1. EVENT HANDLING

Event handling in Java can be considered a paradigm in the interface that enables a listener in your applications and triggers responses based on the input received. An event could be a specific user action, such as pressing a key or clicking the mouse. Now when any of these specified actions take place, a set of pre-determined, pre-programmed actions will set into course. Event handling is used by programmers and developers to make the code more interactive and to optimize and, to a certain degree, even customize user experiences.

To understand the components of event handling in Java, you must focus on three aspects. They are the event source, the event handler, and the event listener. They are tasked with generating, capturing, and executing the set of codes when the event happens, respectively.

Event handling in Java can be defined as the process of designing and deploying user-interaction or system-action-sensitive code. It is a mechanism that aims to identify or "listen" for specific events such as a mouse click, button activations, etc. It differs from traditional programming in that event-driven programming is more user-centered, and the outcomes depend on the user's actions rather than a pre-designed sequence of codes.

Delegation Event Model

If you are in a team organizing an event, it is natural that different team members would be "delegated" different event responsibilities. In the same way, the delegation event model is an architecture of event handling in the Java paradigm that allows objects to delegate the responsibility of handling these events to other objects. The object to which the event is delegated is called the "listener object."

Registering the Source with Listener

Registering the source with a listener in the Delegation Event Model involves connecting the source object to the listener object. This allows the listener to receive and handle events generated by the source. This enables effective communication and event-driven behavior in Java.

How Are Events Handled?

Events are handled through a systematic process in Java. When an event occurs, the event source detects it and notifies the registered event listener. The listener, equipped with the appropriate event handling code, responds accordingly. This structured approach ensures seamless communication and enables applications to react promptly and accurately to user interactions.

Events in Java represent the change in the state of any object. Events occur when the user interacts with the interface. Clicking a button, moving the mouse, typing a character, selecting an item from a list, and scrolling the page are all examples of behaviors that cause an event to occur.

Types of Events in Java:

Foreground Events: These events necessitate the user's direct participation. They are produced as a result of a user interacting with graphical components in a Graphical User Interface.

Background Events: Background events are those that require end-user interaction. Operating system interrupts and hardware or software failures are examples of background events.

Event handling in Java is the process of controlling an event and taking appropriate action if one occurs.

Delegation event model:

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.

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.

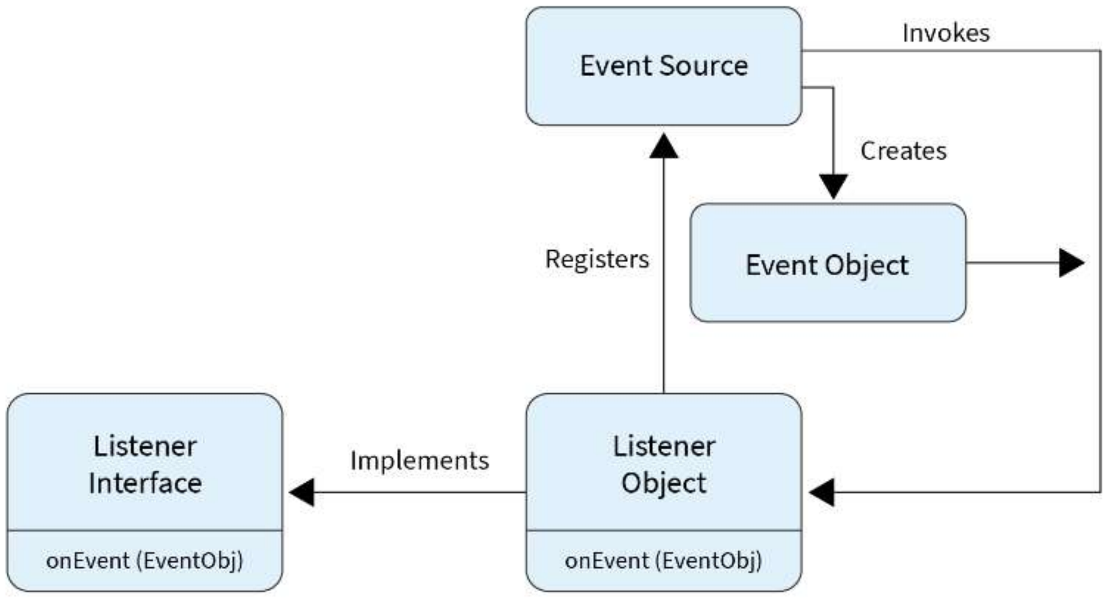


Fig 1: Delegation event model

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:

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.

