

## MODULE IV

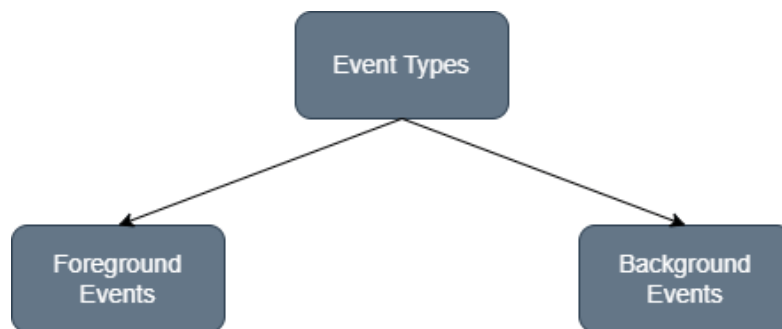
### Types of events in Java

An **event** is one of the most important concepts in Java. The change in the state of an object or behavior by performing actions is referred to as **an Event** in Java. Actions include button click, keypress, page scrolling, or cursor movement.

Java provides a package **java.awt.event** that contains several event classes.

We can classify the events in the following two categories:

1. Foreground Events
2. Background Events



#### Foreground Events

**Foreground events** are those events that require user interaction to generate. In order to generate these foreground events, the user interacts with components in GUI. When a user clicks on a button, moves the cursor, and scrolls the scrollbar, an event will be fired.

#### Background Events

**Background events** don't require any user interaction. These events automatically generate in the background. OS failure, OS interrupts, operation completion, etc., are examples of background events.

### Delegation Event Model

A mechanism for controlling the events and deciding what should happen after an event occur is referred to as event handling. Java follows the **Delegation Event Model** for handling the events.

The **Delegation Event Model** consists of **Source** and **Listener**.

## Source

Buttons, checkboxes, list, menu-item, choice, scrollbar, etc., are the sources from which events are generated.

## Listeners

The events which are generated from the source are handled by the listeners. Each and every listener represents interfaces that are responsible for handling events.

The syntax of registering the source with the listener is as follows:

### 1. addTypeListener

For example, if we need to register **Key** and **Action** events, we use the **addActionListener()** and **addKeyListener()** methods.

These are some of the most used Event classes:

S.N o.	Event Class	Listener Interface	Methods	Descriptions
1.	ActionEvent	ActionListener	actionPerformed()	ActionEvent indicates that a component-defined action occurred.
2.	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged()	Adjustment events occur by an adjustable object like a scrollbar.
3.	ComponentEvent	ComponentListener	componentResized(), componentMoved(), componentShown() and componentHidden()	An event occurs when a component moved, changed its visibility or the size changed.
4.	ContainerEvent	ContainerListener	componentRemoved() and componentAdded()	The event is fired when a component is added or removed from a container.
5.	FocusEvent	FocusListener	focusLost() and focusGained()	Focus events include focus, focusout, focusin, and blur.
6.	ItemEvent	ItemListener	itemStateChanged()	Item event occurs when an item is selected.
7.	KeyEvent	KeyListener	keyPressed(), keyReleased(), and keyTyped().	A key event occurs when the user presses a key on the keyboard.

8.	MouseEvent	MouseListener and MouseMotionListener	mouseClicked(), mousePressed(), mouseEntered(), mouseExited() and mouseReleased() are the mouseListener methods. mouseDragged() and mouseMoved() are the MouseMotionListener() methods.	A mouse event occurs when the user interacts with the mouse.
9.	MouseWheelEvent	MouseWheelListener	mouseWheelMoved().	MouseWheelEvent occurs when the mouse wheel rotates in a component.
10.	TextEvent	TextListener	textChanged()	TextEvent occurs when an object's text change.
11.	WindowEvent	WindowListener	windowActivated(), windowDeactivated(), windowOpened(), windowClosed(), windowClosing(), windowIconified() and windowDeiconified().	Window events occur when a window's status is changed.

Let's take an example to understand how we can work with the events and listeners:

### **EventHandlerExample1.java**

```
// import required classes and package, if any
package JavaTpoint.Examples;
```

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

```
//create class EventHandlerExample1 to perform event handling within the class
public class EventHandlerExample1 {
```

```
    // create variables for Frame, Label and Panel
    private Frame frame;
    private Label header;
    private Label status;
    private Panel panel;
```

```

// default constructor
public EventHandlingExample1(){
    makeGUI();
}

// main() method start
public static void main(String[] args){

    // create an instance of EventHandlingExample1
    EventHandlingExample1 obj = new EventHandlingExample1();
    obj.addButtonsAndLabel();
}

    // create makeGUI() method to create a UI to perform user interaction
private void makeGUI(){
    // initialize Frame
    frame = new Frame("Event Handling Example");
    // set frame size by using setSize() method
    frame.setSize(400,400);
    //set frame layout by using setLayout() method
    frame.setLayout(new GridLayout(3, 1));

    // add window listener to frame
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent windowEvent){
            System.exit(0);
        }
    });

    // initialize header and set alignment to center
    header = new Label();
    header.setAlignment(Label.CENTER);

    //initialize status and set its alignment and size
    status = new Label();
    status.setAlignment(Label.CENTER);
    status.setSize(350,100);

