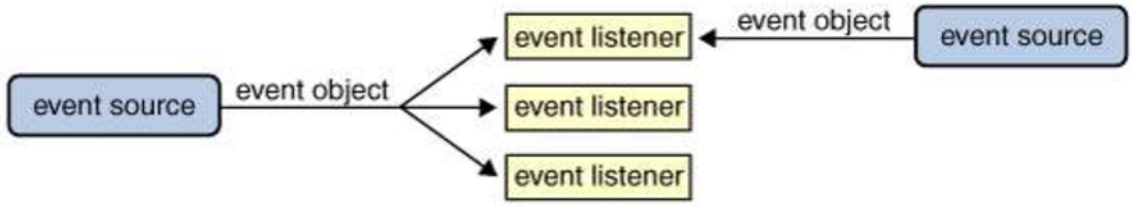
Chapter-2.3
User interface components with AWT and Event Handling

Event Handling Mechanism in java

- One of the important events handling mechanisms in java is handling the events using **Event Delegation Model.**
- This model defines standard and consistent mechanisms to generate and process events.
- The basic idea behind this model is, a source generates event and sends it to one or more event listener. Event listener simply waits until it receives an event. Once received, the listener processes/handles the events and then returns. The user interface elements are able to "delegate" the processing of an event to a separate piece of code hence called as event delegation model.



- In the delegation event model, listeners must be registered with a source in order to receive an event notification.
- The advantage of this design is that the logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code.
- The main components of event delegation model are
  1. Event Classes
  2. Events Sources
  3. Events Listeners

Steps in Event Handling:

- The User interacts with event source for example, clicks the button, and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- The method is now gets executed and returns.

Event Classes

- An event is an object that describes a state change in a source.
- Event represents the activities between the application and user. In GUI application, the application waits for the user to perform some action. When the user performs some action, the application executes the appropriate code for handling that action.
- GUI program in java are also called as event driven program because the sequences of events are created as user interacts with the GUI components determining what happen in the program.
- Some of the actions that cause events to be generated are **pressing a button, entering a character via keyboard, selecting an item in the list, clicking a mouse etc.**
- The **java.awt.event** defines several types of events that are generated by various GUI components. The below table shows different event classes and explains when they are generated.

Event Class	Description of Event Source
ActionEvent	Generated when button is pressed, a list item is double-clicked or menu item is selected
ItemEvent	Generated when checkbox is clicked, list item is selected or choice item is selected
KeyEvent	Generated when input is received from the keyboard i.e. when key is pressed, released, typed
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, released and when mouse enters or exists a components(Hover in and Hover out)
TextEvent	Generated when the value of a Textarea or Textfield is changed
WindowEvent	Generated when a window is opened, closed, activated, deactivated, deiconified, or iconified
FocusEvent	Generated when a components gains or loses keyboard focus

Event Sources

- A source is an object that generates an event.
- Some sources may generate more than one type of events.
- A source must listen in order for the listener to receive notification about a specific type of events.
- Some of the examples of event source are **Button, Choice, TextField, List, Checkbox etc.**

Event Listener

- Event listener are the classes that implement the various **listener interfaces** defined by the **java.awt.event** package.
- When an event occurs, the event source invokes the appropriate method defined by the listener and provide an event object as its argument.

The below table lists different event source classes, their corresponding event types and event listener interfaces.

Event Source	Event Type	Event Listener interface
Button click, list double clicked, menu item selected	ActionEvent	ActionListener
Checkbox clicked, list item selected, choice item selected	ItemEvent	ItemListener
Key pressed, key released, key typed	KeyEvent	KeyListener
Mouse clicked, mouse entered, mouse pressed, mouse exited, mouse released etc.	MouseEvent	MouseListener
Mouse dragged, mouse moved	MouseMotionEvent	MouseMotionListener
Value of Textarea and Textfield changed	TextEvent	TextListener
Window opened, window closed etc	WindowEvent	WindowListener
Gain keyboard focus, lose keyboard focus	FocusEvent	FocusListener

The following sections the specific methods that are contained in each interface

1. **ActionListener interface**
   This interface define only one method **actionPerformed()** to receive action event. This method is invoked when an **action event** occurs.
   The general form is:

```
void actionPerformed(ActionEvent e)
 {

 }
```

Some of the methods provided by ActionEvent object are

i. getSource(): returns the reference name of the object the event happened to.
ii. getActionCommand(): Returns the command string associated with the action.

Note: we use setActionCommand() to set the action command string in the components like Button, List and Menu item.

