CHAPTER-14

AWT 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.

Types of Event

The events can be broadly classified into two categories:

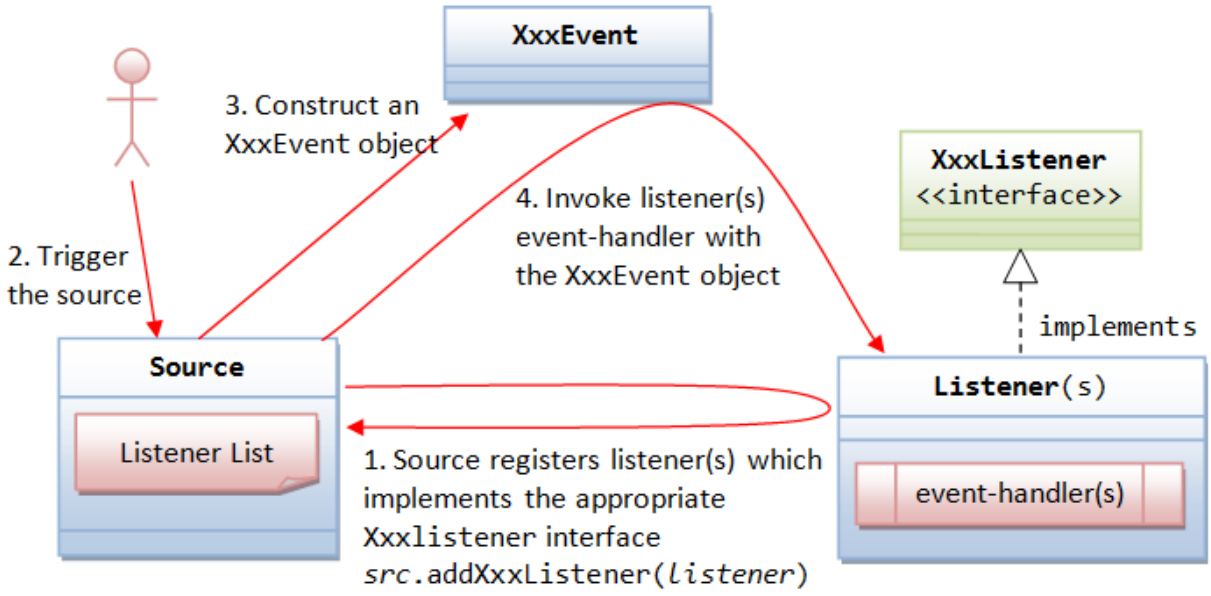
- **Foreground Events** - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page etc.
- **Background Events** - Those events that don't require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

In AWT components, we came to know every component (except Panel and Label) generates events when interacted by the user like clicking over a button or pressing enter key in a text field etc. Listeners handle the events. Let us know the style (or design pattern) Java follows to handle the events.

AWT Event-Handling

Java adopts the so-called "Event-Driven" (or "Event-Delegation") programming model for event-handling, similar to most of the visual programming languages (such as Visual Basic and Dot.NET etc).
The AWT's event-handling classes are kept in package **java.awt.event.**

The *source* object (such as Button and Textfield) interacts with the user. Upon triggered, it creates an *event* object. This *event* object will be messaged to all the *registered listener* object(s), and an appropriate event-handler method of the listener(s) is called-back to provide the response. In other words, *triggering a source fires an event to all its listener(s), and invoke an appropriate handler of the listener(s)*.
Three objects are involved in the event-handling: a *source*, *listener*(s) and an *event* object.



The sequence of steps is illustrated above:
1. *The source object registers its listener(s)* for a certain type of *event*.

Source object fires event upon triggered. For example, clicking an Button fires an **ActionEvent**, mouse-click fires **MouseEvent**, key-type fires **KeyEvent**, etc.

How the source and listener understand each other? The answer is via an agreed-upon interface. For example, if a source is capable of firing an event called XxxEvent (e.g., MouseEvent) involving various operational modes (e.g., mouse-clicked, mouse-entered, mouse-exited, mouse-pressed, and mouse-released). Firstly, we need to declare an interface called XxxListener (e.g., MouseListener) containing the names of the handler methods. Recall that an interface contains only abstract methods without implementation. For example,

```
// A MouseListener interface, which declares the signature of the handlers
// for the various operational modes.

public interface MouseListener
{
public void mousePressed(MouseEvent evt);//Called back upon mouse-button pressed
public void mouseReleased(MouseEvent evt); // Called back upon mouse-button released
public void mouseClicked(MouseEvent evt); // Called back upon mouse-button clicked (pressed and released)
public void mouseEntered(MouseEvent evt); // Called back when mouse pointer entered the component
public void mouseExited(MouseEvent evt); // Called back when mouse pointer exited the component
```

Event Delegation Model

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:

1. Event Sources } **Source**
2. Event Object classes }
3. Event Listeners }
4. Event Adapters } **Listener**

The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

I Event Sources
Event sources are components, subclasses of **java.awt.Component**, capable to generate events. The event source can be a button, TextField or a Frame etc.

