Event Handling in Java

An **event** can be defined as changing the state of an object or behavior by performing actions. Actions can be a button click, cursor movement, keypress through keyboard or page scrolling, etc.
The **java.awt.event** package can be used to provide various event classes.

Classification of Events

- Foreground Events
- Background Events

1. Foreground Events

Foreground events are the events that require user interaction to generate, i.e., foreground events are generated due to interaction by the user on components in Graphic User Interface (**GUI**). Interactions are nothing but clicking on a button, scrolling the scroll bar, cursor moments, etc.

2. Background Events

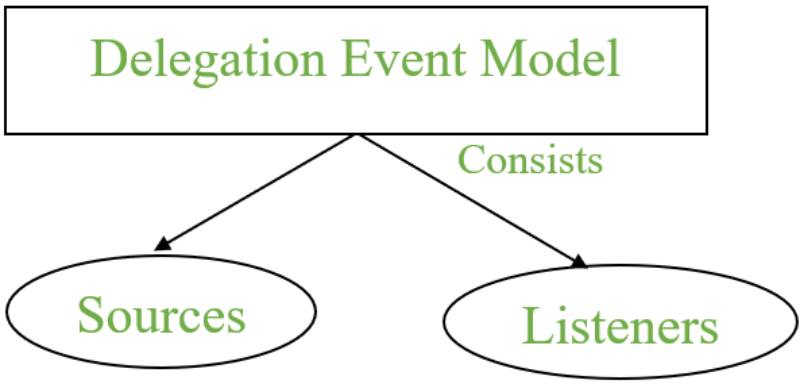
Events that don't require interactions of users to generate are known as background events. Examples of these events are operating system failures/interrupts, operation completion, etc.

Event Handling

It is a mechanism to **control the events** and to **decide what should happen after an event** occur. To handle the events, Java follows the ***Delegation Event model.***

Delegation Event model

- It has Sources and Listeners.



Delegation Event Model

- **Source:** Events are generated from the source. There are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows, etc., to generate events.
- **Listeners:** Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

To perform Event Handling, we need to register the source with the listener.

Registering the Source With Listener

Different Classes provide different registration methods.

Syntax:

```
addTypeListener()
```

where Type represents the type of event.

Example 1: For **KeyEvent** we use *addKeyListener()* to register.
Example 2:that For **ActionEvent** we use *addActionListener()* to register.

Event Classes in Java

Event Class	Listener Interface	Description

ActionEvent	ActionListener	An event that indicates that a component-defined action occurred like a button click or selecting an item from the menu-item list.
AdjustmentEvent	AdjustmentListener	The adjustment event is emitted by an Adjustable object like Scrollbar.
ComponentEvent	ComponentListener	An event that indicates that a component moved, the size changed or changed its visibility.
ContainerEvent	ContainerListener	When a component is added to a container (or) removed from it, then this event is generated by a container object.
FocusEvent	FocusListener	These are focus-related events, which include focus, focusin, focusout, and blur.
ItemEvent	ItemListener	An event that indicates whether an item was selected or not.
KeyEvent	KeyListener	An event that occurs due to a sequence of keypresses on the keyboard.
MouseEvent	MouseListener & MouseMotionListener	The events that occur due to the user interaction with the mouse (Pointing Device).
MouseWheelEvent	MouseWheelListener	An event that specifies that the mouse wheel was rotated in a component.
TextEvent	TextListener	An event that occurs when an object's text changes.
WindowEvent	WindowListener	An event which indicates whether a window has changed its status or not.

Note: *As Interfaces contains abstract methods which need to implemented by the registered class to handle events.*

Different interfaces consists of different methods which are specified below.

Listener Interface	Methods	Listener Interface	Methods
ActionListener	• actionPerformed()	MouseListener	• mousePressed() • mouseClicked() • mouseEntered() • mouseExited() • mouseReleased()
AdjustmentListener	• adjustmentValueChanged()	MouseMotionListener	• mouseMoved() • mouseDragged()
ComponentListener	• componentResized() • componentShown() • componentMoved() • componentHidden()	MouseWheelListener	• mouseWheelMoved()
ContainerListener	• componentAdded() • componentRemoved()	TextListener	• textChanged()
FocusListener	• focusGained() • focusLost()	WindowListener	• windowActivated() • windowDeactivated() • windowOpened() • windowClosed() • windowClosing() • windowIconified() • windowDeiconified()
ItemListener	• itemStateChanged()	KeyListener	• keyTyped() • keyPressed() • keyReleased()

Flow of Event Handling

1. User Interaction with a component is required to generate an event.
2. The object of the respective event class is created automatically after event generation, and it holds all information of the event source.
3. The newly created object is passed to the methods of the registered listener.
4. The method executes and returns the result.

Code-Approaches

The three approaches for performing event handling are by placing the event handling code in one of the below-specified places.

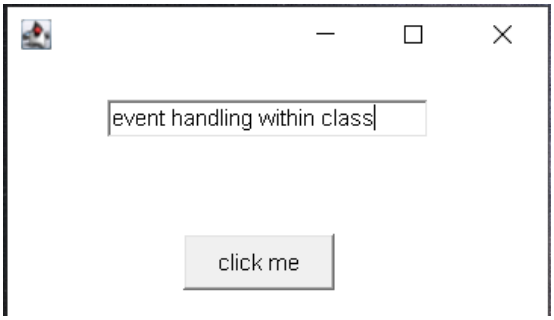
1. Within Class
2. Other Class
3. Anonymous Class

Note: *Use any IDE or install JDK to run the code, Online compiler may throw errors due to the unavailability of some packages.*

Event Handling Within Class

```
// Java program to demonstrate the
// event handling within the class
package w02ex01;
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
	TextField textField;
	AEvent(){
		//create components
		textField =new TextField();
		textField.setBounds(60,50,170,20);
		Button button=new Button("click me");
		button.setBounds(100,120,80,30);
		// Registering component with listener
		// this refers to current instance
		button.addActionListener(this);
		//add components and set size, layout and visibility
		add(button);
		add(textField);
		setSize(300,300);
		setLayout(null);
		setVisible(true);
	}

	public void actionPerformed(ActionEvent e){
		textField.setText("event handling within class");
	}
	public static void main(String args[]){
		new AEvent();
	}
}
```

Output



After Clicking, the text field value is set to GFG!