2. **ItemListener interface**
   This interface defines only one method **itemStateChanged()** to recognize when the state of an item changes. This method is invoked when the state of an item is changes.
   The general form is:

```
void itemStateChanged(ItemEvent e)
 {

 }
```

Some of the methods provided byItemEvents object are

i. int getStateChange(): Returns the new state of the item which can be Selected and DESELECTED.
ii. ItemSelectable getItemSelected(): Returns the component that fired the item event. This is similar to getSource() method.

3. **KeyListener interface**

This interface defines three methods to recognize when a key is pressed, released or typed. The **keyPressed() and keyReleased()** methods are invoked when a key is pressed and released respectively. The **keyTyped()** method is invoked when a character has been entered.

Note: if user press 'A ' and release then it generates key pressed, typed and released events in sequence. If user press 'home' key then it generates key pressed and key released events in sequence.

The general form is:

```
void keyPressed(KeyEvent e)
    {

    }


void keyReleased(KeyEvent e)
    {

    }
void keyTyped(KeyEvent e)
    {

    }
```

Some of the methods provided by KeyEvent object are

i. int getKeycode(): Retrieve the key code for a given character.
ii. int getKeychar(): Retrieve the character equivalent of a given key code.

The keyEvent class provides the following integer constant

VK_A , VK_B……VK_Z, VK_UP, VK_DOWN, VK_ENTER, VK_SHIFT etc.

4. **MouseListener interface**

This interface defines five method to recognize when the mouse is clicked, enters a component, exits a component, is pressed or is released. **mouseClicked()** method is invoked when the mouse is clicked and released at the same point. When a mouse enters a component **mouseEntered()** method is invoked. When it leaves, **mouseExited()** method is invoked. The **mousePressed()** and **mouseReleased()** methods are invoked when the mouse is pressed and released respectively.

The general form is:

```
void mouseClicked(MouseEvent e)
    {

    }
void mouseEntered(MouseEvent e)
    {

    }
void mouseExited(MouseEvent e)
    {

    }
void mousePressed(MouseEvent e)
    {

    }
void mouseReleased(MouseEvent e)
    {

    }
```

Some of the methods provided by MouseEvent object are

i. int getX(): Returns the X coordinate
ii. int getY(): Returns the Y coordinate

5. **MouseMotionListener interface**
   This interface defines two methods to recognize when mouse is dragged or moved. The **mouseDragged()** method is invoked multiple times as the mouse is dragged. The **mouseMoved()** method is invoked multiple times as the mouse is moved.
   The general form is:

```
void mouseDragged(MouseEvent e)
{

}
void mouseMoved(MoseEvent e)
{

}
```

6. **TextListener interface**
   This interface defines only one method **textChanged()** that is invoked when a change occurs in a text area or text field.
   The general form is:

```
void textChanged(TextEvent e)
{

}
```

7. **WindowListener interface**
   This interface defines seven methods to recongnize when a window is activated, closed, deactivated, deiconified, iconified, opened or quit. The **windowActivated() and windowDeactivated()** methods are invoked when a window is activated and deactivated respectively. The **windowIconified() and windowDeiconified()** methods are invoked when the window is iconified and deiconified. The **windowOpened() and windowClosed()** methods are invoked when the window is opened or closed respectively. The **windowClosing()** method is invoked when a window is being closed.
   The general form is:

```
void windowActivated(WindowEvent e)
{
}
void windowDeactivated(WindowEvent e)
{

}
void windowClosed(WindowEvent e)
{

}
void windowOpened(WindowEvent e)
{

}
void windowClosing(WindowEvent e)
{

}
void windowIconified(WindowEvent e)
{

}
void windowDeiconified(WindowEvent e)
{

}
```

8. **FocusListener interface**

This interface defines two methods to recognize when a component gains or loses keyboard focus. The **focusGained() and focusLost()** methods are invoked when a component obtains and loses keyboard focus respectively.
The general form is:

```
void focusGained(FocusEvent e)
{

}
void focusLost(FocusEvent e)
{

}
```

9. ContainerListener

The interface**ContainerListener** is used for receiving container events. The class that process container events needs to implements this interface. The componentAdded () and componentRemoved() method are invoked when the components are added or removed. The general form is

```
void componentAdded(ContainerEvent e)
{

}

Void componentRemoved(ContainerEvent e)
{

}
```

Steps in Event Handling

1. Implement an appropriate Event Listener interface.
2. Register an Event listener
3. Override the requested methods for that listener interface, which handles the delegated event.

