Programming Language Concepts: Lecture 8

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 8, 11 February 2009

GUIs and event driven programming

- How do we design graphical user interfaces?

GUIs and event driven programming

- How do we design graphical user interfaces?
- Multiple applications simultaneously displayed on screen

GUIs and event driven programming

- How do we design graphical user interfaces?
- Multiple applications simultaneously displayed on screen
- Keystrokes, mouse clicks have to be sent to appropriate window

GUIs and event driven programming

- How do we design graphical user interfaces?
- Multiple applications simultaneously displayed on screen
- Keystrokes, mouse clicks have to be sent to appropriate window
- In parallel to main activity, record and respond to these `events`
  - Web browser renders current page
  - Clicking on a link loads a different page

Keeping track of events

Low level solution

- Remember coordinates and extent of each window

Keeping track of events

Low level solution

- Remember coordinates and extent of each window
- Track coordinates of mouse

Keeping track of events

Low level solution

- Remember coordinates and extent of each window
- Track coordinates of mouse
- OS reports mouse click at (x, y)
  - Check which windows are positioned at (x, y)
  - Check if one of them is “active”
  - Inform that window about mouse click

Keeping track of events

Low level solution

- Remember coordinates and extent of each window
- Track coordinates of mouse
- OS reports mouse click at (x, y)
  - Check which windows are positioned at (x, y)
  - Check if one of them is "active"
  - Inform that window about mouse click
- Tedious and error-prone

Keeping track of events . . .

Better solution

- Programming language support for higher level events
  - Button was clicked, box was ticked . . .

Keeping track of events . . .

Better solution

- Programming language support for higher level events
  - Button was clicked, box was ticked . . .
- OS reports low level events
  - Mouse clicked at (x, y), key `'a'` pressed

Keeping track of events . . .

Better solution

- Programming language support for higher level events
  - Button was clicked, box was ticked . . .
- OS reports low level events
  - Mouse clicked at (x, y), key `'a'` pressed
- Run time support for language maps low level events to high level events

Keeping track of events . . .

Better solution . . .

- Programmer directly defines components such as windows, buttons, . . . that "generate" high level events

Keeping track of events . . .

Better solution . . .

- Programmer directly defines components such as windows, buttons, . . . that "generate" high level events
- Each event is associated with a listener that knows what to do
  - e.g., clicking `Close window` exits application

Keeping track of events . . .

Better solution . . .

- Programmer directly defines components such as windows, buttons, . . . that “generate” high level events
- Each event is associated with a listener that knows what to do
  - e.g., clicking `Close window` exits application
- Programming language has mechanisms for
  - Describing what types of events a component can generate
  - Setting up an association between components and listeners

Keeping track of events . . .

Better solution . . .

- Programmer directly defines components such as windows, buttons, . . . that "generate" high level events
- Each event is associated with a listener that knows what to do
  - e.g., clicking `Close window` exits application
- Programming language has mechanisms for
  - Describing what types of events a component can generate
  - Setting up an association between components and listeners
- Different events invoke different functions
  - Window frame has `Maximize`, `Iconify`, `Close` buttons

Keeping track of events . . .

Better solution . . .

- Programmer directly defines components such as windows, buttons, . . . that "generate" high level events
- Each event is associated with a listener that knows what to do
  - e.g., clicking `Close window` exits application
- Programming language has mechanisms for
  - Describing what types of events a component can generate
  - Setting up an association between components and listeners
- Different events invoke different functions
  - Window frame has `Maximize`, `Iconify`, `Close` buttons
- Language "sorts" out events and automatically calls the correct function in the listener

An example

- A Button with one event, the button being pressed

An example

- A Button with one event, the button being pressed
- Pressing the button invokes the function buttonpush(..) in a listener

```
interface ButtonListener{
  public abstract void buttonpush(...);
}

class MyClass implements ButtonListener{
  ...
  public void buttonpush(...){
     ...      // what to do when a button is pushed
  }
}

Button b = new Button();
MyClass m = new MyClass();
b.add_listener(m);   // Tell b to notify m when pushed
```

An example . . .

- We have set up an association between `Button b` and a listener `ButtonListener m`

An example . . .

- We have set up an association between `Button b` and a listener `ButtonListener m`
- Nothing more needs to be done!

An example . . .

- We have set up an association between `Button b` and a listener `ButtonListener m`
- Nothing more needs to be done!
- Communicating each button push to the listener is done automatically by the run-time system
- Information about the button push event is passed as an object to the listener
  - `buttonpush(...)` has arguments
  - Listener can decipher source of event, for instance

Timer

- Recall `Timer` example
- `Myclass m` creates a `Timer t` that runs in parallel
- `Timer t` notifies a `TimerOwner` when it is done via a function `notify()`
- In our example, `Myclass m` was itself the `TimerOwner` to be notified
- In principle, `Timer t` could be passed a reference to any object that implements `TimerOwner` interface

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`

- Relationship between components generating events and listeners is flexible

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible
  - One listener can listen to multiple objects
    - Three buttons on window frame all report to common listener

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible
  - One listener can listen to multiple objects
    - Three buttons on window frame all report to common listener
  - One component can inform multiple listener
    - `Exit browser` reported to all windows currently open

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible
  - One listener can listen to multiple objects
    - Three buttons on window frame all report to common listener
  - One component can inform multiple listener
    - `Exit browser` reported to all windows currently open
- Must explicitly set up association between component and listener

Event driven programming in Java

- `Swing` toolkit to define high-level components
- Built on top of lower level event handling system called `AWT`
- Relationship between components generating events and listeners is flexible
  - One listener can listen to multiple objects
    - Three buttons on window frame all report to common listener
  - One component can inform multiple listener
    - `Exit browser` reported to all windows currently open
- Must explicitly set up association between component and listener
- Events are “lost” if nobody is listening!

