

Chapter 2...

Event Handling

Weightage of Marks = 20, Teaching Hours = 10

Contents

- 2.1 Introduction
 - 2.1.1 What is an Event?
 - 2.1.2 What is Event Handling?
- 2.2 Delegation Event Model
 - 2.2.1 Events
 - 2.2.2 Event Sources
 - 2.2.3 Event Listeners
 - 2.2.4 Event Classes
 - 2.2.4.1 ActionEvent
 - 2.2.4.2 ComponentEvent Class
 - 2.2.4.3 ContainerEvent Class
 - 2.2.4.4 FocusEvent Class
 - 2.2.4.5 ItemEvent Class
 - 2.2.4.6 KeyEvent Class
 - 2.2.4.7 MouseEvent Class
 - 2.2.4.8 TextEvent Class
 - 2.2.4.9 WindowEvent Class
 - 2.2.4.10 Sources of Events
- 2.3 Adapter Classes
- 2.4 Inner Classes
- 2.5 Event Listener Interfaces
 - 2.5.1 ActionListener Interface
 - 2.5.2 ComponentListener Interface
 - 2.5.3 ContainerListener Interface
 - 2.5.4 FocusListener Interface
 - 2.5.5 ItemListener Interface
 - 2.5.6 KeyListener Interface
 - 2.5.7 MouseListener Interface
 - 2.5.8 MouseMotion Interface
 - 2.5.9 TextListener Interface
 - 2.5.10 WindowListener Interface
 - 2.5.11 WindowFocusListener Interface
- Important Points
- Practice Questions

Objectives

- To Write Event Driven Programs using the Delegation Event Model
- To Learn the Concept of Adapter Classes and the Inner Classes

2.1 INTRODUCTION

- Applets and Frame (GUI Program) are event driven. So it is important to understand in a general way how the event-driven architecture works with Window.
- A GUI program contains a set of interrupt service routines.
- A GUI program waits until an event (For example, Button is pressed, List item is selected i.e. some operation is performed on the control) occurs.
- GUIs generate events when we interact with GUI. For example, clicking a button, moving the mouse, closing Window etc. Both AWT and swing components generate events.
- In java, events are represented by Objects. These objects tell us about event and its source. Examples are: ActionEvent (Clicking a button), WindowEvent (Doing something with window e.g. closing, minimizing).
- Some event classes of java.awt.event are shown in Fig. 2.1.

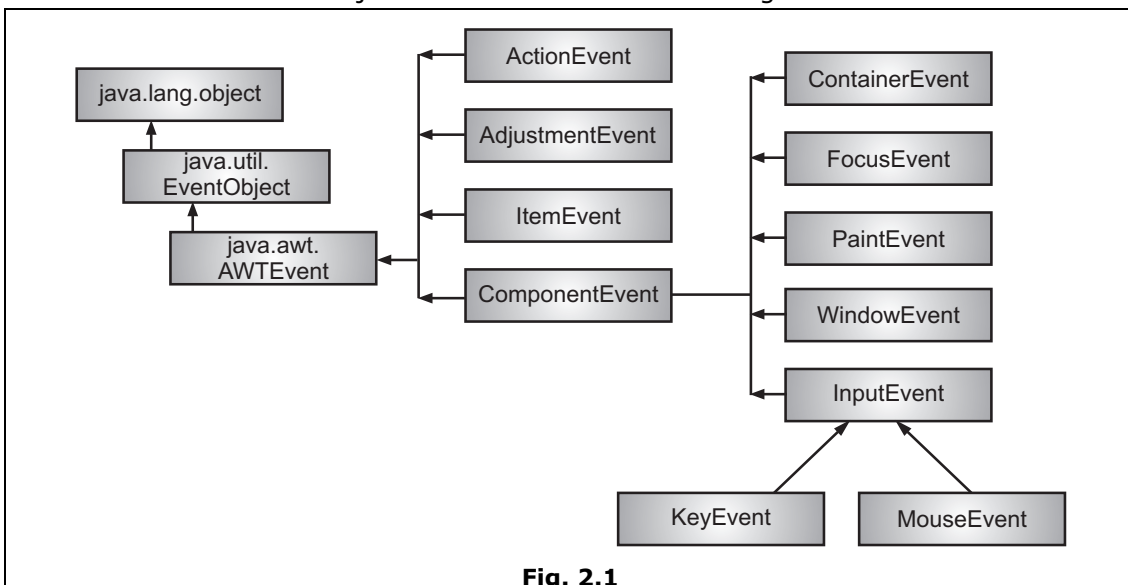


Fig. 2.1

2.1.1 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 events can be broadly classified into two categories:
 - 1. 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 (GUI).
 - For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.
 - 2. 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.

2.1.2 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.

2.2 DELEGATION EVENT MODEL

- Applets are event-driven programs. Thus, event handling is at the core of successful applet programming.
- Most events to which our applet will respond are generated by the user. These events are passed to our applet in a variety of ways, with the specific method depending upon the actual event.
- There are several types of events. The most commonly handled events are those generated by the mouse, the keyboard, and various controls, such as a push button.
- Events are supported by the `java.awt.event` package.
- In Java both AWT and Swing components use Event Delegation Model. In this model, processing of an event is delegated to a particular object (handlers) in the program.
- The modern approach to handling events is based on the delegation event model as shown in Fig. 2.2.

- This model 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 received, the listener processes the event and then returns.

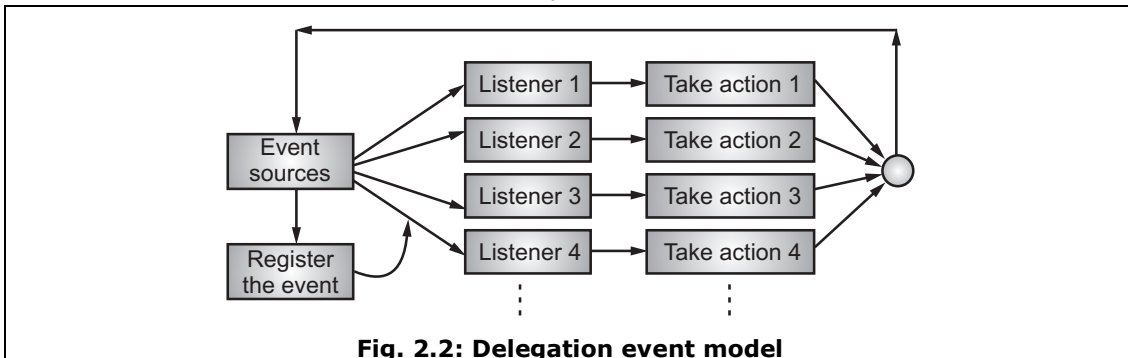


Fig. 2.2: Delegation event model

- 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 i.e. 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 old Java 1.0 approach.
- Java also allows us to process events without using the delegation event model. This can be done by extending an AWT component. However, the delegation event model is the preferred design for the reasons just cited.

2.2.1 Events

- In the delegation model, an event is an object that describes a state change in a source.
- It 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.
- Many other user operations could also be cited or considered as examples.
- Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed.

