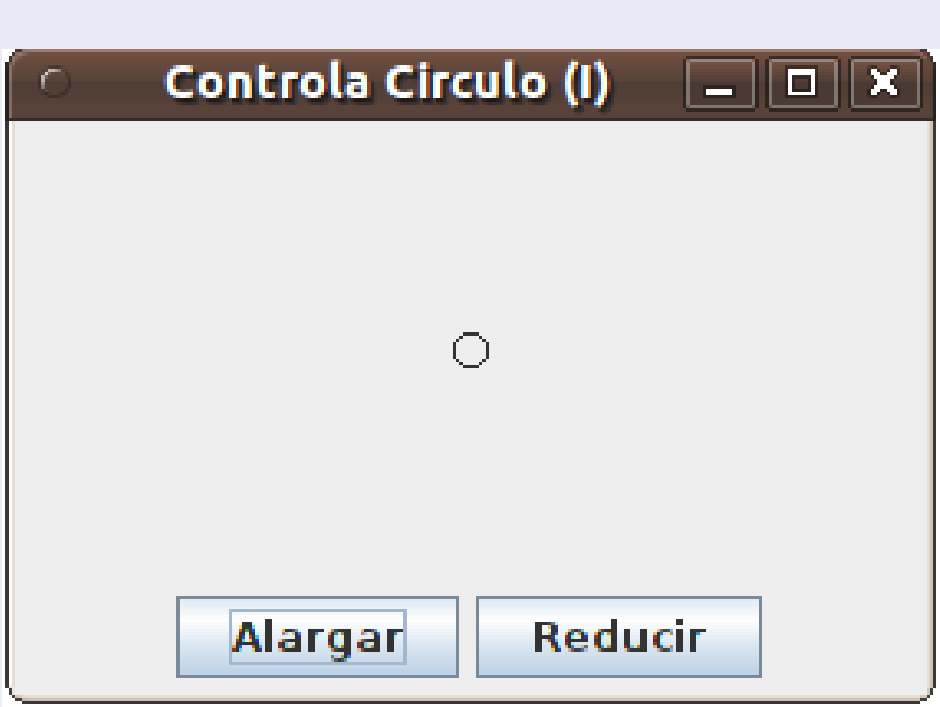
Displays the user interface with a circle in the center and two buttons in the bottom

