320341 Programming in Java

Fall Semester 2014

Lecture 13: Event Handling in Java

Instructor: Jürgen Schönwälder

Slides: Bendick Mahleko

Objectives

JACOBS UNIVERSITY

The objective of this lecture is to

- Introduce basics of event handling in Java
- Introduce **AWT** event model

© Jacobs University Bremen

Overview

JACOBS UNIVERSITY

Events play important roles in:

- Operating systems
- Reactive systems
- Middleware
- Web services etc
- XML processing (SAX parser)

(Hansen & Fossum, 2004)

GUI development is only one source of examples for event-based programming

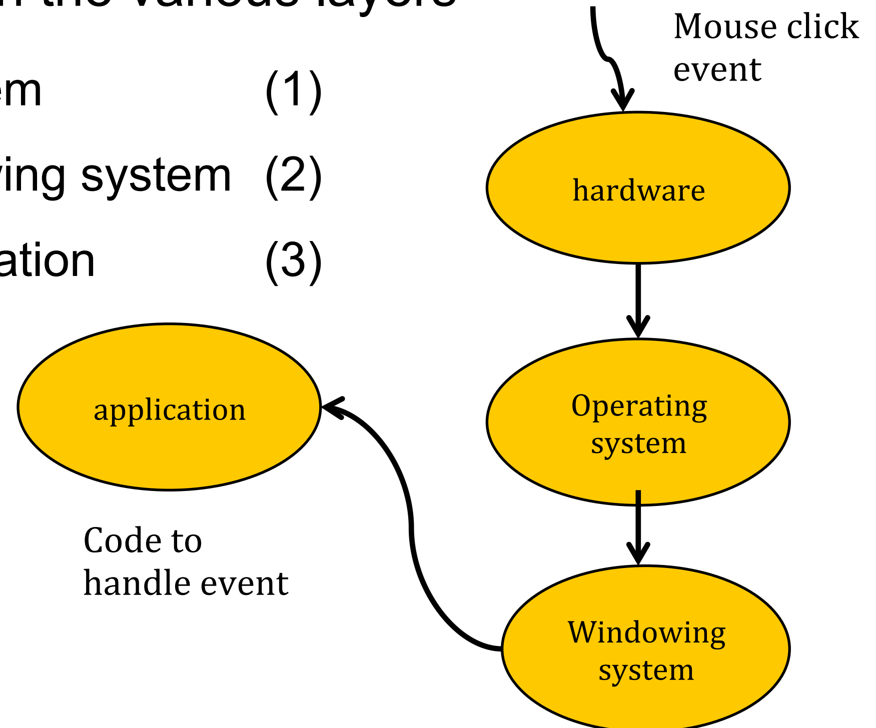
© Jacobs University Bremen

Example

A physical *mouse click* is captured and passed by the operating system (OS) via *interrupts*

The mouse click is passed through the various layers

- From hardware to operating system (1)
- From operating system to windowing system (2)
- From windowing system to application (3)

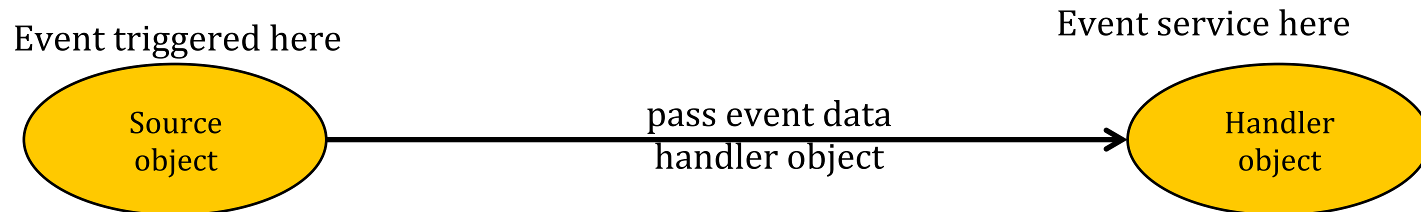


© Jacobs University Bremen

Characteristics

In object oriented systems events have a **source object** and a **handler object**

