CSE 143 Java

Event-Driven Programming

Reading: Chs. 17-18, particularly Sec. 17.4

Overview

- Topics
  - Event-driven programming (review)
  - Events in Java
  - Event listeners
  - Buttons
  - Mice

Event-Driven Programming (Review)

- 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
  - Sensors, lab experiments
  - Timer interrupt
- First demonstrated in the 1960s(!);
- Major developments at Xerox PARC in the 1970s (Alto workstation, Smalltalk, Xerox Star)
- Appeared outside research community in Apple Macintosh (1984)

Events in Java

- An object that is interested in an event must be *registered* with the object (user interface component or other) that generates the event
  - An object may be registered to listen for many kinds of events generated by many other objects
  - There may be many listeners registered to listen for particular kinds of events from a single object
- When an event occurs, all registered listeners are notified by calling the appropriate method in the listener objects

  (Just like the model/viewer architecture)

Event Objects

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

Event Listeners

- An event listener must implement the appropriate *interface* for the events it wishes to receive
  - ActionListener, KeyListener, MouseListener (buttons), MouseMotionListener (move/drag), others …
- When the event occurs, the appropriate method from the interface 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 supplied as a parameter to the receiving method

A First Example – Simple Button Listener

- Idea: Create a JPanel extension with a single button in it
- Create a listener object to receive clicks on the button and print a message when events happen
- Register the listener object with the button

Button Listener

- Simplest part of setup
- Need to implement ActionListener interface and actionPerformed method declared in that interface
- Doesn’t do much – just gets the action command string from the event object e and prints it

```
public class ButtonListener implements ActionListener {
  /** Respond to events generated by the button. */
  public void actionPerformed(ActionEvent e) {
    System.out.println(e.getActionCommand());
  }
}
```

Button Panel

- This panel contains the button; when constructed, it
  - creates the button and a listener
  - adds the button to the panel
  - registers the listener with the button

```
public class ButtonDemo extends JPanel {
/** Construct a new ButtonDemo object */
  public ButtonDemo() {
    JButton button = new JButton("Hit me!");
    button.setActionCommand("OUCH!");     // optional - default is button text
    button.addActionListener(new ButtonListener());
    add(button);
  }
```

Identifying the Button

- Only one button in this example, but what if the listener was registered for ActionEvents from multiple buttons?
- Answer: use method getActionCommand( ) on the event object – returns a string
  - Default value is text in the button, but can set it with setActionCommand on the button object
    - (setActionCommand is a good idea so the program won't break if button text changes later – maybe by translating to another language, but is optional for CSE143)

Second Example: Mice

- A mouse generates an event every time it twitches
  - Every move, every button press, ...
- Sometimes it makes sense to handle every mouse moved/dragged event; other times it's just noise
- Key interfaces associated with mouse events:
  - MouseListener – click, press, release, enter region, exit region
  - MouseMotionListener – mouse moved or dragged
- MouseListener and MouseMotionListener methods receive a MouseEvent parameter
  - Contents: location of the mouse event, which modifier keys were down when it happened, which buttons were pressed, etc.

Example: Mouse Clicks

```
public class Mouser extends JPanel implements MouseListener {
      /** Constructor – register this object to listen for mouse events */
      Mouser( ) {
          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

• Note that this JPanel extension registers itself to listen for the mouse events

–Could be done in other ways, e.g. have a separate listener object as we did with the button

Interactive Bouncing Balls

- Idea: add some interaction to the bouncing ball simulation/animation
- First change: add buttons in a panel at the bottom to pause and resume the simulation
- Steps
  - Create a new JPanel containing the buttons
  - Create a second JPanel BallSimControl containing the original graphics view in the middle and the button JPanel beneath
  - Add this to the top-level JFrame

Button Panel

- In BallSimControl (an extended JPanel) constructor

```
JButton pause = new JButton("pause");
JButton resume = new JButton("resume");
JButton stop = new JButton("stop");
JPanel buttons = new JPanel();
buttons.add(pause);
buttons.add(resume);
buttons.add(stop);
add(buttons, BorderLayout.SOUTH);
```

Handling Button Clicks

- Who should handle the pause/resume button clicks?
  - Not the SimModel object – it shouldn't know about views
- New class: SimButtonListener
- Code in BallSimControl

```
// set up listener for the buttons
buttonListener = new SimButtonListener(…);
pause.addActionListener(buttonListener);
resume.addActionListener(buttonListener);
stop.addActionListener(buttonListener);
```

Listener Object

```
class SimButtonListener implements ActionListener {
  private SimModel world;   // the model
  /** 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();
    } else if (e.getActionCommand().equals("stop")) {
      world.stop();
    }
  }
}
```

- Question: How does the listener know what SimModel object to notify?
- Answer: store a reference to the model in a listener instance variable

Interactive Bouncing Balls (cont.)

- Second change: when the mouse is clicked in the window, add a new bouncing ball with random size, direction, and color
- Steps
  - Create a SimMouseListener class to listen for the clicks
  - Register a listener object to listen for clicks on the view pane
- Same complications as with the buttons – the listener needs to know the model it interacts with

1/20/2005 (c) 2001-5, University of Washington 08-17

Initializing the Mouse Listener

- In BallSimControl

```
// set up listener for mouse clicks on the view
  mouseListener = new SimMouseListener(...);
  viewPane.addMouseListener(mouseListener);
```

1/20/2005 (c) 2001-5, University of Washington 08-18

Mouse Listener Object

```
/** Process mouse click by adding a new ball to the simulation at the
  * location of the click with a random color, size, and velocity   */
 public void mouseClicked(MouseEvent e) {
   world.add(randomBall(e.getX(), e.getY()));
 }

 /** Create a new ball with random color, size, and velocity   */
 public Ball randomBall(int x, int y) {
   return new Ball( ... );
 }
```

1/20/2005 (c) 2001-5, University of Washington 08-19

Summary So Far

- 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

1/20/2005 (c) 2001-5, University of Washington 08-20

Evaluation

- So far, we've implemented listeners as instances of separate stand-alone classes
- Issues
  - Relatively simple, fairly easy to understand, but
  - Somewhat messy to provide listener with access to necessary data (passing around all those references to the SimModel)
  - Creates unnecessary top-level classes
  - Also, had to implement all MouseListener methods even though we only wanted to process clicks

Coming Attractions

- Solutions
  - Event adapter classes – empty implementations of all methods in an interface; extend and override (only) what you want
  - Nested (inner) classes – which can be private
  - Anonymous inner classes – create an extended adapter class without even having to give it a name