2.2.2 Events Sources

- A source is an object that generates an event.
- Event sources are components, subclasses of `java.awt.Component`, capable to generate events.
- Events 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.
- The general form/syntax:

```
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/syntax of such a method is this:

```
public void addTypeListener(TypeListener el)
    throws java.util.TooManyListenersException
```

- 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 uni-casting the event.
- A source must also provide a method that allows a listener to un-register an interest in a specific type of event. The general form/syntax 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, we 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.

2.2.3 Event Listeners

- The events generated by the GUI components are handled by a special group of interfaces known as "listeners".
- A listener is an object that is notified when an event occurs.
- It has two major requirements:
 1. It must have been registered with one or more sources to receive notifications about specific types of events.
 2. It must implement methods to receive and process these notifications.
- The methods that receive and process events are defined in a set of interfaces found in `java.awt.event`.
- For example, the `MouseListener` 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.
- Fig. 2.3 shows Event Listeners.

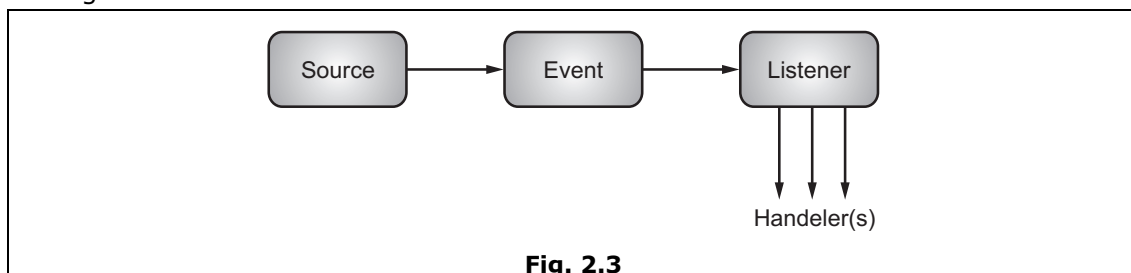


Fig. 2.3

2.2.4 Event Classes

- Event class provide a consistent, easy-to-use means of encapsulating events.
- Almost every event source 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`.
- The classes that represent events are at the core of Java's event handling mechanism. At the root of the Java event class hierarchy is `EventObject`, which is in `java.util`. It is the superclass for all events.
- `EventObject` contains two methods i.e. `getSource()` and `toString()`.
- The `getSource()` method returns the source of the event. Its general form/syntax:

```
Object getSource( )
```

- As expected, `toString()` returns the string equivalent of the event. Its general form/syntax.

```
String toString():
```

- The class `AWTEvent`, defined within the `java.awt` package, is a subclass of `EventObject`. It is the super class, (either directly or indirectly) of all AWT-based events used by the delegation event model.
 - `EventObject` is a superclass of all events.
 - `AWTEvent` is a superclass of all AWT events that are handled by the delegation event model.
- The package `java.awt.event` defines several types of events that are generated by various user interface elements.
- Event class description in Fig. 2.4 are listed below:
 - 1. ActionEvent:** Generated when a button is pressed, a list item is double clicked, or a menu item is selected.
 - 2. AdjustmentEvent:** Generated when a scroll bar is manipulated.
 - 3. ComponentEvent:** Generated when a component is hidden, moved, resized, or becomes visible.
 - 4. ContainerEvent:** Generated when a component is added to or removed from a container.
 - 5. FocusEvent:** Generated when a component gains or losses keyboard focus.
 - 6. InputEvent:** Abstract super class for all component input event classes.
 - 7. ItemEvent:** Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.
 - 8. KeyEvent:** Generated when input is received from the keyboard.
 - 9. MouseEvent:** Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
 - 10. MouseWheelEvent:** Generated when the mouse wheel is moved.
 - 11. TextEvent:** Generated when the value of a text area or text field is changed.
 - 12. WindowEvent:** Generated when a window is activated, closed, deactivated, deiconified, iconified, opened or quit.

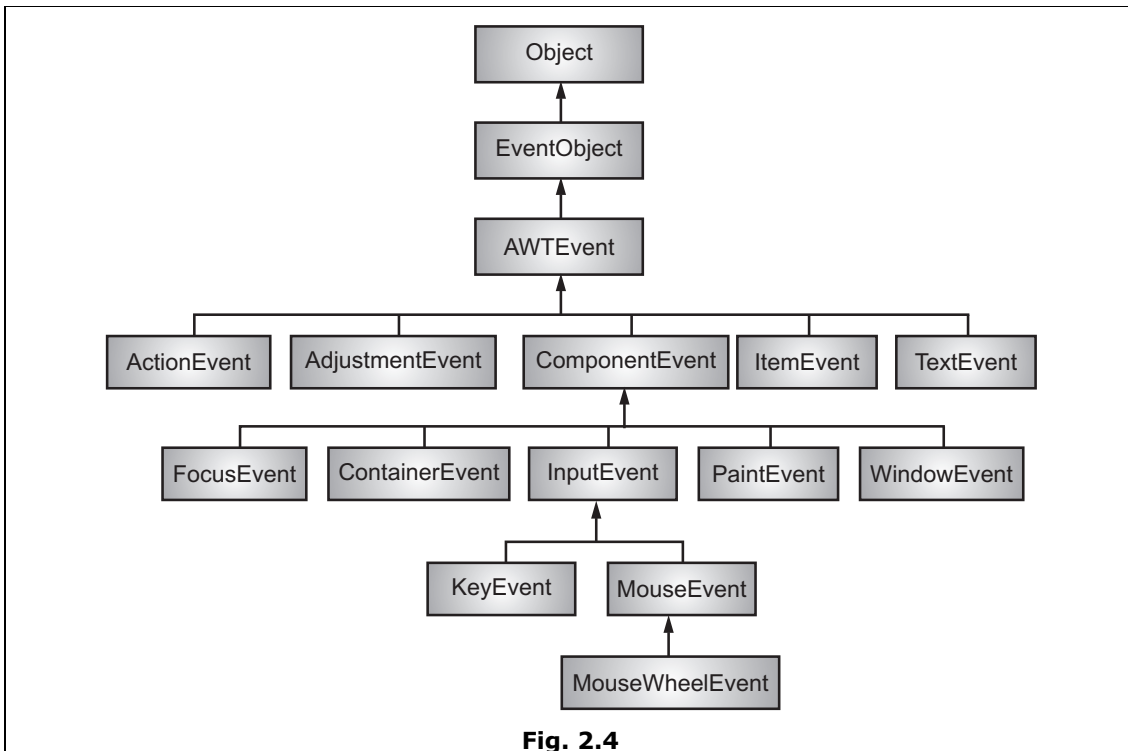


Fig. 2.4

2.2.4.1 ActionEvent Class

- The `ActionEvent` class is defined in `java.awt.event` package.
- An `ActionEvent` is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
- The `ActionEvent` class defines four integer constants that can be used to identify any modifiers associated with an action event:
 `ALT_MASK`, `CTRL_MASK`, `META_MASK`, and `SHIFT_MASK`.
- In addition, there is an integer constant, `ACTION_PERFORMED`, which can be used to identify action events.
- **Constructors:**
 1. `ActionEvent(java.lang.Object source, int id, java.lang.String command)`: Constructs an `ActionEvent` object.
 2. `ActionEvent(java.lang.Object source, int id, java.lang.String command, int modifiers)`: Constructs an `ActionEvent` object with modifier keys.
 3. `ActionEvent(java.lang.Object source, int id, java.lang.String command, long when, int modifiers)`: Constructs an `ActionEvent` object with the specified modifier keys and timestamp.

- We can obtain the command name for the invoking `ActionEvent` object by using the `getActionCommand()` method.

Syntax: `String getActionCommand()`

For example, when a button is pressed, an action event is generated that has a command name equal to the label on that button.

- The `getModifiers()` method returns a value that indicates which modifier keys (ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated.

Syntax: `int getModifiers()`

- The method `getWhen()` that returns the time at which the event took place. This is called the event's timestamp.
- The syntax for `getWhen()` method is shown as: `long getWhen()`.

2.2.4.2 ComponentEvent Class

- The Class `ComponentEvent` represents that a component moved, changed size, or changed visibility.
- A `ComponentEvent` is generated when the size, position or visibility of a component is changed. There are four types of component events.
- The `ComponentEvent` class defines integer constants that can be used to identify them. The constants and their meanings are shown below:

1. **Component_Hidden:** The component was hidden.
2. **Component_Moved:** The component was moved.
3. **Component_Resized:** The component was resized.
4. **Component_Shown:** The component became visible.

- `ComponentEvent` has this constructor:

```
ComponentEvent(Component 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 `getComponent()` method returns the component that generated the event. It is shown below:

```
Component getComponent( )
```

2.2.4.3 ContainerEvent Class

- A `ContainerEvent` is generated when a component is added to or removed from a container. There are two types of container events.
- The `ContainerEvent` class defines `int` constants that can be used to identify them: `Component_Added` and `Component_Removed`. They indicate that a component has been added to or removed from the container.