A detailed example in Swing

A button that paints its background red

- `JButton` is Swing class for buttons
- Corresponding listener class is `ActionListener`
- Only one type of event, button push — invokes `actionPerformed(...)` in listener
- Button push is an `ActionEvent`

A detailed example in Swing . . .

```
class MyButtons{
  private JButton b;
  public MyButtons(ActionListener a){
     b = new JButton("MyButton");  // Set the label on the bu
     b.addActionListener(a);       // Associate an listener
  }
}
```

A detailed example in Swing . . .

```
class MyButtons{
  private JButton b;
  public MyButtons(ActionListener a){
     b = new JButton("MyButton");  // Set the label on the bu
     b.addActionListener(a);       // Associate an listener
  }
}


class MyListener implements ActionListener{
  public void actionPerformed(ActionEvent evt){...}
    // What to do when a button is pressed
}

class XYZ{
  MyListener l = new MyListener(); // ActionListener l
  MyButtons m = new MyButtons(l);  // Button m, reports to l
}
```

A detailed example in Swing . . .

- To actually display the button, we have to do more
- Embed the button in a panel — `JPanel`
- Embed the panel in a frame — `JFrame`
- Display the frame!

A JPanel for our button . . .

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

public class ButtonPanel extends JPanel implements ActionListener{
  private JButton redButton;

  public ButtonPanel(){
      redButton = new JButton("Red");      // Create the button
      redButton.addActionListener(this);  // Make panel a listener
      add(redButton);                      // Embed button in panel
  }

  public void actionPerformed(ActionEvent evt){
      Color color = Color.red;             // Set background colour
      setBackground(color);                // to red when button
      repaint();                           // is clicked
  }
}
```

A JFrame for our panel . . .

- JFrame itself generates seven different types of events
- Corresponding listener class is WindowListener
    - Each of the seven events automatically calls a different function in WindowListener

A JFrame for our panel . . .

- JFrame itself generates seven different types of events
- Corresponding listener class is WindowListener
  - Each of the seven events automatically calls a different function in WindowListener
- Need to implement windowClosing event to terminate the window
- Other six types of events can be ignored

A `JFrame` for our panel . . .

- `JFrame` itself generates seven different types of events
- Corresponding listener class is `WindowListener`
  - Each of the seven events automatically calls a different function in `WindowListener`
- Need to implement `windowClosing` event to terminate the window
- Other six types of events can be ignored
- One more complication
  - `JFrame` is "complex", many layers
  - Items to be displayed have to be added to `ContentPane`

A JFrame for our panel . . .

```
public class ButtonFrame extends JFrame  implements WindowListener {
    Private Container contentPane;

    public ButtonFrame(){
        setTitle("ButtonTest");  setSize(300, 200);
        addWindowListener(this);    /// ButtonFrame listens to itself
        contentPane = this.getContentPane();  // ButtonPanel is added
        contentPane.add(new ButtonPanel());   //   to the contentPane
    }

    // Seven methods required for implementing WindowListener
    // Six out of seven are dummies (stubs)
    public void windowClosing(WindowEvent e){  // Exit when window
        System.exit(0);                        //    is killed
    }

    public void windowActivated(WindowEvent e){}
    ... // 5 more dummy methods
}
```

Finally, a main function

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

public class ButtonTest

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

Three buttons

- A panel with three buttons, to paint the panel red, yellow or blue
- Make the panel listen to all three buttons
- Determine what colour to use by identifying source of the event

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

public class ButtonPanel extends JPanel implements ActionListener{

    private JButton yellowButton;    // Panel has three buttons
    private JButton blueButton;
    private JButton redButton;

    public ButtonPanel(){
        yellowButton = new JButton("Yellow");
        blueButton = new JButton("Blue");
        redButton = new JButton("Red");

        yellowButton.addActionListener(this);  // ButtonPanel is the
        blueButton.addActionListener(this);    //   listener for all
        redButton.addActionListener(this);     //   three buttons

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

```
public class ButtonPanel extends JPanel implements ActionListener{
    ...
    public void actionPerformed(ActionEvent evt){
        Object source = evt.getSource();       // Find the source of th
                                               //   event
        Color color = getBackground();         // Get current backgroun
                                               //   colour

        if (source == yellowButton) color = Color.yellow;
        else if (source == blueButton) color = Color.blue;
        else if (source == redButton) color = Color.red;

        setBackground(color);
        repaint();
    }
}
```