II Event Object classes

Almost every event source(except Panel and Label) generates an event and is named by some Java class. For example, the event generated by button is known as **ActionEvent** and that of Checkbox is known as **ItemEvent**. All the events are listed in **java.awt.event** package. Following list gives a few components and their corresponding listeners.

Component	**Event it generates**
Button, TextField, List, Menu	ActionEvent
Frame	WindowEvent
Checkbox, Choice, List	ItemEvent
Scrollbar	AdjustmentEvent
Mouse (hardware)	MouseEvent
Keyboard (hardware)	KeyEvent

The events generated by hardware components (like **MouseEvent** and **KeyEvent**) are known as **low-level events** and the events generated by software components (like Button, List) are known as **semantic events**.

Event Listeners

The events generated by the GUI components are handled by a special group of interfaces known as "**listeners**". Note, Listener is an interface. Every component has its own listener, say, **AdjustmentListener** handles the events of scrollbar. Some listeners handle the events of multiple components. For example, **ActionListener** handles the events of Button, TextField, List and Menus. Listeners are from **java.awt.event** package.

Event Classe	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released also when the enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

4. Event Adapters

When a listener includes many abstract methods to override, the coding becomes heavy to the programmer. For example, to close the frame, you override seven abstract methods of *WindowListener*, in which, infact you are using only one method. To avoid this heavy coding, the designers come with another group of classes known as "**adapters**". Adapters are abstract classes defined in **java.awt.event** package. Every listener that has more than one abstract method has got a corresponding adapter class.

Adapters are abstract classes for receiving various events. The methods in these classes are empty. These classes exists as convenience for creating listener objects.

Following is the list of commonly used adapters while listening GUI events in AWT.

Sr. No.	Adapter & Description
1	FocusAdapter An abstract adapter class for receiving focus events.
2	KeyAdapter An abstract adapter class for receiving key events.
3	MouseAdapter An abstract adapter class for receiving mouse events.
4	MouseMotionAdapter An abstract adapter class for receiving mouse motion events.
5	WindowAdapter An abstract adapter class for receiving window events.

Steps involved in event handling

- The User clicks the button and the event is generated.
- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.
- Event object is forwarded to the method of registered listener class.
- the method is now get executed and returns.

Points to remember about listener

- In order to design a listener class we have to develop some listener interfaces. These Listener interfaces forecast some public abstract **callback methods** which must be implemented by the listener class.
- If you do not implement the any if the predefined interfaces then your class can not act as a listener class for a source object.

What happens internally at a button click?

We know the events are handled by **listeners** and **ActionListener** handles the events of a button. Observe the following skeleton code.

```
public class ButtonDemo extends Frame implements ActionListener
{
public ButtonDemo()
{
Button btn = new Button("OK");
btn.addActionListener(this);
add(btn);
}
public void actionPerformed(ActionEvent e)
{
String str = e.getActionCommand();
}
}
```

3. Now, the question is, what **ActionListener** does with the **ActionEvent** object it received?
The **ActionListener** simply calls **actionPerformed()** method and passes the **ActionEvent** object to the parameter as follows.

public void actionPerformed(ActionEvent e)

The parameter for the above method comes from ActionListener.

4. Finally the **ActionEvent** object generated by the button **btn** reaches the **e** object of ActionEvent. All this is done by JVM implicitly. For this reason, the **getActionCommand()** method of ActionEvent class knows the label of the button **btn**.

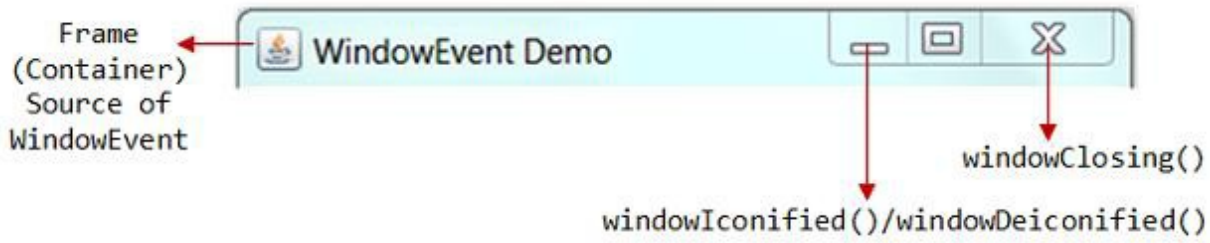
String str = e.getActionCommand();

The **e** represents an object of **ActionEvent** and the value for the **e** is coming from button **btn**. **str** is nothing but the label of the button OK.

