Event handling

Event handling in Java allows developers to create interactive applications by responding to user inputs and other occurrences. The Java programming language provides a robust event-handling model primarily through the Abstract Window Toolkit (AWT) and Swing libraries.

Key words

Events Sources
Events
Event Objects
Event Listeners:
Event Registration

1. Event Sources:
   Definition: Objects that generate events (e.g., buttons, text fields, windows).
2. Events:
   Definition: Occurrences that signify user actions or system changes (e.g., button clicks, mouse movements). That is, change of state of Sources will create Events., Button state chenged from Unclicked to Clicked.

   Eg Types:
   ActionEvents: for Button clicks.
   MouseEvents: for Mouse actions (clicks, movements).
   KeyboardEvents: for Key presses/releases.

3. Event Objects:
   Definition: Objects that encapsulate or describe event details when an event occurs.

   Event details in object:
   Source: The object that generated the event.
   Event Type: The kind of event (e.g., action, mouse).

   Usage: Passed to event-handling methods to determine how to respond.

4. Event Listeners:
   Definition: Interfaces defining methods to handle specific events.

   Examples:
   ActionListener: For action events.
   MouseListener: For mouse events.
   KeyListener:
   For keyboard events.

5. Event Registration: **Listeners** must be registered with **Event sources** to receive **notifications** when specific events occur.

Example: How to program button click.

```
/*--- implemented as anonumous innrer class ---*/
ActionListener listener = new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
                // add code here on what is expected on clicking button
```

```
        }
};

button = new Button("Click Me");
button.addActionListener(listener);
```

button: This is the event source (the object generating the event).
listener: This is an instance of a class that implements the **ActionListener** interface, which defines the method to handle action events (like button clicks).

In this example, when the button is clicked, the registered listener will receive an event notification, allowing it to execute the code defined in it.

This registration mechanism is fundamental to Java's event-handling model, enabling responsive interactions in GUI applications.

By using event sources, listeners, and event objects, Java provides a structured way to manage and respond to events, making it a powerful tool for GUI development.

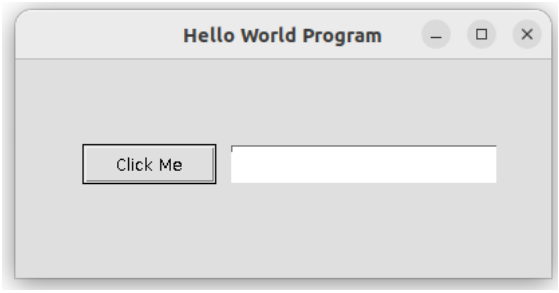
Sample program: Helow World GUI
A button click should print Hello World

- Below is the code for GUI withoout handling the Button click.

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

public class HelloWorld {
    JFrame frame;
    JButton button;
    JTextField textField;

    HelloWorld() {
        frame = new JFrame("Hello World Program");
        frame.setLayout(null);

        // Initialize the button and text field
        button = new JButton("Click Me");
        textField = new JTextField();

        // Position the gui with (x, y, width, height)
        button.setBounds(50, 100, 100, 30);
        textField.setBounds(160, 100, 200, 30);

        // Add components to the frame
        frame.add(button);
        frame.add(textField);

        // Set the size of the frame
        frame.setSize(400, 200);
        frame.setVisible(true);
```

