Unit 5
Applets and Event Handling

Topics:
Applets: Applet class, Applet structure, An example of Applet, Applet life Cycle, Event Delagation Model, Java.awt.event description, Sources of Events, Event Listeners, Adapter class, inner class

Part-1: Applet Programming

Introduction to Applet

Applets are small Java programs that are used in Internet computing. The Applets can be easily transported over the internet from one computer to another computer and can be run using "**appletviewer**" or java enabled "**Web Browser**". An Applet like any application program can do many things for us. It can perform arithmetic operations, display graphics, animations, text, accept the user input and play interactive games.

Applets are created in the following situations:

1. When we need something to be included dynamically in the web page.
2. When we require some flash output
3. When we want to create a program and make it available on the internet.

Types of Applet

There are two types of Applets.

- The first type is created is based on the **Applet** class of java.applet package. These applets use the **Abstract Window Toolkit(AWT)** for designing the graphical user interface.
- The second type of type of the Applets are based on the **Swing** class **JApplet.** The swing Applets use the swing classes to create Graphical User Interface.

The JApplet inherits the properties from the Applet, so all the features of the Applet are available in the JApplet.

Applet Basics

- All applets are subclasses (either directly or indirectly) of **Applet**. Applets are not stand-alone programs. Instead, they run within either a web browser or an applet viewer, which is provided by the JDK.
- Execution of an applet does not begin at **main( )**.
- Output to your applet's window is not performed by **System.out.println( )**. Rather, in non-Swing applets, output is handled with various AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y location
- To use an applet, it is specified in an HTMLfile. One way to do this is by using the APPLET tag. **(HTML stands for Hyper Text Markup Language)**

- The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTMLfile.
- To just test the applet it can be executed using the appletviewer. The applet tag must be included as comment lines in the java source program as follow:
  /*<applet code="applet_name" width=400 height=400 ></applet> */
- To view the applet using the HTML file, it can be included in the HTML file with <applet. Tag as follw:

Filename.HTML

```
<html>
<head><title> The name of the Web Page</title>
</head>
<body>
<applet code="applet_name"   width=400 height=400 ></applet>
</body>
</html>
```

Note: Here, the **<applet>** is the name of the tag, and "code" ,"width" and "height" are called attributes of the Tag, applet_name, 400, 400 are called values respectively.

The Applet class

The Applet class defines several methods that support for execution of the applets, such as starting and stopping. It also provides methods to load and display images. It also provides methods for loading and playing the audio clips. The **Applet** extends the AWT class **Panel**. The **Panel** extends **Container** class, which in turn extends from the **Component.** The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTMLfile.

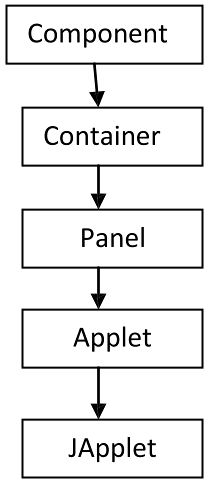


Fig 1: Hierarchy of Applet class

methods of Applet class:

destroy()
Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated.

getAppletContext()
Determines this applet's context, which allows the applet to query and affect the environment in which it runs.

getAppletInfo()
Returns information about this applet.

getAudioClip(URL)
Returns the AudioClip object specified by the URL argument.

getAudioClip(URL, String)
Returns the AudioClip object specified by the URL and name arguments.

getCodeBase()
Gets the base URL.

getDocumentBase()
Gets the document URL.

getImage(URL)
Returns an Image object that can then be painted on the screen.

getImage(URL, String)
Returns an Image object that can then be painted on the screen.

getLocale()
Gets the Locale for the applet, if it has been set.

getParameter(String)
Returns the value of the named parameter in the HTML tag.

getParameterInfo()
Returns information about the parameters than are understood by this applet.

init()
Called by the browser or applet viewer to inform this applet that it has been loaded into the system.

isActive()
Determines if this applet is active.

play(URL)
Plays the audio clip at the specified absolute URL.

play(URL, String)
Plays the audio clip given the URL and a specifier that is relative to it.

resize(Dimension)
Requests that this applet be resized.

resize(int, int)
Requests that this applet be resized.

setStub(AppletStub)
Sets this applet's stub.

showStatus(String)
Requests that the argument string be displayed in the "status window".

start()

Called by the browser or applet viewer to inform this applet that it should start its execution.

stop()

Called by the browser or applet viewer to inform this applet that it should stop its execution.

Applet Architecture

An applet is a window-based program. As such, its architecture is different from the console-based programs. The key concepts are as follow:

- **First, applets are event driven**. An applet waits until an event occurs. The run-time system notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return.
- **Second, the user initiates interaction with an applet**. These interactions are sent to the applet as events to which the applet must respond. For example, when the user clicks the mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated

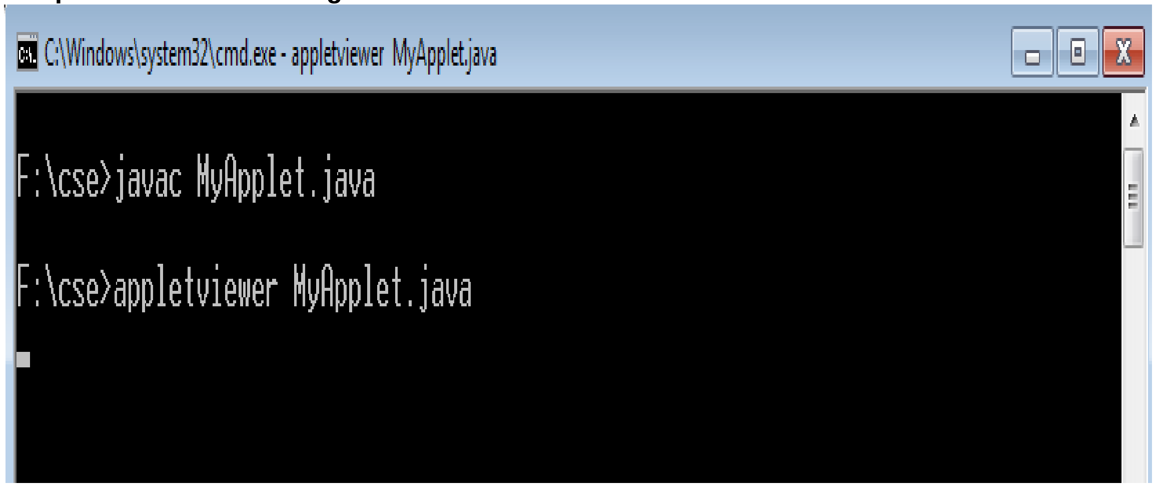
An Applet Skelton –An Example of Applet

Most of the applets override a set of methods of the Applet. Four of these methods, **init( )**, **start( )**, **stop( )**, and **destroy( )**, apply to all applets and are defined by **Applet**. Default implementations for all of these methods are provided. Applets do not need to override those methods they do not use.

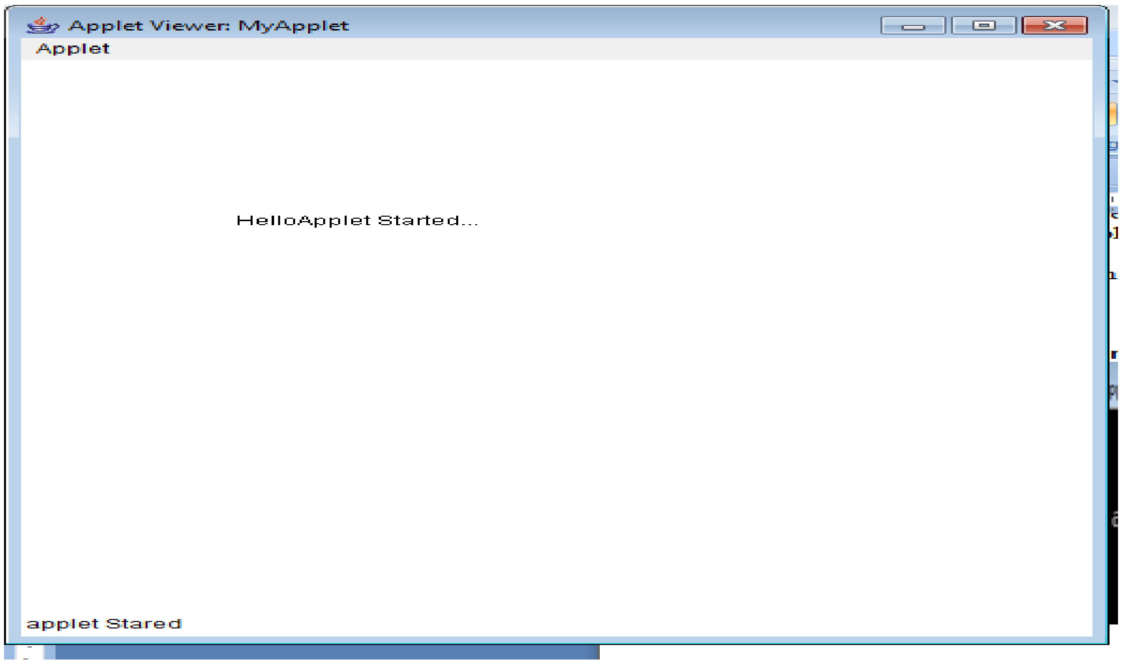
AWT-based applets will also override the **paint( )** method, which is defined by the AWT **Component** class. This method is called when the applet's output must be redisplayed. These five methods can be assembled into the skeleton shown here:

```
// An Applet skeleton.
import java.awt.*;
import java.applet.*;
/*
<applet code="AppletSkel" width=300 height=100>
</applet>
*/
public class AppletSkel extends Applet
{
// Called first.
public void init()
{
        // initialization
}
/* Called second, after init(). Also called whenever
the applet is restarted. */
public void start()
{
        // start or resume execution
}
// Called when the applet is stopped.
public void stop()
{
        // suspends execution
}
/* Called when applet is terminated. This is the last
method executed. */
public void destroy()
 {
        // perform shutdown activities
}
// Called when an applet's window must be restored.
public void paint(Graphics g)
{
        // redisplay contents of window
}
}
```

The procedure for Running



```
F:\cse>javac MyApplet.java

F:\cse>appletviewer MyApplet.java
```



HelloApplet Started...

applet Stared

Life Cycle of an Applet

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the following methods are called, in this sequence:
1. **init( )**
2. **start( )**
3. **paint( )**
When an applet is terminated, the following sequence of method calls takes place:
1. **stop( )**
2. **destroy( )**
Let’s look more closely at these methods.

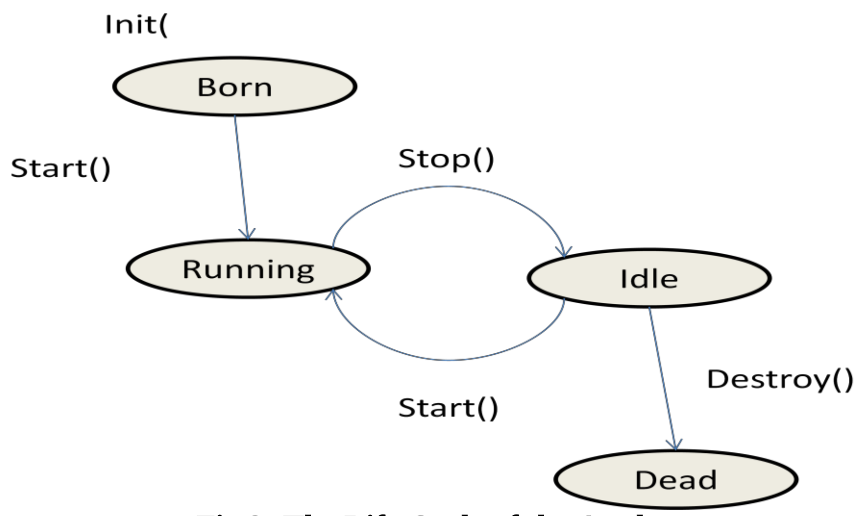


Fig 2: The Life Cycle of the Applet

init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

start( )

The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped. Whereas **init( )** is called once—the first time an applet is loaded—**start( )** is called each time an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at **start( )**.

paint( )

The **paint( )** method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. **paint( )** is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, **paint( )** is called. The **paint( )** method has one parameter of type **Graphics**. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

The **stop( )** method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When **stop( )** is called, the applet is probably running. You should use **stop( )** to suspend threads that don't need to run when the applet is not visible. You can restart them when **start( )** is called if the user returns to the page.

destroy( )

The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The **stop( )** method is always called before **destroy( )**.

Requesting the repaint() method

One of the important architectural constraints that have been imposed on an applet is that it must quickly return control to the AWT run-time system. It cannot create a loop inside paint( ). This would prevent control from passing back to the AWT. Whenever your applet needs to update the information displayed in its window, it simply calls ***repaint( )***. The ***repaint( )*** method is defined by the AWT that causes AWT run-time system to execute a call to your applet's ***update()*** method, which in turn calls ***paint()***. The AWT will then execute a call to paint( ) that will display the stored information. The repaint( ) method has four forms. The simplest version of repaint( ) is:

1. **void repaint ( )**

This causes the entire window to be repainted. Other versions that will cause repaint are:

2. **void repaint(int left, int top, int width, int height)**

If your system is slow or busy, update( ) might not be called immediately. If multiple calls have been made to AWT within a short period of time, then update( ) is not called very frequently. This can be a problem in many situations in which a consistent update time is necessary. One solution to this problem is to use the following forms of **repaint( ):**

3. **void** repaint (long **maxDelay)**
4. **void** repaint (long **maxDelay, int x, int y, int width, int height)**

   Where "maxDelay" is the number milliseconds should be elapsed before updat() method is called.

Using the Status Window

In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call **showStatus( )** with the string that you want displayed. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors. The status window also makes an excellent debugging aid, because it gives you an easy way to output information about your applet.

Example program

StatusWindow.java

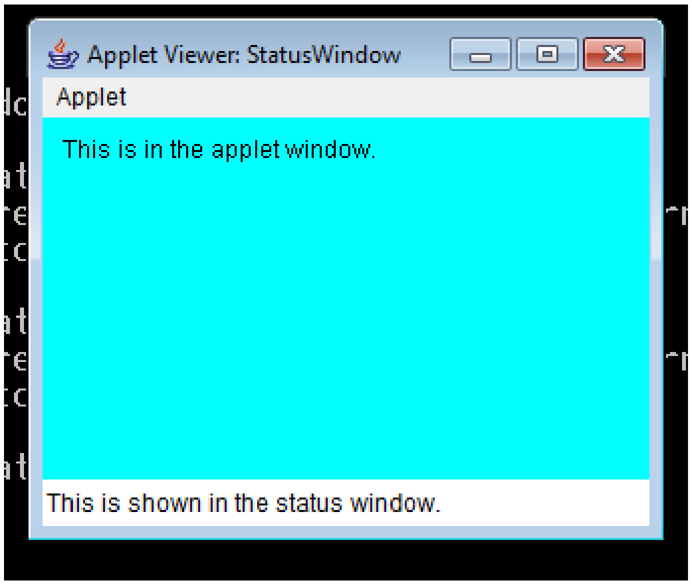
```
// Using the Status Window.
import java.awt.*;
import java.applet.*;
/*
```

```
<applet code="StatusWindow" width=300 height=50>
</applet>
*/
public class StatusWindow extends Applet
{
        public void init()
        {
                setBackground(Color.cyan);
        }
// Display msg in applet window.

        public void paint(Graphics g)
        {
                g.drawString("This is in the applet window.", 10, 20);
                showStatus("This is shown in the status window.");
        }
}
```

Running applet:

1. javac StatusWindow.java
2. appletviewer StatusWindow.java

Output:



```
This is in the applet window.
```

```
This is shown in the status window.
```

Passing parameters to Applet

We can supply user defined parameters to an applet using the **<param>** Tag of HTML. Each <param> Tag has the attributes such as "name" , "value" to which actual values are assigned. Using this tag we can change the text to be displayed through applet. We write the <param> Tag as follow:

<param name="name1" value="Hello Applet" > </param>

Passing parameters to an Applet is something similar to passing parameters to main() method using command line arguments. To set up and handle parameters, we need to do two things:

1. Include appropriate <param> Tag in the HTML file
2. Provide the code in the applet to take these parameters.

Example Program:

ParaPassing.java

```
import java.applet.*;
import java.awt.*;
public class ParaPassing extends Applet
{
        String str;
        public void init()
        {
           str=getParameter("name1");
           if(str==null)
                str="Java";
            str="Hello  "+str;
        }
        public void paint(Graphics g)
        {
                g.drawString(str,50,50);
        }
}
```

para.html

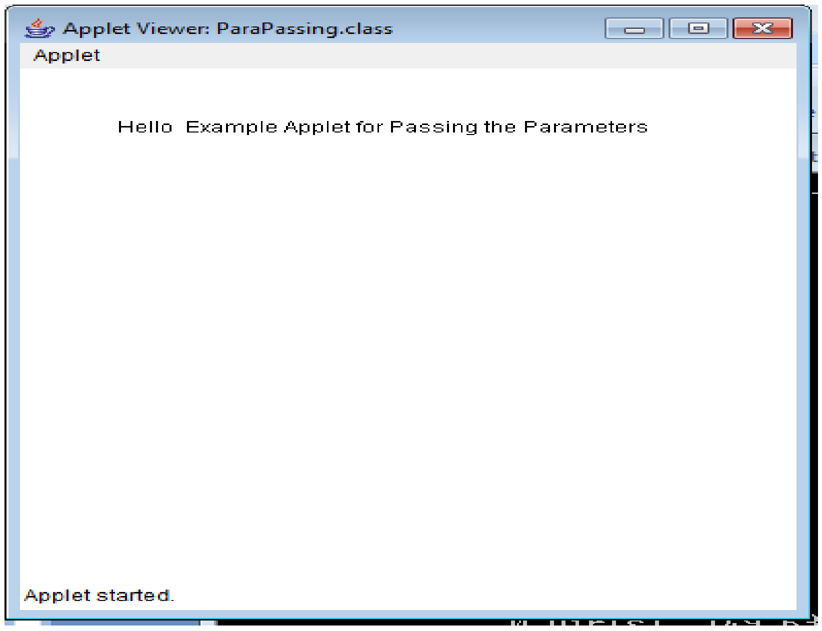
```
<html>
<head>      <title> Passing Parameters</title> </head>
<body bgcolor=pink >
        <applet code="ParaPassing.class"  width=400 height=400 >
        <param name="name1" value="Example Applet for Passing the Parameters" >
        </param>
        </applet>
</body>
</html>
```

Running the Program:

1. Compile the "ParaPassing.java" using the "javac" command, which generates the "ParaPassing.class " file
2. Use "ParaPassing.class" file to code attribute of the <applet> Tag and save it as "para.html"

3. Give "para.html" as input to the "**appletviewer**" to see the output or open the file using the applet enabled web broser.

Output:



```
Hello  Example Applet for Passing the Parameters
```

```
Applet started.
```

Getting the Input from the User

Applets work in the graphical environment. Therefore, applets treat inputs as text strings. we must first create an area of the screen in which user can type and edit input items. we can do this by using the **TextField** class of the applet package. Once text fields are created, user can enter and edit the content.

Next step is to retrieve the contents from the text fields for display of calculations, if any. The text fields contain the item in the form of String. They need to be converted to the right form, before they are used in any computations.

Example Program:

```
import java.applet.*;
import java.awt.*;
public class InputApplet extends Applet
{
        TextField text1,text2;
        Label l1,l2;
        public void init()
        {
                text1=new TextField(8);
                l1=new Label("Enter First No");
                text2=new TextField(8);
                l2=new Label("Enter second No");
                add(l1);
                add(text1);
                add(l2);
                add(text2);
                text1.setText("0");
                text2.setText("0");
        }       public void paint(Graphics g)
        {
                int x=0,y=0,z=0;
                String s1,s2,res;
                g.drawString("Enter a number in each text box",50,100);
                try
                {
                        s1=text1.getText();
                        x=Integer.parseInt(s1);
                        s2=text2.getText();
                        y=Integer.parseInt(s2);
                }
                catch(Exception e)
                {

                }

                        z=x+y;
                        res=String.valueOf(z);
                g.drawString("The Sum is :",50,150);
                g.drawString(res,150,150);
        }
        public boolean action(Event e,Object obj)
        {
                repaint();
                return true;
        }
}
```

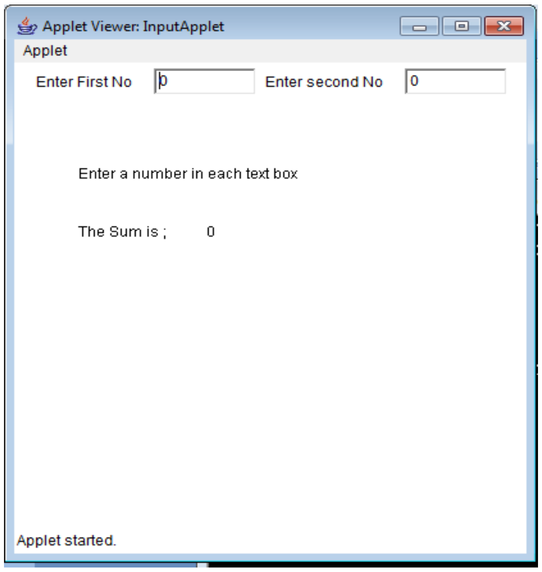
sum.html

```
<html>

<head><title>Geting input from the User </title>

</head>

<body>

        <applet code="InputApplet.class" width=400 height=400 ></applet>
```

```
</body>

</html>
```

Output:

Before Entering the Input **After Entering the input**



Applet Viewer: InputApplet

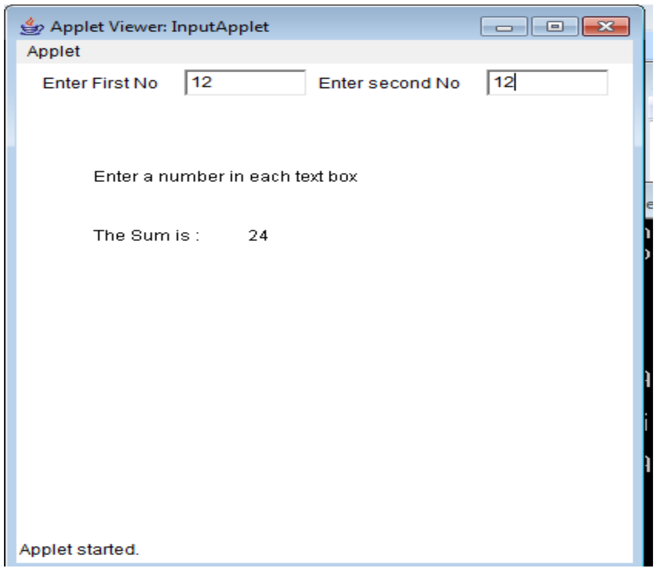
Applet

Enter First No 0 Enter second No 0

Enter a number in each text box

The Sum is ; 0

Applet started.



Applet Viewer: InputApplet

Applet

Enter First No 12 Enter second No 12

Enter a number in each text box

The Sum is : 24

Applet started.

Part –II: Event Handling

Types of Event Handling Mechanisms

There are two types of Event handling mechanisms supported by Java. First, the approach supported by Java 1.0, where the generated event is given hierarchically to the objects until it was handled. This can be also called as ***Hierarchical Event Handling Model***.

Second, the approach supported by Java 1.1, which registers the listeners to the source of the event and the registered listener processes the event and returns response to the source. This is called " ***Event Delegation Model***"

The Event Delegation 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 (1)***. Its concept is quite simple: a ***source (2)*** generates an event and sends it to one or more ***listeners (3)***. 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 response.

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 old 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.

1. What is an Event?

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.

Examples: Pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse etc..

2. 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 add*Type*Listener(*Type*Listener *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 add*Type*Listener(*Type*Listener *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 ***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 remove*Type*Listener(*Type*Listener *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.

3. **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 found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved.

The Event classes

The Event classes represent the event. Java provides us various Event classes but we will discuss those which are more frequently used.

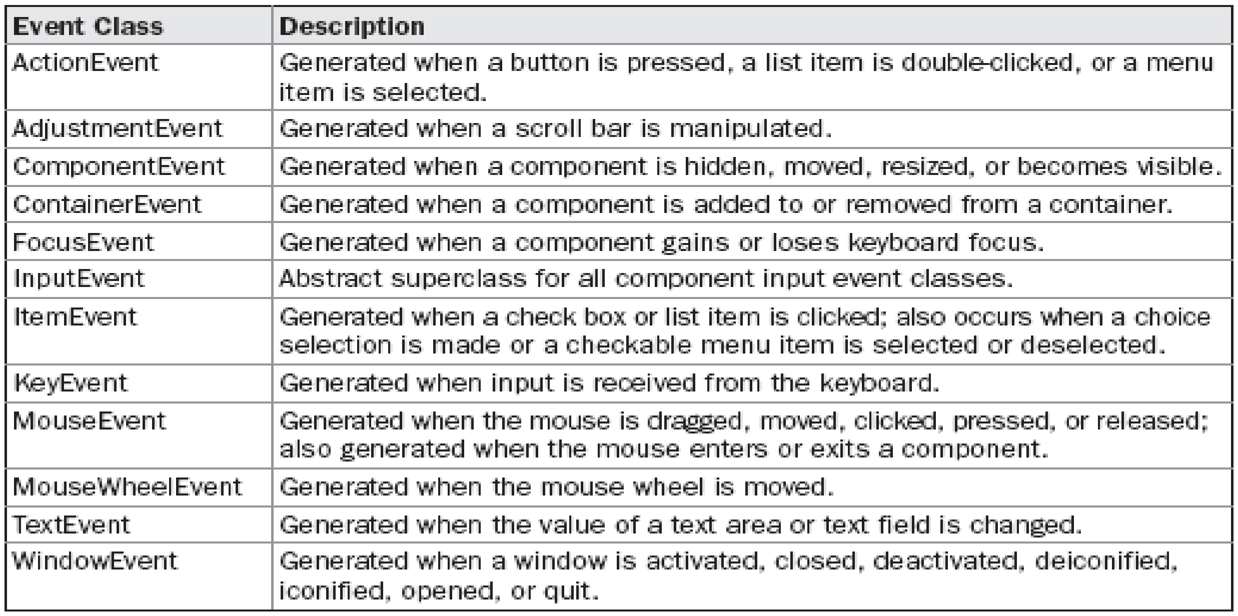
EventObject class

It is the root class from which all event state objects shall be derived. All Events are constructed with a reference to the object, the **source**, that is logically deemed to be the object upon which the Event in question initially occurred upon. This class is defined in java.util package.

The summary

- **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 many types of events that are generated by various user interface elements. Table 1 shows several commonly used event classes and provides a brief description of when they are generated. Commonly used constructors and methods in each class are described in the following sections



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

Table 1: List of Event Classes

1. **The ActionEvent class**

The ActionEvent class defines 5 integer constant and also defines 3 methods.

Integer constants:(instance variable)

ALT_MASK, **CTRL_MASK**, **META_MASK**, and **SHIFT_MASK ,** these are used to identify any modifier associated with an action event.

Methods:

i. ***String getActionCommand()*** - this is used to obtain the command name. For example when a button is pressed, the button contains the label, this method is used to obtain that.

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

iii. ***long getWhen()*** -used to get the time when the event took place.

2. The AdjustmentEvent Class

Integer Constants:

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:

BLOCK_DECREMENT	The user clicked inside the scroll bar to decrease its value.
BLOCK_INCREMENT	The user clicked inside the scroll bar to increase its value.
TRACK	The slider was dragged.
UNIT_DECREMENT	The button at the end of the scroll bar was clicked to decrease its value.
UNIT_INCREMENT	The button at the end of the scroll bar was clicked to increase its value.

Methods:

i. ***Adjustable getAdjustable()*** - returns the object that generated the event.

ii. ***int getAdjustmentType()*** **-** the type of adjustment event can be obtained by this method.

iii. ***int getValue()*** -the amount of adjustment can be obtained by this method

3. The ComponentEvent Class

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 here:

COMPONENT_HIDDEN	The component was hidden.
COMPONENT_MOVED	The component was moved.
COMPONENT_RESIZED	The component was resized.
COMPONENT_SHOWN	The component became visible.

Method:

Component getComponent( ) -this is used to return the component that has generated the event.

4. **The ContainerEvent Class**

   A **ContainerEvent** is generated when a component is added to or removed from a container. There are two types of container events.

Integer Constants:

COMPONENT_ADDED and **COMPONENT_REMOVED**.

The **ContainerEvent** class defines **int** constants that can be used to identify them.

Methods:

i. ***container getContainer()*** - used to get the reference of the container that has generated this event
ii. ***Component getChild()*** **-** used to get the component that has been added or removed.

5. **The FocusEvent Class**

   A **FocusEvent** is generated when a component gains or loses input focus.

Integer Constants:

FOCUS_GAINED and **FOCUS_LOST**.

Methods:

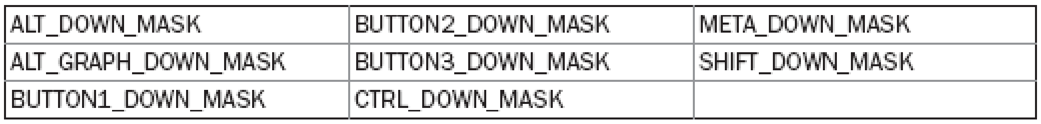
For example, assume that the focus is in a text field. If the user moves the mouse to adjust a scroll bar, the focus is temporarily lost.) The other component involved in the focus change, called the ***opposite component***, is passed in *other*.

i. ***Component getOppositeComponent( )***- used to determine the other opposite component.
ii. ***boolean isTemporary( )***- used to determine focus is changed temporarily

6. **The InputEvent Class**

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass for component input events. Its subclasses are **KeyEvent** and **MouseEvent**.

Integer Constants:



ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

Methods: these are used determine which key is pressed

i. boolean isAltDown( ) -
ii. boolean isAltGraphDown( )
iii. boolean isControlDown( )
iv. boolean isMetaDown( )
v. boolean isShiftDown( )

7. **The 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.

Integer Constants:

DESELECTED	The user deselected an item.
SELECTED	The user selected an item.

Methods:

i. ***Object getItem( )*** –used to get the reference of the item that has generated the event
ii. ***ItemSelectable getItemSelectable( )***- used to get reference to the selectable items.
iii. ***int getStateChange( )***- return the state change for the event.

8. 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.

Integer Constants:

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

Methods:

The **KeyEvent** class defines several methods, but 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( )

9. 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:

Integer Constants:

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.

Methods:

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:

i. **int getX( ) –** used to get the X coordinate
ii. **int getY( )** –used to get the Y coordinate
iii. **Point getPoint( )** –used to obtain the coordinates of the mouse
iv. **getClickCount( ) -** method obtains the number of mouse clicks for this event. Its signature is shown here:
v. **isPopupTrigger() -** method tests if this event causes a pop-up menu to appear on this platform

Java SE 6 added three methods to **MouseEvent** 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( )

10. The MouseWheelEvent Class

The **MouseWheelEvent** class encapsulates a mouse wheel event. It is a subclass of **MouseEvent**. Not all mice have wheels. If a mouse has a wheel, it is located between the left and right buttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integer constants:

WHEEL_BLOCK_SCROLL	A page-up or page-down scroll event occurred.
WHEEL_UNIT_SCROLL	A line-up or line-down scroll event occurred.

Methods:

MouseWheelEvent defines methods that give you access to the wheel event. To obtain the number of rotational units, call **getWheelRotation( )**, shown here:

int getWheelRotation( )

It returns the number of rotational units. If the value is positive, the wheel moved counterclockwise. If the value is negative, the wheel moved clockwise. To obtain the type of scroll, call **getScrollType( )**, shown next:

int getScrollType( )

It returns either **WHEEL_UNIT_SCROLL** or **WHEEL_BLOCK_SCROLL**. If the scroll type is **WHEEL_UNIT_SCROLL**, you can obtain the number of units to scroll by calling **getScrollAmount( )**. It is shown here:

int getScrollAmount( )

11. 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:

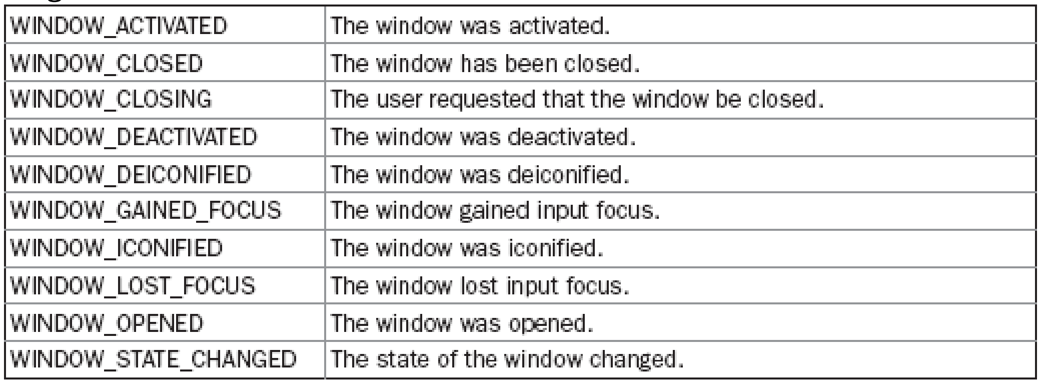
TextEvent(Object *src*, int *type*)

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

12. 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:

Integer Constants:



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.

Methods:

Acommonly 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( )

Event Sources

The following are the event source classes, that actually generate the event.

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar 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.

Interface	Description
ActionListener	Defines 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.
FocusListener	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 changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five 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 the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven 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 AdjustmentListener Interface

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

void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface
This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:
void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

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 FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, **focusGained( )** is invoked. When a component loses keyboard focus, **focusLost( )** is called. Their general forms are shown here:

void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)

The ItemListener Interface

This interface defines the **itemStateChanged( )** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent ie)

The KeyListener Interface

This interface defines three methods. The **keyPressed( )** and **keyReleased( )** methods are invoked when a key is pressed and released, respectively. The **keyTyped( )** method is invoked when a character has been entered. For example, if a user presses and releases the A key, three events are generated in sequence: key pressed, typed, and released. If a user presses and releases the HOME key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

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 MouseMotionListener Interface
This interface defines two methods. The **mouseDragged( )** method is called multiple times as the mouse is dragged. The **mouseMoved( )** method is called multiple times as the mouse is moved. Their general forms are shown here:

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

The MouseWheelListener Interface
This interface defines the **mouseWheelMoved( )** method that is invoked when the mouse wheel is moved. Its general form is shown here:

void mouseWheelMoved(MouseWheelEvent *mwe*)

The TextListener Interface

This interface defines the **textChanged( )** method that is invoked when a change occurs
in a text area or text field. Its general form is shown here:

void textChanged(TextEvent *te*)

The WindowFocusListener Interface

This interface defines two methods: **windowGainedFocus( )** and **windowLostFocus( )**. These are called when a window gains or loses input focus. Their general forms are shown here:

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)

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*)

Using the Delegation Model for Handling the Mouse Events

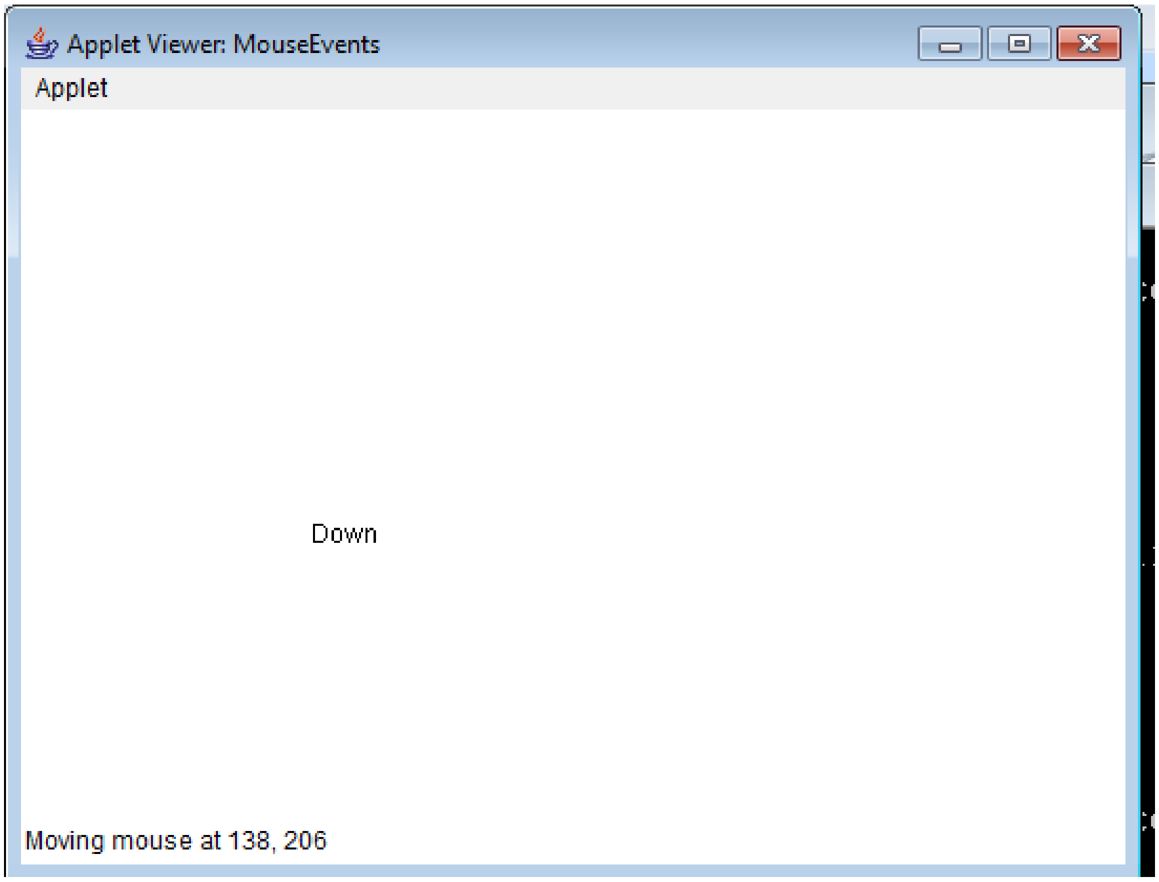
To handle mouse events, you must implement the **MouseListener** and the **MouseMotionListener** interfaces. It displays the current coordinates of the mouse in the applet's status window. Each time a button is pressed, the word "Down" is displayed at the location of the mouse pointer. Each time the button is released, the word "Up" is shown. If a button is clicked, the message "Mouse clicked" is displayed in the upper left corner of the applet display area

As the mouse enters or exits the applet window, a message is displayed in the upper-left corner of the applet display area. When dragging the mouse, a * is shown, which tracks with the mouse pointer as it is dragged. Notice that the two variables, **mouseX** and **mouseY**, store the location of the mouse when a mouse pressed, released, or dragged event occurs. These coordinates are then used by **paint( )** to display output at the point of these occurrences.

Example program for Handling Mouse Events

```
// Demonstrate the mouse event handlers.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</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);
}
// Handle mouse clicked.
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);
}
}
```

OutPut:



```
Down
```

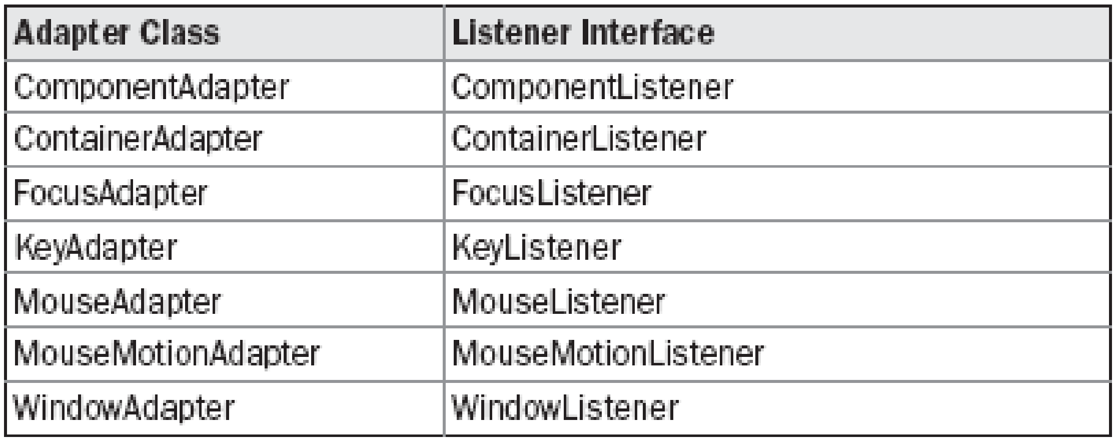
```
Moving mouse at 138, 206
```

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.

For example, the **MouseMotionAdapter** class has two methods, **mouseDragged( )** and **mouseMoved( )**, which are the methods defined by the **MouseMotionListener** interface. If you were interested in only mouse drag events, then you could simply extend **MouseMotionAdapter** and override **mouseDragged( )**. The empty implementation of **mouseMoved( )** would handle the mouse motion events for you.

The Following table provide the commonly used adapter classes:



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

Example program using the Adapter Class

```
// Demonstrate an adapter.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</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");
}
}
```

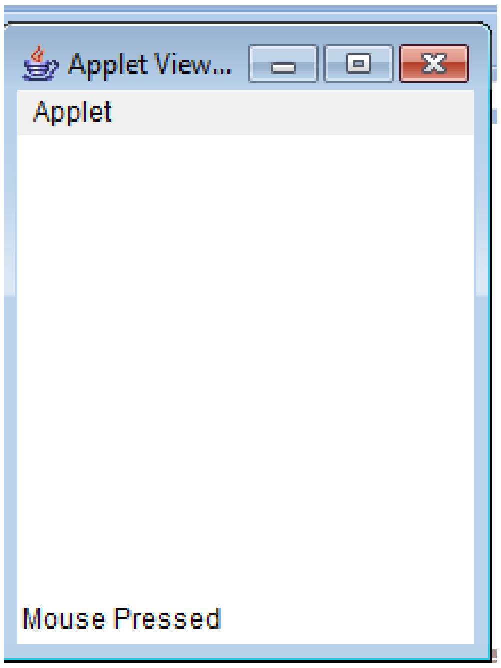
Inner Classes

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

Example program:

```
// 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());
     }

//inner class
class MyMouseAdapter extends MouseAdapter
{
     //overriding the mousePressed() method
     public void mousePressed(MouseEvent me)
     {
          showStatus("Mouse Pressed");
     }
}
}
```

OutPut:



```
Mouse Pressed
```

**************************End of 5th UNIT************************************