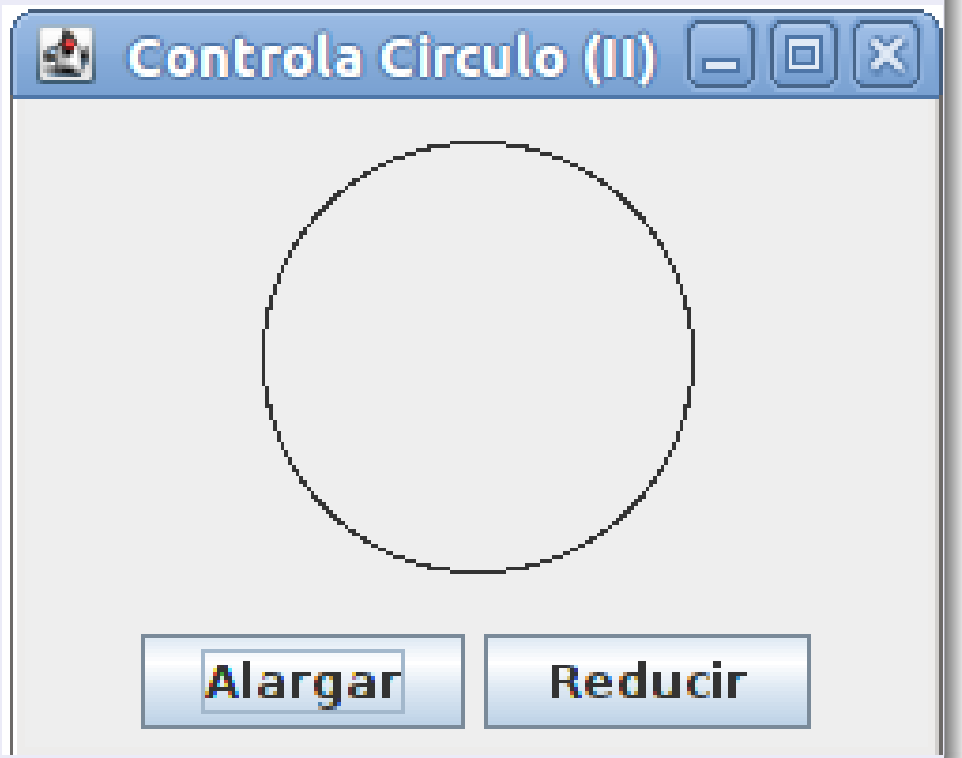




Example: `ControlaCirculo2.java`. A program that uses two buttons to control the size of a circle

Second Version (with listeners for enlarge/reduce)





Example: `ControlaCirculo2.java`

When the *Alargar* button is clicked, you want the circle to be repainted with a larger radius. How can you accomplish this?

1. Define a listener class (`Oyente`) that implements `ActionListener` (lines 31–39).
2. Create a listener and register it with `botonAlargar` (line 18)
3. Add a method named `alargar()` in `PanelCirculo` to increase the radius, then repaint the panel (lines 45–48).
4. Implement the `actionPerformed` method in `Oyente` to invoke `areaDibujo.alargar()` (line 35).
5. To make the reference variable `areaDibujo` accessible from the `actionPerformed` method, define `Oyente` (lines 31–39) as an inner class of the `ControlaCirculo2` class.
   Inner classes are defined inside another class.
   We will introduce inner classes later
6. The `PanelCirculo` class (lines 41–62) now is also defined as an inner class in `ControlaCirculo2`

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 or *Nested 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

Inner classes combine dependent classes into the primary class

```
public class Test {
  ...
}

public class A {
  ...
}
```

```
public class Test {
  ...

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

```
// 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

- It is a way of logically grouping classes that are only used in one place
- It increases encapsulation
- Inner classes can make programs simple, readable, concise and maintainable
- Normally, you define a class as inner class if it is used only by its outer class
- 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

- 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

`http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`

Anonymous Inner Classes

- A listener class is designed specifically to create a listener object for a GUI component
- 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
- 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
}
```

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

The inner class in `ControlaCirculo2.java` can be replaced by an anonymous inner class as:

```
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

Anonymous Listener Demo

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

public class AnonymousListenerDemo extends JFrame {
  public AnonymousListenerDemo() {
