Unit 2: Event Handling

Normally, a user interacts with an application's GUI to indicate the tasks that the application should perform. For example, when you write an e-mail in an e-mail application, clicking the Send button tells the application to send the e-mail to the specified e-mail addresses. GUIs are event driven. When the user interacts with a GUI component, the interaction—known as an event—drives the program to perform a task. Some common user interactions that cause an application to perform a task include clicking a button, typing in a text field, selecting an item from a menu, closing a window and moving the mouse. The code that performs a task in response to an event is called an event handler, and the overall process of responding to events is known as **event handling.**

Event handling is fundamental to Java programming because it is integral to the creation of many kinds of applications, including applets and other types of GUI-based programs. Furthermore, any program that uses a graphical user interface, such as a Java application written for Windows, is event driven. Thus, you cannot write these types of programs without a solid command of event handling. Events are supported by a number of packages, including **java.util, java.awt,** and **java.awt.event.** Most events to which your program will respond are generated when the user interacts with a GUI-based program.

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.

- **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 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.

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.

Two Event Handling Mechanisms

The way in which events are handled changed significantly between the original version of Java (1.0) and all subsequent versions of Java, beginning with version 1.1. Although the 1.0 method of event handling is still supported, it is not recommended for new programs. Also, many of the methods that support the old 1.0 event model have been deprecated. The modern approach is the way that events should be handled by all new programs and thus is the method employed by programs in this topic.

The Delegation Event Model

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

The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code.

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

Events

- In the delegation model, an **event** is an object that describes a state change in a source.
- Among other causes, an event can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
- Events may also occur that are not directly caused by interactions with a user interface.

Event Sources

- ♣ A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.
- ♣ A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- ♣ Each type of event has its own registration method. Here is the general form:

```
public void 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 **addKeyListener( ).** The method that registers a mouse motion listener is called **addMouseMotionListener( ).**

- ♣ When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as **multicasting the event**. In all cases, notifications are sent only to listeners that register to receive them.
- ♣ Some sources may allow only one listener to register. The general form of such a method is this:

  ```
  public void addTypeListener(TypeListener el )
  ```

  Here, **Type** is the name of the event, and **el** is a reference to the event listener.

  When such an event occurs, the registered listener is notified. This is known as **unicasting the event.**

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

  ```
  public void removeTypeListener(TypeListener el )
  ```

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

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

Event Listeners

- ♦ A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.
- ♦ The methods that receive and process events are defined in a set of interfaces, such as those found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface.

Event Classes

The classes that represent events are at the core of Java's event handling mechanism. Widely used Event classes are those defined by AWT and those defined by Swing.

- ➢ At the root of the Java event class hierarchy is **EventObject**, which is in **java.util.** It is the superclass for all events. Its one constructor is shown here:

  ```
  EventObject(Object src)
  ```

Here, **src** is the object that generates this event.

- **EventObject** defines two methods: **getSource( )** and **toString( )**. The **getSource( )** method returns the source of the event. Its general form is shown here:

  ```
  Object getSource( )
  ```

- As expected, **toString( )** returns the string equivalent of the event.
- The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject.** It is the superclass (either directly or indirectly) of all AWT-based events used by the delegation event model. Its **getID( )** method can be used to determine the type of the event. The signature of this method is shown here:
  ```
  int getID( )
  ```

To summarize:

- **EventObject** is a superclass of all events.
- **AWTEvent** is a superclass of all AWT events that are handled by the delegation event model.
- **AWT Event Classes:**

Following is the list of commonly used event classes.

Sr. No.	Control & Description
1	**AWTEvent** It is the root event class for all AWT events. This class and its subclasses supercede the original java.awt.Event class.
2	**ActionEvent** The ActionEvent is generated when button is clicked or the item of a list is double clicked.
3	**InputEvent** The InputEvent class is root event class for all component-level input events.
4	**KeyEvent** On entering the character the Key event is generated.
5	**MouseEvent**

	This event indicates a mouse action occurred in a component.
6	**TextEvent** The object of this class represents the text events.
7	**WindowEvent** The object of this class represents the change in state of a window.
8	**AdjustmentEvent** The object of this class represents the adjustment event emitted by Adjustable objects.
9	**ComponentEvent** The object of this class represents the change in state of a window.
10	**ContainerEvent** The object of this class represents the change in state of a window.
11	**MouseMotionEvent** The object of this class represents the change in state of a window.
12	**PaintEvent** The object of this class represents the change in state of a window.

SUMMARY ABOUT EVENT HANDLING

Event handling has three main components:

- **Events :** An event is a change in state of an object.
- **Events Source :** Event source is an object that generates an event.
- **Listeners :** A listener is an object that listens to the event. A listener gets notified when an event occurs.

Simply Event Handling includes : **Action→ Event→Listener**

- ✓ ***Action:*** What user does is known as action. Example, a click over button. Here, click is the action performed by the user.
- ✓ ***Event:*** The action done by the component when the user's action takes place is known as event. That is, event is generated (not seen, it is software) against action.
- ✓ ***Listener:*** It is an interface that handles the event. That is, the event is caught by the listener and when caught, immediately executes some method filled up with code. Other way, the method called gives life to the user action.

For more clarity,

- ❖ **Action by user:** Mouse click over a button
- ❖ **Event generated:** ActionEvnet
- ❖ **Listener that handles ActionEvent:** ActionListener
- ❖ **Method called implicitly:** actionPerformed() method.

  The code of actionPerformed() contains the actions of what is to be done (like user name and password validation etc.) when the user clicks a button.

Steps involved in event handling(in above case)

- ✓ 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.

The KeyEvent Class

→ A **KeyEvent** is generated when keyboard input occurs.

→ There are three types of key events, which are identified by these integer constants: **KEY_PRESSED, KEY_RELEASED**, and **KEY_TYPED**. The first two events are generated when any key is pressed or released. The last event occurs only when a character is generated. Remember, not all keypresses result in characters. For example, pressing shift does not generate a character.

→ There are many other integer constants that are defined by **KeyEvent**. 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_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

The **VK** constants specify virtual key codes and are independent of any modifiers, such as control, shift, or alt.

→ **KeyEvent** is a subclass of **InputEven**t. Here is one of its constructors:

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

Here, **src** is a reference to the component that generated the event. The type of the event is specified by **type**. The system time at which the key was pressed is passed in **when**. The **modifiers** argument indicates which modifiers were pressed when this key event occurred. The virtual key code, such as **VK_UP**, **VK_A,** and so forth, is passed in code. The character equivalent (if one exists) is passed in **ch**. If no valid character exists, then **ch** contains **CHAR_UNDEFINED**.

For **KEY_TYPED** events, code will contain **VK_UNDEFINED**.

→ **The KeyEvent** class defines several methods, but probably the most commonly used ones are **getKeyChar( ),** which returns the character that was entered, and **getKeyCode(),** which returns the key code. Their general forms are shown here:

```
char getKeyChar( )

int getKeyCode( )
```

If no valid character is available, then **getKeyChar( )** returns **CHAR_UNDEFINED**. When a **KEY_TYPED** event occurs, **getKeyCode( )** returns **VK_UNDEFINED**.

The MouseEvent Class

→ There are eight types of mouse events. The **MouseEvent** class defines the following integer constants that can be used to identify them:

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.
MOUSE_PRESSED	The mouse was pressed.
MOUSE_RELEASED	The mouse was released.
MOUSE_WHEEL	The mouse wheel was moved.

→ **MouseEvent** is a subclass of InputEvent. Here is one of its constructors:

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

Here, **src** is a reference to the component that generated the event. The type of the event is specified by **type**. The system time at which the mouse event occurred is passed in **when**. The **modifiers** argument indicates which modifiers were pressed when a mouse event occurred. The coordinates of the mouse are passed in **x** and **y**. The click count is passed in **clicks**. The **triggersPopup** flag indicates if this event causes a pop-up menu to appear on this platform.

→ Two commonly used methods in this class are **getX( )** and **getY( ).** These return the X and Y coordinates of the mouse within the component when the event occurred. Their forms are shown here:

```
int getX( )
int getY( )
```

Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse. It is shown here:

Point getPoint( )

It returns a **Point** object that contains the **X,Y** coordinates in its integer members: **x** and **y**.

→ The **translatePoint( )** method changes the location of the event. Its form is shown here:

```
void translatePoint(int x, int y)
```

Here, the arguments **x** and **y** are added to the coordinates of the event.

→ The **getClickCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:

```
int getClickCount( )
```

→ The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:
```
boolean isPopupTrigger( )
```
→ Also available is the **getButton( )** method, shown here:
```
int getButton( )
```
It returns a value that represents the button that caused the event. For most cases, the return value will be one of these constants defined by **MouseEvent**:

NOBUTTON, BUTTON1, BUTTON2 and BUTTON3

The **NOBUTTON** value indicates that no button was pressed or released.

→ Also available are three methods that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:

```
Point getLocationOnScreen( )

int getXOnScreen( )

int getYOnScreen( )
```

The **getLocationOnScreen( )** method returns a Point object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

The TextEvent Class

→ Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program.
→ **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED.**
→ The one constructor for this class is shown here:

```
TextEvent(Object src, int type)
```

Here, **src** is a reference to the object that generated this event. The type of the event is specified by **type**.

→ The **TextEvent** object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information. This operation differs from other event objects discussed in this section. Think of a text event notification as a signal to a listener that it should retrieve information from a specific text component.

The WindowEvent Class

❖ There are ten types of window events. The **WindowEvent** class defines integer constants that can be used to identify them. The constants and their meanings are shown here:

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.

→ **WindowEvent** is a subclass of **ComponentEvent**. It defines several constructors. The first is

```
WindowEvent(Window src, int type)
```

Here, **src** is a reference to the object that generated this event. The type of the event is **type**. The next three constructors offer more detailed control:

```
WindowEvent(Window src, int type, Window other)

WindowEvent(Window src, int type, int fromState, int toState)

WindowEvent(Window src, int type, Window other, int fromState,
int toState)
```

Here, **other** specifies the opposite window when a focus or activation event occurs. The **fromState** specifies the prior state of the window, and **toState** specifies the new state that the window will have when a window state change occurs.

→ A commonly used method in this class is **getWindow( ).** It returns the Window object that generated the event. Its general form is shown here:

Window getWindow( )

→ **WindowEvent** also defines methods that return the opposite window (when a focus or activation event has occurred), the previous window state, and the current window state. These methods are shown here:

```
Window getOppositeWindow( )
int getOldState( )
int getNewState( )
```

Sources of Events

→ Objects from which events are generated are source of events .

Event Sources	Description
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Choice	Generates item events when the choice is changed.
MenuItem	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scrollbar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed , deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

As explained, the delegation event model has two parts: sources and listeners. Listeners are created by implementing one or more of the interfaces defined by the **java.awt.event** package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument. Following table lists several commonly used listener interfaces and provides a brief description of the methods that they define.

Interface	Description
ActionListener	Define one method to receive action events
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
Focus Listener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item is changes.
KeyListener	Defines 3 methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines 5 methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when window gains or loses focus.
WindowListner	Defines 7 methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The ActionListener Interface

This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:

```
void actionPerformed(ActionEvent ae)
```

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, **componentAdded( )** is invoked. When a component is removed from a container, **componentRemoved( )** is invoked. Their general forms are shown here:

```
void componentAdded(ContainerEvent ce)

void componentRemoved(ContainerEvent ce)
```

The MouseListener Interface

This interface defines five methods. If the mouse is pressed and released at the same point, **mouseClicked( )** is invoked. When the mouse enters a component, the **mouseEntered( )** method is called. When it leaves, **mouseExited( )** is called. The **mousePressed( )** and **mouseReleased( )** methods are invoked when the mouse is pressed and released, respectively.

The general forms of these methods are shown here:

```
void mouseClicked(MouseEvent me)

void mouseEntered(MouseEvent me)

void mouseExited(MouseEvent me)

void mousePressed(MouseEvent me)

void mouseReleased(MouseEvent me)
```

The WindowListener Interface

This interface defines seven methods. The **windowActivated( )** and **windowDeactivated( )** methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the **windowIconified( )** method is called. When a window is deiconified, the **windowDeiconified( )** method is called. When a window is opened or closed, the **windowOpened( )** or **windowClosed( )** methods are called, respectively. The **windowClosing( )** method is called when a window is being closed. The general forms of these methods are

```
void windowActivated(WindowEvent we)

void windowClosed(WindowEvent we)

void windowClosing(WindowEvent we)

void windowDeactivated(WindowEvent we)

void windowDeiconified(WindowEvent we)

void windowIconified(WindowEvent we)

void windowOpened(WindowEvent we)
```

Category	Event	Method
Windows Events	The user clicks on the cross button.	void windowClosing (WindowEvent e)
	The window opened for the first time.	void windowOpened (WindowEvent e)
	The window is activated.	void windowActivated (WindowEvent e)
	The window is deactivated.	void windowDeactivated (WindowEvent e)
	The window is closed.	void windowClosed (WindowEvent e)
	The window is minimized	void windowIconified (WindowEvent e)
	The window maximized	void windowDeiconified (WindowEvent e)

Using the Delegation Event Model

Using the delegation event model is actually quite easy. Just follow these two steps:

1. Implement the appropriate interface in the listener so that it can receive the type of event desired.
2. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

So, before an application can respond to an event for a particular GUI component, you must:

1. Create a class that represents the event handler and implements an appropriate interface—known as an **event-listener interface**.
2. Indicate that an object of the class from Step 1 should be notified when the event occurs—known as **registering the event handler**.

Points to remember about listener

- In order to design a listener class we have to implement 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.

Where to put event handling codes?

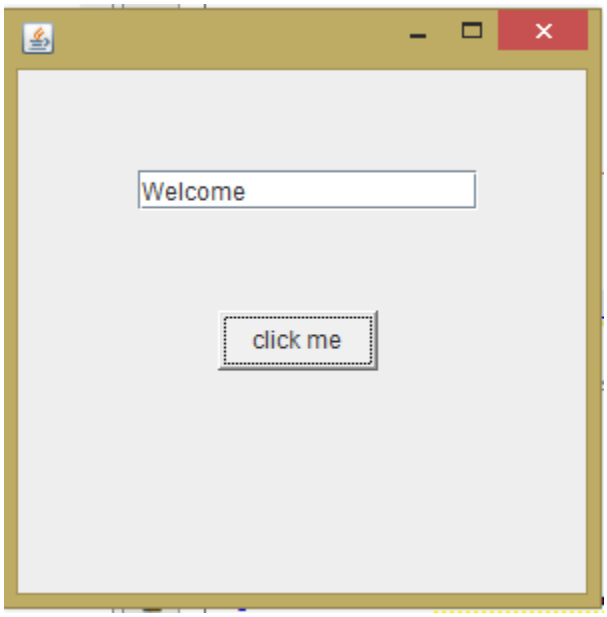
We can put the event handling code into one of the following places:

- Within class
- Other class
- Inner Class / Anonymous inner class

Example of event handling within a class

```
import javax.swing.*;
      import java.awt.event.*;
      class EventDemo extends JFrame implements ActionListener{
      JTextField tf;
      EventDemo(){
        //create components
      tf=new JTextField();
      tf.setBounds(60,50,170,20);
      JButton b=new JButton("click me");
      b.setBounds(100,120,80,30);
        //register listener
      b.addActionListener(this);//passing current instance
        //add components and set size, layout and visibility
      add(b);add(tf);
      setSize(300,300);
      setLayout(null);
      setVisible(true);
```

```
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new EventDemo();
}
}
```



Example of event handling by Outer Class (other class)

```
package eventhandlingpackage;
import javax.swing.*;
class AE2 extends JFrame{
JTextField tf;
AE2(){
//create components
tf=new JTextField();
tf.setBounds(60,50,170,20);
JButton b=new JButton("click me");
b.setBounds(100,120,80,30);
//register listener
Outer o=new Outer(this);
b.addActionListener(o);//passing outer class instance
//add components and set size, layout and visibility
```

```
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public static void main(String args[]){
new AE2();
}
}
```

```
package eventhandlingpackage;

import java.awt.event.*;

/**
 *
 * @author BIPIN
 */
class Outer implements ActionListener{

AE2 obj;

Outer(AE2 obj){

this.obj=obj;

}

public void actionPerformed(ActionEvent e){

obj.tf.setText("welcome");

}

}
```

Example of Event handling by Inner class(Anonymous Class)