```

```
// initialize panel
panel = new Panel();

// set flow layout to panel
panel.setLayout(new FlowLayout());

// add header, status and panel to frame
frame.add(header);
frame.add(panel);
frame.add(status);
frame.setVisible(true);
}

// create addButtonsAndLabel() method
private void addButtonsAndLabel(){

    // set label test
    header.setText("Click a button: ");

    // create ok, submit and cancel buttons
    Button okBtn = new Button("OK");
    Button submitBtn = new Button("Submit");
    Button cancelBtn = new Button("Cancel");

    // use setActionCommand() method to set action for ok, submit and cancel buttons
    okBtn.setActionCommand("OK");
    submitBtn.setActionCommand("Submit");
    cancelBtn.setActionCommand("Cancel");

    // add event listener to buttons using addActionListener() and ButtonClickListener()
    okBtn.addActionListener(new ButtonClickListener());
    submitBtn.addActionListener(new ButtonClickListener());
    cancelBtn.addActionListener(new ButtonClickListener());

    // add buttons to panel
    panel.add(okBtn);
    panel.add(submitBtn);
```

```

panel.add(cancelBtn);

// make frame visible by using setVisible()mmethod
frame.setVisible(true);
}

// implements ActionListener interface
private class ButtonClickListener implements ActionListener{

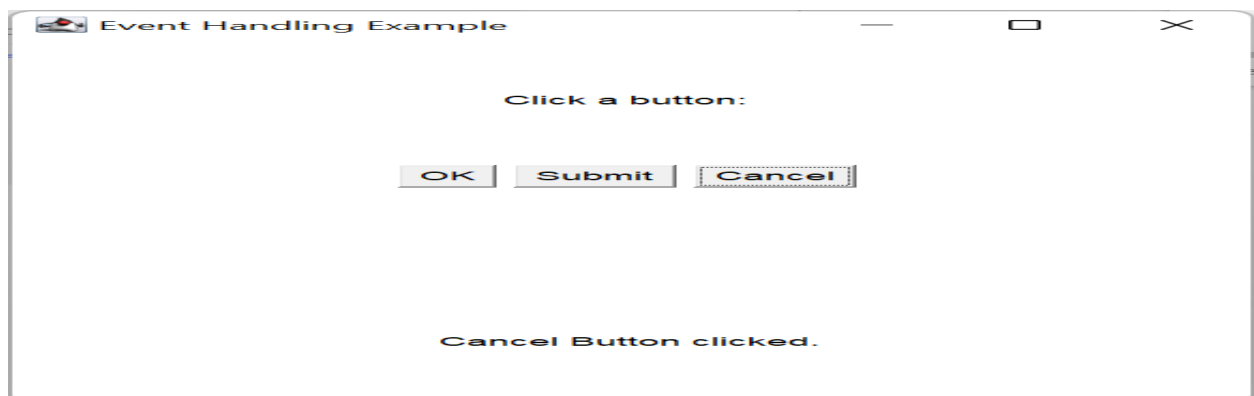
    // define actionPerformed() method of ActionListener
    public void actionPerformed(ActionEvent event) {

        // get action command using getActionCommand() method of Event
        String command = event.getActionCommand();

        // code to check which button is pressed
        if( command.equals( "OK" )) {
            status.setText("Ok Button clicked.");
        } else if( command.equals( "Submit" ) ) {
            status.setText("Submit Button clicked.");
        } else {
            status.setText("Cancel Button clicked.");
        }
    }
}
}

```

### Output:



# 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 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

```
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();
```

```
}
```

```
}
```

**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 event handling by outer class

```
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();
}
}

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");
}
}

```

## **Java event handling by anonymous class**

```

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();
}
}

```

## Delegation Event Model in Java

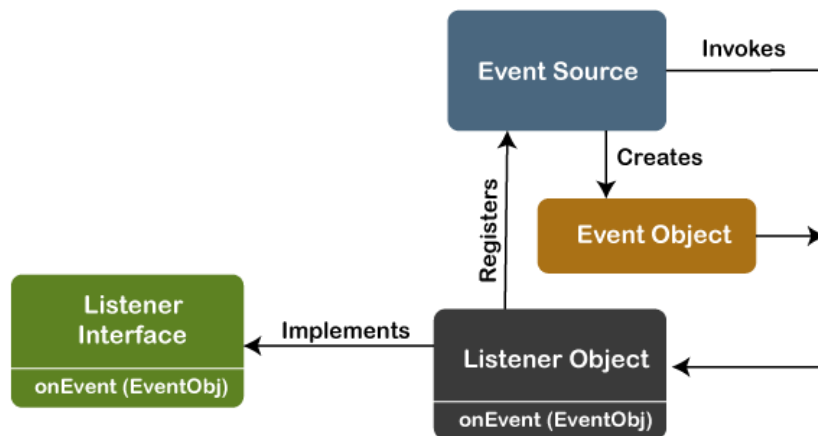
The Delegation Event model is defined to handle events in GUI programming languages.

The GUI stands for Graphical User Interface, where a user graphically/visually interacts with the system.

The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.

## Event Processing in Java

Java support event processing since Java 1.0. It provides support for AWT ( Abstract Window Toolkit) , which is an API used to develop the Desktop application. In Java 1.0, the AWT was based on inheritance. To catch and process GUI events for a program, it should hold subclass GUI components and override action() or handleEvent() methods. The below image demonstrates the event processing.