    // Create four buttons
    JButton botonNew = new JButton("New");
    JButton botonOpen = new JButton("Open");
    JButton botonSave = new JButton("Save");
    JButton botonPrint = new JButton("Print");

    // Create a panel to hold buttons
    JPanel panel = new JPanel();
    panel.add(botonNew);
    panel.add(botonOpen);
    panel.add(botonSave);
    panel.add(botonPrint);

    add(panel);

    // Create and register anonymous inner class listener
    botonNew.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          System.out.println("Process New");
        }
      }
    );

    botonOpen.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent e) {
          System.out.println("Process Open");
        }
```

AnonymousListenerDemo.java

- The program creates four listeners using anonymous inner classes
- 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
- 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

Alternative Ways of Defining Listener Classes

There are different ways to define the listener classes

For example, you may rewrite the `AnonymousListenerDemo` 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

`DetectaFuente.java`

- 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 and determines which button fired the event

Alternative Ways of Defining Listener Classes

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

Frame as Listener Demo

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

public class FrameAsListenerDemo extends JFrame implements ActionListener {
  // Create four buttons
  private JButton botonNuevo = new JButton("Nuevo");
  private JButton botonAbrir = new JButton("Abrir");
  private JButton botonGrabar = new JButton("Grabar");
  private JButton botonImprimir = new JButton("Imprimir");

  public FrameAsListenerDemo() {
    // Create a panel to hold buttons
    JPanel panel = new JPanel();
    panel.add(botonNuevo);
    panel.add(botonAbrir);
    panel.add(botonGrabar);
    panel.add(botonImprimir);

    add(panel);

    // Register listener with buttons
    botonNuevo.addActionListener(this);
    botonAbrir.addActionListener(this);
    botonGrabar.addActionListener(this);
    botonImprimir.addActionListener(this);
  }

  /** Implement actionPerformed */
  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == botonNuevo)
      System.out.println("Procesar Nuevo");
    else if (e.getSource() == botonAbrir)
      System.out.println("Procesar Abrir");
    else if (e.getSource() == botonGrabar)
```

