

# Programming Language Concepts: Lecture 9

Madhavan Mukund

Chennai Mathematical Institute

`madhavan@cmi.ac.in`

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

PLC 2009, Lecture 9, 16 February 2009

# Event driven programming

- ▶ GUI components such as buttons, checkboxes generate high level events

# Event driven programming

- ▶ GUI components such as buttons, checkboxes generate high level events
- ▶ Each event is automatically sent to a listener
  - ▶ Listener capability is described using an interface
  - ▶ Event is sent as an object — listener can query the event to obtain details such as event source

# Event driven programming

- ▶ GUI components such as buttons, checkboxes generate high level events
- ▶ Each event is automatically sent to a listener
  - ▶ Listener capability is described using an interface
  - ▶ Event is sent as an object — listener can query the event to obtain details such as event source
- ▶ In Java, association between event generators and listeners is flexible
  - ▶ One listener can listen to multiple objects
  - ▶ One component can inform multiple listeners

# Event driven programming

- ▶ GUI components such as buttons, checkboxes generate high level events
- ▶ Each event is automatically sent to a listener
  - ▶ Listener capability is described using an interface
  - ▶ Event is sent as an object — listener can query the event to obtain details such as event source
- ▶ In Java, association between event generators and listeners is flexible
  - ▶ One listener can listen to multiple objects
  - ▶ One component can inform multiple listeners
- ▶ Must explicitly set up association between component and listener
  - ▶ Events are “lost” if nobody is listening!

## Swing example: A checkbox

- ▶ `JCheckbox`: a box that can be ticked

# Swing example: A checkbox

- ▶ `JCheckbox`: a box that can be ticked
- ▶ A panel with two checkboxes, `Red` and `Blue`
  - ▶ If only `Red` is ticked, make background red
  - ▶ If only `Blue` is ticked, make background blue
  - ▶ If both are ticked, make background green

# Swing example: A checkbox

- ▶ `JCheckbox`: a box that can be ticked
- ▶ A panel with two checkboxes, `Red` and `Blue`
  - ▶ If only `Red` is ticked, make background red
  - ▶ If only `Blue` is ticked, make background blue
  - ▶ If both are ticked, make background green
- ▶ Only one action — click the box
  - ▶ Listener is again `ActionListener`

# Swing example: A checkbox

- ▶ `JCheckbox`: a box that can be ticked
- ▶ A panel with two checkboxes, `Red` and `Blue`
  - ▶ If only `Red` is ticked, make background red
  - ▶ If only `Blue` is ticked, make background blue
  - ▶ If both are ticked, make background green
- ▶ Only one action — click the box
  - ▶ Listener is again `ActionListener`
- ▶ Checkbox has a state: ticked or not ticked
  - ▶ Method `isSelected()` to determine the current state of the checkbox

## Swing example: A checkbox

- ▶ `JCheckbox`: a box that can be ticked
- ▶ A panel with two checkboxes, `Red` and `Blue`
  - ▶ If only `Red` is ticked, make background red
  - ▶ If only `Blue` is ticked, make background blue
  - ▶ If both are ticked, make background green
- ▶ Only one action — click the box
  - ▶ Listener is again `ActionListener`
- ▶ Checkbox has a state: ticked or not ticked
  - ▶ Method `isSelected()` to determine the current state of the checkbox
- ▶ Rest is very similar to basic button example

# CheckBoxPanel

```
import ...
public class CheckBoxPanel extends JPanel implements ActionListener{
    private JCheckBox redBox;
    private JCheckBox blueBox;

    public CheckBoxPanel(){
        redBox = new JCheckBox("Red");
        blueBox = new JCheckBox("Blue");

        redBox.addActionListener(this);
        blueBox.addActionListener(this);

        redBox.setSelected(false);
        blueBox.setSelected(false);

        add(redBox);
        add(blueBox);
    }
    ...
}
```

## CheckBoxPanel ...

```
public class CheckBoxPanel extends JPanel implements ActionListener{
    ...

    public void actionPerformed(ActionEvent evt){

        Color color = getBackground();

        if (blueBox.isSelected()) color = Color.blue;
        if (redBox.isSelected()) color = Color.red;
        if (blueBox.isSelected() && redBox.isSelected()) color = Color.

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

## A JFrame for our CheckBoxPanel ...