But, the modern approach for event processing is based on the Delegation Model. It defines a standard and compatible mechanism to generate and process events. In this model, a source generates an event and forwards it to one or more listeners. The listener waits until it receives an event. Once it receives the event, it is processed by the listener and returns it. The UI elements are able to delegate the processing of an event to a separate function.

The key advantage of the Delegation Event Model is that the application logic is completely separated from the interface logic. In this model, the listener must be connected with a source to receive the event notifications. Thus, the events will only be received by the listeners who wish to receive them. So, this approach is more convenient than the inheritance-based event model (in Java 1.0).

In the older model, an event was propagated up the containment until a component was handled. This needed components to receive events that were not processed, and it took lots of time. The Delegation Event model overcame this issue.

**Basically, an Event Model is based on the following three components:**

- Events
- Events Sources
- Events Listeners

## Events

The Events are the objects that define state change in a source. An event can be generated as a reaction of a user while interacting with GUI elements. Some of the event generation activities are moving the mouse pointer, clicking on a button, pressing the keyboard key, selecting an item from the list, and so on. We can also consider many other user operations as events.

The Events may also occur that may be not related to user interaction, such as a timer expires, counter exceeded, system failures, or a task is completed, etc. We can define events for any of the applied actions.

## Event Sources

A source is an object that causes and generates an event. It generates an event when the internal state of the object is changed. The sources are allowed to generate several different types of events.

A source must register a listener to receive notifications for a specific event. Each event contains its registration method. Below is an example:

### 1. **public void** addTypeListener (TypeListener e1)

From the above syntax, the Type is the name of the event, and e1 is a reference to the event listener. For example, for a keyboard event listener, the method will be called as **addKeyListener()**. For the mouse event listener, the method will be called as **addMouseMotionListener()**. When an event is triggered using the respected source, all the events will be notified to registered listeners and receive the event object. This process is known as event multicasting. In few cases, the event notification will only be sent to listeners that register to receive them.

### 2. **public void** addTypeListener(TypeListener e2) **throws** java.util.TooManyListenersException

From the above syntax, the Type is the name of the event, and e2 is the event listener's reference. When the specified event occurs, it will be notified to the registered listener. This process is known as **unicasting** events.

A source should contain a method that unregisters a specific type of event from the listener if not needed. Below is an example of the method that will remove the event from the listener.

### 3. **public void** removeTypeListener(TypeListener e2?)

From the above syntax, the Type is an event name, and e2 is the reference of the listener. For example, to remove the keyboard listener, the **removeKeyListener()** method will be called.

The source provides the methods to add or remove listeners that generate the events. For example, the Component class contains the methods to operate on the different types of events, such as adding or removing them from the listener.

## Event Listeners

An event listener is an object that is invoked when an event triggers. The listeners require two things; first, it must be registered with a source; however, it can be registered with several resources to receive notification about the events. Second, it must implement the methods to receive and process the received notifications.

The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.

For example, the **MouseEvent** interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the **MouseEvent** interface.

## Types of Events

The events are categorized into the following two categories:

### The Foreground Events:

The foreground events are those events that require direct interaction of the user. These types of events are generated as a result of user interaction with the GUI component. For example, clicking on a button, mouse movement, pressing a keyboard key, selecting an option from the list, etc.

### The Background Events :

The Background events are those events that result from the interaction of the end-user. For example, an operating system interrupt, system failure (Hardware or Software).

To handle these events, we need an event handling mechanism that provides control over the events and responses.

## The Delegation Model

The Delegation Model is available in Java since Java 1.1. It provides a new delegation-based event model using AWT to resolve the event problems. It provides a convenient mechanism to support complex Java programs.

## Design Goals

The design goals of the event delegation model are as following:

- It is easy to learn and implement
- It supports a clean separation between application and GUI code.
- It provides robust event handling program code which is less error-prone (strong compile-time checking)
- It is Flexible, can enable different types of application models for event flow and propagation.
- It enables run-time discovery of both the component-generated events as well as observable events.
- It provides support for the backward binary compatibility with the previous model.

Let's implement it with an example:

## Java Program to Implement the Event Delegation Model

The below is a Java program to handle events implementing the event delegation model:

### **TestApp.java:**

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

public class TestApp {
    public void search() {
        // For searching
        System.out.println("Searching...");
    }
    public void sort() {
        // for sorting
        System.out.println("Sorting....");
    }

    static public void main(String args[]) {
        TestApp app = new TestApp();
        GUI gui = new GUI(app);
    }
}

class Command implements ActionListener {
    static final int SEARCH = 0;
    static final int SORT = 1;
    int id;
    TestApp app;

    public Command(int id, TestApp app) {
        this.id = id;
        this.app = app;
    }

    public void actionPerformed(ActionEvent e) {
        switch(id) {
```

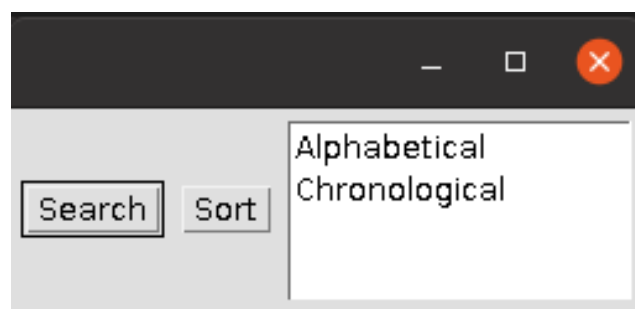
```