Events originate in the **source object** and are handled in the **handler object**



*An source object **fires** an event to a handler object*

*A handler object responds by taking action (a handler object **acts** on the event)*

© Jacobs University Bremen

Characteristics

An event source may be any object, including a:

- A GUI component
- Non-GUI component
- Hardware device
- Remote process

© Jacobs University Bremen

Relationship between Sources and Handlers

JACOBS UNIVERSITY

Runtime registration

- Binding of event sources and handlers is done at runtime
- The event handler objects are registered with the event sources
- Registration is done by passing a reference to the handler to a registration method in the source
- Registration is usually one of the first steps in program execution

© Jacobs University Bremen

Relationship between Sources and Handlers

JACOBS UNIVERSITY

Multicasting

- A single event may be sent to multiple handlers
- Example: An object with multiple views associated with it
- Each of these objects needs to be updated when a property is changed

© Jacobs University Bremen

Relationship between Sources and Handlers

JACOBS UNIVERSITY

Multiplexing

- Occurs when a handler receives events from multiple sources
- GUIs use multiplexing when there exists multiple ways to accomplish the same action
- Example: Choosing File → Save or clicking on a save icon

© Jacobs University Bremen

Event Handling Basics

JACOBS UNIVERSITY

Events from operating environment need to be captured and processed

Events have

- **Sources** - objects that generate events e.g., Buttons, Scrollbars, etc
- **Handlers (Listeners)** – objects that listen and respond to events from certain sources

Listener objects must **register** with source objects to listen to its events

Source objects *provide methods for event listeners to register* with them

© Jacobs University Bremen

Java Event Handling Basics in AWT

JACOBS UNIVERSITY

Event Handling in Abstract Window Toolkit (AWT)

- **Listener object:** an instance of a class that implements a **listener interface**
- **Event source:** An object that registers **listener objects** & sends them **event objects**
- The event source sends out event objects to all registered listeners when that event occurs
- The listener objects use information in the event object to react to the event

© Jacobs University Bremen

Registering Listeners

JACOBS UNIVERSITY

Registering Listener objects with source objects

```
eventSourceObject.addEventListener(eventListenerObject);
```

Event Source e.g., **Button**

registration method

Event listener to be registered

Example

```
ActionListener listener = ….;
JButton button = new JButton(“OK“);
button.addActionListener(listener);
```

- The **listener object** is notified whenever an action event occurs in the button

© Jacobs University Bremen

Listener Implementation

JACOBS UNIVERSITY

Implementing Listener Interface

- The listener object class must implement the appropriate listener interface
- The **ActionListener** interface must have a method called *actionPerformed*

```
class MyListener implements ActionListener {
    …
    public void actionPerformed(ActionEvent event) {
        // reaction to button click goes here
        …
    }
}
```

```
ActionListener listener = new MyListener() ;
JButton button = new JButton("OK");
button.addActionListener(listener);
```