Some listeners allow only one listener to register. Below is an example:

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.

```
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 MouseMotionListener interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

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.

2. ACTION EVENT AND ACTION LISTENER

An action event is a semantic event which indicates that a component-defined action occurred. - The ActionListener interface gets this ActionEvent when the event occurs.

- Event like Button pressed is an action event.

- It is defined in 'java.awt.event' package.

- This event is generated when the button is clicked or the item of a list is double clicked.

Declaration:

```
public class ActionEvent extends AWTEvent
```

Constructors of action event:

- ActionEvent(java.lang.Object source, int id, java.lang.String command)- Constructs an ActionEvent object.
- ActionEvent(java.lang.Object source, int id, java.lang.String command, int modifiers)- Constructs an ActionEvent object with modifier keys.
- ActionEvent(java.lang.Object source, int id, java.lang.String command, long when, int modifiers)- Constructs an ActionEvent object with the specified modifier keys and timestamp.

Class methods

- java.lang.String getActionCommand()- Returns the command string associated with this action.
- int getModifiers()-Returns the modifier keys held down during this action event.
- long getWhen()- Returns the timestamp of when this event occurred.
- java.lang.String paramString()- Returns a parameter string identifying this action event.

Action listener:

Java ActionListener is an interface in java.awt.event package. It is an type of class in Java that receives a notification whenever any action is performed in the application. Java ActionListener is alerted whenever the button or menu item is clicked. It is alerted against ActionEvent. Java ActionListener interface consists of only one method. The following syntax can declare the Java ActionListener interface:

```
public class ActionListenerExample Implements ActionListener
```

Method of Java ActionListener

Whenever a button or menu is clicked, an object must be present in the program, which helps in implementing the interface. An event will be generated by clicking the menu or button; for this, we will use some methods. Java ActionListener consists of only one method, which is as follows:

- String getActionCommand()-Returns the string that corresponds to this action. The setActionCommand method, supported by most objects that may fire action events, enables you to set this string.
- int getModifiers()-The button the user pushed during the action event is returned as an integer. The keys pressed are identified using a few ActionEvent-defined constants, such as SHIFT MASK, CTRL MASK, META MASK, and ALT MASK. For instance, the expression is nonzero if a user chooses a menu item.
- Object getSource()(in java.util.EventObject) - returns the event's event-firing object.

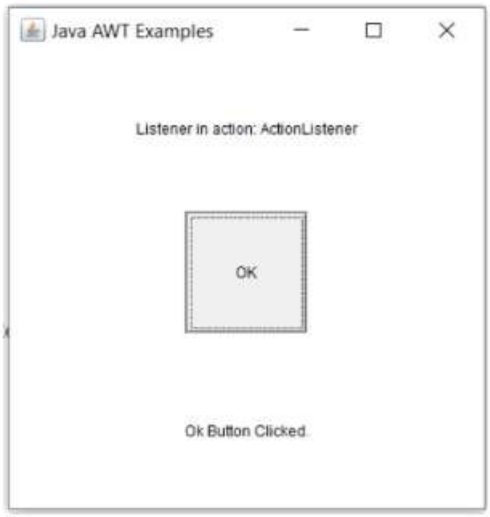
Writing action listener

- Implement ActionListener Interface: Create a class that implements the ActionListener interface.
- Override actionPerformed Method: Implement the actionPerformed(ActionEvent e) method within the class. This method contains the code to execute when the action occurs.
- Register ActionListener: Register the ActionListener to the appropriate GUI component using the addActionListener() method.

- Handle Action Events: Write the logic inside the actionPerformed() method to handle the action events triggered by the user interaction with the GUI component.
- Perform Desired Actions: Define the actions to be performed when the event occurs, such as updating GUI components, performing calculations, or invoking other methods.