    case SEARCH:
        app.search();
        break;
    case SORT:
        app.sort();
        break;
    }
}
}

class GUI {
    public GUI(TestApp app) {
        Frame f = new Frame();
        f.setLayout(new FlowLayout());
        Command searchCmd = new Command(Command.SEARCH, app);
        Command sortCmd = new Command(Command.SORT, app);
        Button b;
        f.add(b = new Button("Search"));
        b.addActionListener(searchCmd);
        f.add(b = new Button("Sort"));
        b.addActionListener(sortCmd);
        List l;
        f.add(l = new List());
        l.add("Alphabetical");
        l.add("Chronological");
        l.addActionListener(sortCmd);
        f.pack();
        f.show();
    } }

```

### Output:





# Java MouseListener Interface

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

## Methods of MouseListener interface

The signature of 5 methods found in MouseListener interface are given below:

1. **public abstract void** mouseClicked(MouseEvent e);
2. **public abstract void** mouseEntered(MouseEvent e);
3. **public abstract void** mouseExited(MouseEvent e);
4. **public abstract void** mousePressed(MouseEvent e);
5. **public abstract void** mouseReleased(MouseEvent e);

## Java MouseListener Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener{
    Label l;
    MouseListenerExample(){
        addMouseListener(this);

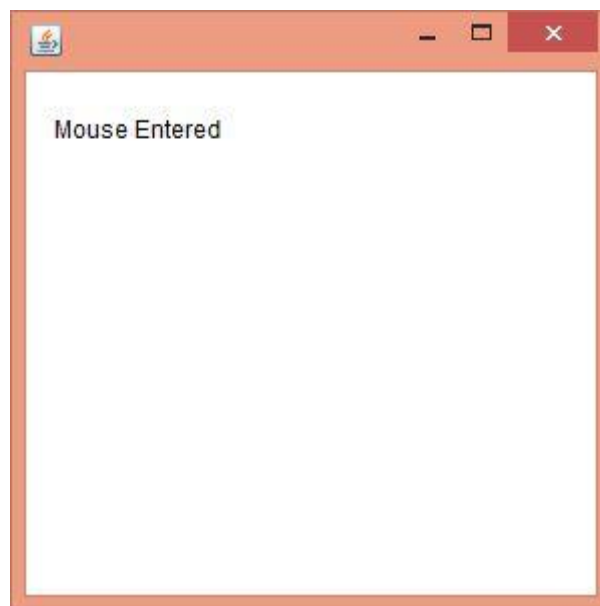
        l=new Label();
        l.setBounds(20,50,100,20);
        add(l);
        setSize(300,300);
        setLayout(null);
        setVisible(true);
    }
    public void mouseClicked(MouseEvent e) {
        l.setText("Mouse Clicked");
    }
    public void mouseEntered(MouseEvent e) {
        l.setText("Mouse Entered");
    }
    public void mouseExited(MouseEvent e) {
```

```

        l.setText("Mouse Exited");
    }
    public void mousePressed(MouseEvent e) {
        l.setText("Mouse Pressed");
    }
    public void mouseReleased(MouseEvent e) {
        l.setText("Mouse Released");
    }
    public static void main(String[] args) {
        new MouseListenerExample();
    }
}

```

**Output:**



## Java MouseListener Example 2

```

import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample2 extends Frame implements MouseListener{
    MouseListenerExample2(){
        addMouseListener(this);

        setSize(300,300);
    }
}

```

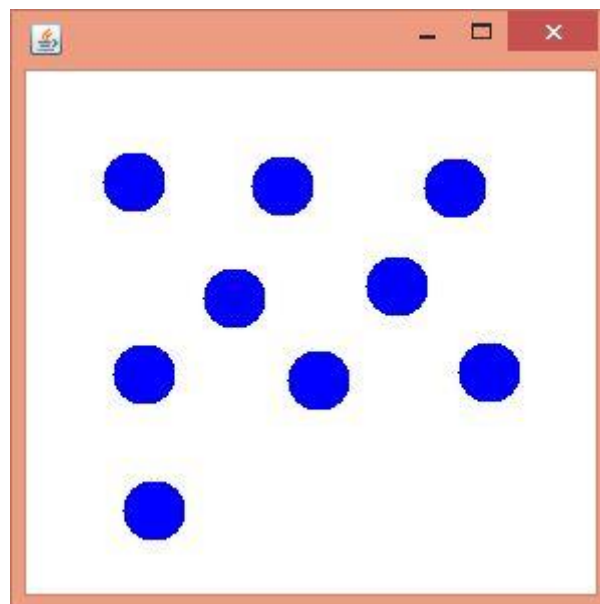
```

    setLayout(null);
    setVisible(true);
}
public void mouseClicked(MouseEvent e) {
    Graphics g=getGraphics();
    g.setColor(Color.BLUE);
    g.fillOval(e.getX(),e.getY(),30,30);
}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mousePressed(MouseEvent e) {}
public void mouseReleased(MouseEvent e) {}

public static void main(String[] args) {
    new MouseListenerExample2();
}
}

```

**Output:**



# Java KeyListener Interface

The **Java KeyListener** is notified whenever you change the state of key. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package, and it has three methods.

