Event Handling

What is an Event?

Change in the state of an object is known as event i.e. event describes the **change in state of source**. Events are generated as result of user interaction with the graphical user interface components.

For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page are the activities that causes an event to happen.

The event object is used to carry the required information about the state change.

The **java.awt.event** package can be used to provide various event classes.

Types of Event

The events can be broadly classified into two categories:

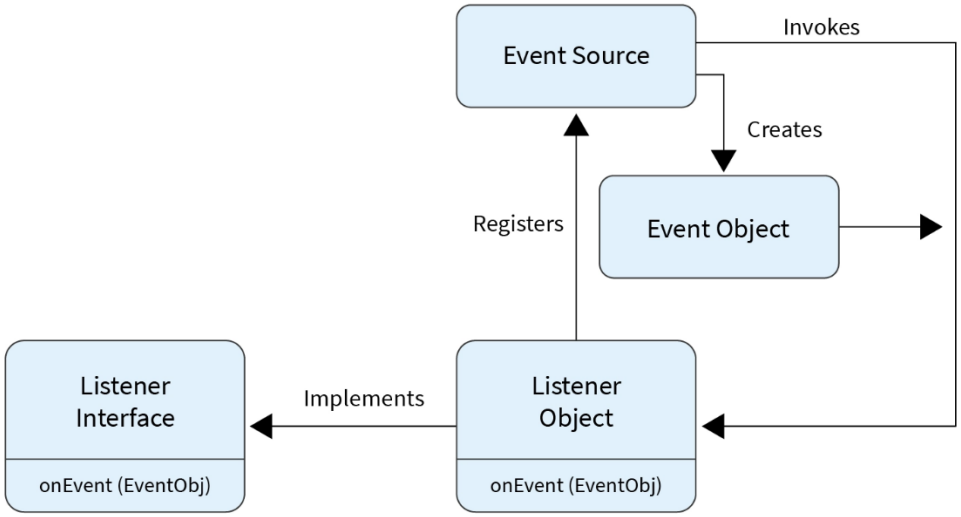
Foreground Events - Those events which **require the direct interaction of user**. They are generated as consequences of **a person interacting with the graphical components** in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.

Background Events - Events that **don't require interactions of users** to generate are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

What is Event Handling?

Event Handling is the **mechanism** that **controls** the **event** and **decides what should happen if an event occurs**. **This mechanism have the code which is known as event handler** that is executed when an event occurs. Java Uses the **Delegation Event Model** to handle the events.

This model defines the **standard mechanism** to **generate** and **handle** the **events**. Let's have a brief introduction to this model.



The Delegation Event Model has the following key participants namely:

- **Source** - 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 addTypeListener(TypeListener 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 add**Key**Listener( ). The method that registers a mouse motion listener is called add**MouseMotion**Listener( ).

- **Listener** - It is also known as **event handler**. Event listeners are objects that are notified as soon as a specific event occurs. Event listeners must define the methods to process the notification. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Steps involved in event handling

- The User clicks the button and the event is generated. (eg. ActionEvent)
- Now the object of concerned event class is created automatically and information about the source and the event get populated within same object.
- Event object (ActionEvent e) is forwarded to the method (eg. actionPerformed) of registered listener class (ActionListener).
- The method is now get executed and returns.

Important Event Classes and Interface in Java

Classes defined in java.awt.event package:-

Event Classes	Description	Listener Interface
ActionEvent	When a button is clicked or a list item is double-clicked, an ActionEvent is triggered.	ActionListener
MouseEvent	This event indicates a mouse action occurred in a component.	MouseListener
KeyEvent	The Key event is triggered when the character is entered using the keyboard.	KeyListener
ItemEvent	An event that indicates whether an item was selected or not.	ItemListener
TextEvent	when the value of a textarea or text field is changed	TextListener
MouseWheelEvent	generated when the mouse wheel is rotated	MouseWheelListener
WindowEvent	The object of this class represents the change in the state of a window and are generated when the window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	when a component is hidden, moved, resized, or made visible	ComponentEventListener
ContainerEvent	when a component is added or removed from a container	ContainerListener
AdjustmentEvent	when the scroll bar is manipulated	AdjustmentListener
FocusEvent	when a component gains or loses keyboard focus	FocusListener

ActionEvent

An *ActionEvent* is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.

