# Java AWT

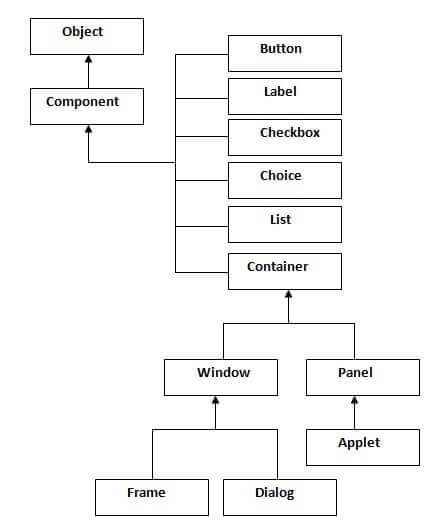
**Java AWT** (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt [package](https://www.javatpoint.com/package) provides [classes](https://www.javatpoint.com/object-and-class-in-java) for AWT api such as [TextField](https://www.javatpoint.com/java-awt-textfield), [Label](https://www.javatpoint.com/java-awt-label), [TextArea](https://www.javatpoint.com/java-awt-textarea), RadioButton, [CheckBox](https://www.javatpoint.com/java-awt-checkbox), [Choice](https://www.javatpoint.com/java-awt-choice), [List](https://www.javatpoint.com/java-awt-list) etc.

### Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



# Delegation Event Model in Java

The Delegation Event model is defined to handle events in GUI [programming languages](https://www.javatpoint.com/programming-language). The [GUI](https://www.javatpoint.com/gui-full-form) 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.**

**Import java.util.EventHandling**

The modern approach to handling events is based on the *delegation event model,* which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a *source* generates an event and sends it to one or more *listeners.* In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application 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.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the original Java 1.0 approach. Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

The following sections define events and describe the roles of sources and listeners.

|  |  |
| --- | --- |
| **Event Source** | * **GUI component that generates the event** * **Example: button** |
| **Event Listener/Handler** | * **Receives and handles events** * **Contains business logic** * **Example: displaying information useful to the user, computing a value.** |
| **Event object** | * **Created when an event occurs (i.e., user interacts with a GUI component).** * **Contains all necessary information about the event that has occurred** * **Type of event that has occurred Source of the event represented by an Event class** |

**Events**

In the delegation model, **an *event* is** an object that describes a state change in a source. Among other causes, an event can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that **cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.** Many other user operations could also be cited as examples.

Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed. You are free to define events that are appropriate for your application.

**Event Sources**

A *source* is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.

A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void add*Type*Listener (*Type*Listener *el*)

Here, ***Type***is the name of the event, and ***el***is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as *multicasting* the event. In all cases, notifications are sent only to listeners that register to receive them.

Some sources may allow only one listener to register. The general form of such a method is this:

public void add*Type*Listener(*Type*Listener *el*) throws java.util.TooManyListenersException

Here, *Type* is the name of the event, and *el* is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as *unicasting* the event.

A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void remove*Type*Listener(*Type*Listener *el*)

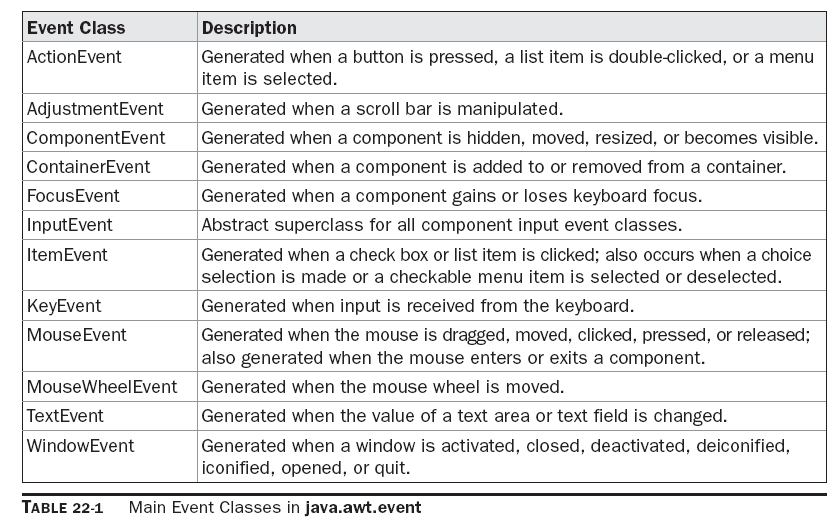
Here, *Type* is the name of the event, and *el* is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**.