```
import javax.swing.*;
import java.awt.event.*;
class AE3 extends JFrame{
JTextField tf;
AE3(){
tf=new JTextField();
tf.setBounds(60,50,170,20);
JButton b=new JButton("click");
b.setBounds(50,120,80,30);
ActionListener ac;
    ac = new ActionListener(){
        public void actionPerformed(ActionEvent e){
            tf.setText("hello");
        }
    };
b.addActionListener(ac);
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public static void main(String args[]){
new AE3();
}
}
```

Important Event Classes and Interface

Event Classes	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 and also when it 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

Handling Mouse Events

To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces

Handling Keyboard Events

To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListener** interface. Before looking at an example, it is useful to review how key events are generated.

→ When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler. When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed. If a character is generated by the keystroke, then a **KEY_TYPED** event is sent and the **keyTyped( )** handler is invoked.

Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the key press and release events. However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed( )** handler.

Adapter Classes

Java provides a special feature, called an adapter class, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface. You 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 you are interested.

What is the problem with listener interfaces?

Actually problem does not occur with every listener, but occurs with a few. A few listeners contain more than one abstract method. For example, **WindowListener**, used for frame closing, comes with 7 abstract methods. We are interested in only one abstract method to close the frame, but being **WindowListener** is an interface, we are forced to override remaining 6 methods also, just at least with empty body. This is the only problem, else fine. Some listeners like **ActionListener** comes with only one abstract method and with them no problem at all.

→ Java **adapter classes** provide the default implementation of listener interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.
→ Adapters are replacement to listeners. The advantage with adapter is we can override any number of methods we would like and not all. For example, if we use **WindowAdapter**(instead of **WindowListener**), we can override only one method to close the frame.
→ Adapters make event handling simple. Any listener has more than one abstract method has got a corresponding adapter class. For example, **MouseListener** with 5 abstract methods has got a corresponding adapter known as **MouseAdapter**.
But **ActionListener** and **ItemListener** do not have corresponding adapter class as they contain only one abstract method.
→ Adapters are abstract classes introduced from JDK 1.1.

The adapter classes are found in ***java.awt.event***, ***java.awt.dnd*** and ***javax.swing.event*** packages. The Adapter classes with their corresponding listener interfaces are given below.

java.awt.event Adapter classes

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

Listener Interfaces vs Adapter Classes

Any class which handles events must implement a listener interface for the event type it wishes to handle. A class can implement any number of listener interfaces. To implement a listener interface the class must implement every method in the interface. Sometimes you may want to implement only one or two of the methods in an interface. You can avoid having to write your

own implementation of all the methods in an interface by using an adapter class. An adapter class is a class that already implements all the methods in its corresponding interface. Your class must extend the adapter class by inheritance so you can extend only one adapter class in your class (Java does not support multiple inheritance). Here are the Java adapter classes and the listener interfaces they implement. Notice that adapters exist for only listener interfaces with more than one method.

LISTENER INTERFACE	CORRESPONDING ADAPTER
WindowListener (7)	WindowAdapter
MouseListener (5)	MouseAdapter
MouseMotionListener (2)	MouseMotionAdapter
KeyListener (3)	KeyAdapter
FocusListener (2)	FocusAdapter

The values in the parentheses indicate the abstract methods available in the listener

Inner Classes

→ An inner class is a class defined within another class, or even within an expression.

→ Listener classes are generally designed with just one purpose in mind: the creation of the listener object for some GUI object (e.g., a button, window, checkbox, etc...). Given this relationship, separating the code for the Listener class from the code for the class of the related GUI object seems to, in a certain sense, work against the notion of the benefits of code encapsulation.

→ Fortunately, in Java, within the scope of one class, we are allowed to define one or more inner classes(or nested classes). (Technically, the compiler turns an inner class into a regular class file named: OuterClassName$InnerClassName.class.

→ Such inner classes can be defined inside the framing class, but outside its methods -- or it may even be defined inside a method, and they can reference data and methods defined in the outer class in which it nests.

Importance of using Inner Class in Event Handling

→ What if you want to use an adapter class, but do not want your public class to inherit from an adapter class? For example, suppose you write an applet, and you want your Applet subclass to contain some code to handle mouse events. Since the Java language does not permit multiple inheritance, your class cannot extend both the Applet and **MouseAdapter** classes. A solution is to define an inner class a class inside of your Applet subclass that extends the **MouseAdapter** class.

→ Inner classes can also be useful for event listeners that implement one or more interfaces directly.

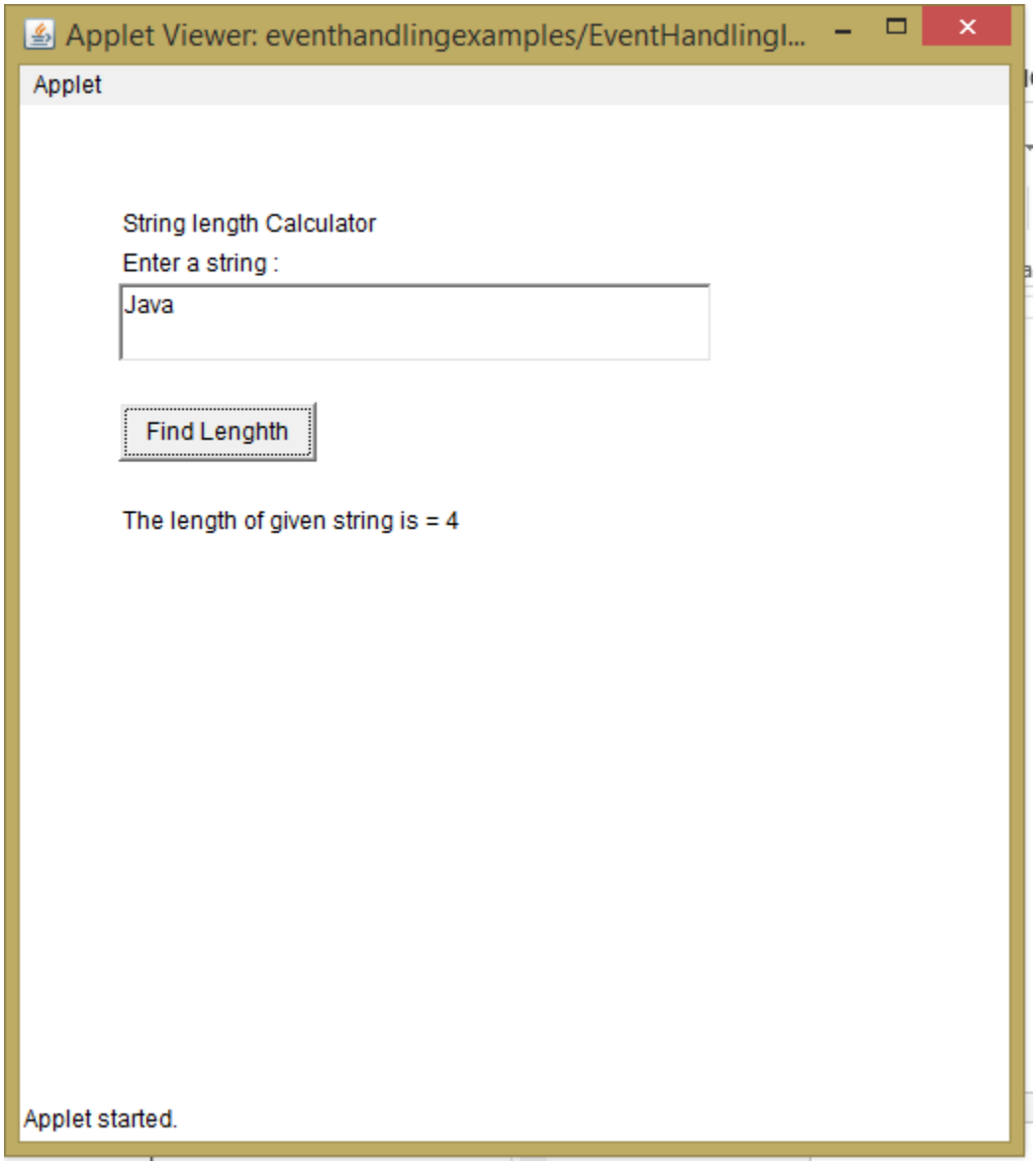
Anonymous inner classes

We can also create **Anonymous** inner classes (i.e., inner classes that have no name) to shorten the process of declaring an inner class and creating an instance of that class into one step. While it might look strange at first glance, anonymous inner classes can make your code easier to read because the class is defined where it is referenced. However, you need to weigh the convenience against possible performance implications of increasing the number of classes.

Some Event handling Examples

1. **Program to compute the length of string entered in an applet.**