Explanation

1. Firstly extend the class with the applet and implement the respective listener.
2. Create Text-Field and Button components.

3. Registered the button component with respective event. i.e. ActionEvent by addActionListener().
4. In the end, implement the abstract method.

Event Handling by Other Class

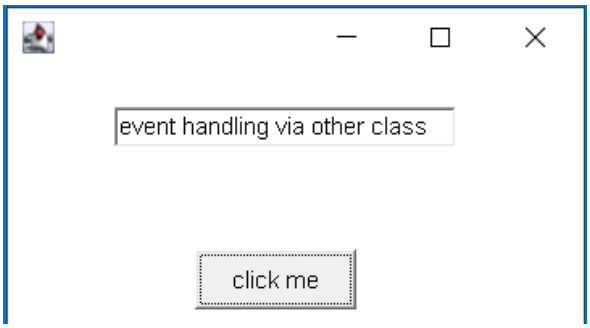
AEvent2.java

```
// Java program to demonstrate the
// event handling by the other class
package w02ex01;
import java.awt.*;
import java.awt.event.*;
class AEvent2 extends Frame{
	TextField textF;
	AEvent2(){
		//create components
		textF=new TextField();
		textF.setBounds(60,50,170,20);
		Button butt=new Button("click me");
		butt.setBounds(100,120,80,30);
		// Passing other class as reference
		Outer o=new Outer(this);
		// Registering component with listener
		butt.addActionListener(o);//passing outer class instance
		//add components and set size, layout and visibility
		add(butt);add(textF);
		setSize(300,300);
		setLayout(null);
		setVisible(true);
	}
	public static void main(String args[]){
		new AEvent2();
	}
}
```

Outer.java

```
 package w02ex01;
import java.awt.event.*;
class Outer implements ActionListener{
	AEvent2 obj;
	Outer(AEvent2 obj){
		this.obj=obj;
	}
	public void actionPerformed(ActionEvent e){
		obj.textF.setText("event handling via other class");
	}
}
```

Output

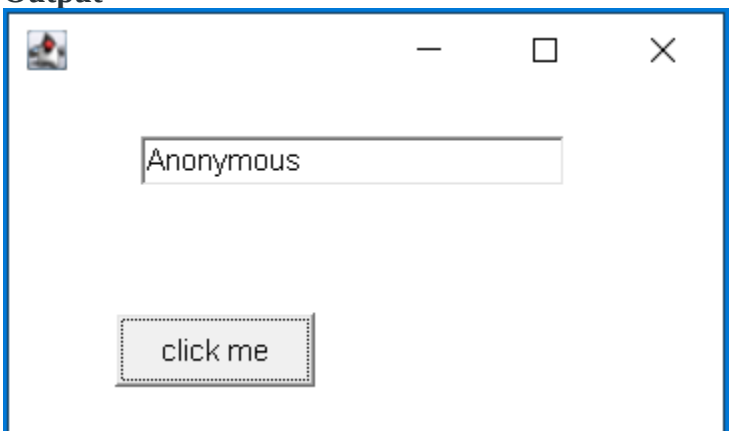


Handling event from different class

Event Handling By Anonymous Class

```
 // Java program to demonstrate the
// event handling by the anonymous class
package w02ex01;

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);

             // Registering component with listener anonymously
         b.addActionListener(new ActionListener() {
             public void actionPerformed(ActionEvent e)
             {
                 // Setting text to field
             tf.setText("Anonymous");
             }
         });

             add(b);add(tf);
             setSize(300,300);
             setLayout(null);
             setVisible(true);
       }

       public static void main(String args[]){
             new AEvent3();
       }
}
```

Output



AWT Event Handling

Points to remember about listener

- In order to design a listener class we 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 the any if the predefined interfaces then your class can not 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 represents 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 will listen to it's events the the source must register itself to the listener.

```
import java.awt.*;
import java.awt.event.*;
public class AwtControlDemo {

   private Frame mainFrame;
   private Label headerLabel;
   private Label statusLabel;
   private Panel controlPanel;

   public AwtControlDemo(){
      prepareGUI();
   }

   public static void main(String[] args){
      AwtControlDemo  awtControlDemo = new AwtControlDemo();
      awtControlDemo.showEventDemo();
   }

   private void prepareGUI(){
      mainFrame = new Frame("Java AWT Examples");
      mainFrame.setSize(400,400);
      mainFrame.setLayout(new GridLayout(3, 1));
      mainFrame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent windowEvent){
            System.exit(0);
         }
      });
      headerLabel = new Label();
      headerLabel.setAlignment(Label.CENTER);
      statusLabel = new Label();
      statusLabel.setAlignment(Label.CENTER);
      statusLabel.setSize(350,100);

      controlPanel = new Panel();
```

```
      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");

      Button okButton = new Button("OK");
      Button submitButton = new Button("Submit");
      Button cancelButton = new Button("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.");
         }
      }
   }
}
```

Output: