OROMIA STATE UNIVERSITY
COLLEGE OF SCIENCE AND TECHNOLOGY
DEPARTEMENTOFINFORMATI ON TECHNOLOGY

Network Programming
Chapter-6
User Interface Design and The Event Model

Dear students would you remember GUI in your object-oriented programming course?

In this Lecture, you will learn about Events, its types, and also learn how to handle an event.

What is an Event?

Change in the state of an object is known as Event, i.e., event describes the change in the state of the source. Events are generated as a result of user interaction with the graphical user interface components. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from the list, and scrolling the page are the activities that causes an event to occur.

Types of Event

The events can be broadly classified into two categories −

Foreground Events − These events require direct interaction of the user. They are generated as consequences of a person interacting with the graphical components in the Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page, etc.

Background Events − These events require the interaction of the end user. Operating system interrupts, hardware or software failure, timer expiration, and operation completion are some examples of background events.

What is Event Handling?

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism has a code which is known as an event handler, that is executed when an event occurs.

Java uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

The Delegation Event Model has the following key participants.

Source − The source is an object on which the event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide us with classes for the source object.

Listener − It is also known as event handler. The listener is responsible for generating a response to an event. From the point of view of Java implementation, the listener is also an object. The listener waits till it receives an event. Once the event is received, the listener processes the event and then returns.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to a separate piece of code.

In this model, the listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listeners who want to receive them.

Steps Involved in Event Handling

Step 1 − The user clicks the button and the event is generated.

Step 2 − The object of concerned event class is created automatically and information about the source and the event get populated within the same object.

Step 3 − Event object is forwarded to the method of the registered listener class.

Step 4 − The method is gets executed and returns.

Points to Remember About the Listener

In order to design a listener class, you have to develop some listener interfaces. These Listener interfaces forecast some public abstract callback methods, which must be implemented by the listener class.

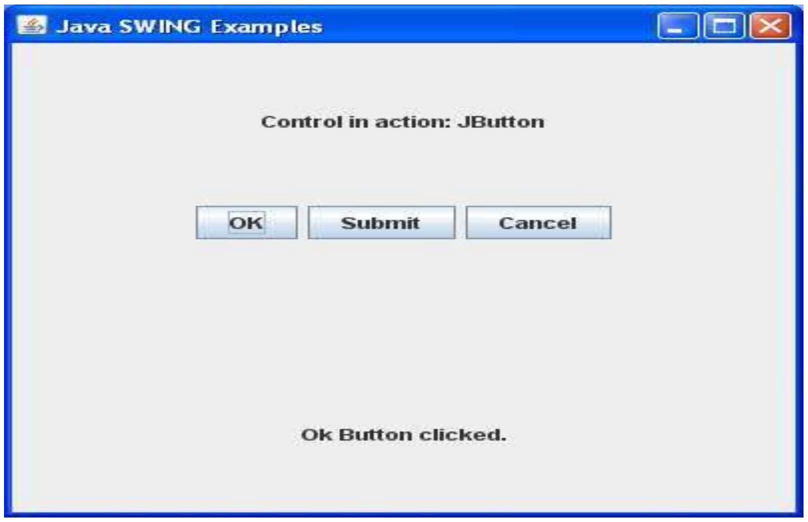
If you do not implement any of the predefined interfaces, then your class cannot act as a listener class for a source object.

Callback Methods

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represent an event method. In response to an event, java jre will fire callback method. All such callback methods are provided in listener interfaces.

If a component wants some listener to listen ot its events, the source must register itself to the listener.

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

public class SwingControlDemo {
   private JFrame mainFrame;
   private JLabel headerLabel;
   private JLabel statusLabel;
   private JPanel controlPanel;

   public SwingControlDemo(){
      prepareGUI();
   }
   public static void main(String[] args){
      SwingControlDemo swingControlDemo = new
SwingControlDemo();
      swingControlDemo.showEventDemo();
   }
   private void prepareGUI(){
      mainFrame = new JFrame("Java SWING Examples");
      mainFrame.setSize(400,400);
      mainFrame.setLayout(new GridLayout(3, 1));

      headerLabel = new JLabel("",JLabel.CENTER );
      statusLabel = new JLabel("",JLabel.CENTER);
      statusLabel.setSize(350,100);

      mainFrame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent windowEvent){
            System.exit(0);
         }
      });
      controlPanel = new JPanel();
      controlPanel.setLayout(new FlowLayout());

      mainFrame.add(headerLabel);
      mainFrame.add(controlPanel);
      mainFrame.add(statusLabel);
      mainFrame.setVisible(true);
   }
   private void showEventDemo(){
      headerLabel.setText("Control in action: Button");

