
CSE 143 Java

Events, Event Handlers, and Threads
(slides not used in lecture 02au)

12/14/2002

(c) 2001, University of Washington

00 1

Overview

- Topics
 - Event-driven programming
 - Events in Java
 - Event Listeners
 - Event Adapters
 - Threads
 - Inner Classes
- Reading:
 - Textbook: Ch. 19 & 20, particularly sec. 19.4

12/14/2002

(c) 2001, University of Washington

00 2

Classic Data Processing

- Input specified as part of the program design
 - Example: process bank account deposits
 - Repeated set of transactions
 - Each transaction consists of a deposit slip (transaction header) followed by 1 or more checks to be deposited to the account
- Program expects input in required order
 - Program structure mirrors input organization

```
while (more input) {
    read and process transaction header
    read and process individual checks
}
```

12/14/2002

(c) 2001, University of Washington

00 3

Event-Driven Programming

- Idea: program initializes itself then accepts *events* in whatever random order they occur
- Kinds of events
 - Mouse move/drag/click, Keyboard, Touch screen, Joystick, game controller
 - Window resized or components changed
 - Activity over network or file stream
 - Timer interrupt
 - (can still think of this as processing an "input stream", but point of view is basically different)
- First demonstrated in the 1960s(!); major developments at Xerox PARC in the 1970s (Alto workstation, Smalltalk)
- Available outside research community with Apple Macintosh (1984)

12/14/2002

(c) 2001, University of Washington

00 4

Java Events

- An event is represented by an event object
 - AWT/Swing events are subclasses of AWTEvent. Some examples:
 - ActionEvent – button pressed
 - KeyEvent – keyboard input
 - MouseEvent – mouse move/drag/click/button press or release
- All user interface components generate events when appropriate
- Event objects contain information about the event
 - User interface object that triggered the event
 - Other information appropriate for the event. Examples:
 - ActionEvent – contents of button text generating event (if from a button)
 - MouseEvent – mouse coordinates of the event
- All in java.util.event – need to import this to handle events

12/14/2002

(c) 2001, University of Washington

00 5

Event Listeners

- Basic idea: any object that is interested in an event registers itself with the component that can generate the event
- The object must implement the appropriate Interface
 - ActionListener, KeyListener, MouseListener (buttons), MouseMotionListener (move/drag), others ...
- When the event occurs, the appropriate method of the object is called
 - actionPerformed, keyPressed, keyReleased, keyTyped, mouseClicked, mouseDragged, etc. etc. etc.
 - Reminder – because these are part of an interface, you can't change their signatures.
- An event object describing the event is a parameter to the receiving method

12/14/2002

(c) 2001, University of Washington

00 6

Example: Mouse Clicks

```
public class Mouser extends JPanel implements MouseListener {
    /** Constructor - register this object to listen for mouse events */
    Mouser() {
        super();
        addMouseListener(this);
    }

    /** Process mouse click */
    public void mouseClicked(MouseEvent e) {
        System.out.println("mouse click at x = " + e.getX() + " y = " + e.getY());
    }
}
```

• Also need to implement the other events in MouseListener

12/16/2002

(c) 2001, University of Washington

08-7

Example: Pause/Resume Buttons

- Idea: add a pair of buttons to the graphical view of the ball simulator to control the simulation
- First, rearrange the code to create an extended JFrame named BallSimControl that contains the JPanel with the bouncing balls plus the pause/resume buttons

12/16/2002

(c) 2001, University of Washington

08-8

Button/View Layout

- In the constructor for BallSimControl

```
Container cp = getContentPane();
BallGraphicsView viewPane = new BallGraphicsView()
cp.add(viewPane, BorderLayout.CENTER);
JButton pause = new JButton("pause");
JButton resume = new JButton("resume");
JPanel buttons = new JPanel();
buttons.add(pause);
buttons.add(resume);
cp.add(buttons, BorderLayout.SOUTH);
```

12/16/2002

(c) 2001, University of Washington

08-9

Handling Button Clicks

- Who should handle the pause/resume button clicks?
 - Not the SimModel object – shouldn't know about views
 - But need to catch the event and then call methods in the SimModel to carry out the pause/resume
 - One solution: create a *listener object*
- New class: SimButtonListener
- Code in BallSimControl

```
SimButtonListener listener = new SimButtonListener(simWorld);
pause.addActionListener(listener);
resume.addActionListener(listener);
```

12/16/2002

(c) 2001, University of Washington

08-10

Listener Object

```
public class SimButtonListener implements ActionListener {
    // instance variables
    SimModel world; // the SimModel we are controlling

    /** Constructor for objects of class SimButton */
    public SimButtonListener(SimModel world) {
        this.world = world;
    }

    /** Process button clicks by turning the simulation on and off */
    public void actionPerformed(ActionEvent e) {
        ???
    }
}
```

12/16/2002

(c) 2001, University of Washington

08-11

Question: Which Button was Pressed?

- Several possible answers – here's one
- Quick & dirty – get the button text from the event object

```
/** Process button clicks by turning the simulation on and off */
public void actionPerformed(ActionEvent e) {
    if (e.getActionCommand().equals("pause")) {
        world.pause();
    } else if (e.getActionCommand().equals("resume")) {
        world.resume();
    }
}
```
- Not terribly portable – what if you wanted to translate the user interface to Chinese? – but good enough for now

12/16/2002

(c) 2001, University of Washington

08-12

Event Adapter Classes

- Interfaces like `MouseListener` and `WindowListener` contain many methods; often we only are interested in one or two
- Alternative to implementing the interface and having to provide empty implementations for uninteresting methods – *adaptor classes*
- `java.awt.event` includes an abstract class with empty implementations of all required methods for each of the event listener interfaces

`KeyAdapter` (for `KeyListener`), `MouseAdapter` (for `MouseListener`), `WindowAdapter` (for `WindowListener`), etc.

- Extend and override only what you need to create a listener object

12/16/2002

(c) 2001, University of Washington

08-13

Threads and The AWT Event Thread

- Java supports "threads": apparently concurrently executing streams of instructions.
- User programs have at least one thread running
 - Not hard to create additional threads
 - Can be tricky to coordinate multiple threads
- The Java system has several threads running all the time
- One important system thread: the AWT event dispatcher
- All AWT/Swing event handlers execute in this thread
- Consequence: your event handlers may be running simultaneously with your application code

12/16/2002

(c) 2001, University of Washington

08-14

Example: Add Balls on Mouse Click

- Would like to create a listener that does something like this:

```
class BallClickListener extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        if (model != null) {
            model.add(randomBall(e.getX(), e.getY()));
        }
    }
}
```

- Listener needs to know about the model, etc.
- We really don't want another top-level class; what we'd like is a class definition nested inside `BallGraphicsView`, with access to instance variables, particularly the model object we're controlling

12/16/2002

(c) 2001, University of Washington

08-15

Towards a Solution: Inner Classes

- Java 1.1 and later allows classes to be nested
 - Inner classes define a new scope nested in the containing class
 - Inner classes can access instance variables and methods of the containing class
 - Inner classes can be public, protected, or private
- Example: `Point2D`
 - has two inner classes, named `Float` and `Double`
 - Are public, so can be used outside of class `Point2D`, as `Point2D.Float` and `Point2D.Double`
- Inner classes in event handling
 - A class like `class BallClickListener extends MouseAdapter {...}` can be a private inner class: is only needed once, and only inside the containing class

12/16/2002

(c) 2001, University of Washington

08-16

Solution: Anonymous Inner Classes

- For the mouse listener, all we need to do is create one instance of the inner class and add it as a mouse listener
 - Doesn't really need a name(!)
 - Solution: create one instance of an **anonymous** inner class
- Warning!!! Ghastly syntax ahead. Here's how to create a new object of an anonymous inner class

```
new <classname> (<constructor parameters>) { <method overrides>
}
```

12/16/2002

(c) 2001, University of Washington

08-17

Example: Constructor for Graphics View

```
public BallGraphicsView() {
    super();
    // Create inner class instance to listen for mouse clicks
    this.addMouseListener(
        new MouseAdapter() { // anon inner class extending MouseAdapter
            public void mouseClicked(MouseEvent e) {
                if (model != null) {
                    model.add(randomBall(e.getX(), e.getY()));
                } //end overridden method mouseClicked
            }
        } //end anon class extending MouseAdapter
    );
} //end method BallGraphicsView
```

12/16/2002

(c) 2001, University of Washington

08-18

Summary

- Event-driven programming
- Event objects
- Event listeners – anything that implements the relevant interface
 - Must register with object generating events as a listener
- Listener objects – handle events by passing them along to other objects
- Event adaptor classes – implementations of event interfaces with empty methods
 - Extend and override only what you want
 - Commonly used to create instances of anonymous inner classes that listen for events