Class constructors :

public ActionEvent(Object source, int id, String command, long when, int modifiers)

Constructs an ActionEvent object with the specified modifier keys and timestamp.

Parameters:

source - The object that originated the event
id - An integer that identifies the event.
command - A string that may specify a command (possibly one of several) associated with the event
modifiers - The modifier keys down during event (shift, ctrl, alt, meta). Passing negative parameter is not recommended. Zero value means that no modifiers were passed.
when - A long that gives the time the event occurred.

Methods

public String getActionCommand()

Returns the string identifying the command for this event.

public long getWhen()

Returns the time at which the event took place. This is called the event's timestamp.

public int getModifiers()

Returns the modifier keys held down during this action event.

For example, when a button is pressed, an action event is generated. The getModifiers( ) method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated.

Example:

```
import java.awt.event.*;
import javax.swing.*;
class Sample extends JFrame implements ActionListener
{
 Sample()
 {
  JButton button = new JButton("Submitted");
  button.addActionListener(this);
  button.setBounds(100,100,100,30);
  add(button);
  setLayout(null);
  setSize(500,500);
  setVisible(true);
 }
 public void actionPerformed(ActionEvent e)
 {
  System.out.println("Command: " + e.getActionCommand());
 }
```

```
 public static void main(String args[])
 {
  Sample frame = new Sample();
 }
}
```

ItemEvent

An ItemEvent is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.

Constructor

ItemEvent(ItemSelectable source, int id, Object item, int stateChange)

Constructs an ItemEvent object.

Parameters:

source - The ItemSelectable object that originated the event
id - The integer that identifies the event type.
item - the item affected by the event
stateChange - An integer that indicates whether the item was selected or deselected.

The stateChange of any ItemEvent instance takes one of the following values:

- ItemEvent.SELECTED :- The user selected an item.
- ItemEvent.DESELECTED :- The user deselected an item.

In addition, ItemEvent defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state.

Methods

getItemSelectable

public ItemSelectable getItemSelectable()

Returns the ItemSelectable object that originated the event.

Lists and choices are examples of user interface elements that implement the ItemSelectable interface.

getItem

public Object getItem()

Returns: the item (object) that was affected by the event

getStateChange

public int getStateChange()

Returns: an integer that indicates whether the item was selected or deselected

paramString

public String paramString()

Returns a parameter string identifying this item event.

Interface:-
ItemListener

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

void itemStateChanged(ItemEvent ie)

Example:

```
import java.awt.*;
import java.awt.event.*;
public class ItemListenerExample extends Frame implements ItemListener
{
   Checkbox checkBox1,checkBox2;
   Label label;
   ItemListenerExample()
   {
      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);
      add(checkBox1);
      add(checkBox2);
      add(label);
      checkBox1.addItemListener(this);
      checkBox2.addItemListener(this);
    }
   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[])
{
   ItemListenerExample f = new ItemListenerExample();
   f.setSize(400,400);
   f.setLayout(null);
   f.setVisible(true);
}
}
```

KeyEvent

An event which indicates that a **keystroke** occurred in a component.
It is generated by a component object (such as a text field) when a **key is pressed, released, or typed**. The event is passed to every **KeyListener** object which registered to receive such events using the component's **addKeyListener** method.

Constructor

KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar)

Parameters:

source - The Component that originated the event
id - An integer indicating the type of event.
when - A long integer that specifies the time the event occurred.
modifiers - The modifier keys down during event (shift, ctrl, alt, meta).
keyCode - The **integer code for** an actual key, or **VK_UNDEFINED** (for a key-typed event)
keyChar - The **Unicode character** generated by this event, or **CHAR_UNDEFINED** (for key-pressed and key-released events which do not map to a valid Unicode character)

KEY_PRESSED : generated when any key is pressed
KEY_RELEASED :- when any key is released
KEY_TYPED :- occurs only if a valid Unicode character could be generated.