Example 3: WindowEvent and WindowListener Interface

A WindowEvent is fired (to all its WindowEvent listeners) when a window (e.g., Frame) has been opened/closed, activated/deactivated, iconified/deiconified via the 3 buttons at the top-right corner or other means. The source of WindowEvent shall be a top-level window-container such as Frame.

A WindowEvent listener must implement WindowListener interface, which declares 7 abstract event-handling methods, as follows. Among them, the windowClosing(), which is called back upon clicking the window-close button, is the most commonly-used.



public void windowClosing**(WindowEvent e)**
// Called-back when the user attempts to close the window by clicking the window close button.
// This is the most-frequently used handler.
public void windowOpened(WindowEvent e)
// Called-back the first time a window is made visible.
public void windowClosed(WindowEvent e)
// Called-back when a window has been closed as the result of calling dispose on the window.
public void windowActivated(WindowEvent e)
// Called-back when the Window is set to be the active Window.

public void windowDeactivated(WindowEvent e)
// Called-back when a Window is no longer the active Window.
public void windowIconified(WindowEvent e)
// Called-back when a window is changed from a normal to a minimized state.
public void windowDeiconified(WindowEvent e)
// Called-back when a window is changed from a minimized to a normal state.

The following program added support for "close-window button" to Example 1: AWTCounter.

```
import java.awt.*; // Using AWT containers and components
import java.awt.event.*; // Using AWT events and listener interfaces
// An AWT GUI program inherits the top-level container java.awt.Frame
public class WindowEventDemo extends Frame
implements ActionListener, WindowListener {
// This class acts as listener for ActionEvent and WindowEvent
// Java supports only single inheritance, where a class can extend
// one superclass, but can implement multiple interfaces.
private TextField tfCount;
private Button btnCount;
private int count = 0; // Counter's value
```

```
/** Constructor to setup the UI components and event handling */
public WindowEventDemo() {
setLayout(new FlowLayout()); // "super" Frame sets to FlowLayout
add(new Label("Counter")); // "super" Frame adds an anonymous Label
tfCount = new TextField("0", 10); // Allocate TextField
tfCount.setEditable(false); // read-only
add(tfCount); // "super" Frame adds tfCount
btnCount = new Button("Count"); // Declare and allocate a Button
add(btnCount); // "super" Frame adds btnCount
btnCount.addActionListener(this);
// btnCount fires ActionEvent to its registered ActionEvent listener
// btnCount adds "this" object as an ActionEvent listener
addWindowListener(this);
// "super" Frame fires WindowEvent to its registered WindowEvent listener
// "super" Frame adds "this" object as a WindowEvent listener
setTitle("WindowEvent Demo"); // "super" Frame sets title
setSize(250, 100); // "super" Frame sets initial size
setVisible(true); // "super" Frame shows
}
/** The entry main() method */

public static void main(String[] args) {
new WindowEventDemo(); // Let the construct do the job
}
/** ActionEvent handler */
@Override
public void actionPerformed(ActionEvent evt) {
++count;
tfCount.setText(count + "");
}
/** WindowEvent handlers */
// Called back upon clicking close-window button
@Override
public void windowClosing(WindowEvent e) {
System.exit(0); // Terminate the program
}
// Not Used, but need to provide an empty body
@Override
public void windowOpened(WindowEvent e) { }
@Override
public void windowClosed(WindowEvent e) { }
@Override
public void windowIconified(WindowEvent e) { }
@Override
public void windowDeiconified(WindowEvent e) { }
@Override
public void windowActivated(WindowEvent e) { }
@Override
public void windowDeactivated(WindowEvent e) { }
}
```

In this example, we shall modify the earlier AWTCounter example to handle the WindowEvent. Recall that pushing the "close-window" button on the AWTCounter has no effect, as it did not handle the WindowEvent of windowClosing(). We included the WindowEvent handling codes in this example.

1. We identify super Frame as the source object.
2. The Frame fires the WindowEvent to all its registered WindowEvent listener(s).
3. We select this object as the WindowEvent listener (for simplicity)
4. We register this object as the WindowEvent listener to the source Frame via method **addWindowListener(this).**
5. The WindowEvent listener (this class) is required to implement the WindowListener interface, which declares 7 abstract methods: **windowOpened(), windowClosed(), windowClosing(), windowActivated(), windowDeactivated(), windowIconified() and windowDeiconified().**
6. We override the **windowClosing()** handler to terminate the program using **System.exit(0)**. We ignore the other 6 handlers, but required to provide an empty body.