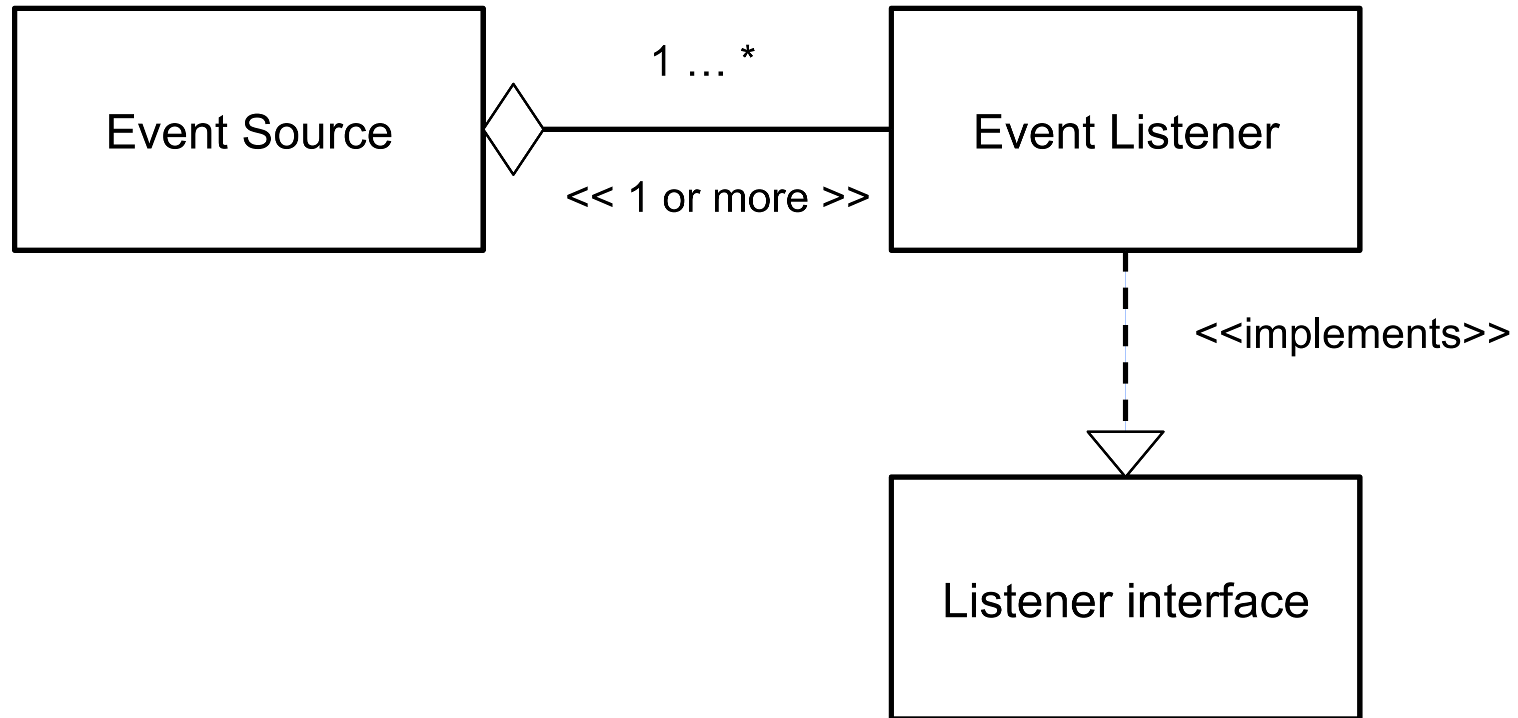
The event object is used to determine reaction to the event

Multiple listener objects can be registered with the event source

Example

JACOBS UNIVERSITY

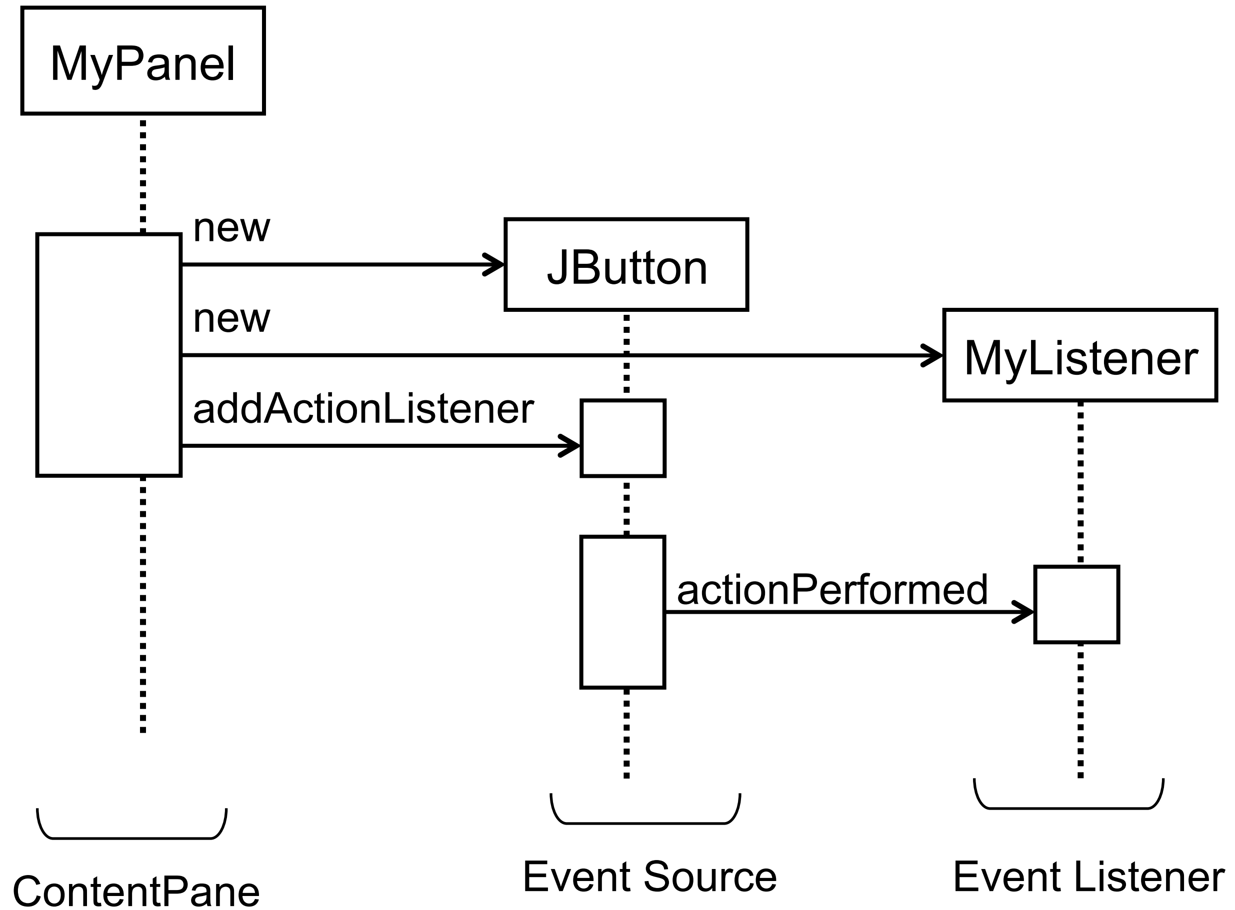
Relationship between event source and listeners