      JButton okButton = new JButton("OK");
      JButton submitButton = new JButton("Submit");
      JButton cancelButton = new JButton("Cancel");

      okButton.setActionCommand("OK");
      submitButton.setActionCommand("Submit");
      cancelButton.setActionCommand("Cancel");

      okButton.addActionListener(new ButtonClickListener());
      submitButton.addActionListener(new
ButtonClickListener());
      cancelButton.addActionListener(new
ButtonClickListener());

      controlPanel.add(okButton);
      controlPanel.add(submitButton);
      controlPanel.add(cancelButton);

      mainFrame.setVisible(true);
   }
   private class ButtonClickListener implements
ActionListener{
      public void actionPerformed(ActionEvent e) {
         String command = e.getActionCommand();

         if( command.equals( "OK" ))  {
            statusLabel.setText("Ok Button clicked.");
         } else if( command.equals( "Submit" ) )  {
            statusLabel.setText("Submit Button clicked.");
         } else {
            statusLabel.setText("Cancel Button clicked.");
         }
      }
   }
}
```



Control in action: JButton

Ok Button clicked.

LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers. There are following classes that represents the layout managers:

1. java.awt.BorderLayout
2. java.awt.FlowLayout
3. java.awt.GridLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout
7. javax.swing.GroupLayout
8. javax.swing.ScrollPaneLayout
9. javax.swing.SpringLayout etc.

Java BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west and center. Each region (area) may contain one component only. It is the default layout of frame or window. The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **JBorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Java Swing | Look and Feel

Swing is **GUI Widget Toolkit** for Java. It is an API for providing Graphical User Interface to Java Programs. Unlike AWT, Swing components are written in Java and therefore are platform-independent. Swing provides platform specific Look and Feel and also an option for pluggable Look and Feel, allowing application to have Look and Feel independent of underlying platform.Initially there were very few options for colors and other settings in Java Swing, that made the entire application look boring and monotonous. With the growth in Java framework, new changes were introduced to make the UI better and thus giving developer opportunity to enhance the look of a Java Swing Application.

"Look" refers to the appearance of GUI widgets and "feel" refers to the way the widgets behave.
Sun's JRE provides the following L&Fs:

1. **CrossPlatformLookAndFeel:** this is the "Java L&F" also known as "Metal" that looks the same on all platforms. It is part of the Java API (javax.swing.plaf.metal) and is the default.
2. **SystemLookAndFeel:** here, the application uses the L&F that is default to the system it is running on. The System L&F is determined at runtime, where the application asks the system to return the name of the appropriate L&F.For Linux and Solaris, the System L&Fs are "GTK+" if GTK+ 2.2 or later is installed, "Motif" otherwise. For Windows, the System L&F is "Windows".
3. **Synth:** the basis for creating your own look and feel with an XML file.
4. **Multiplexing:** a way to have the UI methods delegate to a number of different look and feel implementations at the same time.

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event.

For example, click on button, dragging mouse etc.

The java.awt.event package provides many event classes and Listener interfaces for event handling.

Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Register the component with the Listener

Registration Methods

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

Java Event Handling Code

We can put the event handling code into one of the following places:

1. Within class
2. Other class
3. Anonymous class

Java event handling by implementing ActionListener

```java
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){

//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);

//register listener
b.addActionListener(this);//passing current instance

//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
}
}
```

Java event handling by outer class

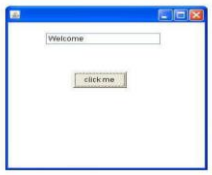
public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.

```java
import java.awt.*;
import java.awt.event.*;
class AEvent2 extends Frame{
TextField tf;
AEvent2(){
//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
//register listener
Outer o=new Outer(this);
b.addActionListener(o);//passing outer class instance
//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public static void main(String args[]){
new AEvent2();
}
}
```

```java
import java.awt.event.*;
class Outer implements ActionListener{
AEvent2 obj;
Outer(AEvent2 obj){
this.obj=obj;
}
public void actionPerformed(ActionEvent e){
obj.tf.setText("welcome");
}
}
```

3) Java event handling by anonymous class

```java
import java.awt.*;
import java.awt.event.*;
class AEvent3 extends Frame{
TextField tf;
AEvent3(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(50,120,80,30);

b.addActionListener(new ActionListener(){
public void actionPerformed(){
tf.setText("hello");
}
});
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public static void main(String args[]){
new AEvent3();
}
}
```


Welcome
click me