2ip90 Lecture 11

Events

Paradigm Shift

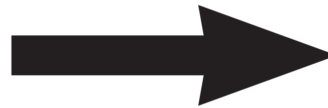
- until now: initiative with the program
  - order of actions completely determined by the program; program **acts**
- GUI: initiative with the user
  - order and choice of actions determines by user
  - consequence: program **reacts**
  - how to achieve this?

Call-back mechanism

event source

user presses button

hardware + OS + java system trigger event



JButton object

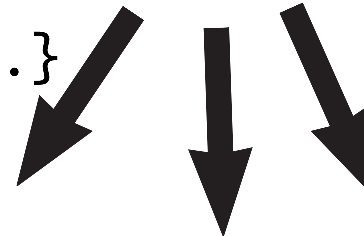
calls actionPerformed

register

listener object (ActionListener)

{...println("au!"); ...}

executes code



au!   all kinds of effects

What is needed?

1. create an event source

```
button = new JButton()...
```

2. make a class that defines the listener:

```
class XYZ implements ActionListener {
        ...
```

3. define the event method(s) in that class

```
public void actionPerformed( ActionEvent evt ) {
    if (evt...
```

4. create an object of that class

```
xyz = new XYZ(...);
```

5. register the object with the source

```
button.addActionListener(xyz);
```

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

class ClickCounter implements ActionListener {
  int nofClicks = 0;

  // this method will be called when a button is clicked; override
  public void actionPerformed(ActionEvent e) {
    nofClicks++;
    System.out.println("you have clicked "+nofClicks+" times");
  }

}
class EventDemo {
  JFrame frame;
  JPanel buttonPanel;
  JButton jbtPlus;
  ClickCounter clickCounter;

  void demo() {
    frame = new JFrame("Event Demo");
    buttonPanel = new JPanel();
    jbtPlus = new JButton("Plus");
      // add button to the panel
    buttonPanel.add(jbtPlus);

    clickCounter = new ClickCounter();
      // register the counter as listener to
      // the button
    jbtPlus.addActionListener(clickCounter);

    frame.add( buttonPanel );
    frame.setDefaultCloseOperation(
        JFrame.EXIT_ON_CLOSE);
    frame.setSize(160, 100);
    frame.setVisible(true);
  }

  public static void main(String[] args) {
          new EventDemo().demo();
    }
}
```

Example

```
// ActionEventPanel.java: simple use of ActionEvent with button
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ActionEventPanel extends JPanel implements ActionListener {
  // Create two buttons
  private JButton jbtOk = new JButton("OK");
  private JButton jbtCancel = new JButton("Cancel");

  /** Default constructor */
  public ActionEventPanel() {
    // Add buttons to the frame
    this.add(jbtOk);
    this.add(jbtCancel);

    // Register at the sources
    jbtOk.addActionListener(this);
    jbtCancel.addActionListener(this);
  }
```

Example ctd.

```
  /* This method will be invoked when a button is clicked */
  public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("OK")) {
      System.out.println("The OK button is clicked");
    }
    else if (e.getActionCommand().equals("Cancel")) {
      System.out.println("The Cancel button is clicked");
    }
  }


  /* Main method */
  public static void main(String[] args) {
    JFrame frame = new JFrame("ActionEventPanel");
    ActionEventPanel aep = new ActionEventPanel();
    frame.getContentPane().add( aep );

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(120, 100);
    frame.setVisible(true);
  }
}
```

- All GUI components are sources of events. There are many different events and associated listeners. See
  - http://java.sun.com/docs/books/tutorial/uiswing/events/api.html
  - http://java.sun.com/docs/books/tutorial/uiswing/events/eventsandcomponents.html
- Naming follows a pattern:
  A BlahBlahListener can register at a source of BlahBlahEvents
- The type of listener determines which methods can be called

ActionEvent

- There are several sources of ActionEvents:
  - JButton – when clicked
  - JComboBox – when item selected
  - JTextField – when return is pressed in field

Mouse interaction

- *MouseListener* for non-moving mouse interaction; often *mousePressed(MouseEvent m)* is the best method to use
- the click spot is recorded in the MouseEvent; use `m.getX()` and `m.getY()` to get coordinates
- *MouseMotionListener* is for moving mouse interaction; often *mouseDragged(MouseEvent m)* is used (dragging is moving the mouse while pressing a button)
- Note: MouseListener and MouseMotionListener use the same MouseEvent as the parameter type.