```
package eventhandlingexamples;

import java.applet.Applet;

import java.awt.*;

import java.awt.event.*;


/**
 *
 * @author BIPIN
 */
public class EventHandlingInApplet extends Applet
implements ActionListener {
/*<applet code="EventHandlingInApplet" width=500
height=500></applet>*/
    Label l1 = new Label("String length Calculator");
    Label l2 =new Label("Enter a string : ");
    Label l3 = new Label();
    TextField t = new TextField();
    Button b = new Button("Find Lenghth");
    public void init() {
        setSize(500,500);
        setLayout(null);
        l1.setBounds(50,50,200,20);
        l2.setBounds(50,70,200,20);
        t.setBounds(50,90,300,40);
        b.setBounds(50,150,100,30);
        //Regestering the listener
        b.addActionListener(this);
        l3.setBounds(50,200,300,20);
    }
    public void start(){
        add(l1);add(l2);add(t);add(b);add(l3);
    }
   public void actionPerformed(ActionEvent ae){
       String s= t.getText();
       int len = s.length();
       l3.setText("The length of given string is = "+ len);
   }

}
```



2. Keyboard Event handling Examples :

2.1.Key Event Demo Program

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

public class KeyEventDemo implements KeyListener {
    JFrame f = new JFrame("KeyEventDemo");
```

```
    JLabel l = new JLabel("");
    public void getReady(){
    f.setSize(500,500);
    f.setLayout(null);
    l.setBounds(20,50,300,20);
    f.addKeyListener(this);
    f.add(l);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setVisible(true);
    }
    @Override
    public void keyPressed(KeyEvent ke){

        l.setText("Key is pressed...");
        if(KeyEvent.VK_ENTER==ke.getKeyChar()){
      int code = ke.getKeyCode();
   l.setText("Enter is Pressed and key Code is "+ code);
        }
    }
    @Override
    public void keyReleased(KeyEvent ke){
    //l.setText("Key is released...");

    }
    @Override
    public void keyTyped(KeyEvent ke){
    }

public static void main(String args[]){
  KeyEventDemo d= new KeyEventDemo();
  d.getReady();

}
}
```

2.2.Keyboard event handling : word counter example

```
public class WordCounter implements KeyListener {
    JFrame f = new JFrame("WordCounterWithKeyListner");
    JLabel l = new JLabel("");
    JTextArea a = new JTextArea();
    public void getReady(){
    f.setSize(500,500);
    f.setLayout(null);
    l.setBounds(20,50,300,20);
```

```
        a.setBounds(50,100,200,100);
        a.addKeyListener(this);
        f.add(l);f.add(a);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
        }
        @Override
        public void keyPressed(KeyEvent e){

            l.setText("Key is pressed...");
        }
        @Override
        public void keyReleased(KeyEvent e){
        //l.setText("Key is released...");
        String text = a.getText();
        String words[]= text.split("\\s");
        l.setText("Words=" + words.length +
    "Characters="+text.length());


        }
        @Override
        public void keyTyped(KeyEvent e){
        l.setText("Typing...");
        }

    public static void main(String args[]){
      WordCounter w= new WordCounter();
      w.getReady();

    }
    }
```

