Event driven programming

Listeners, sources, and events
GUI programming

- Asynchronous programming / event-based programming
- Events
  - Keyboard
  - Mouse
- Must respond to events in arbitrary order
- Simplistic approach: make a class to extend a window object
  - Manage layout
  - Create menus
  - Application areas
  - Manage user interaction
What is the role of an event?

What happens when we click the button?

We need two things:

- A way to know *when* something happens (an event)
- A *method* to be called to perform the appropriate task in response

We’re interested in the user-takes-action-on-a-button event!
Events have sources and handlers/listeners

... private int numClicks = 0;
...
public void processButtonClick() {
    numClicks++;
    button.setText("I’ve been clicked " + numClicks + " times. ");
} // end method processButtonClick

Hey, I care about what happens to you!
I’m listening

The user clicked me!
Here are the details on the event!

- Design/implement the “what to do” code as appropriate
- Then set up a relationship between the code and the event-generating object
An *event source* (like a Swing GUI button) creates an *event object* when the user *does something* that matters (like click the button).

Every *event type* has a matching *listener interface*
- If your code *implements* a listener interface and registers as a listener
- then you’ve set up a contract that allows communication from the event source to your code.

Recall: interfaces are 100% pure abstract classes, you *must* write implementations for the methods declared in the interface

*Register: Add me to your List of listeners!*

- **ActionListener <interface>**
  - `actionPerformed (e : ActionEvent )`

- **ItemListener <interface>**
  - `itemStateChanged (e : ItemEvent )`

- **KeyListener <interface>**
  - `keyPressed (e : KeyEvent)`
  - `keyReleased (e : KeyEvent)`
  - `keyTyped (e : KeyEvent)`

*OK. I’ll call you when I have an event!*
*I’ll use the method name in the contract!*
import javax.swing.*;
import java.awt.event.*;

public class ButtonHandler implements ActionListener {
    private JButton button;
    private int numClicks = 0;
    public void buildGUI () {
        JFrame frame = new JFrame ();
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
        JPanel contentPane = (JPanel) frame.getContentPane ();
        button = new JButton ("Click me");
        contentPane.add (button);
        button.addActionListener (this);
        frame.setSize (300,300);
        frame.setVisible (true);
    } // end method GUI

    public void actionPerformed (ActionEvent event) {
        numClicks++;
        button.setText ("I’ve been clicked " + numClicks + " times. ");
    } // end method actionPerformed
} // end class ButtonHandler
Event processing is *concurrent*

- Multiple objects can produce events
- Each object can have multiple listeners
- Code can run in an arbitrary order, or even simultaneously!
  - More on this later!

- To illustrate this we need to:
  - (1) create a GUI with multiple components (say two buttons)
  - (2) register multiple listeners
GUI Layouts: Multiple components

- GUI Layout in Java SWING is own topic
- By default, JFrame uses a “BorderLayout”
  - 5 regions, specify one with 2-arg add
  - one component to each region
- Components can be nested

```java
import java.awt.*;
import javax.swing.*;
...
button = new JButton("Click me");
contentPane.add(BorderLayout.CENTER, button);
button.addActionListener(this);

JButton exitButton = new JButton("Exit");
contentPane.add(BorderLayout.SOUTH, exitButton);
exitButton.addActionListener(this);

frame.setSize(300,300);
...```
Registering/Handling two event sources

- If you register both buttons, then how does the action handler code know which button caused the event?

```java
...  
button = new JButton("Click me");  
contentPane.add(BorderLayout.CENTER,button);  
button.addActionListener(this);  
JButton exitButton = new JButton("Exit");  
contentPane.add(BorderLayout.SOUTH,exitButton);  
exitButton.addActionListener(this);  
...

public void actionPerformed(ActionEvent event) {
    if (event.getSource() == button) {
        numClicks++;  
        button.setText("I've been clicked 
        + numClicks + " times.");
    } else {
        System.exit(0);
    }
} // end method actionPerformed
...```

The event object has all sorts of useful information!
Most OO-languages allow you to nest object code structures
  - In Java, this is often referred to as an inner class
Inner classes have access to all members (even private members) of the outer class

```java
class Outer {
    private int x = 0;
    public void go() {
        Inner inner = new Inner();
        inner.y = 3;
        inner.go();
    } // end method go

    class Inner {
        private int y = 0;
        public void go() {
            x = y;
        } // end method go
    } // end nested class Inner
} // end class Outer
```

Using inner classes can greatly simplify registering multiple listeners.