For example: To write an Action Listener, follow the steps given below:

1. Declare an event handler class and specify that the class either implements an ActionListener interface or extends a class that implements an ActionListener interface. For example:

```
public class MyClass extends Frame implements ActionListener
{

}
```

2. Register an instance of the event handler class as a listener on one or more components.
   For example: someComponent.addActionListener(instanceOf**MyClass**);
3. Include code that implements the methods in listener interface.
   For example:

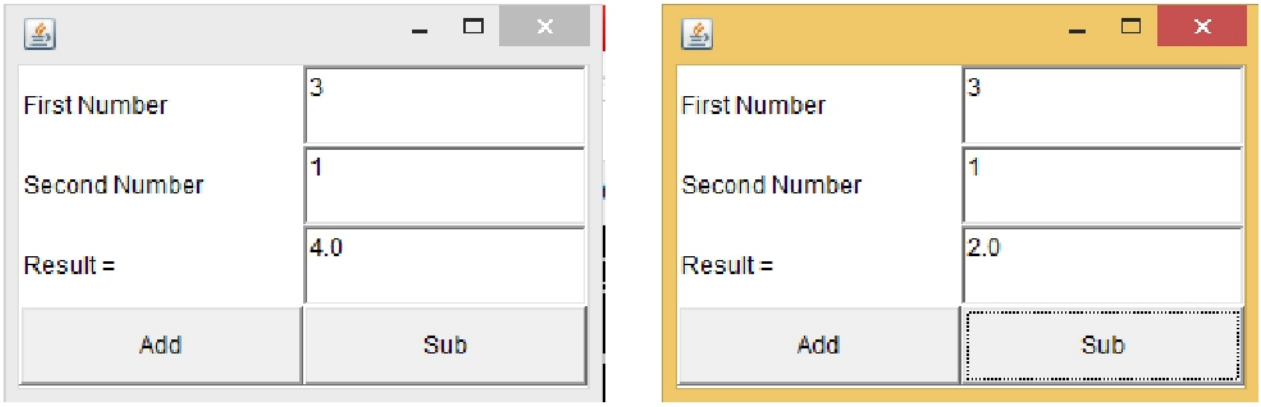
```
public void actionPerformed(ActionEvent e)
{
        ...//code that reacts to the action...
}
```

1. Create A GUI from that takes two number as input and find the sum and difference. Use GridLayout manager.

```
import java.awt.*;
import java.awt.event.*;
public class AddDemo1 extends Frame implements ActionListener // Event Listener
{
    GridLayout glayout;
    Label lbl_fn,lbl_sn,lbl_res;
    Button btn_add,btn_sub;
    TextField txt_fn,txt_sn,txt_res;
```

```
        public AddDemo1()
        {
                setSize(300,200);
                glayout=new GridLayout(4,2);
                setLayout(glayout);
                lbl_fn=new Label("First Number");
                add(lbl_fn);
                txt_fn=new TextField(20);
                add(txt_fn);
                lbl_sn=new Label("Second Number");
                add(lbl_sn);
                txt_sn=new TextField(20);
                add(txt_sn);
                lbl_res=new Label("Result =");
                add(lbl_res);
                txt_res=new TextField(20);
                add(txt_res);
                btn_add=new Button("Add");
                btn_add.addActionListener(this);//Registering event
                add(btn_add);
                btn_sub=new Button("Sub");
                btn_sub.addActionListener(this);//Registering event
                add(btn_sub);
                show();
        }
        public static void main(String [] args)
        {
                new AddDemo1();
        }
        public void actionPerformed(ActionEvent e)// overriding method
        {
                double fn=Double.parseDouble(txt_fn.getText());
                double sn=Double.parseDouble(txt_sn.getText());
                if(e.getSource()==btn_add)
                {
                        double sum=fn+sn;
                        txt_res.setText(Double.toString(sum));
                }
                else if(e.getSource()==btn_sub)
                {
                        double dif=fn-sn;
                        txt_res.setText(Double.toString(dif));
                }
        }
}
```

Output:



Handling the event using Anonymous Inner Class and Adapter class

- Anonymous inner class are useful to handle the event.
- **Many event listener interfaces, such as mouse listener, window listener contain multiple methods all of which must be defined if such interfaces are implemented.**
- **But, if we want only one KeyPressed handler among available three. For this we use Adapter class which implements an interface an allow us to declare only required methods in the interface**
- The following table shows some important listener and their corresponding adapters.

