

Core Java Course
Java Delegation Event Model
 (Chapter 8)
 Prof. Jeff Blessing
 MSOE

Agenda

- ❖ Homework
- ❖ Event Handling Model
- ❖ Events and Event Objects
- ❖ Swing Components & the 1.1 Event Model
- ❖ Summary
- ❖ Code examples
- ❖ Home assignment

2

Generic Event Handling Model

- ❖ The Operating System continuously monitors hardware status (all kinds of user input)
- ❖ If the OS notices any changes in the HW state (mouse click, keystroke, etc.) it puts that information into the system event queue
- ❖ A program running all the time periodically checks the queue for new events and dispatches them by calling registered functions (C) or predefined procedures (VB)

3

Pros and Cons of Event Models

- ❖ The approach adopted in C (callback functions) is very flexible but extremely complex
 - The burden is on the program to check for events!
- ❖ The VB approach (predefined procedures) is simple to use but it is restricted to the existing implementation and cannot be extended any further.
- ❖ The Java 1.1 event model is simpler than C's but much more flexible than the VB model

4

Java 1.1 Event Model: Glossary

- ❖ **Event sources** - AWT components capable of generating *event objects*, based on input activity (ie. buttons, scrollbars, listboxes, text fields, etc)
- ❖ **Event listeners** - a Java class object that has a proper interface for receiving events, knows how to process them, and is also registered with an event source(s) for a specific group of events to be sent to it

5

Java 1.1 Event Model: Details

- ❖ The AWT is the front end for "screening" events
- ❖ Event Source objects have special methods that allow Event Listener objects to *register* for notification of an event


```
ActionFrame frame = new ActionFrame();
Button b = new Button("OK");
b.addActionListener(frame);
```
- ❖ Class `ActionFrame` must have the `ActionListener` interface implemented that includes the method `void actionPerformed(ActionEvent e)`

6

Java 1.1 Event Model (cont.)

- ❖ Event object `e` carries all the useful information about the event
- ❖ The details of the event can be obtained from `e` through accessor functions, e.g.


```
e.getActionCommand();
e.getModifiers();
```
- ❖ The Event Source and Event Listener can be the same object. In this case it registers with itself and listens to itself!

7

Listener interfaces

● ActionListener	● KeyListener
● AdjustmentListener	● MouseListener
● ComponentListener	● MouseMotionListener
● ContainerListener	● TextListener
● FocusListener	● WindowListener
● ItemListener	

8

Back to the Closeable Frame Solution

- ❖ Who is the Listener?
 - (the frame object)
- ❖ Who is the Source?
 - (the frame object)
- ❖ The `closingWindow` event is delivered to
 - The registered `WindowListener` interface object.
- ❖ In order to handle closing events our frame class should implement all methods of that interface.
- ❖ Also, we have to make a call to `addWindowListener(this); //register`

9

Implementing WindowListener: Brute Force

- ❖ Go ahead and implement all seven methods in the `CloseableFrame` class itself

```
void windowClosing(WindowEvent e)
{System.exit(0)};
void windowClosed(WindowEvent e);
void windowIconified(WindowEvent e);
void windowOpened(WindowEvent e);
void windowDeiconified(WindowEvent e);
void windowActivated(WindowEvent e);
void windowDeactivated(WindowEvent e);
```

10

Implementing WindowListener: Adapter Classes

- ⊕ The previous approach looks wasteful: only one method out of 7 is actually doing something
- ⊕ The good news: Every Listener interface in Java 1.1 has a companion **adapter** class
 - `ComponentAdapter`, `ContainerAdapter`, `FocusAdapter`, `KeyAdapter`, `MouseAdapter`, `MouseMotionAdapter`, `WindowAdapter`
- ❖ Adapter classes have required methods already implemented (of course, they still do nothing)

11

Adapter Classes

- ❖ One could derive our listener class from the `WindowAdapter` class and override just one `windowClosing()` method
- ⊖ Disadvantage: we would have to deal with another class since the frame itself cannot inherit from a `WindowAdapter` derivative (multiple inheritance!)
- ⊕ The good news: Anonymous inner adapter classes can be used to provide an additional interface to the frame object

12

Implementing WindowListener

Inner Adapter Classes

```
public class CloseableFrame extends Frame
{ public CloseableFrame()
  { addWindowListener(new WindowAdapter()
  {
    public void windowClosing
      (WindowEvent e)
      { System.exit(0); }
  } );
  }
}
```

13

Implementing WindowListener:

Inner Adapter Classes (cont.)