```
import java.awt.*;
import java.awt.event.*;
public class AwtListenerDemo {
  private Frame mainFrame;
  private Label headerLabel;
  private Label statusLabel;
  private Panel controlPanel;
  public AwtListenerDemo(){
    prepareGUI();
  }
  public static void main(String[] args){
    AwtListenerDemo  awtListenerDemo = new AwtListenerDemo();
    awtListenerDemo.showActionListenerDemo();
  }
  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 showActionListenerDemo(){
      headerLabel.setText("Listener in action: ActionListener");
      ScrollPane panel = new ScrollPane();
      panel.setBackground(Color.magenta);
      Button okButton = new Button("OK");
      okButton.addActionListener(new CustomActionListener());
      panel.add(okButton);
      controlPanel.add(panel);
      mainFrame.setVisible(true);
   }
   class CustomActionListener implements ActionListener{
      public void actionPerformed(ActionEvent e) {
         statusLabel.setText("Ok Button Clicked.");
      }
   }
}
```



3. ITEM EVENT AND LISTENER

ItemListener is an interface that listens for the item event, basically it refers to the item selection. The ItemListener interface mainly maintains an on/off state for one or more items.

Methods used in ItemListener Interface

- itemStateChanged(ItemEvent e)- This method is called when the item is selected or deselected by the user.
- getState()- It is a predefined method of the checkbox class, using this method we check whether the checkbox is selected or not. If the checkbox is selected then it returns true otherwise it returns false.

Packages Imported

java.awt.*;

AWT stands for Abstract Window Toolkit, it is a package in Java that is imported to provide a graphical user interface to programs. The AWT classes falls under the following categories.

Containers

The containers consist of the following things:

- Frame
- Dialog
- Window
- Panel
- ScrollPane
- Components

The Components of AWT are as follows:

- Button
- Checkbox
- Choice
- List
- Label
- TextArea
- PopupMenu
- LayoutManager

The LayoutManager has the following things:

- FlowLayout
- Borderlayout
- GridLayout
- Gridbaglayout
- Cardlayout

java.awt.event.*;

This package is imported to handle various types of events fired by AWT components. Events are basically fired by event sources.

Item Listener

ItemListener mainly used for item selection, and it is an interface that listens for the item event. The interface ItemListener basically manages the on or off status for one or several items. The Java ItemListener notifies each and every click on the checkbox, and it let you know against the ItemEvent. The package used for the ItemListener was found in the java.awt.event. The method used by the ItemListener was ItemStateChanged() and getState().

Methods used in ItemListener Interface

- itemStateChanged(ItemEvent e): The ItemStateChanged(ItemEvent e) method is called only when the item is checked or unchecked on the user's registered checkbox component.
- getState(): The getState() method is the predefined method for the class of checkbox, by using this getState() method we can check whether the checkbox selected. It returns the result as true when the checkbox selected, and it returns false.
- Java ItemListener will use the method itemStateChanged(); this method will invoke the registered checkbox component at every check/uncheck and the syntax as follows,
- void itemStateChanged(ItemEvent e): The method triggered only when an item been clicked or un-licked by the user.

Working:

It implements an interface called ItemListener, by setting implementation of its method as follows,

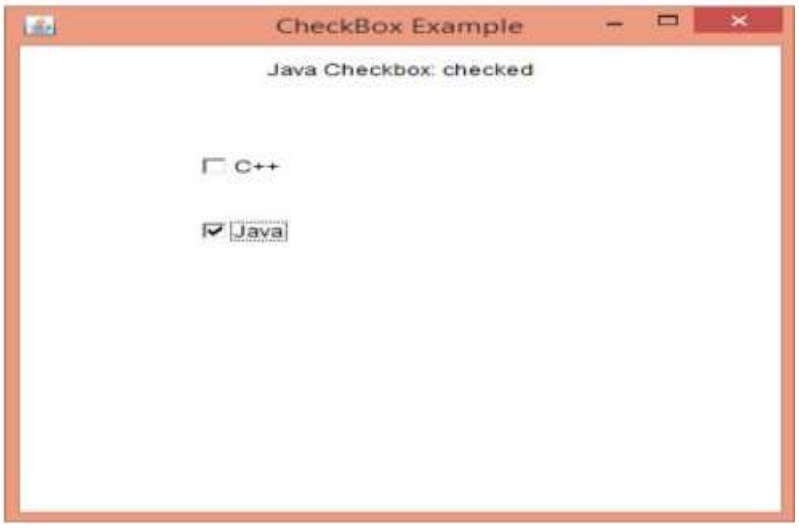
- ItemStateChanged(ItemEvent e): the method will be invoked only when an item been selected or deselected by the user. The ItemEvent will be even a checkbox, or it may be a list, the event source will make a call of its method by
- addItemListener(ItemListener e), where the object of the class implemented by its interface called ItemListener whenever a click made on the specific source it registers the class corresponds to the ItemEvent. The AWT package was imported to provide the GUI (Graphical User Interface) for the program.

The ItemEvent class has several methods:

- public Object getItem(): Which returns the item when it was triggered or clicked the ItemEvent.

- public ItemSelectable getItemSelectable(): Which returns when an item been clicked.
- public int getStateChange(): Which returns the current/latest state of the item been clicked.