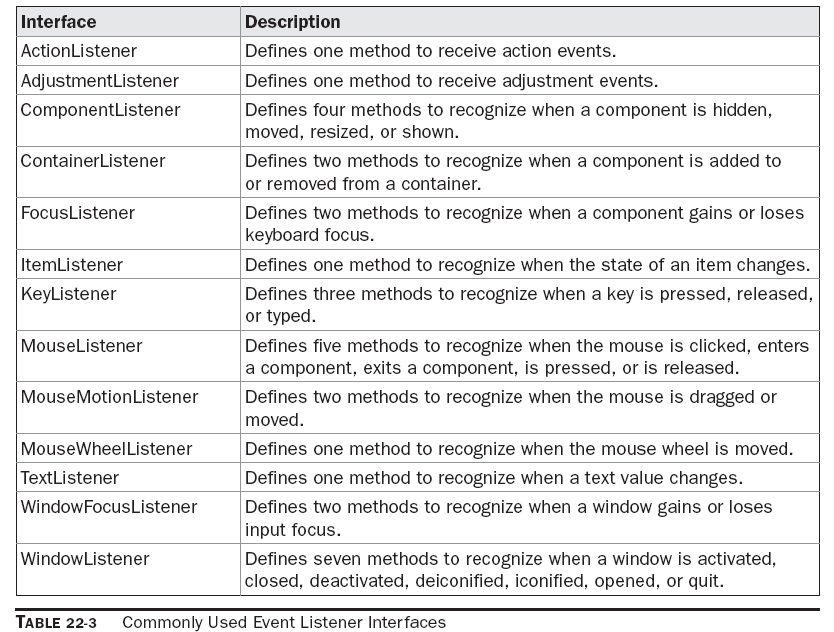
The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

**Event Listeners**

A *listener* is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.

The methods that receive and process events are defined in a set of interfaces, such as those found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. Other listener interfaces are discussed later in this and other chapters.





The event delegation model contains two main components. First are the event sources and second are the listeners. Most of the listener interfaces are available in the java.awt.event package. In Java, there are several event listener interfaces which are listed below:

**ActionListener**

This interface deals with the action events. Following is the event handling method available in the ActionListener interface:

void actionPerformed(ActionEvent ae)

**AdjustmentListener**

This interface deals with the adjustment event generated by the scroll bar. Following is the event handling method available in the AdjustmentListener interface:

void adjustmentValueChanged(AdjustmentEvent ae)

**ComponentListener**

 Components :**Text field ,button,label,radiobutton etc.**

This interface deals with the component events. Following are the event handling methods available in the ComponentListener interface:

void componentResized(ComponentEvent ce)

void componentMoved(ComponentEvent ce)

void componentShown(ComponentEvent ce)

void componentHidden(ComponentEvent ce)

**ContainerListener**

**Text field ,button,label,radiobutton etc. inside frame : applet**

This interface deals with the events that can be generated on containers. Following are the event handling methods available in the ContainerListener interface:

void componentAdded(ContainerEvent ce)

void componentRemoved(ContainerEvent ce)

**FocusListener**

This interface deals with focus events that can be generated on different components or containers. Following are the event handling methods available in the FocusListener interface:

void focusGained(FocusEvent fe)

void focusLost(FocusEvent fe)

**ItemListener**

This interface deals with the item event. Following is the event handling method available in the ItemListener interface:

void itemStateChanged(ItemEvent ie)

**KeyListener**

This interface deals with the key events. Following are the event handling methods available in the KeyListener interface:

void keyPressed(KeyEvent ke)

void keyReleased(KeyEvent ke)

void keyTyped(KeyEvent ke)