```
public class CheckBoxFrame extends JFrame implements WindowListener{
    private Container contentPane;

    public CheckBoxFrame(){
        setTitle("ButtonTest");  setSize(300, 200);
        addWindowListener(this);
        contentPane = this.getContentPane();
        contentPane.add(new CheckBoxPanel());
    }

    public void windowClosing(WindowEvent e){ // Exit when window
        System.exit(0);                       // is killed
    }
    public void windowActivated(WindowEvent e){}
    ... // 5 more dummy methods
}
```

## Swing example: Multicasting

- ▶ Two panels, each with three buttons, **Red**, **Blue**, **Yellow**
- ▶ Clicking a button in **either** panel changes background colour in **both** panels

# Swing example: Multicasting

- ▶ Two panels, each with three buttons, Red, Blue, Yellow
- ▶ Clicking a button in either panel changes background colour in both panels
- ▶ Both panels must listen to all six buttons

# Swing example: Multicasting

- ▶ Two panels, each with three buttons, Red, Blue, Yellow
- ▶ Clicking a button in either panel changes background colour in both panels
- ▶ Both panels must listen to all six buttons
  - ▶ However, each panel has references only for its local buttons
  - ▶ How do we determine the source of an event from a remote button?

# Swing example: Multicasting

- ▶ Two panels, each with three buttons, `Red`, `Blue`, `Yellow`
- ▶ Clicking a button in `either` panel changes background colour in `both` panels
- ▶ Both panels must listen to all six buttons
  - ▶ However, each panel has references only for its local buttons
  - ▶ How do we determine the source of an event from a remote button?
- ▶ Associate an `ActionCommand` with a button
  - ▶ Assign the same action command to both `Red` buttons, ...
- ▶ Choose colour according to `ActionCommand`

# Swing example: Multicasting

- ▶ Two panels, each with three buttons, **Red**, **Blue**, **Yellow**
- ▶ Clicking a button in **either** panel changes background colour in **both** panels
- ▶ Both panels must listen to all six buttons
  - ▶ However, each panel has references only for its local buttons
  - ▶ How do we determine the source of an event from a remote button?
- ▶ Associate an **ActionCommand** with a button
  - ▶ Assign the same action command to both **Red** buttons, ...
- ▶ Choose colour according to **ActionCommand**
- ▶ Need to add both panels as listeners for each button
  - ▶ Add a public function to add a new listener to all buttons in a panel

# Multicast JPanel

```
import ...
public class ButtonPanel extends JPanel implements ActionListener{
    private JButton yellowButton;
    private JButton blueButton;
    private JButton redButton;

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

        yellowButton.setActionCommand("YELLOW");
        blueButton.setActionCommand("BLUE");
        redButton.setActionCommand("RED");

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

# Multicast JPanel

```
public class ButtonPanel extends JPanel implements ActionListener{
    ...
    public void actionPerformed(ActionEvent evt){
        Color color = getBackground();
        String cmd = evt.getActionCommand();    // Use ActionCommand to
                                                // determine what to do

        if (cmd.equals("YELLOW")) color = Color.yellow;
        else if (cmd.equals("BLUE")) color = Color.blue;
        else if (cmd.equals("RED")) color = Color.red;

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

# Multicast ButtonPanel

```
public class ButtonPanel extends JPanel implements ActionListener{
    ...

    public void addListener(ActionListener o){
        yellowButton.addActionListener(o);    // Add a common listener
        blueButton.addActionListener(o);      // for all buttons in
        redButton.addActionListener(o);       // this panel
    }
}
```

## The JFrame for the multicast example