```
import java.awt.*;
import java.awt.event.*;
public class ItemListenerExample implements ItemListener{
    Checkbox checkBox1,checkBox2;
    Label label;
    ItemListenerExample(){
        Frame f= new Frame("CheckBox Example");
        label = new Label();
        label.setAlignment(Label.CENTER);
        label.setSize(400,100);
        checkBox1 = new Checkbox("C++");
        checkBox1.setBounds(100,100, 50,50);
        checkBox2 = new Checkbox("Java");
        checkBox2.setBounds(100,150, 50,50);
        f.add(checkBox1); f.add(checkBox2); f.add(label);
        checkBox1.addItemListener(this);
        checkBox2.addItemListener(this);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
     }
    public void itemStateChanged(ItemEvent e) {
        if(e.getSource()==checkBox1)
            label.setText("C++ Checkbox: "
            + (e.getStateChange()==1?"checked":"unchecked"));
        if(e.getSource()==checkBox2)
        label.setText("Java Checkbox: "
        + (e.getStateChange()==1?"checked":"unchecked"));
     }
public static void main(String args[])
{
    new ItemListenerExample();
}
}
```



Java Checkbox: checked

4. KEY EVENTS AND LISTENER

On entering the character the Key event is generated.There are three types of key events which are represented by the integer constants. These key events are following

- KEY_PRESSED
- KEY_RELASED
- KEY_TYPED

Declaration:

```
public class KeyEvent    extends InputEvent
```

Constructors:

- KeyEvent(Component source, int id, long when, int modifiers, int keyCode)- Deprecated. as of JDK1.1
- KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar)- Constructs a KeyEvent object.
- KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar, int keyLocation)-

Class methods

- char getKeyChar()- Returns the character associated with the key in this event.
- int getKeyCode()- Returns the integer keyCode associated with the key in this event.
- int getKeyLocation()- Returns the location of the key that originated this key event.
- static String getKeyModifiersText(int modifiers)- Returns a String describing the modifier key(s), such as "Shift", or "Ctrl+Shift".
- static String getKeyText(int keyCode)- Returns a String describing the keyCode, such as "HOME", "F1" or "A".
- boolean isActionKey()- Returns whether the key in this event is an "action" key.
- String paramString()-Returns a parameter string identifying this event.
- void setKeyChar(char keyChar)- Set the keyChar value to indicate a logical character.
- void setKeyCode(int keyCode)- Set the keyCode value to indicate a physical key.
- void setModifiers(int modifiers)-Deprecated. as of JDK1.1.4

Key Listener:

Java KeyListener is an interface in java.awt.event package. Java KeyListener operates the event whenever the state of the Key is changed. The class that processes the KeyEvent implement the Java KeyListener.Java KeyListener interface consists of 3 Methods.

The following syntax can declare the java KeyListener interface:

```
public interface KeyListener extends EventListener
```

Methods of Java KeyListener

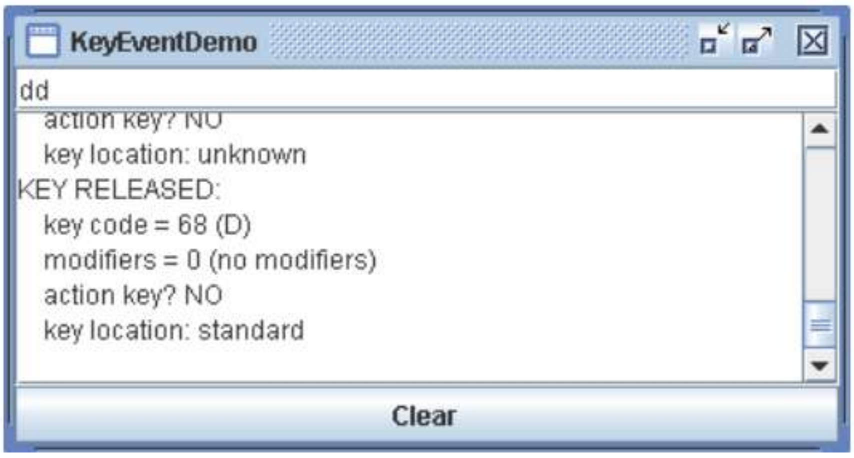
Whenever the state of the key is changed, an object must be present in the program, which helps in implementing the interface. An event will be generated by typing the key, pressing the key, or releasing the key; for this, we will use some methods. Java MouseMotionListener consists of three methods, which are as follows:

- public abstract void keyPressed(KeyEvent e)   - It is initiated when a key has been pressed.
- public abstract void keyTyped(KeyEvent e)- It is initiated when a key has been typed.
- public abstract void keyReleased(KeyEvent e)- It is initiated when a key has been released.

```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

import javax.swing.JButton;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
```

```
public class KeyEventDemo extends JPanel implements KeyListener, ActionListener
{
  JTextArea displayArea;

  JTextField typingArea;

  static final String newline = "\n";

  public KeyEventDemo() {
    super(new BorderLayout());

    JButton button = new JButton("Clear");
    button.addActionListener(this);

    typingArea = new JTextField(20);
    typingArea.addKeyListener(this);

    //Uncomment this if you wish to turn off focus
    //traversal. The focus subsystem consumes
    //focus traversal keys, such as Tab and Shift Tab.
    //If you uncomment the following line of code, this
    //disables focus traversal and the Tab events will
    //become available to the key event listener.
    //typingArea.setFocusTraversalKeysEnabled(false);

    displayArea = new JTextArea();
    displayArea.setEditable(false);
    JScrollPane scrollPane = new JScrollPane(displayArea);
    scrollPane.setPreferredSize(new Dimension(375, 125));

    add(typingArea, BorderLayout.PAGE_START);
    add(scrollPane, BorderLayout.CENTER);
  add(button, BorderLayout.PAGE_END);
}

/** Handle the key typed event from the text field. */
public void keyTyped(KeyEvent e) {
  displayInfo(e, "KEY TYPED: ");
}

/** Handle the key pressed event from the text field. */
public void keyPressed(KeyEvent e) {
  displayInfo(e, "KEY PRESSED: ");
}

/** Handle the key released event from the text field. */
public void keyReleased(KeyEvent e) {
  displayInfo(e, "KEY RELEASED: ");
}

/** Handle the button click. */
public void actionPerformed(ActionEvent e) {
  //Clear the text components.
  displayArea.setText("");
  typingArea.setText("");