**MouseListener**

This interface deals with five of the mouse events. Following are the event handling methods available in the MouseListener interface:

void mouseClicked(MouseEvent me)

void mousePressed(MouseEvent me)

void mouseReleased(MouseEvent me)

void mouseEntered(MouseEvent me)

void mouseExited(MouseEvent me)

**MouseMotionListener**

This interface deals with two of the mouse events. Following are the event handling methods available in the MouseMotionListener interface:

void mouseMoved(MouseEvent me)

void mouseDragged(MouseEvent me)

**MouseWheelListener**

This interface deals with the mouse wheel event. Following is the event handling method available in the MouseWheelListener interface:

void mouseWheelMoved(MouseWheelEvent mwe)

**TextListener**

This interface deals with the text events. Following is the event handling method available in the TextListener interface:

void textValueChanged(TextEvent te)

**WindowFocusListener**

This interface deals with the window focus events. Following are the event handling methods available in the WindowFocusListener interface:

void windowGainedFocus(WindowEvent we)

void windowLostFocus(WindowEvent we)

**WindowListener**

This interface deals with seven of the window events. Following are the event handling methods available in the WindowListener interface:

void windowActivated(WindowEvent we)

void windowDeactivated(WindowEvent we)

void windowIconified(WindowEvent we)

void windowDeiconified(WindowEvent we)

void windowOpened(WindowEvent we)

void windowClosed(WindowEvent we)

void windowClosing(WindowEvent we)

**Mouse Event Handling**

In order to handle mouse events, Mouse Listener has to be defined. Mouse listener is the handler which will handle mouse events. There are two types of mouse listener  
  
**1. MouseListener**  
This listener is used to track click,press, release, entered and exited events.  
  
**2. MouseMotionListener**  
This listener is used to track mouse movements, drag movements etc.

### Methods of MouseListener Interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void mouseClicked() | This method is called when mouse button will be clicked. Mouse click event actually composed of two event button pressing and releasing. |
| public void mousePressed() | This method is called when mouse button is pressed. |
| public void mouseReleased() | This method is called when a pressed mouse button is released. |
| public void mouseEntered() | This method is called when mouse pointer moved in area of component for which listener is registered. |
| public void mouseExited() | This method is called when mouse pointer is moved out from area of components for which listener is registered. |

### Example

First define mouse event handler by implementing MouseListener interface

import javax.swing.\*;

import java.awt.event.\*;

class mouseHandler implements MouseListener

{

public void mouseClicked(MouseEvent me)

{

System.out.println("Mouse Button is Clicked");

}

public void mousePressed(MouseEvent me)

{

System.out.println("Mouse Button is Pressed");

}

public void mouseReleased(MouseEvent me)

{

System.out.println("Mouse Button is Released");

}

public void mouseEntered(MouseEvent me)

{

System.out.println("Mouse Pointer Entered in Frame");

}

public void mouseExited(MouseEvent me)

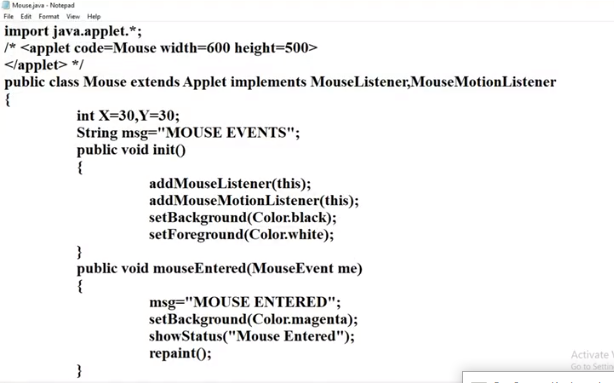
{

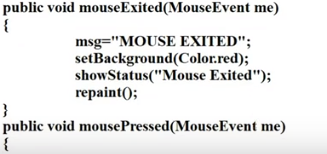
System.out.println("Mouse Pointer is Exited");