© Jacobs University Bremen

Example

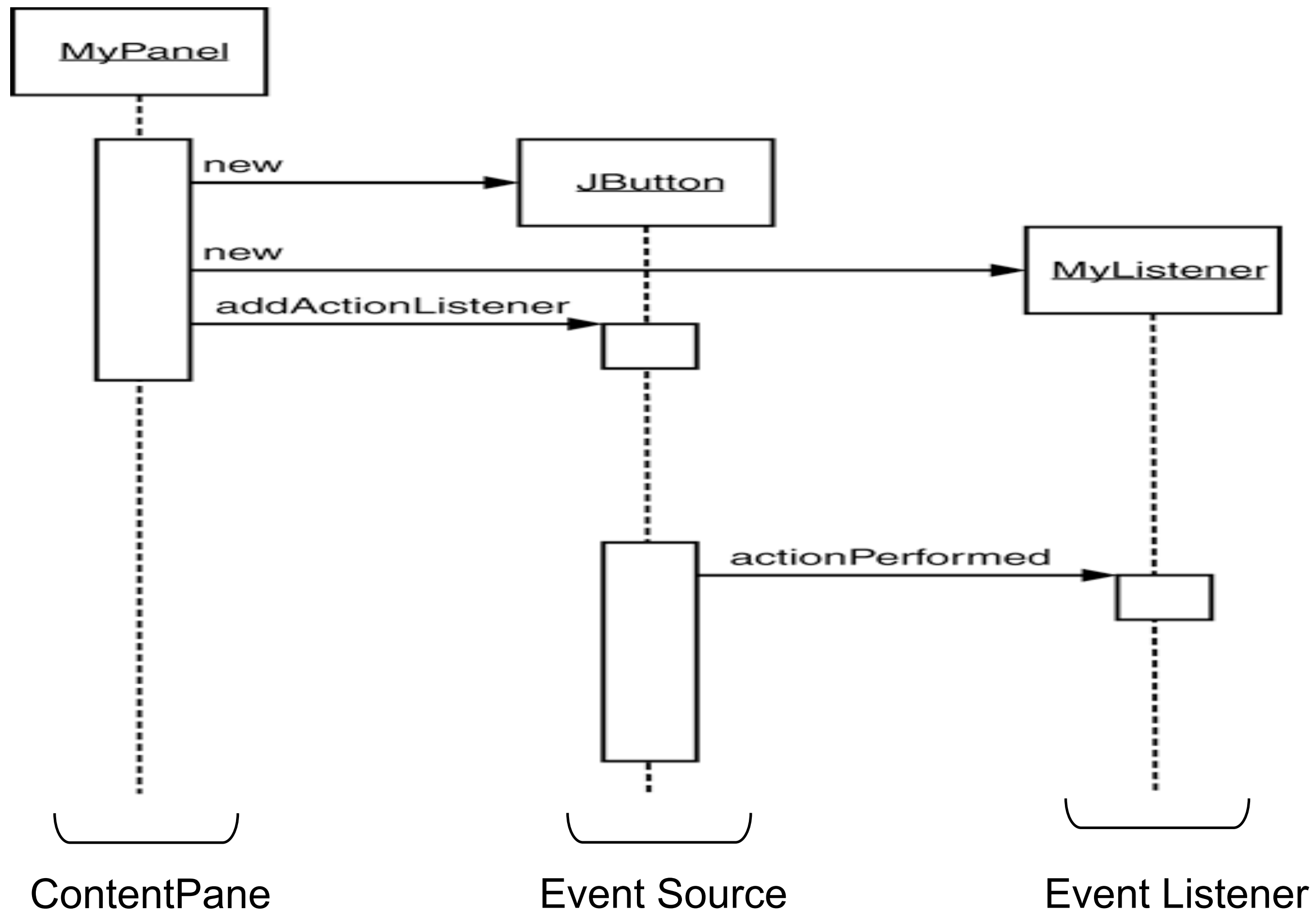
Event Notification Example



© Jacobs University Bremen

Example

Event Notification Example



© Jacobs University Bremen

Handling a Button Click

Step 1: Creating and adding buttons to a **Container**

Extend **JPanel** class

Call **JButton** constructor to create button objects

Use *add* method to add button components to **JPanel** container

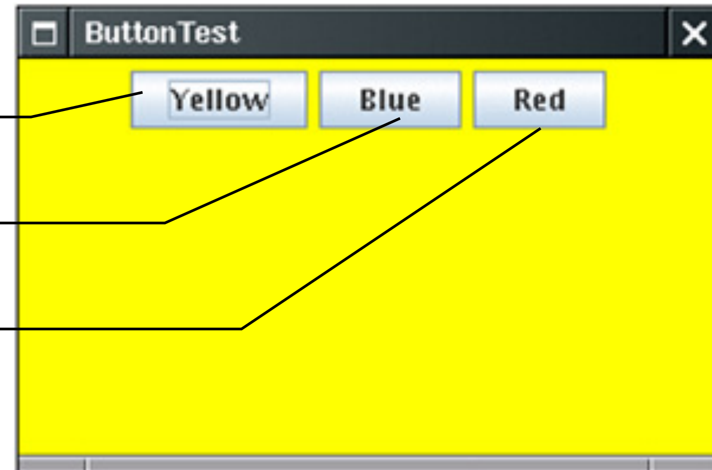
```
class ButtonPanel extends JPanel {
   public ButtonPanel() {
      JButton yellowButton = new JButton("Yellow");
      JButton blueButton    = new JButton("Blue");
      JButton redButton      = new JButton("Red");

      add(yellowButton);
      add(blueButton);
      add(redButton);
   }
}
```

Result

Need a listener for this button

Need a listener for this button

Need a listener for this button



© Jacobs University Bremen

Handling a Button Click

JACOBS UNIVERSITY

Step 2: Implement a **Listener Class**

- Desired action: when button is clicked, set the background color of the panel to the particular color