FrameAsListenerDemo.java

- The frame class extends `JFrame` and implements `ActionListener`
- So the class is a listener class for action events
- The listener is registered to four buttons
- 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 and determines which button fired the event
- This design is not desirable **Why?**

FrameAsListenerDemo.java

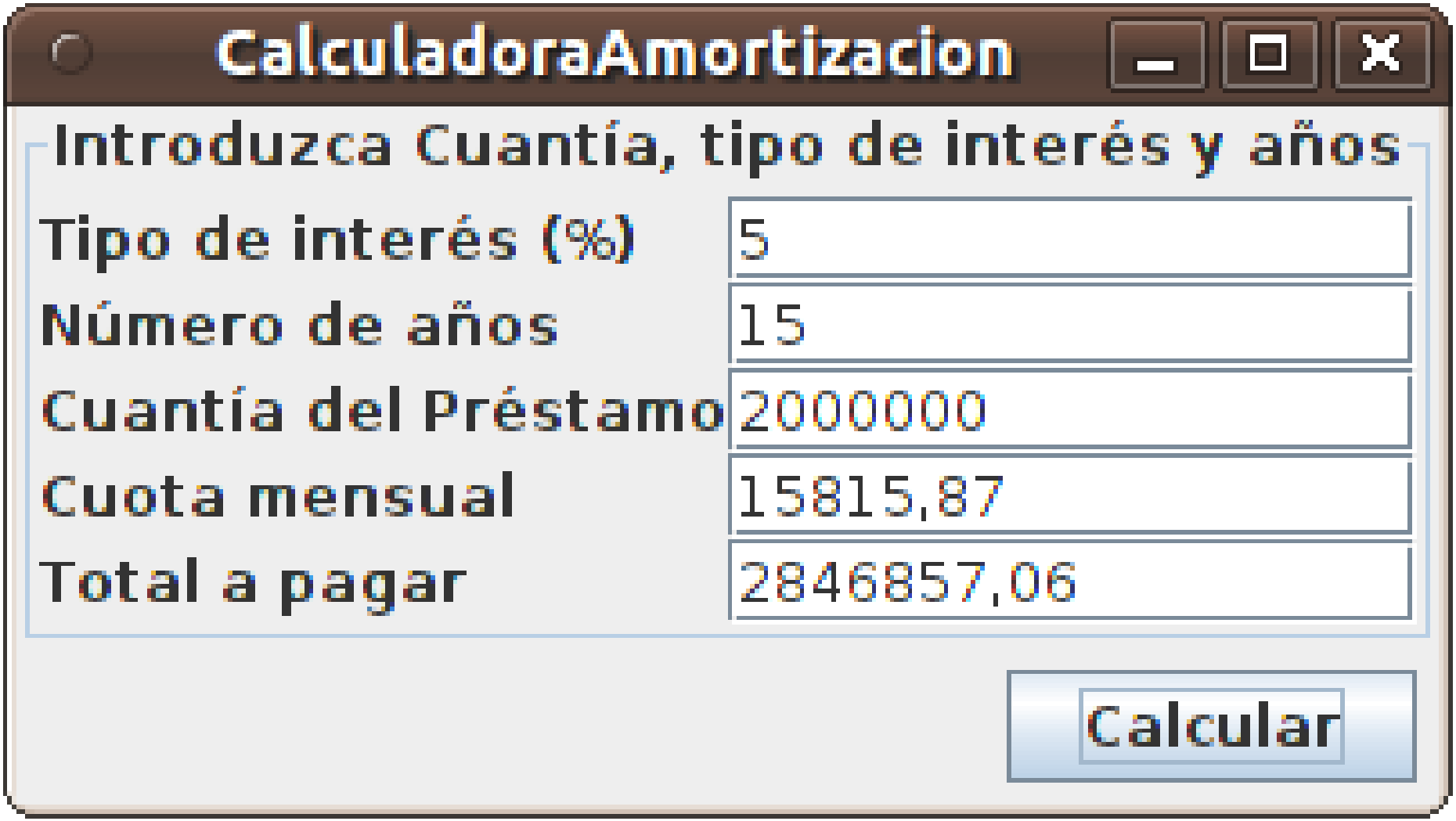
- Remember the Single Responsibility Principle:
- This design is not desirable because it places too many responsibilities into one class: each class should have only one responsibility
- It is better to design a listener class that is solely responsible for handling events