```
public class ButtonFrame extends JFrame implements WindowListener{
    private Container contentPane;
    private ButtonPanel b1, b2;

    public ButtonFrame(){
        ...
        b1 = new ButtonPanel();    // Create two button panels
        b2 = new ButtonPanel();

        b1.addListener(b1);        // Make each panel listen
        b1.addListener(b2);        // to both sets of buttons
        b2.addListener(b1);
        b2.addListener(b2);

        contentPane = this.getContentPane();
        contentPane.setLayout(new BorderLayout()); // Set layout to
        contentPane.add(b1,"North");           // ensure that
        contentPane.add(b2,"South");           // panels don't
                                                // overlap
    } ...
}
```

# The event queue

- ▶ OS passes on low-level events to run-time support for event-driven components
- ▶ Run-time support generates high level events from low level events

# The event queue

- ▶ OS passes on low-level events to run-time support for event-driven components
- ▶ Run-time support generates high level events from low level events
- ▶ Events are stored in an **event queue**
  - ▶ Can optimize — e.g., combine consecutive mouse movements
- ▶ All events, low and high level, go into the queue

# The event queue

- ▶ OS passes on low-level events to run-time support for event-driven components
- ▶ Run-time support generates high level events from low level events
- ▶ Events are stored in an **event queue**
  - ▶ Can optimize — e.g., combine consecutive mouse movements
- ▶ All events, low and high level, go into the queue
- ▶ Application may have a need to capture low level events as well
  - ▶ May want to “capture” the mouse in an application
  - ▶ In a line drawing program, after selecting the first point, must select the target point
  - ▶ All other mouse events are captured and “consumed”

# The event queue

- ▶ OS passes on low-level events to run-time support for event-driven components
- ▶ Run-time support generates high level events from low level events
- ▶ Events are stored in an **event queue**
  - ▶ Can optimize — e.g., combine consecutive mouse movements
- ▶ All events, low and high level, go into the queue
- ▶ Application may have a need to capture low level events as well
  - ▶ May want to “capture” the mouse in an application
  - ▶ In a line drawing program, after selecting the first point, must select the target point
  - ▶ All other mouse events are captured and “consumed”
- ▶ Low level events have listener interfaces, like high level events

# Manipulating the event queue

- ▶ Normally, a Java Swing program interacts with the queue implicitly
  - ▶ Identify and associate listeners to events
  - ▶ When an event reaches the head of the event queue, it is dispatched to all listed listeners
  - ▶ If there are no listeners, the event is discarded
- ▶ Can also explicitly manipulate event queue in Java

# Manipulating the event queue

- ▶ Normally, a Java Swing program interacts with the queue implicitly
  - ▶ Identify and associate listeners to events
  - ▶ When an event reaches the head of the event queue, it is despatched to all listed listeners
  - ▶ If there are no listeners, the event is discarded
- ▶ Can also explicitly manipulate event queue in Java
- ▶ Accessing the queue

```
EventQueue evtq =  
    Toolkit.getDefaultToolkit().getSystemEventQueue();
```

# Manipulating the event queue

- ▶ Normally, a Java Swing program interacts with the queue implicitly
  - ▶ Identify and associate listeners to events
  - ▶ When an event reaches the head of the event queue, it is despatched to all listed listeners
  - ▶ If there are no listeners, the event is discarded
- ▶ Can also explicitly manipulate event queue in Java
- ▶ Accessing the queue

```
EventQueue evtq =  
    Toolkit.getDefaultToolkit().getSystemEventQueue();
```

- ▶ Consuming an event from the queue

```
AWTEvent evt = eq.getNextEvent();
```

# Manipulating the event queue

- ▶ Normally, a Java Swing program interacts with the queue implicitly
  - ▶ Identify and associate listeners to events
  - ▶ When an event reaches the head of the event queue, it is dispatched to all listed listeners
  - ▶ If there are no listeners, the event is discarded
- ▶ Can also explicitly manipulate event queue in Java
- ▶ Accessing the queue

```
EventQueue evtq =  
    Toolkit.getDefaultToolkit().getSystemEventQueue();
```

- ▶ Consuming an event from the queue