## Interface declaration

Following is the declaration for **java.awt.event.KeyListener** interface:

1. **public interface** KeyListener **extends** EventListener

## Methods of KeyListener interface

The signature of 3 methods found in KeyListener interface are given below:

| Sr. no. | Method name                                    | Description                                 |
|---------|------------------------------------------------|---------------------------------------------|
| 1.      | public abstract void keyPressed (KeyEvent e);  | It is invoked when a key has been pressed.  |
| 2.      | public abstract void keyReleased (KeyEvent e); | It is invoked when a key has been released. |
| 3.      | public abstract void keyTyped (KeyEvent e);    | It is invoked when a key has been typed.    |

## Methods inherited

This interface inherits methods from the following interface:

java.awt.EventListener

## Java KeyListener Example

In the following example, we are implementing the methods of the KeyListener interface.

**KeyListenerExample.java**

```
// importing awt libraries
import java.awt.*;
import java.awt.event.*;
// class which inherits Frame class and implements KeyListener interface
```

```

public class KeyListenerExample extends Frame implements KeyListener {
// creating object of Label class and TextArea class
Label l;
    TextArea area;
// class constructor
    KeyListenerExample() {
        // creating the label
        l = new Label();
// setting the location of the label in frame
        l.setBounds (20, 50, 100, 20);
// creating the text area
        area = new TextArea();
// setting the location of text area
        area.setBounds (20, 80, 300, 300);
// adding the KeyListener to the text area
        area.addKeyListener(this);
// adding the label and text area to the frame
        add(l);
        add(area);
// setting the size, layout and visibility of frame
        setSize (400, 400);
        setLayout (null);
        setVisible (true);
    }
// overriding the keyPressed() method of KeyListener interface where we set the text of the label
when key is pressed
    public void keyPressed (KeyEvent e) {
        l.setText ("Key Pressed");
    }
// overriding the keyReleased() method of KeyListener interface where we set the text of the label
when key is released
    public void keyReleased (KeyEvent e) {
        l.setText ("Key Released");
    }
// overriding the keyTyped() method of KeyListener interface where we set the text of the label when
a key is typed
    public void keyTyped (KeyEvent e) {

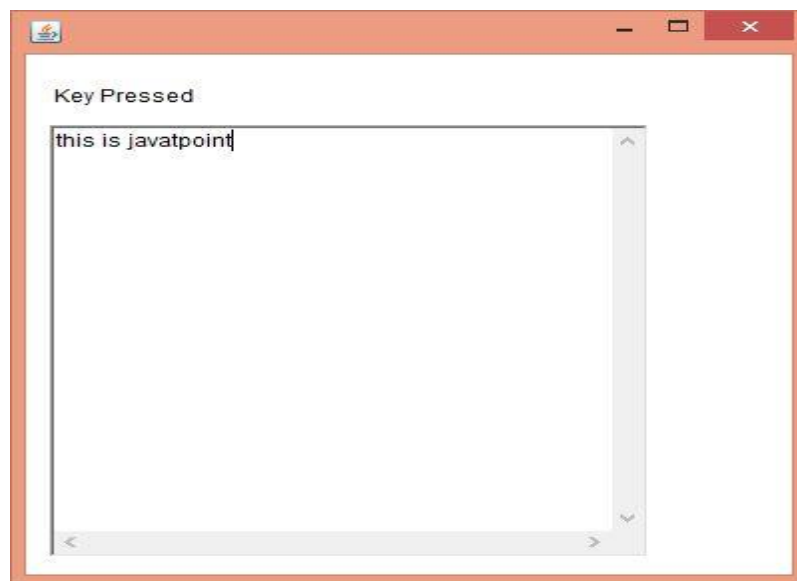
```

```

        l.setText ("Key Typed");
    }
// main method
    public static void main(String[] args) {
        new KeyListenerExample();
    }
}

```

**Output:**



## Java KeyListener Example 2: Count Words & Characters

In the following example, we are printing the count of words and characters of the string. Here, the string is fetched from the TextArea and uses the `KeyReleased()` method of `KeyListener` interface.

### KeyListenerExample2.java

```

// importing the necessary libraries
import java.awt.*;
import java.awt.event.*;
// class which inherits Frame class and implements KeyListener interface
public class KeyListenerExample2 extends Frame implements KeyListener {
// object of Label and TextArea