- ❖ Inner class extends the WindowAdapter class
- ❖ Only the overwritten method needs to be added to the inner class; other six methods are just inherited from WindowAdapter and do nothing
- ❖ An internal object of that class is added without a special (named) object variable
- ❖ The new inner object is passed to the addWindowListener() method of the frame
- ❖ The resulting code is shorter and pretty straightforward, each time following the same pattern

14

Which Button Was “Clicked”?

- ❖ This example adds a Button Panel to a Frame (a kind of Window)
- ❖ The purpose of the application is to change the background color of the Frame to match the color which labels the button
- ❖ To test this out, run ButtonTest . java

15

ButtonTest.java

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

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

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

16

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

public void actionPerformed(ActionEvent evt)
{ Object source = evt.getSource();
  Color color = getBackground();
  if (source == yellowButton) color = Color.yellow;
  else if (source == blueButton) color = Color.blue;
  else if (source == redButton) color = Color.red;
  setBackground(color);
  repaint();
}

private JButton yellowButton;
private JButton blueButton;
private JButton redButton;
}
```

17

The Frame to Hold the Panel

```
class ButtonFrame extends JFrame
{ public ButtonFrame()
  { setTitle("ButtonTest");
    setSize(300, 200);
    addWindowListener(new WindowAdapter()
    { public void windowClosing(WindowEvent e)
      { System.exit(0); }
    } );

    Container contentPane = getContentPane();
    contentPane.add(new ButtonPanel());
  }
}
```

18

The "Driver" Class (ie. main)

```

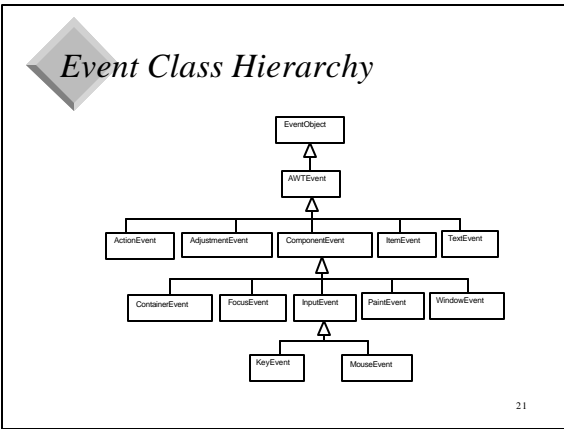
public class ButtonTest
{
    public static void main(String[] args)
    {
        JFrame frame = new ButtonFrame();
        frame.show();
    }
}
    
```

19

The EventObject Class

- ❖ Source objects have to be able to generate EventObjects and register listeners
- ❖ Class EventObject provides a string representation to describe the event, and a getSource() method which returns a reference to the source object which detected the event.
- ❖ ActionEvent is derived from EventObject

20



Semantic and Low Level events

- ❖ Semantic events are explicitly related to user input via AWT/GUI controls on the screen
 - ActionEvent, AdjustmentEvent, ItemEvent, TextEvent
- ❖ Low-level event classes are not necessarily related to particular AWT/GUI components
 - ComponentEvent, KeyEvent, MouseEvent, FocusEvent, WindowEvent, ContainerEvent

22

Individual Events

See Table 8-1 on page 334

- ❖ Focus Events
 - The focus belongs to the component that can receive keyboard input
- ❖ Window Events
- ❖ Keyboard Events
 - keyPressed(), keyReleased(), keyTyped()
- ❖ Mouse Events
 - mousePressed(), mouseReleased(), mouseEntered(), mouseExited(), mouseClicked()

23

Action Interface

- ❖ Different user interactions result in the system performing the same action
 - (ie. A button press, a menu selection, or a key pressed could all initiate an action)
- ❖ Create an Action object and associate it with a GUI object (or objects)
- ❖ This practice separates the GUI from the application code which responds to it

24

Action Interface

- ❖ Extends the ActionListener Interface
- ❖ Contains the following methods:


```
void actionPerformed(ActionEvent evt);
void setEnabled(boolean b);
boolean isEnabled();
void putValue(String key, Object value);
Object getValue(String key);
void addPropertyChangeListener(...);
void removePropertyChangeListener(...);
```

25

Action Interface

- ❖ Enabling/disabling Action objects affects whether they are “grayed out” in a menu or on a toolbar
- ❖ Name/value pairs store the state information
 - Action.NAME and Action.SMALL_ICON store a string and an icon to represent the object in a menu or on a toolbar
- ❖ *Property Change Listeners* are typically the container which holds the action object

26

AbstractAction Base Class

- ❖ The Action interface contains 7 methods!
 - That’s a lot of work to implement in code
 - ◆ especially the property change listeners!
- ❖ Fortunately Java provides an abstract class which implements all methods except the actionPerformed() method
- ❖ AbstractAction is meant to be subclassed in the application code

27

SeparateGUITest.java

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

class ColorAction extends AbstractAction
{
    public ColorAction(String name, Icon icon,
        Color c, Component comp)
    {
        putValue(Action.NAME, name);
        putValue(Action.SMALL_ICON, icon);
        putValue("Color", c);
        target = comp;
    }
}
```

28

```
public void actionPerformed(ActionEvent evt)
{
    Color c = (Color)getValue("Color");
    target.setBackground(c);
    target.repaint();
}

private Component target;
}

class ActionButton extends JButton
{
    public ActionButton(Action a)
    {
        setText(((String)a.getValue(Action.NAME));
        Icon icon = (Icon)a.getValue(Action.SMALL_ICON);
        if (icon != null)
            setIcon(icon);
        addActionListener(a);
    }
}
```

29

```
class SeparateGUIFrame extends JFrame
{
    public SeparateGUIFrame()
    {
        setTitle("SeparateGUITest");
        setSize(300, 200);
        addWindowListener(new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        });
        JPanel panel = new JPanel();
        Action blueAction = new ColorAction("Blue",
            new ImageIcon("blue-ball.gif"),
            Color.blue, panel);
        Action yellowAction = new ColorAction("Yellow",
            new ImageIcon("yellow-ball.gif"),
            Color.yellow, panel);
        Action redAction = new ColorAction("Red",
            new ImageIcon("red-ball.gif"),
            Color.red, panel);
    }
}
```

30



```


panel.add(new JButton(yellowAction));
panel.add(new JButton(blueAction));
panel.add(new JButton(redAction));

panel.registerKeyboardAction(yellowAction,
    KeyStroke.getKeyStroke(KeyEvent.VK_Y, 0),
    JComponent.WHEN_IN_FOCUSED_WINDOW);
panel.registerKeyboardAction(blueAction,
    KeyStroke.getKeyStroke(KeyEvent.VK_B, 0),
    JComponent.WHEN_IN_FOCUSED_WINDOW);
panel.registerKeyboardAction(redAction,
    KeyStroke.getKeyStroke(KeyEvent.VK_R, 0),
    JComponent.WHEN_IN_FOCUSED_WINDOW);

Container contentPane = getContentPane();
contentPane.add(panel);

```

31



```

JMenu m = new JMenu("Color");
m.add(yellowAction);
m.add(blueAction);
m.add(redAction);
JMenuBar mbar = new JMenuBar();
mbar.add(m);
setJMenuBar(mbar);
}

public class SeparateGUITest
{
    public static void main(String[] args)
    {
        JFrame frame = new SeparateGUIFrame();
        frame.show();
    }
}

```

32



Homework Assignment

- ❖ Enhance the `ButtonTest` and `PlafTest` examples by adding more components to the frame, and adding `Motif`, `Windows`, and `MetalWorks` “Look and Feel” buttons to change the “look” and “feel” of the user’s interface on the fly.

33

34