- ContainerEvent is a subclass of ComponentEvent and has this constructor.

```
ContainerEvent(Component src, int type, Component comp)
```
- Here, src is a reference to the container that generated this event. The type of the event is specified by type and the component that has been added to or removed from the container is comp.

AdjustmentEvent:

- The Class AdjustmentEvent represents adjustment event emitted by Adjustable objects.
- An AdjustmentEvent is generated by a scroll bar.
- There are five types of adjustment events. The AdjustmentEvent class defines integer constants that can be used to identify them.
- The constants and their meanings are shown here:
 1. **BLOCK_DECREMENT:** The user clicked inside the scroll bar to decrease its value.
 2. **MouseEvent:** Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
 3. **MouseWheelEvent:** Generated when the mouse wheel is moved.
 4. **TextEvent:** Generated when the value of a text area or text field is changed.
 5. **WindowEvent:** Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Constructors:

1. AdjustmentEvent(Adjustable source, int id, int type, int value): Constructs an AdjustmentEvent object with the specified Adjustable source, event type, adjustment type, and value.
2. AdjustmentEvent(Adjustable source, int id, int type, int value, boolean isAdjusting): Constructs an AdjustmentEvent object with the specified Adjustable source, event type, adjustment type and value.

2.2.4.4 FocusEvent Class

- A FocusEvent is generated when a component gains or loses input focus.
- These events are identified by the integer constants FOCUS_GAINED and FOCUS_LOST.
- FocusEvent is a subclass of ComponentEvent. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.)
- The other component involved in the focus change, called the opposite component, is passed in other.
- Therefore, if a FOCUS_GAINED event occurred, other will refer to the component that lost focus. Conversely, if a FOCUS_LOST event occurred, other will refer to the component that gains focus.

2.2.4.5 ItemEvent Class

- An ItemEvent is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.
- There are two types of item events, which are identified by the following integer constants:
 1. **DESELECTED:** The user deselected an item.
 2. **SELECTED:** The user selected an item.
- In addition, ItemEvent defines one integer constant, ITEM_STATE_CHANGED, that signifies a change of state.
- The getItem() method can be used to obtain a reference to the item that generated an event. Its syntax is shown here:

```
Object getItem( )
```

- The getItemSelectable() method can be used to obtain a reference to the ItemSelectable object that generated an event. Its general form/syntax is shown here:
- Lists and choices are examples of user interface element that implement the ItemSelectable interface.
- The getStateChange() method returns the state change (i.e., SELECTED or DESELECTED) for the event. It is shown here:

```
int getStateChange( )
```

2.2.4.6 KeyEvent Class

- On entering the character the Key event is generated.
- 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 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.
- 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 and are independent of any modifiers, such as control, shift or alt.
- KeyEvent have following **constructor**:
 1. `KeyEvent(Component src, int type, long when, int modifiers, int code)`
 2. `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.

2.2.4.7 MouseEvent Class

- This event indicates a mouse action occurred in a component.
- There are eight types of mouse events. The `MouseEvent` class defines the following integer constants that can be used to identify them:
 1. **MOUSE_CLICKED**: The user clicked the mouse.
 2. **MOUSE_DRAGGED**: The user dragged the mouse.
 3. **MOUSE_ENTERED**: The mouse entered a component.
 4. **MOUSE_EXITED**: The mouse exited from a component.
 5. **MOUSE_MOVED**: The mouse moved.
 6. **MOUSE_PRESSED**: The mouse was pressed.
 7. **MOUSE_RELEASED**: The mouse was released.
 8. **MOUSE_WHEEL**: The mouse wheel was moved.
- The most commonly used methods in this class are `getX()` and `getY()`. These returns the X and Y coordinate of the mouse when the event occurred.
- Their forms/syntaxs are shown below:

```
int getX( )
```

And

```
int getY( )
```

- Alternatively, we can use the `getPoint()` method to obtain the coordinates of the mouse. It is shown below:

```
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/syntax 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 syntax is shown below:

```
int getClickCount( )
```

2.2.4.8 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 below:

```
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.

2.2.4.9 WindowEvent Class

- The object of this class represents the change in state of a window.
- This low-level event is generated by a `Window` object when it is opened, closed, activated, deactivated, iconified, or deiconified, or when focus is transferred into or out of the `Window`.
- 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:

1. **WINDOW_ACTIVATED:** The window was activated.
2. **WINDOW_CLOSED:** The window has been closed.
3. **WINDOW_CLOSING:** The user requested that the window be closed.
4. **WINDOW_DEACTIVATED:** The window was deactivated.
5. **WINDOW_DEICONIFIED:** The window was deiconified.
6. **WINDOW_GAINED_FOCUS:** The window gained input focus.
7. **WINDOW_ICONIFIED:** The window was iconified.
8. **WINDOW_LOST_FOCUS:** The window lost input focus.
9. **WINDOW_OPENED:** The window was opened.
10. **WINDOW_STATE_CHANGED:** The state of the window changed.

- `WindowEvent` is a subclass of `ComponentEvent`. The most commonly used method in this class is `getWindow()`. It returns the `Window` object that generated the event. Its general form/syntax is shown below:

```
Window getWindow( )
```

2.2.4.10 Sources of Events

- Following is list of some of the user interface components that can generate the events described in the previous section. In addition to these graphical user interface elements, other components, such as an applet, can generate events.
- For example, we receive key and mouse events from an applet, (We may also build our own components that generate events.)

Event Source Description:

1. **Button:** Generates action events when the button is pressed.
2. **Checkbox:** Generates item events when the check box is selected or deselected.
3. **Choice:** Generates item events when the choice is changed.
4. **List:** Generates action events when an item is double-clicked; Generates item events when an item is selected or deselected.
5. **Menu Item:** Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
6. **Scrollbar:** Generates adjustment events when the scroll bar is manipulated.
7. **Text components:** Generates text events when the user enters a character.
8. **Window:** Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened or quit.

2.3 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 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. For example, the `MouseMotionAdapter` class has two methods, `mouseDragged()` and `mouseMoved()`.
- The signatures of these empty methods are exactly as defined in the `MouseMotionListener` interface. If you were interested in only mouse drag events, then you could simply extend `MouseMotionAdapter` and implement `mouseDragged()`. The empty implementation of `mouseMoved()` would handle the mouse motion events for you.

- List below shows the commonly used adapter classes in java.awt.event and notes the interface that each implements.

| Adapter Class | Listener Interface |
|--------------------|---------------------|
| ComponentAdapter | ComponentListener |
| ContainerAdapter | ContainerListener |
| FocusAdapter | FocusListener |
| KeyAdapter | KeyListener |
| MouseAdapter | MouseListener |
| MouseMotionAdapter | MouseMotionListener |
| WindowAdapter | WindowListener |

- The following example demonstrates an adapter. It displays a message in the status bar of an applet viewer or browser when the mouse is clicked or dragged. However, all other mouse events are silently ignored.
- The program has three classes. AdapterDemo extends Applet. Its init() method creates an instance of MyMouseAdapter and registers that object to receive notifications of mouse events. It also creates an instance of MyMouseMotionAdapter and registers that object to receive notifications of mouse motion events.
- Both of the constructors take a reference to the applet as an argument. MyMouseAdapter implements the mouseClicked() method. The other mouse events are silently ignored by code inherited from the MouseAdapter class. MyMouseMotionAdapter implements the mouseDragged() method.
- The other mouse motion event is silently ignored by code inherited from the MouseMotionAdapter class.

Program 2.1: Program to demonstrate adapter classes.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class AdapterDemo extends Applet
{
public void init()
{
addMouseListener(new MyMouseAdapter(this));
addMouseMotionListener(new MyMouseMotionAdapter(this));
}
}
```

```
class MyMouseAdapter extends MouseAdapter
{
    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo adapterDemo)
    {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse clicked
    public void mouseClicked(MouseEvent me)
    {
        adapterDemo.showStatus("Mouse clicked");
    }
}

class MyMouseMotionAdapter extends MouseMotionAdapter
{
    AdapterDemo adapterDemo;
    public MyMouseMotionAdapter(AdapterDemo adapterDemo)
    {
        this.adapterDemo = adapterDemo;
    }
    // Handle mouse dragged
    public void mouseDragged(MouseEvent me)
    {
        adapterDemo.showStatus("Mouse dragged");
    }
}

/*<applet code="AdapterDemo" width=300 height=100>
</applet> */
```

-
- As we can see by looking at the program, not having to implement all of the methods defined by the `MouseMotionListener` and `MouseListener` interfaces saves our considerable amount of effort and prevents our code from becoming cluttered with empty methods.

2.4 INNER CLASSES

- Inner classes are class within class.
- Inner class instance has special relationship with Outer class. This special relationship gives inner class access to member of outer class as if they are the part of outer class.
- There are four types of inner classes i.e. member, static member, local and anonymous.

1. Member Inner Class:

- A member class is defined at the top level of the class. It may have the same access modifiers as variables (public, protected, package, static, final), and is accessed in much the same way as variables of that class.

Program 2.2: Program to demonstrate member inner class.

```
public class OuterClass {
    int outerVariable = 100;
    class MemberClass {
        int innerVariable = 20;
        int getSum(int parameter) {
            return innerVariable + outerVariable + parameter;
        }
    }
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        MemberClass inner = outer.new MemberClass();
        System.out.println(inner.getSum(3));
        outer.run();
    }
    void run() {
        MemberClass localInner = new MemberClass();
        System.out.println(localInner.getSum(5));
    }
}
```

2. Static Member Inner Class:

- A static member class is defined like a member class, but with the keyword static. Despite its position inside another class, a static member class is actually an "outer" class--it has no special access to names in its containing class.

- To refer to the static inner class from a class outside the containing class, use the syntax, `OuterClassName.InnerClassName`. A static member class may contain static fields and methods.

Program 2.3: Program to demonstrate static member inner class.

```
public class OuterClass
{
    int outerVariable = 100;
    static int staticOuterVariable = 200;
    static class StaticMemberClass {
        int innerVariable = 20;
        int getSum(int parameter) {
            // Cannot access outerVariable here
            return innerVariable + staticOuterVariable + parameter;
        }
    }
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        StaticMemberClass inner = new StaticMemberClass();
        System.out.println(inner.getSum(3));
        outer.run();
    }
    void run() {
        StaticMemberClass localInner = new StaticMemberClass();
        System.out.println(localInner.getSum(5));
    }
}
```

3. Local Inner Class:

- A local inner class is defined within a method, and the usual scope rules apply to it. It is only accessible within that method, therefore access restrictions (public, protected, package) do not apply.
- However, because objects (and their methods) created from this class may persist after the method returns, a local inner class may not refer to parameters or non-final local variables of the method.

Program 2.4: Program to demonstrate local inner class.

```
public class OuterClass
{
    int outerVariable = 10000;
    static int staticOuterVariable = 2000;
    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        System.out.println(outer.run());
    }
    Object run() {
        int localVariable = 666;
        final int finalLocalVariable = 300;
        class LocalClass {
            int innerVariable = 40;
            int getSum(int parameter) {
                // Cannot access localVariable here
                return outerVariable + staticOuterVariable +
                    finalLocalVariable + innerVariable + parameter;
            }
        }
        LocalClass local = new LocalClass();
        System.out.println(local.getSum(5));
        return local;
    }
}
```

4. Anonymous Inner Class:

- An anonymous inner class is one that is not assigned a name.
- An anonymous inner class is one that is declared and used to create one object (typically as a parameter to a method), all within a single statement.
- An anonymous inner class may extend a class:

```
new SuperClass(parameters){ class body }
```
- Here, SuperClass is not the name of the class being defined, but rather the name of the class being extended. The parameters are the parameters to the constructor for that superclass.

- An anonymous inner class may implement an interface:
`new Interface(){ class body }`
- Because anonymous inner classes are almost always used as event listeners, the example below uses an anonymous inner class as a button listener.

Program 2.5: Program to demonstrate anonymous inner class.

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

public class OuterClass extends JFrame
{
    public static void main(String[] args)
    {
        OuterClass outer = new OuterClass();
        JButton button = new JButton("Don't click me!");
        button.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent event)
            {
                System.out.println("Ouch!");
            }
        });
        outer.add(button);
        outer.pack();
        outer.setVisible(true);
    }
}
```

-
- For understanding the benefit provided by inner classes, consider the applet shown in the following listing. It does not use an inner class. Its goal is to display the string "Mouse Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.
 - There are two top-level classes in this program. `MousePressedDemo` extends `Applet`, and `MyMouseAdapter` extends `MouseAdapter`.
 - The `init()` method of `MousePressedDemo` instantiates `MyMouseAdapter` and provides this object as an argument to the `addMouseListener()` method.

- Notice that a reference to the applet is supplied as an argument to the `MyMouseAdapter` constructor. This reference is stored in an instance variable for later use by the `mousePressed()` method.
- When the mouse is pressed, it invokes the `showStatus()` method of the applet through the stored applet reference. In other words, `showStatus()` is invoked relative to the applet reference stored by `MyMouseAdapter`.

```
// This applet does NOT use an inner class
import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/
public class MousePressedDemo extends Applet
{
    public void init()
    {
        addMouseListener(new MyMouseAdapter(this));
    }
}
class MyMouseAdapter extends MouseAdapter
{
    MousePressedDemo mousePressedDemo;
    public MyMouseAdapter(MousePressedDemo mousePressedDemo)
    {
        this.mousePressedDemo = mousePressedDemo;
    }
    public void mousePressed(MouseEvent me)
    {
        mousePressedDemo.showStatus("Mouse Pressed.");
    }
}
```

- The following listing shows how the preceding program can be improved by using an inner class. Here, `InnerClassDemo` is a top-level class that extends `Applet`. `MyMouseAdapter` is an inner class that extends `MouseAdapter`. Because `MyMouseAdapter` is defined within the scope of `InnerClassDemo`, it has access to all of the variables and methods within the scope of that class.

- Therefore, the `mousePressed()` method can call the `showStatus()` method directly. It no longer needs to do this via a stored reference to the applet. Thus, it is no longer necessary to pass `MyMouseAdapter()` a reference to the invoking object.

```
// Inner class demo.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/
public class InnerClassDemo extends Applet
{
public void init()
{
addMouseListener(new MyMouseAdapter());
}
class MyMouseAdapter extends MouseAdapter
{
public void mousePressed(MouseEvent me)
{
showStatus("Mouse Pressed");
}
}
}
```

2.5 EVENT LISTENER INTERFACES

- The delegation event model has two parts i.e. sources (an object on which event occurs.) and listeners (responsible for generating response to an event).
- Listener is also known as event handler.
- The Event listener represent the interfaces responsible to handle events.
- Listeners are created by implementing one or more of the interfaces defined by the `java.awt.event` package.
- Following is the declaration for `java.util.EventListener` interface:

```
public interface EventListener
```

- When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its argument.

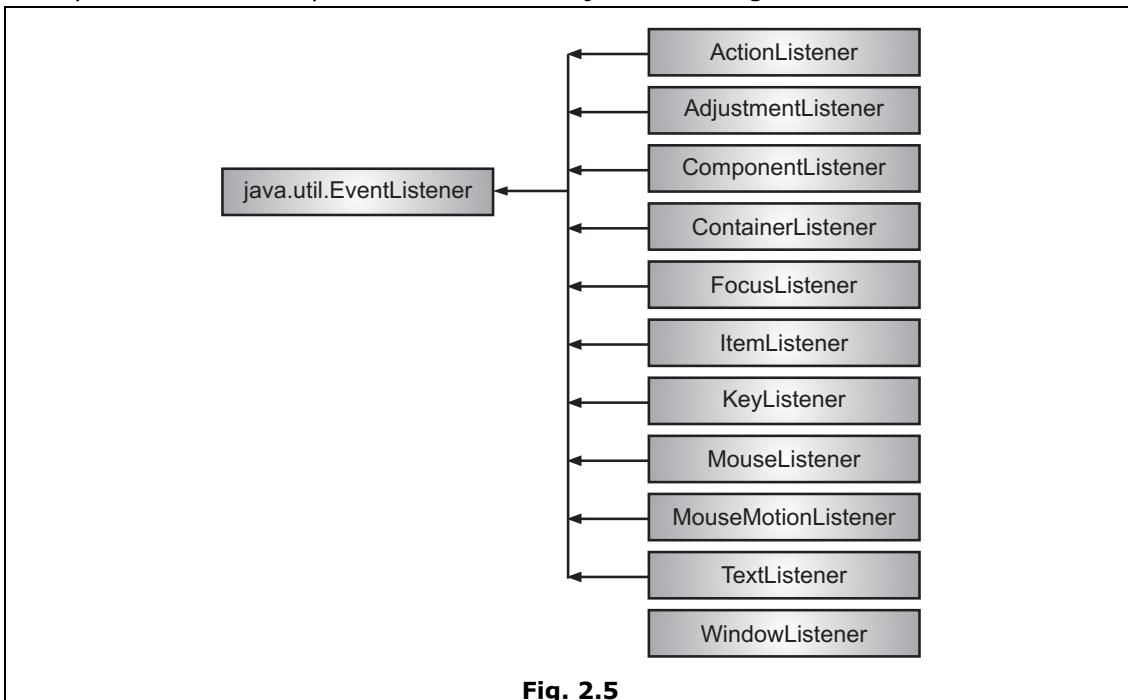


Fig. 2.5

2.5.1 ActionListener Interface

- This interface is used for receiving the action events.
- This interface defines the `actionPerformed()` method that is invoked when an action event occurs.
- Its general form/syntax is shown below:

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

2.5.2 ComponentListener Interface

- The class which processes the `ComponentEvent` should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the `addComponentListener()` method. Component event are raised for information only.
- **Methods:**
 1. `void componentHidden(ComponentEvent e)`: Invoked when the component has been made invisible.
 2. `void componentMoved(ComponentEvent e)`: Invoked when the component's position changes.

3. `void componentResized(ComponentEvent e):` Invoked when the component's size changes.
4. `void componentShown(ComponentEvent e):` Invoked when the component has been made visible.

2.5.3 ContainerListener Interface

- The interface `ContainerListener` is used for receiving container events.
- The class that process container events needs to implements this interface.
- **Methods:**
 1. `void componentAdded (ContainerEvent e):` Invoked when a component has been added to the container.
 2. `void componentRemoved(ContainerEvent e):` Invoked when a component has been removed from the container.

2.5.4 FocusListener Interface

- This interface is used for receiving the focus events.
- This interface defines two methods. Their general forms are shown here:
 1. `void focusGained(FocusEvent fe):` This method is called, when a component gain keyboard focus.
 2. `void focusLost(FocusEvent fe):` This method is called when a component loses keyboard focus.

AdjustmentListener Interface:

- This interface is used for receiving the adjusmtent events.
- This interface defines the `adjustmentValueChanged()` method that is invoked when an adjustment event occurs.
- Its general form/syntax is shown below:

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

2.5.5 ItemListener Interface

- This interface is used for receiving the item events.
- This interface defines the `itemStateChanged()` method that is invoked when the state of an item changes.
- Its general form/syntax is shown below:

```
void itemStateChanged(ItemEvent ie)
```

2.5.6 KeyListener Interface

- The class which processes the `KeyEvent` should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the `addKeyListener()` method.

- **Methods:**

1. `void keyPressed(KeyEvent e)`: Invoked when a key has been pressed.
2. `void keyReleased(KeyEvent e)`: Invoked when a key has been released.
3. `void keyTyped(KeyEvent e)`: Invoked when a key has been typed.

2.5.7 MouseListener Interface

- This interface is used for receiving the key events.
- 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 below:

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

2.5.8 MouseMotionListener Interface

- This interface is used for receiving the mouse motion events.
- This interface defines two methods.
 1. The `mouseDragged()` method is called multiple times as the mouse is dragged.
 2. The `mouseMoved()` method is called multiple times as the mouse is moved.
- Their general forms are shown here:

```
void mouseDragged(MouseEvent me)
And
void mouseMoved(MouseEvent me)
```

2.5.9 TextListener Interface

- This interface is used for receiving the text events.
- This interface defines only one method `textChanged()` method. The general form is shown here:

```
void textChanged(TextEvent te)
```
- This method is called when a change occurs in a text area or text field.

2.5.10 WindowListener Interface

- This interface is used for receiving the window events.
- This interface defines seven methods, these method perform operation on the window. The general forms of these methods are:
 1. `void windowActivated(WindowEvent we)`: This method is Invoked when a window is activated.
 2. `void windowClosed(WindowEvent we)`: This method is called when a window is closed.
 3. `void windowClosing(WindowEvent we)`: This method is called when a window is being closed.
 4. `void windowDeactivated(WindowEvent we)`: This method is Invoked when a window is deactivated.
 5. `void windowDeiconified(WindowEvent we)`: This method is called when a window is deiconified.
 6. `void windowIconified(WindowEvent we)`: This method is called when a window is iconified.
 7. `void windowOpened(WindowEvent we)`: This method is called when a window is opened.

2.5.11 WindowFocusListener Interface

- Focus events are fired whenever a component gains or losses the keyboard focus.
- The listener interface for receiving WindowEvents. When the Window's status changes by virtue of it being opened, closed, activated, deactivated, iconified or deiconified, or by focus being transferred into or out of the Window, the relevant method in the listener object is invoked and the WindowEvent is passed to it.

Methods:

1. `void windowGainedFocus(WindowEvent e)`: Invoked when the Window is set to the focused Window, which means that the Window, or one of its subcomponents, will received keyboard events.
2. `void windowLostFocus(WindowEvent e)`: Invoked when the Window is no longer the focused Window, which means that keyboard events will no longer be delivered to the Window or any of its subcomponents.

Programs:

Program 1: Handling Mouse Events. In order to handle mouse events, we must implement the `MouseListener` and the `MouseMotionListener` interfaces.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener
{
String msg = "";
int mouseX = 0, mouseY = 0; // coordinates of mouse
public void init()
{
addMouseListener(this);
addMouseMotionListener(this);
}
public void mouseClicked(MouseEvent me)
{
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse clicked.";
repaint();
}
// Handle mouse entered.
public void mouseEntered(MouseEvent me)
{
// save coordinates
mouseX = 0;
mouseY = 10;
msg = "Mouse entered.";
repaint();
}
// Handle mouse exited.
```

```
public void mouseExited(MouseEvent me)
{
    // save coordinates
    mouseX = 0;
    mouseY = 10;
    msg = "Mouse exited.";
    repaint();
}
// Handle button pressed.
public void mousePressed(MouseEvent me)
{
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Down";
    repaint();
}
// Handle button released.
public void mouseReleased(MouseEvent me)
{
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "Up";
    repaint();
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
    // save coordinates
    mouseX = me.getX();
    mouseY = me.getY();
    msg = "*";
    showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
    repaint();
}
// Handle mouse moved.
```

```
public void mouseMoved(MouseEvent me)
{
    // show status
    showStatus("Moving mouse at " + me.getX() + ", " +
me.getY());
}
// Display msg in applet window at current X,Y location.
public void paint(Graphics g)
{
    g.drawString(msg, mouseX, mouseY);
}
}
/*<applet code="MouseEvents" width=300 height=100>
</applet>*/
```

- Here, the `MouseEvents` class extends `Applet` and implements both the `MouseListener` and `MouseMotionListener` interfaces. These two interfaces contain methods that receive and process the various types of mouse events.
- Notice that the applet is both the source and the listener for these events. This works because `Component`, which supplies the `addMouseListener()` and `addMouseMotionListener()` methods, is a superclass of `Applet`. Being both the source and the listener for events is a common situation for applets. Inside `init()`, the applet registers itself as a listener for mouse events. This is done by using `addMouseListener()` and `addMouseMotionListener()`, which as mentioned, are members of `Component`. They are shown here:

```
void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)
```

- Here, `ml` is a reference to the object receiving mouse events, and `mml` is a reference to the object receiving mouse motion events. In this program, the same object is used for both. The applet then implements all of the methods defined by the `MouseListener` and `MouseMotionListener` interfaces. These are the event handlers for the various mouse events. Each method handles its event and then returns.

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

```
public class AdapterDemo extends Applet
{
public void init()
{
addMouseListener(new MyMouseAdapter(this));
addMouseMotionListener(new MyMouseMotionAdapter(this));
}
}

class MyMouseAdapter extends MouseAdapter
{
AdapterDemo adapterDemo;
public MyMouseAdapter(AdapterDemo adapterDemo)
{
this.adapterDemo = adapterDemo;
}
// Handle mouse clicked.
public void mouseClicked(MouseEvent me)
{
adapterDemo.showStatus("Mouse clicked");
}
}

class MyMouseMotionAdapter extends MouseMotionAdapter
{
AdapterDemo adapterDemo;
public MyMouseMotionAdapter(AdapterDemo adapterDemo)
{
this.adapterDemo = adapterDemo;
}
// Handle mouse dragged.
public void mouseDragged(MouseEvent me)
{
adapterDemo.showStatus("Mouse dragged");
}
}

/*<applet code="AdapterDemo" width=300 height=100>
</applet>*/
```

- As we can see by looking at the program, not having to implement all of the methods defined by the `MouseMotionListener` and `MouseListener` interfaces saves our considerable amount of effort and prevents our code from becoming cluttered with empty methods.

Program 2: Handling Buttons Events.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="ButtonDemo" width=250 height=150>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener
{
    String msg = "";
    Button yes, no, maybe;
    public void init()
    {
        yes = new Button("Yes");
        no = new Button("No");
        maybe = new Button("Undecided");
        add(yes);
        add(no);
        add(maybe);
        yes.addActionListener(this);
        no.addActionListener(this);
        maybe.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        String str = ae.getActionCommand();
        if(str.equals("Yes"))
        {
            msg = "You pressed Yes.";
        }
    }
}
```

```
else if(str.equals("No"))
{
msg = "You pressed No.";
}
else
{
msg = "You pressed Undecided.";
}
repaint();
}
public void paint(Graphics g)
{
g.drawString(msg, 6, 100);
}
}
```

Program 3: Handling Checkboxes Events.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CheckboxDemo extends Applet implements ItemListener
{
String msg = "";
Checkbox Win98, winNT, solaris, mac;
public void init()
{
Win98 = new Checkbox("Windows 98/XP", null, true);
winNT = new Checkbox("Windows NT/2000");
solaris = new Checkbox("Solaris");
mac = new Checkbox("MacOS");
add(Win98);
add(winNT);
add(solaris);
add(mac);
Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
}
```

```
public void itemStateChanged(ItemEvent ie)
{
    repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g)
{
    msg = "Current state: ";
    g.drawString(msg, 6, 80);
    msg = " Windows 98/XP: " + Win98.getState();
    g.drawString(msg, 6, 100);
    msg = " Windows NT/2000: " + winNT.getState();
    g.drawString(msg, 6, 120);
    msg = " Solaris: " + solaris.getState();
    g.drawString(msg, 6, 140);
    msg = " MacOS: " + mac.getState();
    g.drawString(msg, 6, 160);
}
}
```

Program 4: Handling Radio Buttons Events.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CBGroup extends Applet implements ItemListener
{
    String msg = "";
    Checkbox Win98, winNT, solaris, mac;
    CheckboxGroup cbg;
    public void init()
    {
        cbg = new CheckboxGroup();
        Win98 = new Checkbox("Windows 98/XP", cbg, true);
        winNT = new Checkbox("Windows NT/2000", cbg, false);
        solaris = new Checkbox("Solaris", cbg, false);
    }
}
```

```
mac = new Checkbox("MacOS", cbg, false);
add(Win98);
add(winNT);
add(solaris);
add(mac);
Win98.addItemListener(this);
winNT.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g)
{
msg = "Current selection: ";
msg += cbg.getSelectedCheckbox().getLabel();
g.drawString(msg, 6, 100);
}
}
/*<applet code="CBGroup" width=250 height=200>
</applet>*/
```

Program 5: Handling Choice Controls Events.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ChoiceDemo extends Applet implements ItemListener
{
Choice os, browser;
String msg = "";
public void init()
{
os = new Choice();
browser = new Choice();
```

```
// add items to os list
os.add("Windows 98/XP");
os.add("Windows NT/2000");
os.add("Solaris");
os.add("MacOS");
// add items to browser list
browser.add("Netscape 3.x");
browser.add("Netscape 4.x");
browser.add("Netscape 5.x");
browser.add("Netscape 6.x");
browser.add("Internet Explorer 4.0");
browser.add("Internet Explorer 5.0");
browser.add("Internet Explorer 6.0");
browser.add("Lynx 2.4");
browser.select("Netscape 4.x");
// add choice lists to window
add(os);
add(browser);
// register to receive item events
os.addItemListener(this);
browser.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
repaint();
}
// Display current selections.
public void paint(Graphics g)
{
msg = "Current OS: ";
msg += os.getSelectedItem();
g.drawString(msg, 6, 120);
msg = "Current Browser: ";
msg += browser.getSelectedItem();
g.drawString(msg, 6, 140);
}
}
/*<applet code="ChoiceDemo" width=300 height=180>
</applet>*/
```

Program 6: Handling Lists Events.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ListDemo extends Applet implements ActionListener
{
List os, browser;
String msg = "";
public void init()
{
os = new List(4, true);
browser = new List(4, false);
// add items to os list
os.add("Windows 98/XP");
os.add("Windows NT/2000");
os.add("Solaris");
os.add("MacOS");
// add items to browser list
browser.add("Netscape 3.x");
browser.add("Netscape 4.x");
browser.add("Netscape 5.x");
browser.add("Netscape 6.x");
browser.add("Internet Explorer 4.0");
browser.add("Internet Explorer 5.0");
browser.add("Internet Explorer 6.0");
browser.add("Lynx 2.4");
browser.select(1);
// add lists to window
add(os);
add(browser);
// register to receive action events
os.addActionListener(this);
browser.addActionListener(this);
}
```

```
public void actionPerformed(ActionEvent ae)
{
    repaint();
}
// Display current selections.
public void paint(Graphics g)
{
    int idx[];
    msg = "Current OS: ";
    idx = os.getSelectedIndexes();
    for(int i=0; i<idx.length; i++)
        msg += os.getItem(idx[i]) + " ";
    g.drawString(msg, 6, 120);
    msg = "Current Browser: ";
    msg += browser.getSelectedItem();
    g.drawString(msg, 6, 140);
}
}
/*<applet code="ListDemo" width=300 height=180>
</applet>*/
```

Program 6: Handling Scrollbars Events.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class SBDemo extends Applet
implements AdjustmentListener, MouseMotionListener
{
    String msg = "";
    Scrollbar vertSB, horzSB;
    public void init()
    {
        int width = Integer.parseInt(getParameter("width"));
        int height = Integer.parseInt(getParameter("height"));
        vertSB = new Scrollbar(Scrollbar.VERTICAL,
        0, 1, 0, height);
        horzSB = new Scrollbar(Scrollbar.HORIZONTAL,
        0, 1, 0, width);
        add(vertSB);
        add(horzSB);
    }
}
```

```
// register to receive adjustment events
vertSB.addAdjustmentListener(this);
horzSB.addAdjustmentListener(this);
addMouseMotionListener(this);
}
public void adjustmentValueChanged(AdjustmentEvent ae)
{
    repaint();
}
// Update scroll bars to reflect mouse dragging.
public void mouseDragged(MouseEvent me)
{
    int x = me.getX();
    int y = me.getY();
    vertSB.setValue(y);
    horzSB.setValue(x);
    repaint();
}
// Necessary for MouseMotionListener
public void mouseMoved(MouseEvent me)
{
}
// Display current value of scroll bars.
public void paint(Graphics g)
{
    msg = "Vertical: " + vertSB.getValue();
    msg += ", Horizontal: " + horzSB.getValue();
    g.drawString(msg, 6, 160);
    // show current mouse drag position
    g.drawString("*", horzSB.getValue(),
    vertSB.getValue());
}
}
/*<applet code="SBDemo" width=300 height=200>
</applet>*/
```

Program 7: Handling Menus Events.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
class MenuFrame extends Frame {
String msg = "";
CheckboxMenuItem debug, test;
MenuFrame(String title) {
super(title);
// create menu bar and add it to frame
MenuBar mbar = new MenuBar();
setMenuBar(mbar);
// create the menu items
Menu file = new Menu("File");
MenuItem item1, item2, item3, item4, item5;
file.add(item1 = new MenuItem("New..."));
file.add(item2 = new MenuItem("Open..."));
file.add(item3 = new MenuItem("Close"));
file.add(item4 = new MenuItem("-"));
file.add(item5 = new MenuItem("Quit..."));
mbar.add(file);
Menu edit = new Menu("Edit");
MenuItem item6, item7, item8, item9;
edit.add(item6 = new MenuItem("Cut"));
edit.add(item7 = new MenuItem("Copy"));
edit.add(item8 = new MenuItem("Paste"));
edit.add(item9 = new MenuItem("-"));
Menu sub = new Menu("Special");
MenuItem item10, item11, item12;
sub.add(item10 = new MenuItem("First"));
sub.add(item11 = new MenuItem("Second"));
sub.add(item12 = new MenuItem("Third"));
edit.add(sub);
```

```
// these are checkable menu items
debug = new CheckboxMenuItem("Debug");
edit.add(debug);
test = new CheckboxMenuItem("Testing");
edit.add(test);
mbar.add(edit);
// create an object to handle action and item events
MyMenuHandler handler = new MyMenuHandler(this);
// register it to receive those events
item1.addActionListener(handler);
item2.addActionListener(handler);
item3.addActionListener(handler);
item4.addActionListener(handler);
item5.addActionListener(handler);
item6.addActionListener(handler);
item7.addActionListener(handler);
item8.addActionListener(handler);
item9.addActionListener(handler);
item10.addActionListener(handler);
item11.addActionListener(handler);
item12.addActionListener(handler);
debug.addItemListener(handler);
test.addItemListener(handler);
// create an object to handle window events
MyWindowAdapter adapter = new MyWindowAdapter(this);
// register it to receive those events
addWindowListener(adapter);
}
public void paint(Graphics g) {
g.drawString(msg, 10, 200);
if(debug.getState())
g.drawString("Debug is on.", 10, 220);
else
g.drawString("Debug is off.", 10, 220);
if(test.getState())
g.drawString("Testing is on.", 10, 240);
else
g.drawString("Testing is off.", 10, 240);
}
}
```

```
class MyWindowAdapter extends WindowAdapter {
    MenuFrame menuFrame;
    public MyWindowAdapter(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    public void windowClosing(WindowEvent we) {
        menuFrame.setVisible(false);
    }
}

class MyMenuHandler implements ActionListener, ItemListener {
    MenuFrame menuFrame;
    public MyMenuHandler(MenuFrame menuFrame) {
        this.menuFrame = menuFrame;
    }
    // Handle action events
    public void actionPerformed(ActionEvent ae) {
        String msg = "You selected ";
        String arg = (String)ae.getActionCommand();
        if(arg.equals("New..."))
            msg += "New.";
        else if(arg.equals("Open..."))
            msg += "Open.";
        else if(arg.equals("Close"))
            msg += "Close.";
        else if(arg.equals("Quit..."))
            msg += "Quit.";
        else if(arg.equals("Edit"))
            msg += "Edit.";
        else if(arg.equals("Cut"))
            msg += "Cut.";
        else if(arg.equals("Copy"))
            msg += "Copy.";
        else if(arg.equals("Paste"))
            msg += "Paste.";
```

```
else if(arg.equals("First"))
msg += "First.";
else if(arg.equals("Second"))
msg += "Second.";
else if(arg.equals("Third"))
msg += "Third.";
else if(arg.equals("Debug"))
msg += "Debug.";
else if(arg.equals("Testing"))
msg += "Testing.";
menuFrame.msg = msg;
menuFrame.repaint();
}
// Handle item events
public void itemStateChanged(ItemEvent ie) {
menuFrame.repaint();
}
}
// Create frame window.
public class MenuDemol extends Applet {
Frame f;
public void init() {
f = new MenuFrame("Menu Demo");
int width = Integer.parseInt(getParameter("width"));
int height = Integer.parseInt(getParameter("height"));
setSize(new Dimension(width, height));
f.setSize(width, height);
f.setVisible(true);
}
public void start() {
f.setVisible(true);
}
public void stop() {
f.setVisible(false);
}
}
/*<applet code="MenuDemol" width=250 height=250>
</applet>*/
```

Handling Events by Extending AWT Components:

- Java also allows us to handle events by sub classing AWT components. Doing so allows us to handle events in much the same way as they were handled under the original 1.0 version of Java.
- Of course, this technique is discouraged, because it has the same disadvantages of the Java 1.0 event model, the main one being inefficiency.
- In order to extend an AWT component, we must call the `enableEvents()` method of `Component`. Its general form is shown below:

```
protected final void enableEvents(long eventMask)
```

- The `eventMask` argument is a bit mask that defines the events to be delivered to this component. The `AWTEvent` class defines `int` constants for making this mask. Several are shown here:

```
ACTION_EVENT_MASK KEY_EVENT_MASK  
ADJUSTMENT_EVENT_MASK MOUSE_EVENT_MASK  
COMPONENT_EVENT_MASK MOUSE_MOTION_EVENT_MASK  
CONTAINER_EVENT_MASK MOUSE_WHEEL_EVENT_MASK  
FOCUS_EVENT_MASK TEXT_EVENT_MASK
```

- We must also override the appropriate method from one of our superclasses in order to process the event. Methods listed below most commonly used and the classes that provide them.

```
Button processActionEvent( )  
Checkbox processItemEvent( )  
CheckboxMenuItem processItemEvent( )  
Choice processItemEvent( )  
Component processComponentEvent( ), processFocusEvent( ),  
processKeyEvent( ), processMouseEvent( ),  
processMouseMotionEvent( ),  
processMouseWheelEvent ( )  
List processActionEvent( ), processItemEvent( )  
MenuItem processActionEvent( )  
Scrollbar processAdjustmentEvent( )  
TextComponent processTextEvent( )
```

Extending Button:

- The following program creates an applet that displays a button labeled "Test Button". When the button is pressed, the string "action event:" is displayed on the status line of the applet viewer or browser, followed by a count of the number of button presses. The program has one top-level class named `ButtonDemo2` that extends `Applet`.

- A static integer variable named `i` is defined and initialized to zero. It records the number of button pushes. The `init()` method instantiates `MyButton` and adds it to the applet. `MyButton` is an inner class that extends `Button`. Its constructor uses `super` to pass the label of the button to the superclass constructor. It calls `enableEvents()` so that action events may be received by this object. When an action event is generated, `processActionEvent()` is called.
- That method displays a string on the status line and calls `processActionEvent()` for the superclass. Because `MyButton` is an inner class, it has direct access to the `showStatus()` method of `ButtonDemo2`.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ButtonDemo2 extends Applet
{
    MyButton myButton;
    static int i = 0;
    public void init()
    {
        myButton = new MyButton("Test Button");
        add(myButton);
    }
    class MyButton extends Button
    {
        public MyButton(String label)
        {
            super(label);
            enableEvents(AWTEvent.ACTION_EVENT_MASK);
        }
        protected void processActionEvent(ActionEvent ae)
        {
            showStatus("action event: " + i++);
            super.processActionEvent(ae);
        }
    }
}
```

Extending Checkbox:

- The following program creates an applet that displays three check boxes labeled "Item 1", "Item 2", and "Item 3". When a check box is selected or deselected, a string containing the name and state of that check box is displayed on the status line of the applet viewer or browser.
- The program has one top-level class named CheckboxDemo2 that extends Applet. Its init() method creates three instances of MyCheckbox and adds these to the applet. MyCheckbox is an inner class that extends Checkbox.
- Its constructor uses super to pass the label of the check box to the superclass constructor. It calls enableEvents() so that item events may be received by this object. When an item event is generated, processItemEvent() is called.
- That method displays a string on the status line and calls processItemEvent() for the superclass.

```
/*<applet code=CheckboxDemo2 width=300 height=100>
</applet>*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class CheckboxDemo2 extends Applet
{
    MyCheckbox myCheckbox1, myCheckbox2, myCheckbox3;
    public void init()
    {
        myCheckbox1 = new MyCheckbox("Item 1");
        add(myCheckbox1);
        myCheckbox2 = new MyCheckbox("Item 2");
        add(myCheckbox2);
        myCheckbox3 = new MyCheckbox("Item 3");
        add(myCheckbox3);
    }
    class MyCheckbox extends Checkbox
    {
        public MyCheckbox(String label)
        {
            super(label);
            enableEvents(AWTEvent.ITEM_EVENT_MASK);
        }
    }
}
```

```
protected void processItemEvent(ItemEvent ie)
{
    showStatus("Checkbox name/state: " + getLabel() +
        "/" + getState());
    super.processItemEvent(ie);
}
}
```

Extending List:

- The following program modifies the preceding example so that it uses a list instead of a choice menu. There is one top-level class named ListDemo2 that extends Applet. Its init() method creates a list element and adds it to the applet. MyList is an inner class that extends List.
- It calls enableEvents() so that both action and item events may be received by this object. When an entry is selected or deselected, processItemEvent() is called. When an entry is double clicked, processActionEvent() is also called. Both methods display a string and then hand control to the superclass.

```
/*<applet code=ListDemo2 width=300 height=100>
</applet>*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class ListDemo2 extends Applet
{
    MyList list;
    public void init()
    {
        list = new MyList();
        list.add("Red");
        list.add("Green");
        list.add("Blue");
        add(list);
    }
}
```

```
class MyList extends List
{
public MyList()
{
enableEvents(AWTEvent.ITEM_EVENT_MASK |
AWTEvent.ACTION_EVENT_MASK);
}
protected void processActionEvent(ActionEvent ae)
{
showStatus("Action event: " +
ae.getActionCommand());
super.processActionEvent(ae);
}
protected void processItemEvent(ItemEvent ie)
{
showStatus("Item event: " + getSelectedItem());
super.processItemEvent(ie);
}
}
}
```

Important Points

- Event class provide a consistent, easy-to-use means of encapsulating events.
- An ActionEvent is generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
- An AdjustmentEvent is generated by a scroll bar.
- A ComponentEvent is generated when the size, position or visibility of a component is changed.
- A ContainerEvent is generated when a component is added to or removed from a container.
- A FocusEvent is generated when a component gains or loses input focus.
- An ItemEvent is generated when a check box or a list item is clicked or when a checkable menu item is selected or deselected.
- A KeyEvent is generated when keyboard input occurs.
- An event is an object that describes a state change in a source.

- A source is an object (For example, button, textfield) 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.
- Java provides a special classes, called an adapter class, that can simplify the creation of event handlers in certain situations.
- GUIs generate events when we interact with GUI. For example, clicking a button, moving the mouse, closing Window etc. Both AWT and swing components generate events.
- Change in the state of an object is known as event.
- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- In Java both AWT and Swing components use Event Delegation Model. In this model, processing of an event is delegated to a particular object (handlers) in the program.
- In the delegation model, an event is an object that describes a state change in a source.
- Event sources are components, subclasses of java.awt.Component, capable to generate events.
- The events generated by the GUI components are handled by a special group of interfaces known as "listeners".

Practice Questions

1. What is event? How to handle an event.
2. Explain what is the meaning of event sources.
3. List out the components, who supports ActionEvent ActionListener events ?
4. What are different events? Explain any one in detail.
5. Explain event delegation model with diagram.
6. What are the event classes in Java?
7. Explain the following event listener:
 - (i) ActionListener
 - (ii) FocusListener
8. What is adopter classes? Explain in detail.
9. Describe inner classe in detail.
10. Explain event listener in detail?