```
                // handle window closing events
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
        public static void main(String[] args) {
                HelloWorld test = new HelloWorld();
        }
}
```

Output



Enhancement to do Button Click

The below highlighted code are added to handle Button click feature

1. implemet ActionListener interface, which is the interface for Button click event
   `public class HelloWorld implements ActionListener { }`
2. write function public void actionPerformed(ActionEvent e)

3. add logic to be implemented in button click in above function

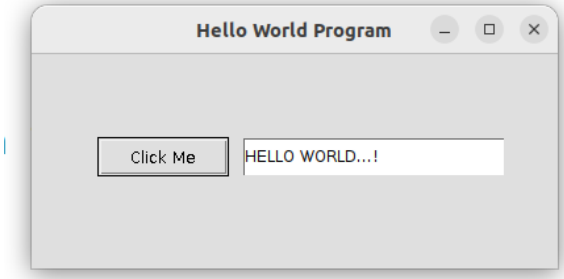
```
public void actionPerformed(ActionEvent e) {
        if (e.getSource() == button) {
                textField.setText("HELLO WORLD ...!");
        }
}
```

Complete Program

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

public class HelloWorld implements ActionListener {
        JFrame frame;
        JButton button;
        JTextField textField;
        HelloWorld() {
                frame = new JFrame("Hello World Program");
                frame.setLayout(null);

                // Initialize the button and text field
                button = new JButton("Click Me");
                textField = new JTextField();

                // Position the gui with (x, y, width, height)
                button.setBounds(50, 100, 100, 30);
                textField.setBounds(160, 100, 200, 30);
```

```
                // Add components to the frame
                frame.add(button);
                frame.add(textField);

                // register event to button
                button.addActionListener(this);

                // Set the size of the frame
                frame.setSize(400, 200);
                frame.setVisible(true);

                // handle window closing events
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
        // abstract function declared in ActionListener interface
        // when user clicks the Button, control will come to this function...!
        public void actionPerformed(ActionEvent e) {
                if (e.getSource() == button) {
                        textField.setText("HELLO WORLD ...!");
                }
        }
        public static void main(String[] args) {
                HelloWorld test = new HelloWorld();
        }
}
```

Output



In above program

Events Sources : ***Button button;***
Events Event Objects : **ActionEvent**
Event Listeners: ***public void actionPerformed(ActionEvent e) {}***
Event Registration : ***button.addActionListener(this);***

Delegation Event Model

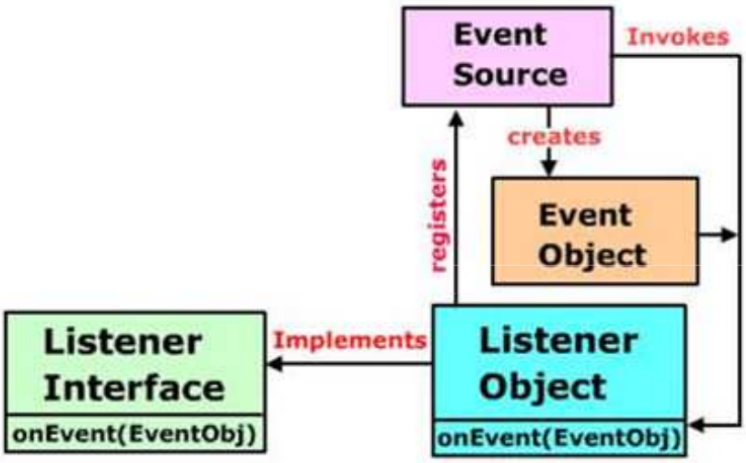
The Delegation Event Model in Java: An Overview

1. The Delegation Event Model is one of the most powerful mechanisms for handling events in Java, particularly within graphical user interface (GUI) applications built using frameworks such as **Swing** or **AWT**.
2. It provides a structured way for **separating the event generation and the handling of those events**.
3. At the core of this model lies the **delegation of responsibility for handling events from the source** (the object generating the event) **to one or more listeners** (the objects that handle the event).

What Is the Delegation Event Model?

1. In Java's **event-driven programming model**, interactions between the user and the application (such as clicking a button or moving the mouse) generate events.
2. These events need to be responded to, typically by updating the user interface, performing computations, or triggering actions in the program. (eg, clicking "Add" button in a Caculator application do addition operation and display in edit box)
3. The Delegation Event Model provides a mechanism to respond to these user-generated events by **delegating event handling to listener objects.**

How the Delegation Event Model Works



Three key components of the Delegation Event Model involves

1. **Event Source** – The object that generates an event. This could be a button, a text field, or any other component that the user interacts with., **Eg, a Button**
2. **Event Listener** – A predefined **interface** in Java that must be implemented by any object that wants to handle specific events generated by an event source. **Eg, ActionListener interface.** Below is the abstarct function declared in interface. This **must be** implemented in the class.

   public void actionPerformed(ActionEvent e) {}

3. **Event Object** – The object encapsulating the event details (e.g., what kind of event occurred, which component generated it, and when it happened). Eg **ActionEvent** in above interface function constructor

Its working involves.