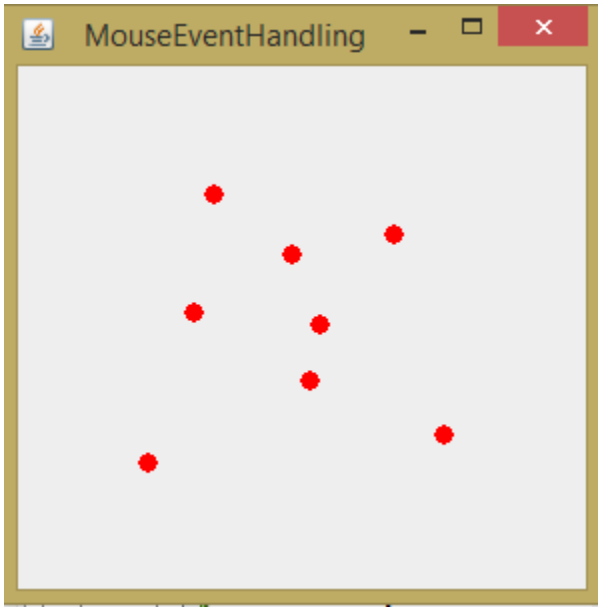
3. Example of window event handling using WindowListener interface

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

/**
 *
 * @author BIPIN
 */
public class WindowEventExample implements WindowListener {
    JFrame f = new JFrame("WindowEvents");
    public void prepareGui(){
```

```
        f.addWindowListener(this);
        f.setSize(400,400);
        f.setVisible(true);
    }
     @Override
    public void windowActivated(WindowEvent e){
        System.out.println("Activated");

    }
     @Override
    public void windowDeactivated(WindowEvent e){
         System.out.println("DeActivated");
    }
     @Override
    public void windowOpened(WindowEvent e){
         System.out.println("Opened");
    }
     @Override
    public void windowClosed(WindowEvent e){
         System.out.println("Closed");
    }
     @Override
    public void windowClosing(WindowEvent e){
         System.out.println("Closing");
         f.dispose();

    }
     @Override
    public void windowIconified(WindowEvent e){
         System.out.println("Iconified");

    }
     @Override
    public void windowDeiconified(WindowEvent e){
         System.out.println("DeIconified");

    }
    public static void main(String[] args){
       WindowEventExample e = new WindowEventExample();
       e.prepareGui();
    }


}
```

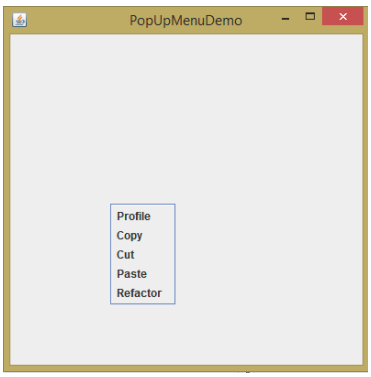
4. Handling Mouse Event with MouseAdapter class

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

/**
 *
 * @author BIPIN
 */
public class MouseEventWithAdapter extends MouseAdapter {
  JFrame f = new JFrame("MouseEventHandling");

  void getReady(){
      f.setSize(300,300);
      f.setLayout(null);
      f.addMouseListener(this);
      f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      f.setVisible(true);
  }
  @Override
  public void mouseClicked(MouseEvent e){
   Graphics g= f.getGraphics();
      g.setColor(Color.red);

      g.fillOval(e.getX(), e.getY(), 10, 10);


  }
  public static void main(String args[]){
      MouseEventWithAdapter me =new
MouseEventWithAdapter();
      me.getReady();
  }
}
```


MouseEventHandling

5. Popup menu Example with mouse event handling

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


/**
 *
 * @author BIPIN
 */
public class PopupMenuDemo extends MouseAdapter{
    JFrame f= new JFrame("PopUpMenuDemo");
    JPopupMenu pm = new JPopupMenu("DoSomething");
    JMenuItem i1 = new JMenuItem("Profile");
    JMenuItem i2 = new JMenuItem("Copy");
    JMenuItem i3 = new JMenuItem("Cut");
    JMenuItem i4 = new JMenuItem("Paste");
    JMenuItem i5 = new JMenuItem("Refactor");
    PopupMenuDemo(){
        pm.add(i1); pm.add(i2); pm.add(i3); pm.add(i4);
pm.add(i5);
        f.setSize(400,400);
        f.addMouseListener(this);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setLayout(null);
        f.setVisible(true);
        f.add(pm);