Listener classes

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

Calculadora

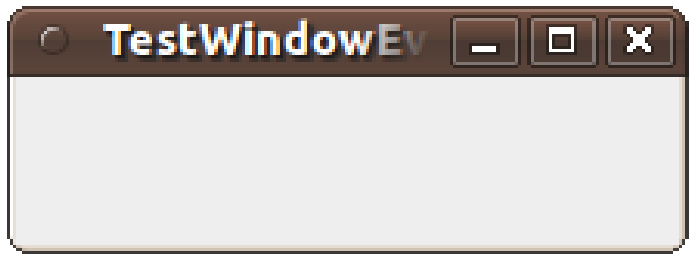


Example: Handling Window Events

Objective: Demonstrate handling the window events

- Other events can be processed similarly
- Any subclass of the `Window` class can generate the following window events: window opened, closing, closed, activated, deactivated, iconified, and deiconified
- `TestWindowEvent.java` creates a frame, listens to the window events, and displays a message to indicate the occurring event

```
{sande@icaro}[~/doc/clases/PAI/transparencias_PAI/document/T4_Events/java]$>
Window opened
Window activated
Window deactivated
Window activated
```



TestWindowEvent.java

- 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

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 `XAdapter` for `XListener`

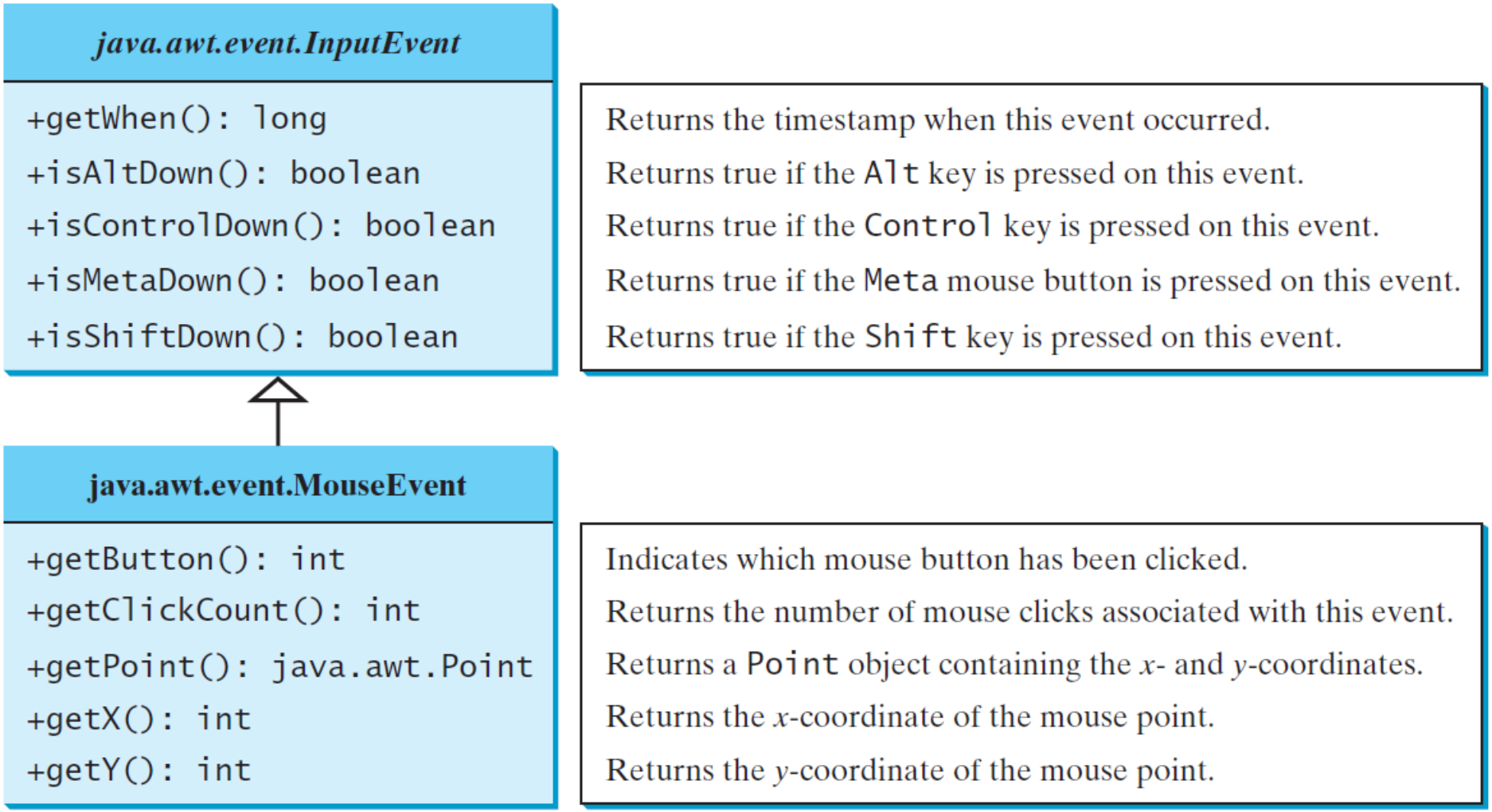
Convenience Adapters

For example, `WindowAdapter` is a convenience listener adapter for `WindowListener`

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 `AdapterDemo.java`

The `MouseEvent` class encapsulates information for mouse events



Mouse Events

- 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 coordinates
- Normally, the data fields in a class should be private, but this class has two public data fields

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

«interface» *java.awt.event.MouseListener*	
+mousePressed(e: MouseEvent): void	Invoked after the mouse button has been pressed on the source component.
+mouseReleased(e: MouseEvent): void	Invoked after the mouse button has been released on the source component.
+mouseClicked(e: MouseEvent): void	Invoked after the mouse button has been clicked (pressed and released) on the source component.
+mouseEntered(e: MouseEvent): void	Invoked after the mouse enters the source component.
+mouseExited(e: MouseEvent): void	Invoked after the mouse exits the source component.

«interface» *java.awt.event.MouseMotionListener*	
+mouseDragged(e: MouseEvent): void	Invoked after a mouse button is moved with a button pressed.
+mouseMoved(e: MouseEvent): void	Invoked after a mouse button is moved without a button pressed.