1. The program/class must implement required listener interface. For eg, Button

Eg: *public class HelloWorld **implements ActionListener** {}*

```
// override interface function
public void actionPerformed(ActionEvent e) {
        if (e.getSource() == buttonObj) {
            textField.setText("HELLO WORLD ...!");
        }
}
```

2. A source must **register** listeners

eg: buttonObj.addActionListener(this)

3. When button is pressed, the button objects state changed from unclicked to clicked, thereby generating and Event.

4. A listener is an object that is notified when an event is generated.

- It has two major requirements.
  - First, it must have been registered with one or more sources to receive notifications about specific types ofevents.
  - Second, it must implement methods to receive and process these notifications..

Programmer will be wrting the business logic of button click in this function. Eventualy, in below example, *"HELLO WORLD ...!" will be printed on buton click.*

```
public void actionPerformed(ActionEvent e) {
        if (e.getSource() == buttonObj) {
            textField.setText("HELLO WORLD ...!");
        }
    }
```

Sample interfaces

ActionListener *- for button click*
MouseListener *- for mouse click, wheel movemnt*
KeyListener *- for keyboard press/release*

Sample events

`ActionEvent`
`MouseEvent`,
`KeyEvent`

Advantages of the Delegation Event Model

1. **Loose Coupling:** Event sources don't need to know how events are handled, only that a listener is registered. This leads to more modular, maintainable code.

2. **Extensibility**: New listeners can be added without changing the event source. Multiple listeners can handle the same event in different ways, supporting flexible designs.

3. **Reusability**: A single listener can handle events for multiple components, making it reusable across different parts of the application.

4. **Simplified Event Management**: Event handling is separated from the source, keeping the code clean and focused on the main functionality.

Event Listeners & Description

Source of Event	Action	Event	Event Listeners	
			Event Interface	**Implementing Functins**
Button	click	ActionEvent	ActionListener	actionPerformed(ActionEvent e)
Scrollbar	scroll	AdjustmentEvent	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent e)
TextArea	Component Hide, Show, Resize, Hidden	ComponentEvent	ComponentListener	componentHidden(ComponentEvent e) componentMoved(ComponentEvent e) componentResized(ComponentEvent e) componentShown(ComponentEvent e)
Checkbox	Check item	ItemEvent	ItemListener	itemStateChanged(ItemEvent e)
Frame	Component Add, Remove	ContainerEvent	ContainerListener	componentAdded(ContainerEvent e) componentRemoved(ContainerEvent e)
Button/ AnyControl	Focus gain, Focus Lost	FocusEvent	FocusListener	focusGained(FocusEvent e) focusLost(FocusEvent e)
TextField	Key Press, Key Release	KeyEvent	KeyListener	keyPressed(KeyEvent e) keyReleased(KeyEvent e) keyTyped(KeyEvent e)
TextField/ AnyControl	Mouse enter, exit, click, press	MouseEvent	MouseListener	mouseClicked(MouseEvent e) mouseEntered(MouseEvent e) mouseExited(MouseEvent e) mousePressed(MouseEvent e) mouseReleased(MouseEvent e)
TextField/ AnyControl	Mouse wheel move	MouseWheelEvent	MouseWheelListener	mouseWheelMoved(MouseWheelEvent e)
TextField	Text Value change	TextEvent	TextListener	textValueChanged(TextEvent e)

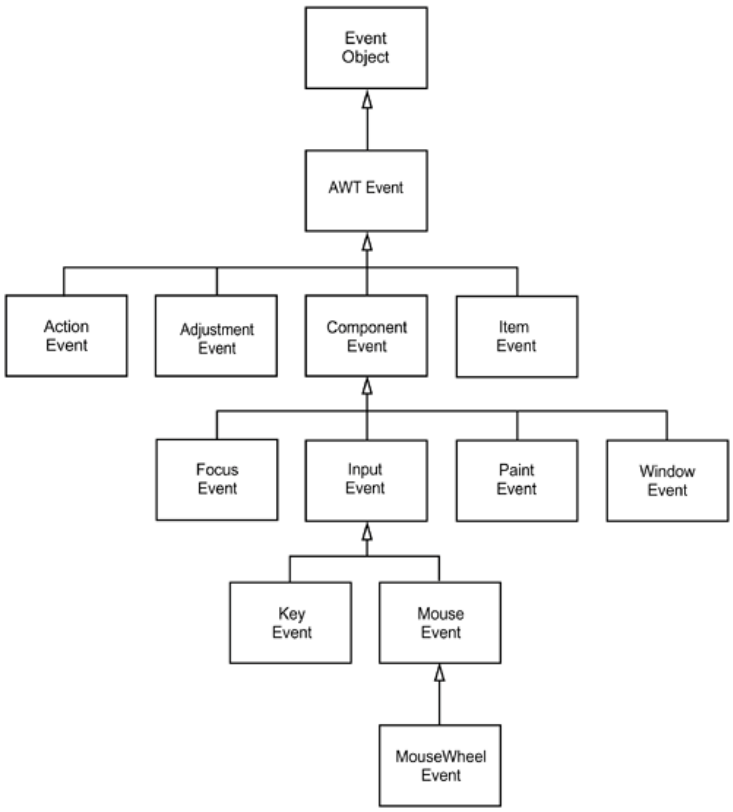
General form of listener registration is:

public void add TypeListener**(TypeListener el)**

– 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 **add**KeyListener( ).
– The method that registers a mouse motion listener is called **add**MouseMotionListener( ).

Event Classes

Hirarchy



ActionEvent	Generated when a **button is pressed, a list item is double-clicked, or a menu item is selected.**
AdjustmentEvent	Generated when a **scroll bar is manipulated**.
ComponentEvent	Generated when a component is **hidden, moved, resized, or becomes visible**.
ContainerEvent	Generated when a component is **added to or removed from a container**.
FocusEvent	Generated when a component **gains or loses keyboard focus.**
ItemEvent	Generated when a **check box or list item is clicked**; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released;also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved.
TextEvent	Generated when the value of a text area or text field is changed.
InputEvent	Abstract superclass for all component input event classes
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit

1. ActionEvent

event generated by user action on a component, typically due to interaction with a GUI control such as a button, menu item, or checkbox. These events occur when the user performs an action, like clicking a button or pressing the "Enter" key while a component has focus.

Constructore:

ActionEvent(Object src, int type, String cmd)

Commonly used functions/methods supported by `ActionEvent` include:

- **getSource():** This method returns the object that generated the event, i.e., the component on which the action occurred.
- **getActionCommand():** For components like buttons, this method returns the string associated with the action (e.g., the label of the button). This helps in identifying which component triggered the action, especially when handling events from multiple components.
- **getModifiers():** It returns an integer representing any modifier keys (such as Ctrl, Alt, Shift) held down during the action event. The `InputEvent` class constants can be used to check for specific modifiers.
- **getID():** This method returns the unique ID of the event, which can be used to differentiate between different types of events.
- **getWhen():** It retrieves the time when the event occurred, represented as the number of milliseconds since the system started (useful for timing or logging).
- **isActionCommand(String command):** It checks if the action command matches the specified string.

2. AdjustmentEvent

represents an event that occurs when the value of an adjustable component changes. It is primarily associated with components that allow users to adjust a value within a range, such as **scrollbars** and **sliders**

Constructore:

AdjustmentEvent(Adjustable src, int id, int type, int data)

Commonly used functions/methods supported by **`AdjustmentEvent`** include:

1. **`getAdjustable()`**: Retrieves the adjustable object responsible for the event. For instance, it could return the scroll bar or scroll pane that triggered the adjustment event.
2. **`getAdjustmentType()`**: Returns an integer constant representing the type of adjustment event that occurred. It may indicate whether the adjustment is due to a unit increment, unit decrement, block increment, block decrement, or other types of changes.
3. **`getValue()`**: Retrieves the current value of the adjustable component after the adjustment event. This method returns an integer representing the updated value after the adjustment.

4. **`getAdjustmentID()`:** Returns an integer constant indicating the type of adjustment event. It specifies whether the event represents a unit increment, unit decrement, block increment, block decrement, or a tracking event (such as continuous scrolling or dragging).

3. ComponentEvent

represents events generated by components (GUI elements) or their containers. It encapsulates events associated with the manipulation or behavior of components, such as moving, resizing, showing, hiding, enabling, disabling,

Constructor:

ComponentEvent(Component src, int type)

Commonly used functions/methods supported by `ComponentEvent` include:

- **getSource():** This method returns the source component that triggered the event.
- **getID():** Returns the unique ID that identifies the specific type of component event that occurred.