Implement a listener interface

Background color is set here

```
class ColorAction implements ActionListener {
   public ColorAction(Color c) {
      backgroundColor = c;
   }

   public void actionPerformed(ActionEvent event) {
      // set panel background color
       setBackgroundColor(backgroundColor);
   }

   private Color backgroundColor;
}
```

© Jacobs University Bremen

Handling a Button Click

Step 3: Construct an object for each color and set them as listeners

Listener objects

```
ColorAction yellowAction = new ColorAction(Color.YELLOW);
ColorAction blueAction = new ColorAction(Color.BLUE);
ColorAction redAction = new ColorAction(Color.RED);
```

Listener objects register with source objects

```
yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);
```

© Jacobs University Bremen

Handling a Button Click

Placing listener class as inner class

Listener class placed inside class whose state it modifies

```
class ButtonPanel extends JPanel  {
    . . .

    private class ColorAction implements ActionListener  {
    . . .

    public void actionPerformed(ActionEvent event)  {
       setBackground(backgroundColor);
       // i.e., outer.setBackground(...)
    }

    private Color backgroundColor;
  }
}
```

Using Inner Classes for Event Handling

JACOBS UNIVERSITY

The following steps are needed to make UI components handle events

1. Construct UI component (**JButton**)
2. Add UI component to the container (**JPanel**)
3. Construct a listener (**ActionListener**)
4. Register the action listener with the event source

© Jacobs University Bremen

Using Inner Classes for Event Handling

JACOBS UNIVERSITY

You can implement a helper method as follows:

```
void makeButton(String name, Color backgroundColor)  {
    JButton button = new JButton(name);
    add(button);
    ColorAction action = new ColorAction(backgroundColor);
    button.addActionListener(action);
}
```

© Jacobs University Bremen

Using Inner Classes for Event Handling

The **ColorAction** is needed only once (in the *makeButton*() method)

- Make **ColorAction** into an anonymous class

```
void makeButton(String name, final Color backgroundColor)  {
   JButton button = new JButton(name);
   add(button);
   button.addActionListener(new
      ActionListener()  {
         public void actionPerformed(ActionEvent event)  {
            setBackground(backgroundColor);
         }
      });
}
```

- **backgroundColor** is declared final because it's a local variable that is accessed in the inner class
- If the event handler consists of a few statements, the code is easy to read

© Jacobs University Bremen

Making Components Event Listeners

JACOBS UNIVERSITY

The event listener class must be expressively created to carry out desired button actions

Some programmers are not comfortable with inner classes

- Make the **Component** that changes due to the event to implement the listener interface

```
class ButtonPanel extends JPanel implements ActionListener {
   . . .
   public void actionPerformed(ActionEvent event)  {
      // set background color
      . . .
   }
}
```

- The panel is then a listener to all three buttons

© Jacobs University Bremen

Making Components Event Listeners

Install listeners for the three buttons as follows

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

- The three buttons no longer have individual listeners
- The *actionPerformed* method must figure out which button was clicked
  - The *getSource()* method of the **EventObject** class tells the event source

```
Object source = event.getSource();
      if (source == yellowButton) . . .
     else if (source == blueButton) . . .
     else if (source == redButton ) . . .
```

© Jacobs University Bremen

Capturing Window Events

JACOBS UNIVERSITY

When a user tries to close a frame window, the **JFrame** object is the source of a **WindowEvent**

- An appropriate **window listener** is needed to capture the window events

```
WindowListener listener = . . .;
frame.addWindowListener(listener);
```

© Jacobs University Bremen

Capturing Window Events

JACOBS UNIVERSITY

The listener class must implement the **WindowListener** interface

There are seven (7) methods in the **WindowListener** interface

```
public interface WindowListener {
   void windowOpened(WindowEvent e);
   void windowClosing(WindowEvent e);
   void windowClosed(WindowEvent e);
   void windowIconified(WindowEvent e);
   void windowDeiconified(WindowEvent e);
   void windowActivated(WindowEvent e);
   void windowDeactivated(WindowEvent e);
}
```