Remember, not all key presses result in characters. For example, pressing the SHIFT key does not generate a character. There are many other integer constants that are defined by KeyEvent.
The VK constants specify **virtual key codes**.
For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters. Here are some others:

VK_ENTER	VK_ESCAPE	VK_CANCEL	VK_UP
VK_DOWN	VK_LEFT	VK_RIGHT	VK_PAGE_DOWN
VK_PAGE_UP	VK_SHIFT	VK_ALT	VK_CONTROL

The VK constants specify virtual key codes.

Method

public int getKeyCode()
Returns the integer keyCode associated with the key in this event.
When a KEY_TYPED event occurs, getKeyCode( ) returns VK_UNDEFINED.

public char getKeyChar()
Returns the character associated with the key in this event. For example, the KEY_TYPED event for shift + "a" returns the value for "A".
If no valid character is available, then getKeyChar( ) returns CHAR_UNDEFINED.

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

Methods of KeyListener interface

Sr.	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.

Program:

```
import java.awt.*;
import java.awt.event.*;
public class Sample extends Frame implements KeyListener
{
 Label l1,l2,l3;
 TextArea area;
   Sample()
   {
   l1 = new Label();
   l1.setBounds (20, 50, 1000, 20);
   l2 = new Label();
   l2.setBounds (20, 80, 1000, 20);
   l3 = new Label();
   l3.setBounds (20, 110, 1000, 20);
   area = new TextArea();
   area.setBounds (20, 150, 300, 100);
   area.addKeyListener(this);
   add(l1);
   add(l2);
   add(l3);
   add(area);
   }
   public void keyPressed (KeyEvent e)
   {
      l1.setText ("Key Pressed : "+ e.getKeyChar() + " and has code: "+e.getKeyCode());
      //l1.setText (e.paramString());
   }
   public void keyReleased (KeyEvent e)
   {
      l2.setText ("Key Released: "+ e.getKeyChar() + " and has code: "+e.getKeyCode());
      //l2.setText (e.paramString());
   }
   public void keyTyped (KeyEvent e)
   {
      l3.setText ("Key Typed : "+ e.getKeyChar());
      //l3.setText (e.paramString());
```

```
    }

    public static void main(String[] args)
    {
    Sample f = new Sample();
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    }
}
```

Example 2: Count Words & Characters

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

public class Sample extends Frame implements KeyListener {
    Label l;
    TextArea area;
    Sample()
    {
        l = new Label();
        l.setBounds (20, 50, 200, 20);
        area = new TextArea();
        area.setBounds (20, 80, 300, 300);
        area.addKeyListener(this);
    add(l);
    add(area);
        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)
    {

    }

    public void keyReleased (KeyEvent e)
    {
        String text = area.getText();
        String words[] = text.split ("\\s");
        l.setText ("Words: " + words.length + " Characters:" + text.length());
    }
    public void keyTyped(KeyEvent e)
    {
```

```
  }
  public static void main(String[] args)
  {
  Sample s = new Sample();
  }
}
```

TextEvent

The object of this class represents the text events. The TextEvent is generated when character is entered in the text fields or text area. The TextEvent instance does not include the characters currently in the text component that generated the event rather we are provided with other methods to retrieve that information.

Constructor

public TextEvent(Object source, int id)

Constructs a TextEvent object.

Parameters:

source - The (TextComponent) object that originated the event
id - An integer that identifies the event type
TextEvent defines the integer constant TEXT_VALUE_CHANGED.

Interface

TextListener

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

void textValueChanged(TextEvent te)

Example:

```
import java.awt.*;
import java.awt.event.*;
public class TextEventEx1 extends Frame implements TextListener
{
Label label1, label2;
TextField field1;
TextEventEx1()
{
label1= new Label("Type in the textfield, to see the text events it generates -");
label1.setBounds(100,100,300,40);
label2= new Label();
label2.setBounds(100,150,300,40);
field1 = new TextField(25);
field1.setBounds(100,200,100,40);
add(label1);
add(field1);
add(label2);
```