4. ItemEvent

In Java, the `ItemEvent` class represents an event that occurs when an item (such as a checkbox, choice, list item, etc.) state changes in an AWT or Swing component.

Constructor:

ItemEvent(ItemSelectable src, int type, Object entry, int state)

Commonly used functions/methods supported by `ItemEvent` include:

- **`getItem()`**: Retrieves the item that generated the event. This method returns an object representing the item whose state changed, such as a checkbox or list item.
- **`getStateChange()`:** Returns the type of state change that occurred for the item. It specifies whether the item's state was selected or deselected. This method returns `ItemEvent.SELECTED` or `ItemEvent.DESELECTED`.
- **`getItemSelectable()`**: Returns the object that generated the event. This method returns the item's parent component that is selectable (like a checkbox, choice, etc.).

5. ContainerEvent

is an event class representing events that occur within a container (e.g., `Panel`, `Frame`, etc.). This event is generated when components are added to or removed from a container.

Constructor:

ContainerEvent(Component src, int type, Component comp)

Commonly used functions/methods supported by `ContainerEvent` include:

1. **getSource():** `Retrieves the object that generated the event, typically the container that triggered the event.`

2. **getID():** Gets the ID of the event, indicating the type of container event that occurred (e.g., COMPONENT_ADDED or COMPONENT_REMOVED).
3. **getChild():** Gets the component that was added or removed from the container. This method is used to obtain the specific component affected by the event.
4. **getContainer():** Retrieves the container on which the event occurred. It returns the container in which the component addition or removal took place.
5. **paramString():** Provides a string representation of the event, displaying details like the event ID, source, and affected child component.

6. FocusEvent

represents the events generated when a component gains or loses the keyboard focus. These events indicate the component's change in focus state within a graphical user interface

Constructor:

FocusEvent(Component src, int type)

Commonly used functions/methods supported by Focus`Event` include:

- **getSource():** This method returns the object that originated the event, indicating the component that gained or lost focus.
- **getID():** Returns an integer representing the type of event. For `FocusEvent`, this could be `FOCUS_GAINED` or `FOCUS_LOST`.
- **isTemporary():** Indicates whether the focus change is temporary or permanent. Returns `true` if the event is temporary and `false` if it is a permanent change.
- **getOppositeComponent():** Returns the opposite `Component` involved in the focus change. For example, if one component gains focus, this method returns the component that lost focus, and vice versa.
- **getComponent():** Returns the component that is currently focused.

7. KeyEvent

represents events that are generated when a keyboard key is pressed or released.

Constructor:

KeyEvent(Component src, int type, long when, int modifiers, int code, char ch)

Commonly used functions/methods supported by Key`Event` include:

- **`getKeyChar()`**: Returns the Unicode character generated by the key event. It provides the character input when a key is pressed.
- **`getKeyCode()`**: Returns the integer key code associated with the key event. It provides the virtual key code for the key that was pressed or released.
- **`getKeyLocation()`**: Returns the location of the key on the keyboard

8. MouseEvent

represents events generated by user interactions with a mouse (such as clicking, moving, dragging, etc.) on a GUI component. This class provides various methods to retrieve information about mouse events and handle them effectively.

Constructor:

MouseEvent(Component src, int type, long when, int modifiers,
int x, int y, int clicks, boolean triggersPopup)

Commonly used functions/methods supported by MouseEvent include:

Getting Mouse Position:

- `int` **`getX()` and `int getY()`:** Retrieve the x and y coordinates of the mouse pointer when the event occurred relative to the source component.
- **`Point getPoint()`:** Retrieve the x and y coordinates as a `Point` object.

Getting Click Count:

- `int` **`getClickCount()`**: Retrieve the number of clicks associated with the event.

8. WindowEvent

used to handle events related to window operations, such as when a window is opened, closed, activated, minimized, or resized.

Constructor:

WindowEvent(Window src, int type)

There are ten types of window events
– WINDOW_ACTIVATED The window was activated.
– WINDOW_CLOSED The window has been closed.
– WINDOW_CLOSING The user requested that the window be closed.
– WINDOW_DEACTIVATED The window was deactivated.
– WINDOW_DEICONIFIED The window was deiconified.
– WINDOW_GAINED_FOCUS The window gained input focus.
– WINDOW_ICONIFIED The window was iconified.
– WINDOW_LOST_FOCUS The window lost input focus.
– WINDOW_OPENED The window was opened.
– WINDOW_STATE_CHANGED The state of the window changed.