```

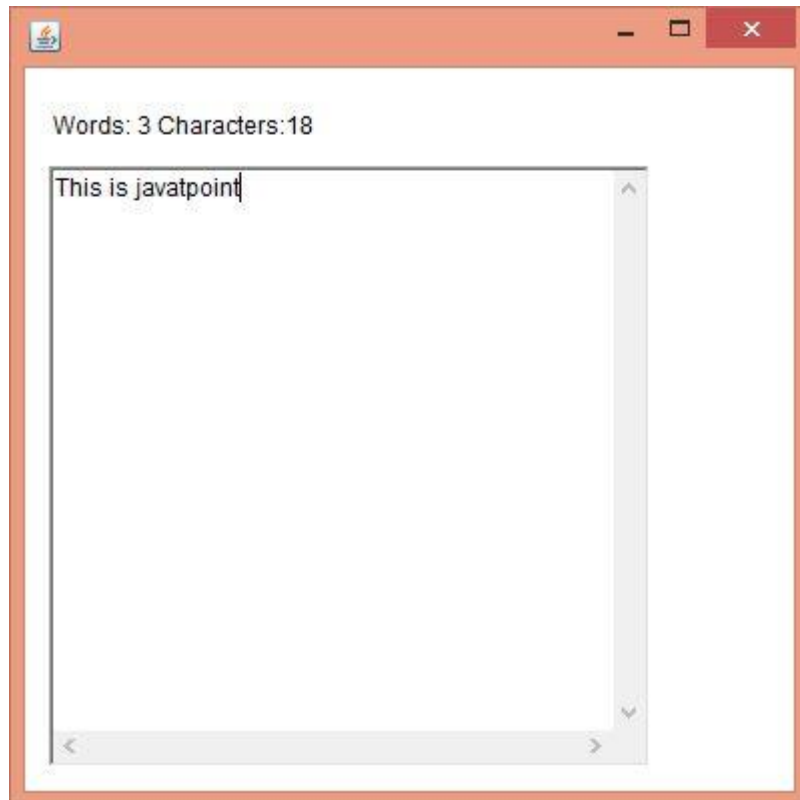
```

    Label l;
    TextArea area;
// class constructor
    KeyListenerExample2() {
        // creating the label
        l = new Label();
// setting the location of label
        l.setBounds (20, 50, 200, 20);
// creating the text area
        area = new TextArea();
// setting location of text area
        area.setBounds (20, 80, 300, 300);
// adding KeyListener to the text area
        area.addKeyListener(this);
// adding label and text area to frame
        add(l);
        add(area);
// setting size, layout and visibility of frame
        setSize (400, 400);
        setLayout (null);
        setVisible (true);
    }
// even if we do not define the interface methods, we need to override them
    public void keyPressed(KeyEvent e) {}
// overriding the keyReleased() method of KeyListener interface
    public void keyReleased (KeyEvent e) {
// defining a string which is fetched by the getText() method of TextArea class
        String text = area.getText();
// splitting the string in words
        String words[] = text.split ("\\s");
// printing the number of words and characters of the string
        l.setText ("Words: " + words.length + " Characters:" + text.length());
    }
    public void keyTyped(KeyEvent e) {}
// main method
    public static void main(String[] args) {
        new KeyListenerExample2();
    }

```

```
}  
}
```

**Output:**



## **Java Inner Classes (Nested Classes)**

**Java inner class** or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

### ***Syntax of Inner class***

```
class Java_Outer_class{  
    //code  
    class Java_Inner_class{  
        //code  
    } }  
}
```



## Advantage of Java inner classes

There are three advantages of inner classes in Java. They are as follows:

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class**, including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization**: It requires less code to write.

## Need of Java Inner class

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class. That way all the outer class can access all the objects of the inner class.

## Difference between nested class and inner class in Java

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

## Types of Nested classes

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

### ➤ Non-static nested class (inner class)

1. Member inner class
2. Anonymous inner class
3. Local inner class

### ➤ Static nested class

| Type                         | Description                                                                                           |
|------------------------------|-------------------------------------------------------------------------------------------------------|
| <u>Member Inner Class</u>    | A class created within class and outside method.                                                      |
| <u>Anonymous Inner Class</u> | A class created for implementing an interface or extending class. The java compiler decides its name. |
| <u>Local Inner Class</u>     | A class was created within the method.                                                                |
| <u>Static Nested Class</u>   | A static class was created within the class.                                                          |
| <u>Nested Interface</u>      | An interface created within class or interface.                                                       |

## Java Member Inner class

A non-static class that is created inside a class but outside a method is called **member inner class**. It is also known as a **regular inner class**. It can be declared with access modifiers like public, default, private, and protected.

### Syntax:

```
class Outer{
//code
class Inner{
//code
}
}
```

## Java Member Inner Class Example

In this example, we are creating a msg() method in the member inner class that is accessing the private data member of the outer class.

### TestMemberOuter1.java

```
class TestMemberOuter1 {
    private int data=30;
    class Inner{
        void msg(){System.out.println("data is "+data);}
    }
    public static void main(String args[]){
```

```

TestMemberOuter1 obj=new TestMemberOuter1();
TestMemberOuter1.Inner in=obj.new Inner();
in.msg();
}
}

```

### Output:

```
data is 30
```

## How to instantiate Member Inner class in Java?

An object or instance of a member's inner class always exists within an object of its outer class. The new operator is used to create the object of member inner class with slightly different syntax.

The general form of syntax to create an object of the member inner class is as follows:

### Syntax:

1. OuterClassReference.**new** MemberInnerClassConstructor();

### Example:

1. obj.**new** Inner();

Here, OuterClassReference is the reference of the outer class followed by a dot which is followed by the new operator.

## Internal working of Java member inner class

The java compiler creates two class files in the case of the inner class. The class file name of the inner class is "Outer\$Inner". If you want to instantiate the inner class, you must have to create the instance of the outer class. In such a case, an instance of inner class is created inside the instance of the outer class.