```
AWTEvent evt = eq.getNextEvent();
```

- ▶ Adding an event to the queue

```
evtq.postEvent(new ActionEvent(.....));
```

# Adding customized events

- ▶ Recall the timer example
- ▶ Suppose we want `Timer` to generate an event when its countdown expires

# Adding customized events

- ▶ Recall the timer example
- ▶ Suppose we want `Timer` to generate an event when its countdown expires
- ▶ Java events extend class `AWTEvent`
- ▶ Each event is identified by a unique numeric id

# Adding customized events

- ▶ Recall the timer example
- ▶ Suppose we want `Timer` to generate an event when its countdown expires
- ▶ Java events extend class `AWTEvent`
- ▶ Each event is identified by a unique numeric id

```
class TimerEvent extends AWTEvent{  
  
    public static final int  
        TIMER_EVENT = AWTEvent.RESERVED_ID_MAX + 5555;  
  
    public TimerEvent(Timer t) {  
        super(t, TIMER_EVENT);  
    }  
}
```

- ▶ Constructor sets source of event and its type (numeric id)

## Adding customized events . . .

- ▶ An interface defining a listener for `TimerEvents`

```
interface TimerListener extends EventListener{  
    public void timeElapsed(TimerEvent evt);  
}
```

## Adding customized events ...

- ▶ An interface defining a listener for `TimerEvents`

```
interface TimerListener extends EventListener{  
    public void timeElapsed(TimerEvent evt);  
}
```

- ▶ ...and a class implementing the interface

```
class MyClass implements TimerListener{  
    ...  
    public void timeElapsed(TimerEvent evt){  
        ...  
    }  
}
```

## Adding customized events ...

- ▶ Generate `TimerEvents` by manipulating event queue

```
class Timer extends Component implements Runnable{
    // Runnable is to ensure that Timer can run concurrently
    ...
    private static EventQueue evtq =
        Toolkit.getDefaultToolkit().getSystemEventQueue();
    // There is only one copy of the EventQueue, shared by all T

    public Timer(){
        ...
    }

    public void f(){
        ...
        // Generate a TimerEvent when done
        TimerEvent te = new TimerEvent(this);
        evtq.postEvent(te);
    }
}
```

## Adding customized events ...

- ▶ Each `Timer` must keep track of its listeners
  - ▶ Recall `addListener` for `JButton` ...

```
class Timer extends Component implements Runnable{
    ...
    private TimerListener listener;

    public addListener(TimerListener t){
        listener = t;
    }
    ...
}
```

## Adding customized events . . .

- ▶ Typical code that uses the timer

```
...  
Timer t = new Timer();  
MyClass m = new MyClass(..);  
t.addListener(m);  
t.start();  
...
```

- ▶ After `t.start()` completes, how does it know what to do to notify `m`?

## Adding customized events . . .

- ▶ It does not know! We have to tell it what to do

## Adding customized events . . .

- ▶ It does not know! We have to tell it what to do
- ▶ AWT promises that a fixed function `processEvent` will be called within an object whenever it generates an `AWTEvent`

## Adding customized events . . .

- ▶ It does not know! We have to tell it what to do
- ▶ AWT promises that a fixed function `processEvent` will be called within an object whenever it generates an `AWTEvent`
- ▶ Add the following to `Timer`

```
class Timer extends Component implements Runnable{
    ...

    public void processEvent(AWTEvent evt)
    { if (evt instanceof TimerEvent)
      { if (listener != null)
          listener.timeElapsed((TimerEvent)evt);
        }
      else super.processEvent(evt);
    }
    ...
}
```

# Adding customized events . . .

## Putting it all together

- ▶ When a `Timer t` generates a `TimerEvent`, AWT calls the (fixed) method `processEvent` within `t`
- ▶ In `Timer`, we write explicit in `processEvent` to call any listener associated with `t`
  - ▶ To support multicasting, allow `Timer` to store an array of listeners and notify each in turn.