  //Return the focus to the typing area.
  typingArea.requestFocusInWindow();
}

/*
 * We have to jump through some hoops to avoid trying to print non-printing
 * characters such as Shift. (Not only do they not print, but if you put
 * them in a String, the characters afterward won't show up in the text
 * area.)
 */
protected void displayInfo(KeyEvent e, String s) {
  String keyString, modString, tmpString, actionString, locationString;

  //You should only rely on the key char if the event
  //is a key typed event.
  int id = e.getID();
  if (id == KeyEvent.KEY_TYPED) {
    char c = e.getKeyChar();
    keyString = "key character = '" + c + "'";
  } else {
    int keyCode = e.getKeyCode();
    keyString = "key code = " + keyCode + " ("
        + KeyEvent.getKeyText(keyCode) + ")";
  }

  int modifiers = e.getModifiersEx();
  modString = "modifiers = " + modifiers;
  tmpString = KeyEvent.getModifiersExText(modifiers);
  if (tmpString.length() > 0) {
    modString += " (" + tmpString + ")";
  } else {
    modString += " (no modifiers)";
  }

  actionString = "action key? ";
  if (e.isActionKey()) {
    actionString += "YES";
  } else {
    actionString += "NO";
  }
  locationString = "key location: ";
  int location = e.getKeyLocation();
  if (location == KeyEvent.KEY_LOCATION_STANDARD) {
   locationString += "standard";
  } else if (location == KeyEvent.KEY_LOCATION_LEFT) {
   locationString += "left";
  } else if (location == KeyEvent.KEY_LOCATION_RIGHT) {
   locationString += "right";
  } else if (location == KeyEvent.KEY_LOCATION_NUMPAD) {
   locationString += "numpad";
  } else { // (location == KeyEvent.KEY_LOCATION_UNKNOWN)
   locationString += "unknown";
  }

  displayArea.append(s + newline + "    " + keyString + newline + "    "
     + modString + newline + "    " + actionString + newline
     + "    " + locationString + newline);
  displayArea.setCaretPosition(displayArea.getDocument().getLength());
}

  private static void createAndShowGUI() {
  //Make sure we have nice window decorations.
  JFrame.setDefaultLookAndFeelDecorated(true);

  //Create and set up the window.
  JFrame frame = new JFrame("KeyEventDemo");
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  //Create and set up the content pane.
  JComponent newContentPane = new KeyEventDemo();
  newContentPane.setOpaque(true); //content panes must be opaque
```

```
    frame.setContentPane(newContentPane);

    //Display the window.
    frame.pack();
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    //Schedule a job for the event-dispatching thread:
    //creating and showing this application's GUI.
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createAndShowGUI();
      }
    });
  }
}
```



```
dd
    action key? NO
    key location: unknown
KEY RELEASED:
    key code = 68 (D)
    modifiers = 0 (no modifiers)
    action key? NO
    key location: standard
```

5. TEXT EVENT

TextEvent class and TextListener interface are associated with both textfield and textarea. If we want to performed any operation, if the value of textfield and textarea are changed than the logic should be written in the textValueChanged(). Text events are sent to text components(text field and text areas) when a change occurs to the text they contain. This happens when the user types something or when a program executes a method such as a setText().

```
public class TextEvent extends AWTEvent
{
 public static final int TEXT_VALUE_CHANGED;
   public TextEvent(Object source, int id);
   public String paramString();
}
```

A text component can process its own text events by calling enableEvents(AWTEvent.TEXT_EVENT_MASK) and providing a processTextEvent() method. Alternatively, a text component can delegate text events to a listener that implements the TextListener interface:

```
public interface TextListener extends EventListener
{
 public void textValueChanged(TextEvent te);
}
```

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

class TextEventEx extends Frame implements TextListener
{
TextField tf1,tf2,tf3;
Label l1,l2,l3;
String nm="",xc="",yc="";
int x,y;
Panel p1;
TextEventEx()
{
setBackground(Color.cyan);
p1=new Panel();
p1.setBackground(Color.yellow);
l1=new Label("Name");
l2=new Label("X-axies");
l3=new Label("y-aixes");

tf1=new TextField(15);
tf2=new TextField(5);
tf3=new TextField(5);

p1.add(l1);
p1.add(tf1);
p1.add(l2);
p1.add(tf2);
p1.add(l3);
p1.add(tf3);
add("North",p1);

tf1.addTextListener(this);
tf2.addTextListener(this);
tf3.addTextListener(this);
}

public void textValueChanged(TextEvent e)
{
nm=tf1.getText();
xc=tf2.getText();
x=Integer.parseInt(xc);
yc=tf3.getText();
y=Integer.parseInt(yc);
repaint();
}
```

```
public  void paint(Graphics g)
{
setFont(new Font("TimesRoman",Font.BOLD,30));
g.setColor(Color.red);
g.drawString("Name"+nm,x,y);
}

public static void main(String args[])
{
TextEventEx t=new TextEventEx();
Toolkit tx=Toolkit .getDefaultToolkit();
t.setSize(tx.getScreenSize());
t.setVisible(true);
}
}
```

6. WINDOW EVENT

When window-related actions like closing, activating, or deactivating a window are initiated, a window event is triggered. WindowEvent class is used to generate objects that represent window events. When we alter the state of a window, the Java WindowListener gets informed. It is also notified when a WindowEvent occurs. The java.awt.event package contains the WindowListener interface.

Methods of window event:

- WindowEvent(Window source, int id)- Constructs a WindowEvent object.
- WindowEvent(Window source, int id, int oldState, int newState)- Constructs a WindowEvent object with the specified previous and new window states.
- WindowEvent(Window source, int id, Window opposite)- Constructs a WindowEvent object with the specified opposite Window.
- WindowEvent(Window source, int id, Window opposite, int oldState, int newState)- Constructs a WindowEvent object.

Event Handling Interfaces

Window events in Java are typically handled using the following interfaces:

- WindowListener: The interface defines methods to handle various window-related events, such as window opening, closing, activation, deactivation, and more. It includes the following methods:
    - windowOpened(WindowEvent e): Invoked when the window is first opened or displayed.
    - windowClosing(WindowEvent e): Triggered when the user attempts to close the window.
    - windowClosed(WindowEvent e): Sent after the window has been closed.
    - windowIconified(WindowEvent e): Occurs when the window is minimized.
    - windowDeiconified(WindowEvent e): Triggered when the window is restored from a minimized state.
    - windowActivated(WindowEvent e): Fired when the window receives focus and becomes active.
    - windowDeactivated(WindowEvent e): Sent when the window loses focus and becomes inactive.
- WindowAdapter:

It is an abstract class that implements the WindowListener interface. It provides default empty implementations for all the methods of the WindowListener interface. We can extend WindowAdapter and override only the methods you need, making it convenient for handling specific window events without implementing all the methods.

Working of WindowListener interface

The WindowListener interface enables a class to handle various window-related events effectively. To process these events, an object must be available that implements this interface. Once this object is registered as a listener, it becomes capable of responding to window events that occur during different states of a window's lifecycle.

By registering this object as a listener, events are automatically generated in response to user actions on the window, such as opening, closing, resizing, or moving it. These events trigger the corresponding methods in the listener's object, allowing developers to define custom actions in response to these events.

After the relevant method in the listener's object is invoked, a WindowEvent is generated, encapsulating information about the event. The event object can be examined and further processed by the developer as needed, providing a structured way to interact with and respond to window-related actions in Java applications.

Registering Window Listeners

To handle window events, you need to register a WindowListener or a subclass of WindowAdapter to a window component (usually a JFrame, JDialog, or JWindow). You can do this using the addWindowListener(WindowListener l) method.