Listener Interface	Corresponding Adapter
WindowListener (7)	WindowAdapter
MouseListener (5)	MouseAdapter
MouseMotionListener (2)	MouseMotionAdapter
KeyListener (3)	KeyAdapter
FocusListener (2)	FocusAdapter

- A special form of an inner class that is declared without a name and typically appears inside a method declaration is called **Anonymous inner class.** They allow us to declare and instantiate a class at the same time. **They are used whenever we need to override the method of class or an interface.**

Syntax:

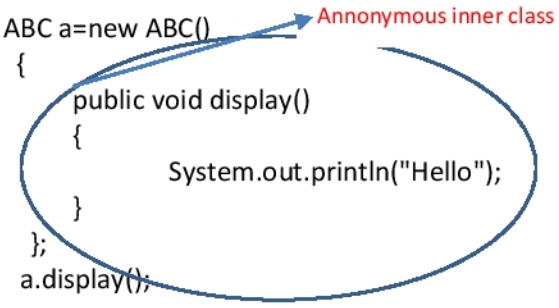
```
Class_name obj=new Class_name ()
{
        public void method_defination()
        {

        }
};
```

Example:

```
//abstact class can contain both abstract and concrete methods
abstract class ABC
{
        public abstract void display();
}
class PQR extends ABC
{
        public void display()
        {
                System.out.println("Hello");
        }
}
class Demo
{
        public static void main(String [] args)
        {
                PQR p=new PQR();
                p.display();
        }
}
```

Example: Approach without Annonymous inner class

```
abstract class ABC
{
        public abstract void display();
}
class Demo
{
        public static void main(String [] args)
        {
           ABC a=new ABC()
            {
                public void display()
                {
                        System.out.println("Hello");
                }
            };
            a.display();
        }
}
```



Example: Approach with Annonymous inner class

Example: writing the above program using the concept of anonymous inner class and adapter class. Also the program must quit whent the user presses close button in window.

```
import java.awt.*;
import java.awt.event.*;
public class AddDemo11 extends Frame
{
        GridLayout glayout;
        Label lbl_fn,lbl_sn,lbl_res;
        Button btn_add,btn_sub;
        TextField txt_fn,txt_sn,txt_res;

        public AddDemo11()
        {
                setSize(300,200);
                addWindowListener(new WindowAdapter()// here we use new WindowAdpater() class so only the
                                //required method among many abstract methods can be defined as below
                {
                        public void windowClosing(WindowEvent e)
                        {
                                System.exit(0);// by default you can't quit the window
                        }
                }
                );

        /*
        addWindowListener(new WindowListener()  // here we use new WindowListener interface so all  its
                                                abstracts method must be defined as below

        {
                public void windowClosing(WindowEvent e)
                {
                        System.exit(0);// by default you can't quit the window
                }
                public void windowActivated(WindowEvent e)
```

```
		{
		}
		public void windowDeactivated(WindowEvent e)
		{
		}
		public void windowClosed(WindowEvent e)
		{
		}

		public void windowOpened(WindowEvent e)
		{
		}
		public void windowIconified(WindowEvent e)
		{
		}
		public void windowDeiconified(WindowEvent e)
		{
		}
}
);
*/

glayout=new GridLayout(4,2);
setLayout(glayout);
lbl_fn=new Label("First Number");
add(lbl_fn);
		txt_fn=new TextField(20);
		add(txt_fn);
		lbl_sn=new Label("Second Number");
		add(lbl_sn);
		txt_sn=new TextField(20);
		add(txt_sn);
		lbl_res=new Label("Result =");
		add(lbl_res);
		txt_res=new TextField(20);
		add(txt_res);
		btn_add=new Button("Add");
		btn_add.addActionListener( new ActionListener()
			{
				public void actionPerformed(ActionEvent e)
				{
					double fn=Double.parseDouble(txt_fn.getText());
					double sn=Double.parseDouble(txt_sn.getText());
					double sum=fn+sn;
					txt_res.setText(Double.toString(sum));
				}
			}
		);
		add(btn_add);
		btn_sub=new Button("Sub");
		btn_sub.addActionListener( new ActionListener()
			{
				public void actionPerformed(ActionEvent e)
				{
						double fn=Double.parseDouble(txt_fn.getText());
						double sn=Double.parseDouble(txt_sn.getText());
						double dif=fn-sn;
						txt_res.setText(Double.toString(dif));
				}
			}
		);
```

```
                add(btn_sub);
                show();
        }

        public static void main(String [] args)
        {
                new AddDemo11();
        }
}
```

In the above program, the Adapter class for implementing different event is anonymous class.