© Jacobs University Bremen

Capturing Windows Events

JACOBS UNIVERSITY

If we are interested in the **windowClosing** method only

- We use *do nothing functions* for the other six methods

```
class Terminator implements WindowListener  {
   public void windowClosing(WindowEvent e)  {
      System.exit(0);
   }

   public void windowOpened(WindowEvent e) {}
   public void windowClosed(WindowEvent e) {}
   public void windowIconified(WindowEvent e) {}
   public void windowDeiconified(WindowEvent e) {}
   public void windowActivated(WindowEvent e) {}
   public void windowDeactivated(WindowEvent e) {}
}
```

© Jacobs University Bremen

Capturing Windows Events

Adapter Classes

- Each AWT listener interface with more than one method comes with an Adapter class

- The Adapter class implements all the interface methods using do-nothing functions

- Extend the Adapter class and specify desired reactions to some event types in the class by **overriding** some methods

© Jacobs University Bremen

Capturing Windows Events

JACOBS UNIVERSITY

Example

- Override **windowClosing** method in **WindowAdapter** class

```
class Terminator extends WindowAdapter  {
   public void windowClosing(WindowEvent e) {
      System.exit(0);
   }
}
```

- Now install event listener

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

© Jacobs University Bremen

Capturing Windows Events

JACOBS UNIVERSITY

Make Listener class into an anonymous class

```
frame.addWindowListener(new
    WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
```

The above code does the following

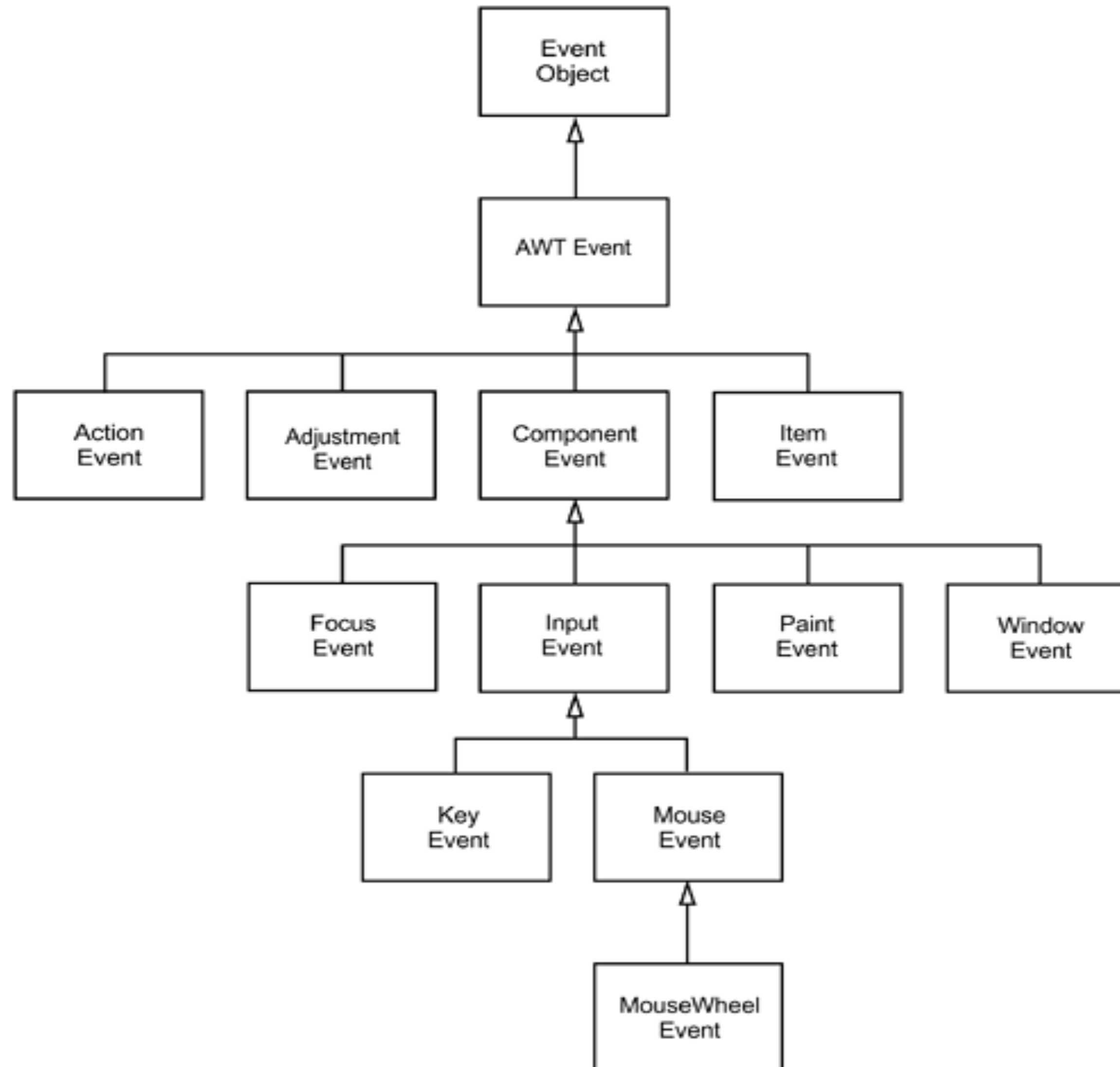
- Defines a class without a name that extends the **WindowAdapter** class
- Adds a *windowClosing* method to that anonymous class
- Inherits the remaining six do-nothing methods from **WindowAdapter**
- Creates an object of this class; that object does not have a name, either
- Passes that object to the *addWindowListener* method