## Internal code generated by the compiler

The Java compiler creates a class file named Outer\$Inner in this case. The Member inner class has the reference of Outer class that is why it can access all the data members of Outer class including private.

```

import java.io.PrintStream;
class Outer$Inner
{

```

```

final Outer this$0;
Outer$Inner()
{
    super();
    this$0 = Outer.this;
}
void msg()
{
    System.out.println((new StringBuilder()).append("data is ")
        .append(Outer.access$000(Outer.this)).toString());
}
}

```

## Java AWT Tutorial

**Java AWT** (Abstract Window Toolkit) is *an API to develop Graphical User Interface (GUI) or windows-based applications* in Java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).

The java.awt package provides classes for AWT API such as TextField , Label , TextArea , RadioButton, CheckBox , Choice , List etc.

The AWT tutorial will help the user to understand Java GUI programming in simple and easy steps.

### **Why AWT is platform independent?**

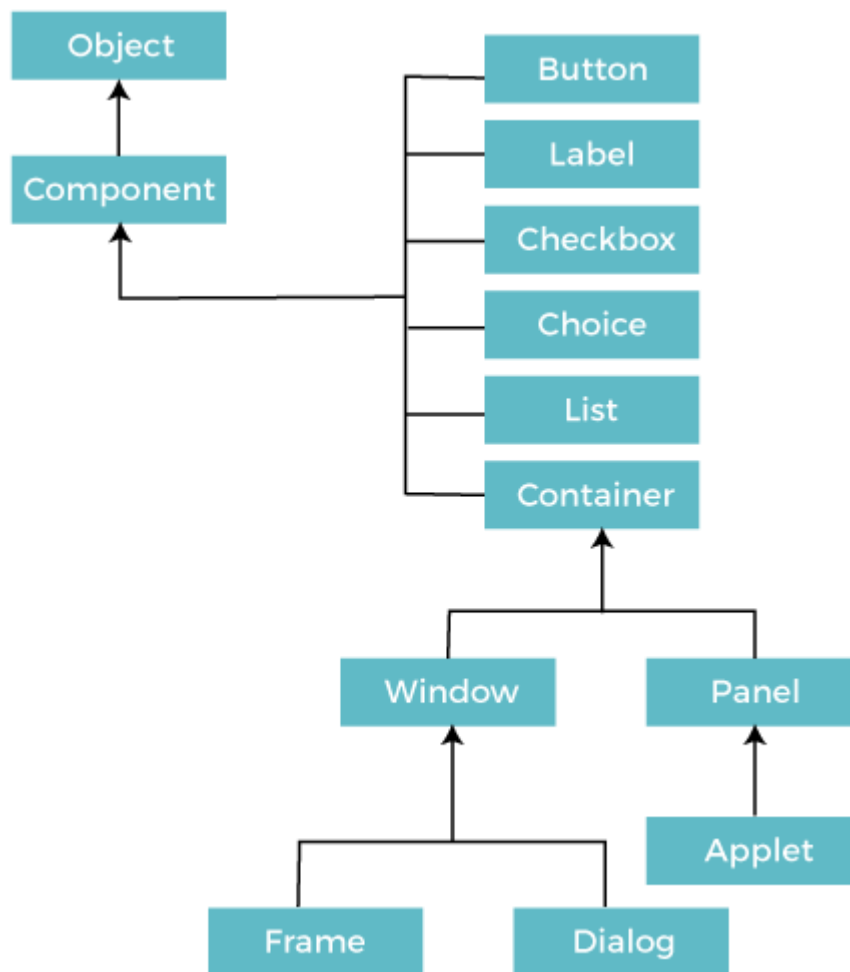
Java AWT calls the native platform calls the native platform (operating systems) subroutine for creating API components like TextField, ChechBox, button, etc.

For example, an AWT GUI with components like TextField, label and button will have different look and feel for the different platforms like Windows, MAC OS, and Unix. The reason for this is the platforms have different view for their native components and AWT directly calls the native subroutine that creates those components.

In simple words, an AWT application will look like a windows application in Windows OS whereas it will look like a Mac application in the MAC OS.

### **Java AWT Hierarchy**

The hierarchy of Java AWT classes are given below.



## COMPONENTS

All the elements like the button, text fields, scroll bars, etc. are called components. In Java AWT, there are classes for each component as shown in above diagram. In order to place every component in a particular position on a screen, we need to add them to a container.

### Container

The Container is a component in AWT that can contain another components like buttons , textfields, labels etc. The classes that extends Container class are known as container such as **Frame**, **Dialog** and **Panel**.

It is basically a screen where the where the components are placed at their specific locations. Thus it contains and controls the layout of components.

### Types of containers:

There are four types of containers in Java AWT:

1. Window
2. Panel
3. Frame
4. Dialog

## Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window. We need to create an instance of Window class to create this container.

## Panel

The Panel is the container that doesn't contain title bar, border or menu bar. It is generic container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.

## Frame

The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc. Frame is most widely used container while developing an AWT application.

## Useful Methods of Component Class

| Method                                    | Description                                                |
|-------------------------------------------|------------------------------------------------------------|
| public void add(Component c)              | Inserts a component on this component.                     |
| public void setSize(int width,int height) | Sets the size (width and height) of the component.         |
| public void setLayout(LayoutManager m)    | Defines the layout manager for the component.              |
| public void setVisible(boolean status)    | Changes the visibility of the component, by default false. |

## Java AWT Example

To create simple AWT example, you need a frame. There are two ways to create a GUI using Frame in AWT.