```
field1.addTextListener(this);
}

public void textValueChanged(TextEvent te)
{
label2.setText(te.paramString());
}
public static void main(String args[])
{
TextEventEx1 f = new TextEventEx1();
f.setSize(500,500);
f.setLayout(null);
f.setVisible(true);
}
}
```

MouseEvent

This event indicates a mouse action occurred in a component. This low-level event is generated by a component object for Mouse Events and Mouse motion events.

Constructor

public MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger, int button)

Parameters:

source - the Component that originated the event
id - the integer that identifies the event
when - a long int that gives the time the event occurred
modifiers - the modifier keys down during event (e.g. shift, ctrl, alt, meta)
x - the horizontal x coordinate for the mouse location
y - the vertical y coordinate for the mouse location
clickCount - the number of mouse clicks associated with event
popupTrigger - a boolean, true if this event is a trigger for a popup menu
button - which of the mouse buttons has changed state.

MouseEvent class defines the following integer constants that can be used to identify them:

- MouseEvents

MOUSE_PRESSED The mouse was pressed.
MOUSE_RELEASED The mouse was released.
MOUSE_CLICKED The user clicked the mouse (pressed and released)
MOUSE_ENTERED The mouse entered a component.
MOUSE_EXITED The mouse exited from a component.

- Mouse Motion Events

MOUSE_DRAGGED The user dragged the mouse.
MOUSE_MOVED The mouse moved.

ClassMethods

S.N.	Method & Description
1	**int getButton()** Returns which of the mouse buttons has changed state. The return value will be one of these constants defined by MouseEvent. NOBUTTON BUTTON1 BUTTON2 BUTTON3 The NOBUTTON value indicates that no button was pressed or released.
2	**int getClickCount()** Returns the number of mouse clicks associated with this event.
3	**Point getPoint()** Returns the x,y position of the event relative to the source component.
4	**int getX()** Returns the horizontal x position of the event relative to the source component.
5	**int getY()** Returns the vertical y position of the event relative to the source component.
6	**boolean isPopupTrigger():** tests if this event causes a pop-up menu to appear on this platform.

Interface:-
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)

Exanple:

```
import java.awt.*;
import java.awt.event.*;
public class Sample extends Frame implements MouseListener
{
   Label l1,l2,l3,l4;
```

```
    Sample()
    {
       addMouseListener(this);
       l1=new Label("Label 1");
       l2=new Label("Label 2");
       l3=new Label("Label 3");
       l4=new Label("Label 4");
       l1.setBounds(20,50,100,20);
       l2.setBounds(20,100,100,20);
       l3.setBounds(20,150,100,20);
       l4.setBounds(20,200,100,20);
       add(l1);
       add(l2);
       add(l3);
       add(l4);
       setSize(300,300);
       setLayout(null);
       setVisible(true);
    }
     public void mouseEntered(MouseEvent e)
    {
       l1.setText("Mouse Entered");
    }
    public void mouseExited(MouseEvent e)
    {
       l1.setText("Mouse Exited");
    }
    public void mousePressed(MouseEvent e)
    {
       l2.setText("Mouse Pressed");
    }
    public void mouseReleased(MouseEvent e)
    {
       l2.setText("Mouse Released");
    }
    public void mouseClicked(MouseEvent e)
    {
      l3.setText("Mouse Clicked");
      int i = e.getButton();
      String res = Integer.toString(i);
      l4.setText(res);