Example: Moving Message Using Mouse

MoverMensaje.java

- 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

MoverMensaje.java

- 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

MoverMensaje.java

- Because the listener is interested only in the mouse-dragged event, the anonymous innerclass 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

Handling Keyboard Events

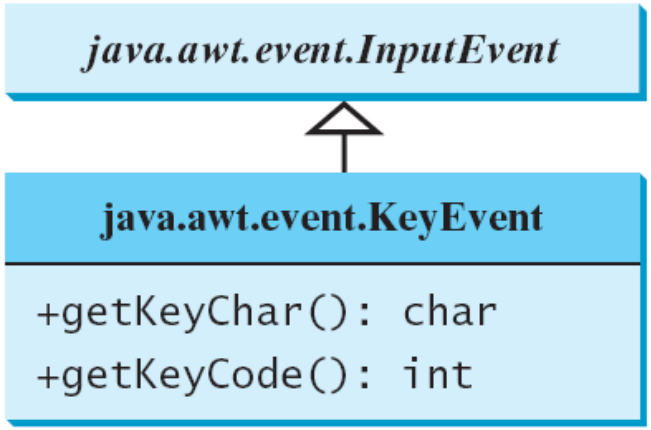
To process a keyboard event, use the following handlers in the `KeyListener` interface:

- `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



Returns the character associated with the key in this event.	
Returns the integer key code associated with the key in this event.	

The KeyEvent Class

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

Example: Keyboard Events Demo

`EventoTeclado.java`
Objective: Display a user-input character

The user can also move the character up, down, left, and right using the arrow keys

Example: Keyboard Events Demo

EventoTeclado.java

- 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 42)
- When an arrow key is pressed, the character moves in the direction indicated by the arrow key (lines 38-41)
- Every time the component is repainted, a new font is created for the `Graphics` object in line 54.
  This is not efficient.
  It is better to create the font once as a data field

Example: Keyboard Events Demo

Focus

Focus is the mechanism that determines which of the components in a window will receive keyboard input events.
A focus manager looks for special keystrokes that change the focus (usually the Tab and Shift-Tab keys), and then decides which component will next get the focus.
Often, you'll want to control how the focus moves between components, especially if you're designing a complex form or dialog for users to enter information

`EventoTeclado.java`

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

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 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.

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

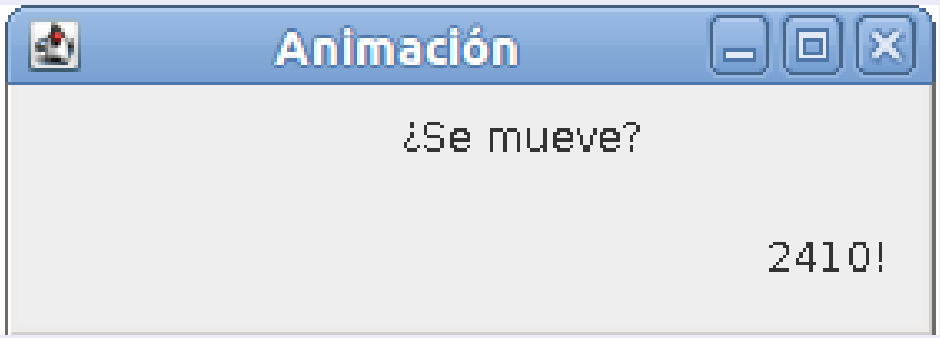
The `Timer` class

- 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

Some non-GUI components can fire events

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.



Animacion.java

- An inner class listener is defined in line 50 to listen for `ActionEvent`
- Line 33 creates a `Timer` for the listener
- The timer is started in line 36
- The timer fires an `ActionEvent` every second, and the listener responds in line 53 to repaint the panel
- When a panel is painted, its x-coordinate is increased (line 46), so the message is displayed to the right
- When the x-coordinate exceeds the bound of the panel, it is reset to -20 (line 44), so the message continues moving from left to right

Clock Animation

- In Lesson 3, you drew a Clock 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

AnimacionReloj.java

- Line 7 creates a `Timer` for a Clock Animation
- The timer is started in line 8
- The timer fires an `ActionEvent` every second, and the listener responds to set a new time (line 15) and repaint the clock (line 16)
- The setCurrentTime() method defined in `Reloj` sets the current time in the clock