Commonly used functions/methods supported by WindowEvent include

`Window getWindow( ):` returns the Window object that generated the event.

`Window getOppositeWindow( )` : return the opposite window (when a focus or activation event has occurred)

`int getOldState( )` : return the previous window state

`int getNewState( )` : return the current window state.

Sources of Event

The components in user interface that can generate the events are the sources of the event.

Event Source are:

1. Button
2. Check box
3. Choice
4. List
5. Menu Item
6. Scroll bar
7. Text components
8. Window

No	Source	Description
1	**Button**	Generates A**ctionEvent** when the **button is pressed**.
2	**Check box**	Generates **ItemEvent** when the **check box is selected or deseected**
3	**Choice**	Generates **ItemEvent** when the **choice is changed.**
4	**List**	Generates A**ctionEvent** when an i tem is **double-clicked**; generates **ItemEvent** when an **item is selected or deselected**
5	**Menu Item**	Generates A**ctionEvent** when a **menu item is selected;** generates **ItemEvent** when a checkable menu item is selected or deselected.
6	**Scroll bar**	Generates **AdjustmentEvent** when the **scroll bar is manipulated.**
7	**Text components**	Generates **TextEvent** when the **user enters a character**.
8	**Window**	Generates **WindowEvent** when a window **is activated, closed, deactivated, deiconified, iconified, opened, or quit**

Event Listener Interfaces

1. ActionListener

The **ActionListener** interface in Java is part of the java.awt.event package and is used to handle action events, such as button clicks, menu item selections, or any other component that generates an ActionEvent.

Probable Sources:

- `JButton`
- `JToggleButton`
- `JComboBox`
- `JMenuItem`
- `JTextField`

Event Generated by Source : `ActionEvent`

Abstract Method:

- `void actionPerformed(ActionEvent e);`: This method is invoked automatically when an action event occurs (e.g., when a button is clicked).

Register ActionListener with a Source:

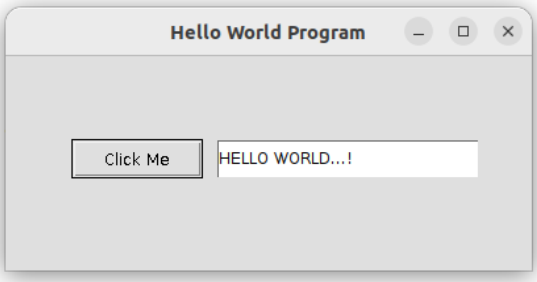
To register an `ActionListener` with a source component (like a button), you use the `addActionListener()` method.

Example:

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

public class HelloWorldSwing implements ActionListener {
    JFrame frame;
    JButton button;
    JTextField textField;

    public HelloWorldSwing() {
        frame = new JFrame("Hello World Program");
        frame.setLayout(null);

        // Initialize the button and text field
        button = new JButton("Click Me");
        textField = new JTextField();

        // Position the GUI components (x, y, width, height)
        button.setBounds(50, 50, 100, 30);
        textField.setBounds(160, 50, 200, 30);
```

```
        // Add components to the frame
        frame.add(button);
        frame.add(textField);

        // Register the ActionListener with the button
        button.addActionListener(this);

        // Frame settings
        frame.setSize(400, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        // Check if the event source is the button
        if (e.getSource() == button) {
            textField.setText("HELLO WORLD ...!");
        }
    }