    }

  public static void main(String[] args)
  {
    Sample s = new Sample();
```

```
  }
}
```

WindowEvent

This Event indicates that a window has changed its status. This is generated by a Window object when it is opened, closed, activated, deactivated, iconified, or deiconified, or when focus is transfered into or out of the Window.

Constructor
public WindowEvent(Window source, int id, Window opposite, int oldState, int newState)

Constructs a WindowEvent object.

Parameters:
source - The Window object that originated the event
id - An integer indicating the type of event.
opposite - The other window involved in the focus or activation change, or null
oldState - Previous state of the window for window state change event.
newState - New state of the window for window state change event.

WindowEvent class defines the following integer constants that can be used to identify them:

WINDOW_ACTIVATED The window was activated.
WINDOW_DEACTIVATED The window was deactivated.
WINDOW_OPENED The window was opened
WINDOW_CLOSED The window has been closed.
WINDOW_CLOSING The user requested that the window be closed.
The Closing event raises immediately after Close() is called or user tried to close the window, and ***can be handled to cancel window closure***. We can close the AWT Window or Frame by calling dispose() or System.exit() inside windowClosing() method.
The Closed raises after the window closed and ***cannot be cancelled.***

WINDOW_ICONIFIED The window was iconified. The window iconified event. This event is delivered when the window has been changed from a normal to a minimized state. For many platforms, a minimized window is displayed as the icon specified in the window's iconImage property.

WINDOW_DEICONIFIED The window was deiconified. This event is delivered when the window has been changed from a minimized to a normal state.

WINDOW_GAINED_FOCUS The window gained input focus.
WINDOW_LOST_FOCUS The window lost input focus.
WINDOW_STATE_CHANGED The state of the window changed.

Interface
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)

Example:

```
import java.awt.*;
import java.awt.event.*;
public class WindowExample extends Frame implements WindowListener {

   WindowExample()
   {
      addWindowListener(this);
   }
public void windowActivated (WindowEvent w) {
   System.out.println("activated");
}

public void windowClosed (WindowEvent w) {
   System.out.println("closed");
}

public void windowClosing (WindowEvent w) {
   System.out.println("closing");
   dispose();
}

public void windowDeactivated (WindowEvent w) {
   System.out.println("deactivated");
}

public void windowDeiconified (WindowEvent w) {
   System.out.println("deiconified");
}

public void windowIconified(WindowEvent w) {
   System.out.println("iconified");
}

public void windowOpened(WindowEvent w) {
   System.out.println("opened");
}
```

```
public static void main(String[] args)
{
  WindowExample f = new WindowExample();
     f.setSize (400, 400);
     f.setLayout (null);
     f.setVisible (true);
}
}
```

Adapter classes

The listener class that implements the Listener interface must provide bodies for all of the methods of that interface. It is not a problem for all the semantic listener interfaces such as ActionEvent, ItemEvent, TextEvent, AdapterEvent as each of them declares only one method. However, for all the low-level listener interfaces where each interface contains multiple methods, implementing each method can be somewhat tedious, especially when we have to define methods in which we are not interested. For example: Suppose we are interested in setting up only one listener interface method windowClosing() of the WindowListener interface that causes the program to terminate. In that case, we would not only need to provide code for windowClosing() method but also need to write empty bodies for the other methods available in the WindowListener interface.

To avoid this unnecessary code, the java.awt.event package provides adapter classes for various event-listener types. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when we want to receive and process only some of the events that are handled by a particular event listener interface. We can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which we are interested.

java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener

Example of KeyAdapter

```
import java.awt.*;
import java.awt.event.*;
// class which inherits the KeyAdapter class
public class KeyAdapterExample extends KeyAdapter
{
   Label l;
   TextArea area;
   Frame f;
   KeyAdapterExample()
   {
      f = new Frame();
      l = new Label();
      l.setBounds (20, 50, 200, 20);
      area = new TextArea();
      area.setBounds (20, 80, 300, 300);
      area.addKeyListener(this);
      f.add(l);
      f.add(area);
      f.setSize (400, 400);
      f.setLayout (null);
      f.setVisible (true);
   }
// overriding the keyReleased() method
   public void keyReleased (KeyEvent e)
   {
      String text = area.getText();
      String words[] = text.split ("\\s");
      l.setText ("Words: " + words.length + " Characters:" + text.length());
   }
   public static void main(String[] args)
   {
    new KeyAdapterExample();
   }
}
```

Example of MouseAdapter

```
import java.awt.*;
import java.awt.event.*;
public class MouseAdapterExample extends MouseAdapter
{
   MouseAdapterExample()
   {
      Frame f = new Frame();
      f.addMouseListener(this);
      f.setSize (400, 400);
      f.setLayout (null);
```

```
        f.setVisible (true);
    }
    public void mouseReleased (MouseEvent e)
    {
        System.out.println(e);
    }
    public static void main(String[] args)
    {
     new MouseAdapterExample();
    }
}
```