Each button can have:
  - its own (inner) class that
  - it registers as an action listener
  - handles only that button’s function
  - With full access to outer class members
Handling multiple event sources using inner classes

```java
... button = new JButton("Click me");
contentPane.add(BorderLayout.CENTER,button);
button.addActionListener(new ClickListener());

JButton exitButton = new JButton("Exit");
contentPane.add(BorderLayout.SOUTH,exitButton);
exitButton.addActionListener(new ExitListener());

class ClickListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        numClicks++;
        button.setText("I’ve been clicked " + numClicks + " times. ");
    } // end method actionPerformed
} // end inner class ClickListener

class ExitListener implements ActionListener {
    public void actionPerformed(ActionEvent event) {
        System.exit(0);
    } // end method actionPerformed
} // end inner class ExitListener
...
Examples

- Add “Speed up” and “Slow down” buttons to our simple ball animation

- Use the keyboard to move in a general frame w/o buttons

- SWING Layout
  - Add a button to each region
  - Next multiple buttons in one panel in central region
  - Etc
  - JLabel
  - JTextField
  - JCheckbox
  - JList
Example: Simple Windows-style GUI

Swing buttons
Swing Menus
Responding to button click events
Creating a simple Windows-Style GUI

We will create a simple GUI application that changes the color in the central panel on demand.

We need to know:

- How to add buttons
- How to create a menu
- How to respond to events
  - button clicks and menu selections
- How to output text to a text field

This is our goal!
Example Goal
Create a red panel

```java
import javax.swing.*;
...
// panel for displaying color
JPanel colorPanel;
JFrame frame;
JPanel contentFrame;
...
// create the color panel
colorPanel = new JPanel();
colorPanel.setBackground(Color.RED);

contentFrame.add(BorderLayout.CENTER, colorPanel);
...
frame.setVisible(true);
}
```

Just a red panel…

Time to do the layout!
Nesting JButtons in a JPanel

```java
import javax.swing.*;
...
// panel for buttons
private JPanel buttonPanel;
private JButton redButton;
private JButton greenButton;
private JButton blueButton;
...
buttonPanel = new JPanel();
redButton = new JButton("RED");
blueButton = new JButton("BLUE");
greenButton = new JButton("GREEN");
// next buttons in panel (Flow Layout!)
buttonPanel.add(redButton);
buttonPanel.add(blueButton);
buttonPanel.add(greenButton);
// add button panel to Layout
add(buttonPanel, BorderLayout.SOUTH);
...
import javax.swing.*;
...
JMenuBar menubar; // menu bar
JMenu colorMenu; // dropdown list
JMenuItem redMenuItem; // menu items
JMenuItem greenMenuItem;
JMenuItem blueMenuItem;
...
menubar = new JMenuBar();
colorMenu = new JMenu("Color");
redMenuItem = new JMenuItem("Red");
colorMenu.add(redMenuItem);
... // repeat for each color
menubar.add(colorMenu);
frame.setJMenuBar(menubar);...
import javax.swing.*;
...
JTextField colorName;
JPanel colorNamePanel;
public ColorWindow() {
...
    colorNamePanel = new JPanel();
    JLabel colorLabel = new JLabel("Current color: ");
    colorNamePanel.add(colorLabel);

    colorName = new JTextField(5);//5char wide
    colorName.setEditable(false); // read only
    colorName.setText("Red");
    colorNamePanel.add(colorName);

    // add to Layout
    add(colorNamePanel, BorderLayout.NORTH);
...
Inner class implements ActionListener