    public static void main(String[] args) {
        new HelloWorldSwing(); // Create an instance of HelloWorldSwing
    }
}
```

Output



2. MouseListener

The MouseListener interface in Java is part of the java.awt.event package and is used to handle mouse events, such as mouse clicks, mouse enters and exits, and mouse button presses.

Probable Sources:

- **`JButton`**
- `JPanel`
- `JFrame`
- `JLabel`
- `JComponent`

Event Generated by Source : `MouseEvent`

Abstract Functions:

The `MouseListener` interface defines the following five abstract methods:

1. **`void mouseClicked(MouseEvent e);`**
    - Invoked when the mouse button is clicked (pressed and released) on a component.
2. **`void mousePressed(MouseEvent e);`**
    - Invoked when a mouse button is pressed on a component.
3. **`void mouseReleased(MouseEvent e);`**
    - Invoked when a mouse button is released on a component.
4. **`void mouseEntered(MouseEvent e);`**
    - Invoked when the mouse cursor enters the component.
5. **`void mouseExited(MouseEvent e);`**
    - Invoked when the mouse cursor exits the component.

Register MouseListener with a Source:

To register a `MouseListener` with a source component, use the `addMouseListener()` method.

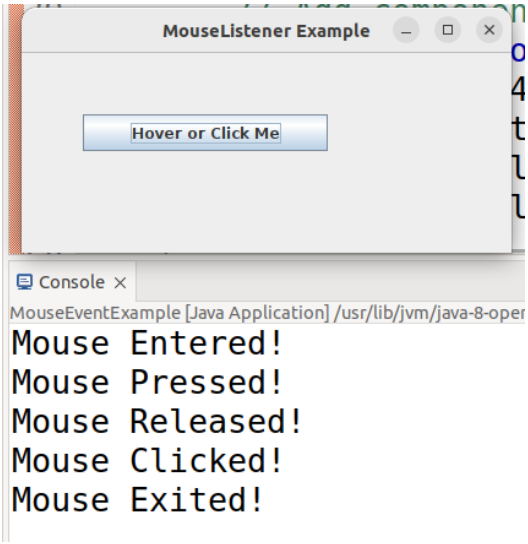
Example:

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

public class MouseEventExample implements MouseListener {
    JFrame frame;
    JButton button;
```

```
public MouseEventExample() {
    frame = new JFrame("MouseListener Example");
    button = new JButton("Hover or Click Me");

    // Set button bounds
    button.setBounds(50, 50, 200, 30);

    // Add MouseListener to the button
    button.addMouseListener(this);

    // Add components to the frame
    frame.add(button);
    frame.setSize(400, 200);
    frame.setLayout(null);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

@Override
public void mouseClicked(MouseEvent e) {
    System.out.println("Mouse Clicked!");
}

@Override
public void mousePressed(MouseEvent e) {
    System.out.println("Mouse Pressed!");
}

@Override
public void mouseReleased(MouseEvent e) {
    System.out.println("Mouse Released!");
}

@Override
public void mouseEntered(MouseEvent e) {
    System.out.println("Mouse Entered!");
}

@Override
public void mouseExited(MouseEvent e) {
```

```
        System.out.println("Mouse Exited!");
    }

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

Output



3. KeyListener

The KeyListener interface in Java is part of the java.awt.event package and is used to handle keyboard events, such as key presses, releases, and typing actions.

Probable Sources:

- `JFrame`
- `JPanel`
- **`JTextField`**
- `JTextArea`
- `JButton` (when focused)

Event Generated by Source : KeyEvent

Abstract Functions:

The `KeyListener` interface defines three abstract methods:

1. **`void keyTyped(KeyEvent e);`**
    - Invoked when a key is typed (pressed and released).
2. **`void keyPressed(KeyEvent e);`**

- Invoked when a key is pressed down.

3. **`void keyReleased(KeyEvent e);`**

   - Invoked when a key is released.

Register KeyListener with a Source:

To register a `KeyListener` with a source component, use the `addKeyListener()` method.

Example:

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

public class KeyEventListenerTest_Swing implements KeyListener{
    JFrame frame;
    JLabel label;
    JTextField textField;

    KeyEventListenerTest_Swing() {
        frame = new JFrame("Hello World Program");
        frame.setLayout(null);

        // Initialize the button and text field
        label = new JLabel("Enter data :");
        textField = new JTextField();

        // Position the gui with (x, y, width, height)
        label.setBounds(50, 50, 100, 30);
        textField.setBounds(160, 50, 200, 30);

        // Add components to the frame
        frame.add(label);
        frame.add(textField);

        // register event to textfield
        textField.addKeyListener(this);

        frame.setSize(400, 200);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }

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

```
        @Override
        public void keyPressed(KeyEvent e) {
            System.out.println("Key Pressed: " + e.getKeyCode());
        }

        @Override
        public void keyReleased(KeyEvent e) {
            System.out.println("Key Released: " + e.getKeyCode());
        }

        @Override
        public void keyTyped(KeyEvent e) {

        }
    }
```

Output