    }
    public void mouseClicked(MouseEvent e){
        if(e.getButton()== MouseEvent.BUTTON3){
        pm.show(f, e.getX(),e.getY());
        }
    }

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

}
```



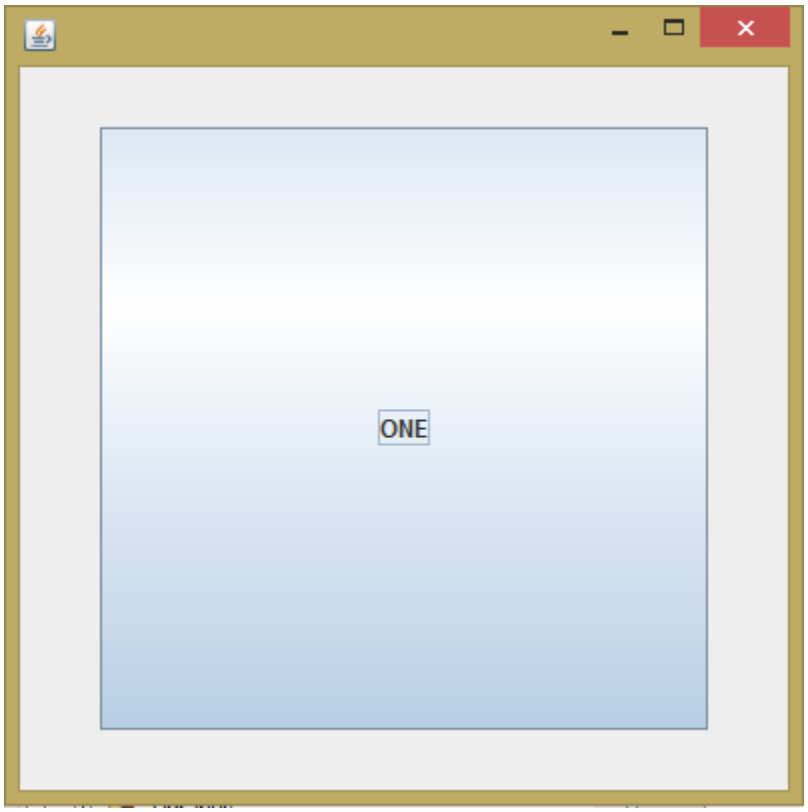
6. Handling AWT Frame event with inner class

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

public class WindowAdapterDemo extends Frame {
    void prepareWindow(){
        setTitle("Window event in AWT frame");
        setSize(400,400);
        setVisible(true);
        MyHandler mh = new MyHandler();
        addWindowListener(mh);
    }
    //using inner class
    public class MyHandler extends WindowAdapter{
        @Override
    public void windowClosing(WindowEvent we){
        setVisible(false);
        dispose();
    }
    }

    public static void main(String[] args) {
        WindowAdapterDemo wad= new WindowAdapterDemo();
        wad.prepareWindow();

    }
}
```

7. Using anonymous inner class for event handling (Card Layout Demo)

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

import javax.swing.*;

public class CardLayoutDemo extends JFrame {
CardLayout card;
JButton b1,b2,b3;
Container c;
    CardLayoutDemo(){
        c=getContentPane();
        card=new CardLayout(40,30);
//create CardLayout object with 40 hor space and 30 ver
space
        c.setLayout(card);

        b1=new JButton("ONE");
        b2=new JButton("TWO");
        b3=new JButton("THREE");

        ActionListener ac;
        //using anonymous inner class
        ac = new ActionListener(){
        @Override
        public void actionPerformed(ActionEvent e) {
        card.next(c);
        }
       };

        b1.addActionListener(ac);
        b2.addActionListener(ac);
        b3.addActionListener(ac);

        c.add(b1);c.add(b2);c.add(b3);

    }




    public static void main(String[] args) {
        CardLayoutDemo cd=new CardLayoutDemo();
```

```
        cd.setSize(400,400);
        cd.setVisible(true);
        cd.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
}
```



8. Mouse Motion Event example

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

/**
 *
 * @author BIPIN
 */
public class MouseMotionExample implements
MouseMotionListener {
    JFrame f = new JFrame("MouseMotionEvent");
    JPanel p = new JPanel();
    MouseMotionExample(){
        p.setSize(400,400);
```

```
        p.addMouseMotionListener(this);
        f.add(p);
        f.setLayout(null);
        f.setSize(400,400);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setVisible(true);
    }

    public void mouseDragged(MouseEvent e){
        Graphics g = p.getGraphics();
        g.setColor(Color.blue);
        g.fillOval(e.getX(), e.getY(), 10, 10);
    }
    public void mouseMoved( MouseEvent e){
    }
    public static void main(String[] args){
        new MouseMotionExample();
    }

}
```