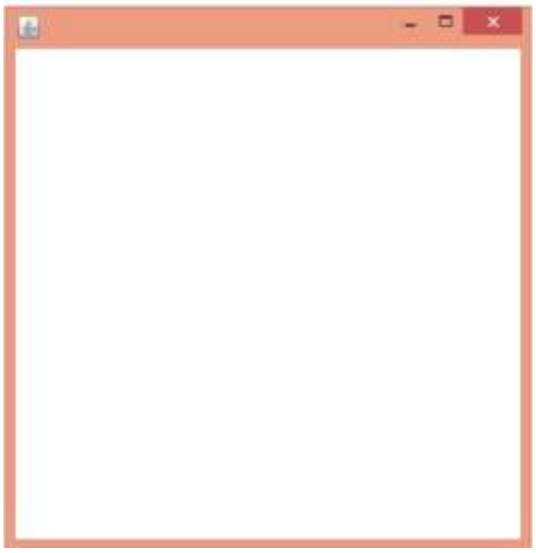
```
frame.addWindowListener(new WindowAdapter() {
   // Override window event methods as needed
});
```

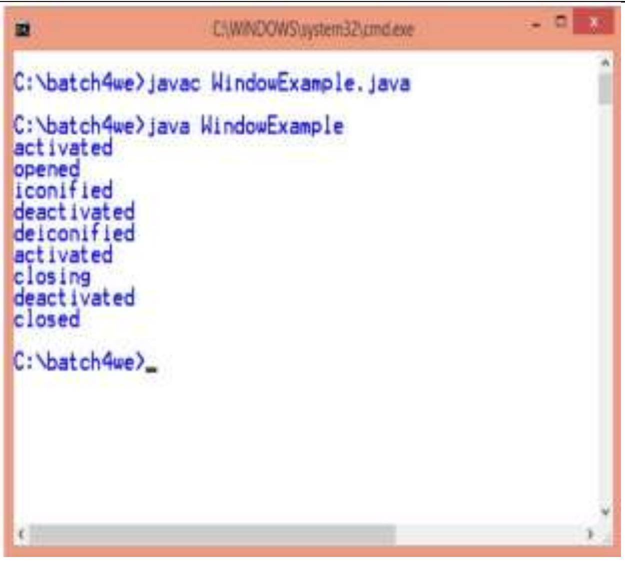
Handling Window Events in Java

To handle window events in Java, you need to implement the WindowListener or WindowAdapter interface. The WindowAdapter class provides default empty implementations for all methods of the WindowListener interface, allowing us to override only the methods relevant to your application.

```
// importing necessary libraries of awt
import java.awt.*;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;

// class which inherits Frame class and implements WindowListener interface
```

```
public class WindowExample extends Frame implements WindowListener {

// class constructor
   WindowExample() {

   // adding WindowListener to the frame
      addWindowListener(this);
      // setting the size, layout and visibility of frame
      setSize (400, 400);
      setLayout (null);
      setVisible (true);
   }
// main method
public static void main(String[] args) {
   new WindowExample();
}

// overriding windowActivated() method of WindowListener interface which prints
the given string when window is set to be active
public void windowActivated (WindowEvent arg0) {
   System.out.println("activated");
}

// overriding windowClosed() method of WindowListener interface which prints the
given string when window is closed
public void windowClosed (WindowEvent arg0) {
   System.out.println("closed");
}

// overriding windowClosing() method of WindowListener interface which prints the
given string when we attempt to close window from system menu
public void windowClosing (WindowEvent arg0) {
    System.out.println("closing");
    dispose();
}

// overriding windowDeactivated() method of WindowListener interface which
prints the given string when window is not active
public void windowDeactivated (WindowEvent arg0) {
    System.out.println("deactivated");
}

// overriding windowDeiconified() method of WindowListener interface which prints
the given string when window is modified from minimized to normal state
public void windowDeiconified (WindowEvent arg0) {
    System.out.println("deiconified");
}

// overriding windowIconified() method of WindowListener interface which prints
the given string when window is modified from normal to minimized state
public void windowIconified(WindowEvent arg0) {
    System.out.println("iconified");
}

// overriding windowOpened() method of WindowListener interface which prints
the given string when window is first opened
public void windowOpened(WindowEvent arg0) {
    System.out.println("opened");
}
}
```





```
C:\batch4we>javac WindowExample.java

C:\batch4we>java WindowExample
activated
opened
iconified
deactivated
deiconified
activated
closing
deactivated
closed

C:\batch4we>_
```