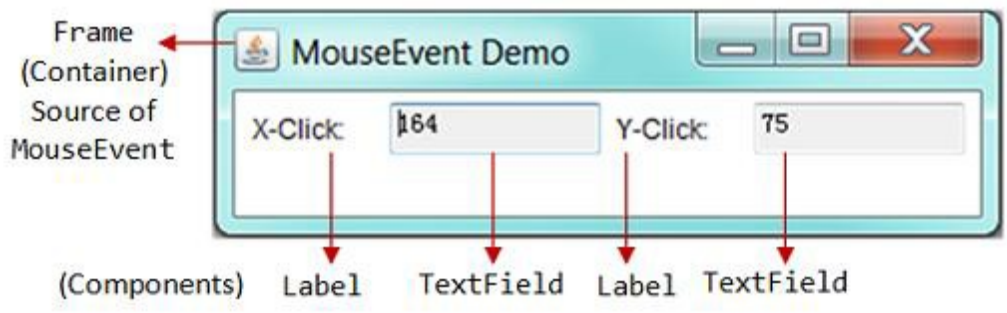
Adapter Classes

Adapters are abstract classes for receiving various events. The methods in these classes are empty. These classes exists as convenience for creating listener objects.

Example 4: MouseEvent and MouseListener Interface

A **MouseEvent** is fired to all its registered listeners, when you press, release, or click (press followed by release) a mouse-button (left or right button) at the source object; or position the mouse-pointer at (enter) and away (exit) from the source object.
A MouseEvent listener must implement the *MouseListener* interface, which declares the following five abstract methods:

public void mouseClicked(MouseEvent e)
// Called-back when the mouse-button has been clicked (pressed followed by released) on the source.
public void mousePressed(MouseEvent e)
public void mouseReleased(MouseEvent e)
// Called-back when a mouse-button has been pressed/released on the source.
// A mouse-click invokes mousePressed(), mouseReleased() and mouseClicked().
public void mouseEntered(MouseEvent e)
public void mouseExited(MouseEvent e)
// Called-back when the mouse-pointer has entered/exited the source.




```
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
public class MouseEventDemo extends Frame implements MouseListener {
// Private variables
private TextField tfMouseX; // to display mouse-click-x
private TextField tfMouseY; // to display mouse-click-y
// Constructor - Setup the UI
public MouseEventDemo() {
setLayout(new FlowLayout()); // "super" frame sets layout
// Label
add(new Label("X-Click: ")); // "super" frame adds component
// TextField
tfMouseX = new TextField(10); // 10 columns
tfMouseX.setEditable(false); // read-only

add(tfMouseX); // "super" frame adds component
// Label
add(new Label("Y-Click: ")); // "super" frame adds component
// TextField
tfMouseY = new TextField(10);
tfMouseY.setEditable(false); // read-only
add(tfMouseY); // "super" frame adds component
addMouseListener(this);
// "super" frame fires the MouseEvent
// "super" frame adds "this" object as MouseEvent listener
setTitle("MouseEvent Demo"); // "super" Frame sets title
setSize(350, 100); // "super" Frame sets initial size
setVisible(true); // "super" Frame shows
}
public static void main(String[] args) {
new MouseEventDemo(); // Let the constructor do the job
}

// MouseEvent handlers
@Override
public void mouseClicked(MouseEvent e) {
tfMouseX.setText(e.getX() + "");
```

```
tfMouseY.setText(e.getY() + "");
}

@Override
public void mousePressed(MouseEvent e) { }
@Override
public void mouseReleased(MouseEvent e) { }
@Override
public void mouseEntered(MouseEvent e) { }
@Override
public void mouseExited(MouseEvent e) { }
}
```

In this example, we setup a GUI with 4 components (two Labels and two non-editable TextFields), inside a top-level container Frame, arranged in FlowLayout.
To demonstrate the MouseEvent:

1. We identity super Frame as the

2. The Frame fires a MouseEvent to all its MouseEvent listener(s) when you click/press/release a mouse-button or enter/exit with the mouse-pointer.
3. We select this object as the MouseEvent listener (for simplicity).
4. We register this object as the MouseEvent listener to super Frame (source) via the method addMouseListener(this).
5. The listener (this class) is required to implement the MouseListener interface, which declares 5 abstract methods: mouseClicked(), mousePressed(), mouseReleased(), mouseEntered(), and mouseExit(). We override the mouseClicked() to display the (x, y) co-ordinates of the mouse click on the two displayed TextFields. We ignore all the other handlers (for simplicity - but you need to provide an empty body for compilation).

Try: Include a WindowListener to handle the close-window button.

Example 5: MouseEvent and MouseMotionListener Interface

A MouseEvent is also fired when you moved and dragged the mouse pointer at the source object. But you need to use MouseMotionListener to handle the mouse-move and mouse-drag. The MouseMotionListener interface declares the following two abstract methods:

public void mouseDragged(MouseEvent e)
// Called-back when a mouse-button is pressed on the source component and then dragged.
public void mouseMoved(MouseEvent e)
// Called-back when the mouse-pointer has been moved onto the source component but no buttons have been pushed.