1. By extending Frame class (**inheritance**)

2. By creating the object of Frame class (**association**)

## AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

### AWTExample1.java

```
// importing Java AWT class
import java.awt.*;

// extending Frame class to our class AWTExample1
public class AWTExample1 extends Frame {
    // initializing using constructor
    AWTExample1() {
        // creating a button
        Button b = new Button("Click Me!!");
        // setting button position on screen
        b.setBounds(30,100,80,30);
        // adding button into frame
        add(b);
        // frame size 300 width and 300 height
        setSize(300,300);
        // setting the title of Frame
        setTitle("This is our basic AWT example");

        // no layout manager
        setLayout(null);

        // now frame will be visible, by default it is not visible
        setVisible(true);
    }

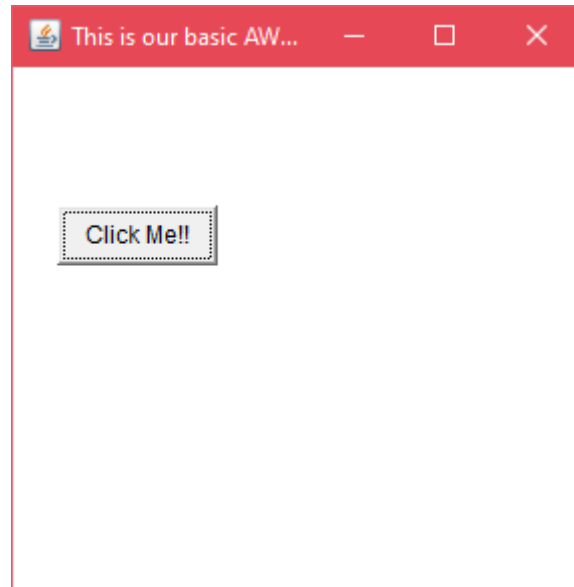
    // main method
    public static void main(String args[]) {

        // creating instance of Frame class
```

```
AWTExample1 f = new AWTExample1();  
}  
}
```

The `setBounds(int x-axis, int y-axis, int width, int height)` method is used in the above example that sets the position of the awt button.

### Output:



## AWT Example by Association

Let's see a simple example of AWT where we are creating instance of Frame class. Here, we are creating a TextField, Label and Button component on the Frame.

### AWTExample2.java

- `// importing Java AWT class`
- `import java.awt.*;`
- 
- `// class AWTExample2 directly creates instance of Frame class`
- `class AWTExample2 {`
- 
- `// initializing using constructor`
- `AWTExample2() {`
- 
- `// creating a Frame`



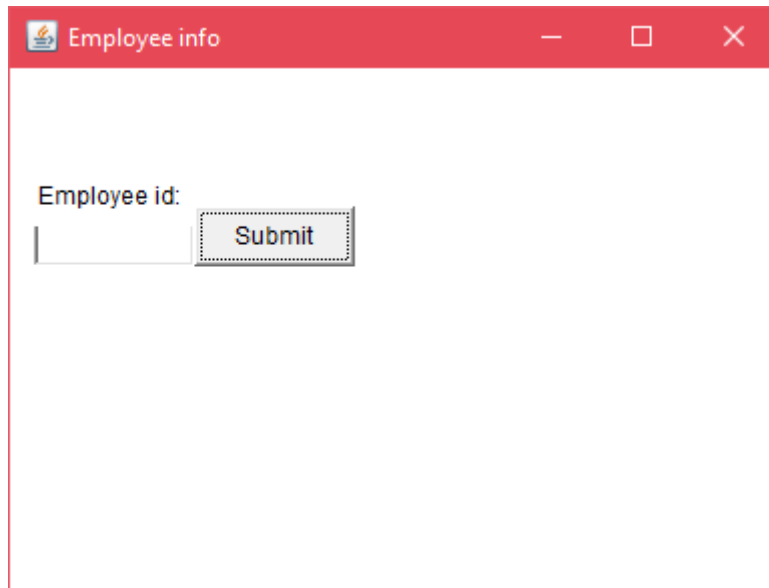
```

➤   Frame f = new Frame();
➤
➤   // creating a Label
➤   Label l = new Label("Employee id:");
➤
➤   // creating a Button
➤   Button b = new Button("Submit");
➤
➤   // creating a TextField
➤   TextField t = new TextField();
➤
➤   // setting position of above components in the frame
➤   l.setBounds(20, 80, 80, 30);
➤   t.setBounds(20, 100, 80, 30);
➤   b.setBounds(100, 100, 80, 30);
➤   // adding components into frame
➤   f.add(b);
➤   f.add(l);
➤   f.add(t);
➤
➤   // frame size 300 width and 300 height
➤   f.setSize(400,300);
➤
➤   // setting the title of frame
➤   f.setTitle("Employee info");
➤
➤   // no layout
➤   f.setLayout(null);
➤
➤   // setting visibility of frame
➤   f.setVisible(true);
➤ }
➤
➤ // main method
➤ public static void main(String args[]) {
➤
➤   // creating instance of Frame class

```

- `AWTExample2 awt_obj = new AWTExample2();`
- 
- `}`
- 
- `}`
- [download this example](#)

**Output:**



The screenshot shows a Java AWT window titled "Employee info" with a red title bar. Inside the window, there is a label "Employee id:" followed by a text input field and a "Submit" button. The button has a dotted border and is positioned to the right of the input field.