© Jacobs University Bremen

AWT Event Hierarchy

JACOBS UNIVERSITY

All events in Java descend from the **java.util.EventObject** class



AWT Event Hierarchy

JACOBS UNIVERSITY

Commonly used AWT Events

ActionEvent	KeyEvent
AdjustmentEvent	MouseEvent
FocusEvent	MouseWheelEvent
ItemEvent	WindowEvent

Listener interfaces

ActionListener	MouseMotionListener
AdjustmentListener	MouseWheelListener
FocusListener	WindowListener
ItemListener	WindowFocusListener
KeyListener	WindowStateListener
MouseListener	

Commonly used adapter classes

FocusAdapter	MouseMotionAdapter
KeyAdapter	WindowAdapter
MouseAdapter	

© Jacobs University Bremen

Semantic versus Low-Level Events

Semantic Events

- Express what the user is doing
- Ex. Clicking a button, thus an **ActionEvent** is a semantic event

Low-Level Events

- Make semantic events possible
- Ex. In case of button click, a series of mouse moves e.g. mouse down event

© Jacobs University Bremen

Semantic versus Low-Level Events

JACOBS UNIVERSITY

Commonly used **Semantic Events**

- **ActionEvent** (Button click, menu selection, selecting list item, ENTER typed)
- **ItemEvent** (The user made a selection from a set of checkbox or list items)
- **AdjustmentEvent** (The user adjusted a scrollbar)

© Jacobs University Bremen

Semantic versus Low-Level Events

JACOBS UNIVERSITY

Commonly used Low-Level Events

- **KeyEvent** (A key was pressed or released)
- **MouseEvent** (Mouse button was pressed, released, moved, or dragged)
- **MouseWheelEvent** (The mouse wheel was rotated)
- **FocusEvent** (A component got focus, or lost focus)
- **WindowEvent** (The window state changed)

© Jacobs University Bremen

Semantic versus Low-Level Events

Event handling summary

- Event sources are UI components, windows and menus
- The OS notifies an event source about interesting activities
- The event source (ES) describes the nature of the event in an **event object**
- The ES also keeps a set of listeners
- The ES calls appropriate methods of the listener interface to deliver information about the event to various listeners
  - ❑ It passes an event object to a method in the listener class
- The listener analyzes the event object to find out more about the event

© Jacobs University Bremen

Reading Material

JACOBS UNIVERSITY

Core Java 2 Volume I, Chapter 8. Event Handling by Horstmann and Cornell

Hansen, S., Fossum, T. (2004) 'Events not equal to GUIs', SIGCSE '04, ACM.

© Jacobs University Bremen