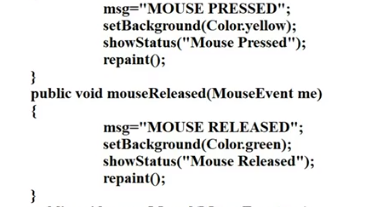
}

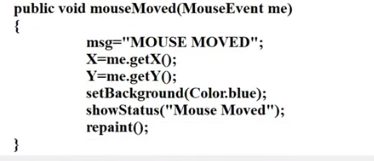
}

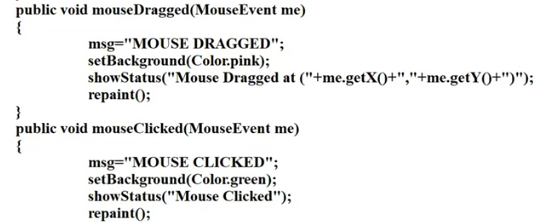
In above code MouseListener interface is implemented in class mouseHandler



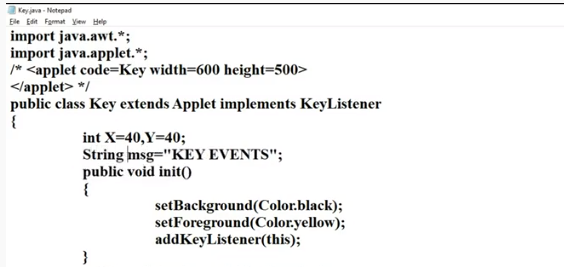


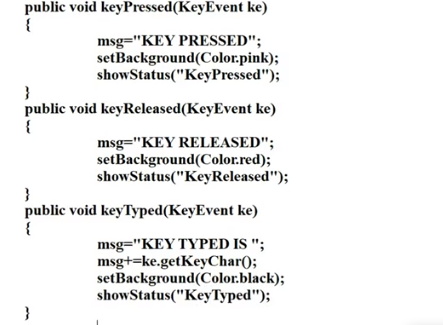


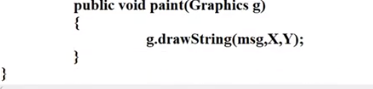




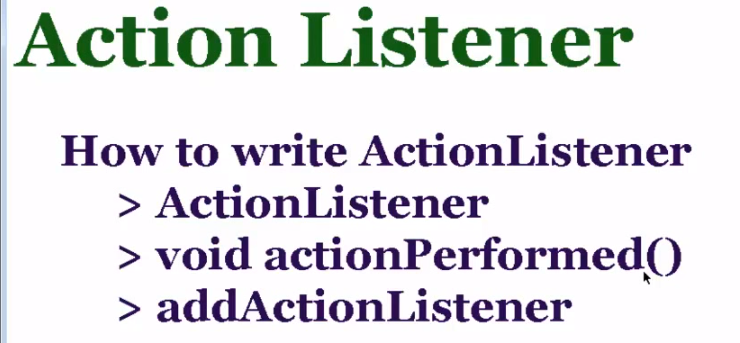
**Key event class and listener**









ActionListener in Java is a class that is responsible in handling all action events such as when the user clicks on a component. Mostly, action listeners are used for JButtons. An ActionListener can be used by the implements keyword to the class definition. It can also be use separately to the class by creating a new class that implements it. It should also be imported to your project. The method actionPerformed handles the all the actions, and also here, you will defined or write your own codes what will happened when an action occured.

### Implementing ActionListener in Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38 | package com.javapointers.javase;    import java.awt.BorderLayout;  import java.awt.event.ActionEvent;  import java.awt.event.ActionListener;  import javax.swing.JButton;  import javax.swing.JFrame;  import javax.swing.JTextArea;    public class ActionListenerTest implements ActionListener {        JButton button;      JFrame frame;      JTextArea textArea;        public ActionListenerTest() {          button = new JButton("Click Me");          frame = new JFrame("ActionListener Test");          textArea = new JTextArea(5, 40);            button.addActionListener(this);          textArea.setLineWrap(true);          frame.setLayout(new BorderLayout());          frame.add(textArea, BorderLayout.NORTH);          frame.add(button, BorderLayout.SOUTH);          frame.pack();            frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);          frame.setVisible(true);      }        @Override      public void actionPerformed(ActionEvent e) {          textArea.setText(textArea.getText().concat("You have clicked          the button\n"));      }        public static void main(String args[]) {          ActionListenerTest test = new ActionListenerTest();      }  } |