- Recall: A listener can be implemented using an *inner class*, implementing the ActionListener interface
  - An inner class is encapsulated within a larger class
  - An inner class may only be used within the larger class
  - There can be several inner classes inside a outer class
class Listener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        // find source of event and respond accordingly
        if (e.getSource() == redButton ||
            e.getSource() == redMenuItem) {
            colorPanel.setBackground(Color.RED);
            colorName.setText("Red");
        }
        // repeat for blue and green colors
    }
}

All listeners must implement the actionPerformed(ActionEvent e) method defined in the ActionListener interface.

This would work… can we do better?
Register the listener

- The listener must be registered with any buttons and menu items that use the listener.
- If desired, you can implement different listeners with different GUI components.
- When registering an ActionListener, create a new instance of the Listener class.
  - Repeat for each button and menu item

```java
import javax.swing.*;
...
// repeat for each color:

// register the menu item
redMenuItem.addActionListener(new Listener());

// register the button
redButton.addActionListener(new Listener());
...
```

An elegant way to implement listeners is to extend the JObjects and implement the interface in the extended objects!
Example 2: a Keyboard-based “game”

import java.awt.*;
import javax.swing.*;
public class GameFrame {
    public GameFrame() {
        play();
    } // end constructor
    private void pause(int millis) {
        try {
            Thread.sleep(millis);
        } catch (Exception e) {
            e.printStackTrace();
        }
    } // method pause
    ... // needs play()
} // end class GameFrame

public class Main {
    public static void main(String[] args) {
        GameFrame game = new GameFrame();
    } // end method main
} // end class Main

- The basic form of a game is pretty standard
  - Make a JFrame
  - Make a JPanel
    - move()
    - paintComponent()
  - Have an animation loop
    - force move
    - check game logic
    - force repaint
    - pause/repeat
  - Process inputs
Main loop skeleton

private void play() {
    JFrame frame = new JFrame("CS241 Game");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    GamePanel panel = new GamePanel(40,4);
    frame.getContentPane().add(BorderLayout.CENTER,panel);
    frame.addKeyListener(panel);
    frame.setSize(1024,600);
    frame.setVisible(true);
    while (true) {
        pause(5);
        panel.move();
        panel.repaint();
    }
} // end method play
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class GamePanel extends JPanel {
    int x; int y; int speed; int size;
    Image image;
    public GamePanel(int size, int speed) {
        this.size = size;
        this.speed = speed;
        image = new ImageIcon("ship.gif").getImage();
    } // end constructor
    public void paintComponent(Graphics g) {
        g.setColor(Color.BLACK);
        g.fillRect(0, 0, getWidth(), getHeight());
        g.setColor(Color.RED);
        //g.fillOval(x, y, ballSize, ballSize);
        g.drawImage(image, x, y, this);
        //System.out.println("(" + x + "," + y + ")");
    } // end method paintComponent
    // code for move()
} // end class GamePanel
Keyboard events

public class GamePanel extends JPanel implements KeyListener {
    boolean left = false;
    boolean right = false;
    boolean up = false;
    boolean down = false;

    public void move() {
        if (left) { x -= speed; }
        if (right) { x += speed; }
        if (up) { y -= speed; }
        if (down) { y += speed; }
    }

    public void keyPressed(KeyEvent e) {
        if (e.getKeyCode() == e.VK_LEFT) {left = true; }
        if (e.getKeyCode() == e.VK_RIGHT) {right = true;}
        if (e.getKeyCode() == e.VK_UP) {up = true;}
        if (e.getKeyCode() == e.VK_DOWN) {down = true;}
    }

    public void keyReleased(KeyEvent e) {
        if (e.getKeyCode() == e.VK_LEFT) {left = false; }
        if (e.getKeyCode() == e.VK_RIGHT) {right = false;}
        if (e.getKeyCode() == e.VK_UP) {up = false;}
        if (e.getKeyCode() == e.VK_DOWN) {down = false;}
    }
}

...