```
// MouseDemo - show some mouse interaction
import javax.swing.*; import java.awt.*; import java.awt.event.*;

class MouseDemo extends JPanel implements MouseListener {
  int x, y; // location of mouse
  int sx=10, sy=10; // size of shape

  MouseDemo() {
    super();
    this.addMouseListener(this); // MouseDemo is its own MouseListener!
  }

  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor( Color.blue );
    g.fillRect(x - sx/2, y - sy/2, sx, sy);
  }

  // the method from MouseListener we're interested in this time
  public void mousePressed( MouseEvent e) {
    int mouseX = e.getX();
    int mouseY = e.getY();
    x = mouseX;
    y = mouseY;
    repaint();
  }
```

```
  // the other four methods from MouseListener
  // we don't use them, but they have to appear here
  public void mouseReleased( MouseEvent e) { }
  public void mouseClicked( MouseEvent e) { }
  public void mouseEntered( MouseEvent e) { }
  public void mouseExited( MouseEvent e) { }

  public static void main(String[] args) {
    JFrame frame = new JFrame("MouseDemo");
    MouseDemo md= new MouseDemo();
    frame.getContentPane().add( md );

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize(300, 300);
    frame.setLocation(200, 200);
    frame.setVisible(true);
  }
}
```

Slider

- JSlider in javax.swing
  - JSlider(int orientation, int min, int max, int value)
  - int getValue()
  - addChangeListener(ChangeListener changeListener)
- ChangeListener
  - stateChanged(ChangeEvent e)

Miscellaneous

Interfaces, references, animation, static methods

Interfaces

- Wet dream of managers: flexibility **and** control
- flexibility: you can register at run-time for events; body of reaction method is free
- control: when registering, you have to guarantee that you can handle the events; by adding **implements ActionListener**
  - the compiler checks your claim

Code by Me

```
public class ActionEventPanel extends JPanel implements ActionListener {
  JButton jbtOk = new JButton("OK");

  public ActionEventPanel() {
    this.add(jbtOk);

    // Register at the source
    jbtOk.addActionListener(this);
  }

  public void actionPerformed(ActionEvent e) {
    ....
  }
}
```

and guaranteed by this

ActionEventPanel doesn't work here, because...

... Oracle knows nothing about it

Code by Sun

```
public class JButton extends AbstractButton {
  ActionListener[ ] listeners;
  ....
  public addActionListener(ActionListener al) {
    listeners[i++] = al;
  }

  notifyListeners() {
    ActionEvent a = new ActionEvent
    for (int j ...) {
      listeners[j].actionPerformed( a );
    }
  }
}
```

which is enforced by this

APPROXIMATION

yet, Sun needs to be sure that actionPerformed can be safely called

References

- A variable (or parameter) of class type holds a *reference* to an object; think of a reference as a phone number, an id. You can use it to call a method on the object or access the data in the object.
- Changing a reference does *not* change the data in the referred object
- A change in the data of an object will be visible by *all* reference holders
- **reference ≠ object**

Reference example

```
Garden g = new Garden(1,1);
Garden h = new Garden(5, 5);
Garden i = g;

g = h;
// (1,1) is not destroyed; i still points at it

g.length = 10;
// h.length is now 10.0 !
```

- References are very useful, though they can be confusing; draw pictures when in doubt!

Animation

- When things have to move by themselves, a *Timer* object is very useful.
- It sends *actionPerformed* events to its ActionListeners with regular intervals
- create a timer that will send every second an event to `listener` with :
```
Timer t = new Timer(1000, listener);
```
- start with:
```
t.start();
```

```java
//
//  TimerTest.java
//
// Beep every second and print a dot every tenth of a second
//
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class TimerTest implements ActionListener {
  JFrame frame = new JFrame("TimerTest");
  Timer timer = new Timer(1000, this);
  Timer timer2 = new Timer(100, this);

  void run() {
    frame.setSize(200, 200);
    frame.setVisible(true);
    timer.start();
    timer2.start();
  }

  public void actionPerformed(ActionEvent e) {
    if (e.getSource() == timer) {
      Toolkit.getDefaultToolkit().beep();
      System.out.println("beep");
      System.out.println(System.currentTimeMillis()); // print a time stamp
    } else {
      System.out.print(" . ");
    }
  }

  public static void main(String[] args) {
    new TimerTest().run();
  }
}
```

Remark

The term *event* is overloaded; it is used for:

1. something happening in the system or even the world (user presses a button, etc.)

2. a method call of a listener method, such as *actionPerformed* or *mousePressed*

3. the object that is passed as a parameter to the call of 2. containing information *about* 1 (of type, e.g., *ActionEvent* or *MouseEvent)*.

Static methods and variables

- not necessary for us right now, but not well explained in Liang
- the keyword `static` before a variable or method declares that the thing does *not* belong to an object, but to the class
- a static variable gives only one place to store data, not one for every object
- a static method can *not* use `this`
- access is via the name of the class: `Math.PI`, `Garden.main(...)`
- Goal of static variables: store data for the whole class (e.g., constants)
- Goal of static methods: use code without the need to create object

Access Modifiers

- Goal: decrease *dependency* between classes

- Means: restrict accessibility (visibility) of methods and variables

- Syntax: use a p-word (one of *public, protected*, <none>, *private*) in front of the declaration

Typical structure

- make variables private or protected
- make some methods public
- make internal helper methods private or protected
- make constants (final variables) public

- **public**: no limitations, visible in all classes
- **protected**: visible in the classes of the same package and in all subclasses
- <**none**> (“package”): visible in the classes of the same package
- **private**: visible in the same class only