Chapter - 16

Event Handling

Event Sources 476
Event 476
Event Listeners 477
Handlers 477
Event classes in Java 477
Class EventObject 477
Class AWTEvent 477
AWT Event classes in Java 478
The ComponentEvent class 478
The MouseEvent class 479
Listeners 481
Interface MouseListener 482
Delegation Event Model 488
Adapter classes 488
Class MouseAdapter 489
Delegation event model using inner classes 492
Anonymous Inner classes 493
Interface MouseMotion Listener 494
Class MouseMotion Adapter 494
The KeyEvent class 496
Class KeyAdapter 498

Event Handling

Windows programming is based on event handling mechanism. Event handling is a mechanism where programs execute based on events performed. For example, let us consider some operations:

1. When mouse click is performed on the mycomputer object of desktop, the object is immediately object gets selected.
2. When mouse right-click is performed on mycomputer object, a short cut menu is displayed.
3. When mouse double-click is performed on mycomputer object, opens the mycomputer window,

In the above three operations, **mycomputer** is the object or source. **Click, right-click** and **double-click** are the events. **Selection of object, displaying short cut menu** and **opening of mycomputer window** are the programs(handlers) that are associated to mycomputer object.

The **mycomputer** object is associated with three programs and each executes for different event. For example, when mouse click event is performed on the object, the event is received by the **listener** which receives the events. The listener is the actual one which processes the event and can calls the program that is associated with this event to handle the event, in this case it calls selection of object program.

These programs execute based on event that is performed on the object. This mechanism of executing the programs based on the events performed is known as event handling mechanism. An object can contain any number of events and each event can have one program associated with it.

Event Sources

An event source is an object that generates event. A single source can generate more than one event. In the above example, mycomputer is an event source. Other event sources are: mouse, keyboard, button, checkbox etc.

Event

An event is an object that describes a state change in a source. In the above examples mouse click, mouse right-click, mouse double-click are the events. Other events are: **keydown**, **keyup**, **actionevent** etc.

Event Listeners

A listener is an object that is notified when an event occurs. Listeners are the interfaces. Events are received by the listeners and it is the one which call the handlers to handle the event. Examples of listeners are: **MouseListener**, **KeyListener** etc.

Handlers

Handlers are the methods that are executed for the events. These handlers are the actual ones that perform the operation for the event. Examples of handlers are: **mouseClicked()** , **KeyDown(),** etc,.

Event classes in Java

class EventObject

The superclass of all event classes is **java.util.EventObject**. This class contains two methods **getSource()** and **toString()**. The **getSource()** method returns object on which the event initially occurred.

class AWTEvent

The **AWTEvent** class is an abstract class and is present **in java.awt** package. This class is a subclass of **EventObject**. The **AWTEvent** is the superclass of all AWT event classes that are handled by delegation event model. These subclasses of **AWTEvent** are concrete classes and are present in **java.awt.event** package.

Methods	Meaning
protected void **consume**()	Consumes this event, if this event can be consumed.
int **getID**()	Returns the id of event type as integer.
protected boolean **isConsumed**()	Returns whether this event has been consumed.
String **paramString**()	Returns a string representing the state of this Event.
void **setSource**(Object newSource)	Retargets an event to a new source.
String **toString**()	Returns a String representation of this object.

AWT Event classes in JAVA

The event classes present in **java.awt.event** package are:

Classes	Meaning
ActionEvent	A semantic event which indicates that a component-defined action occurred.
AdjustmentEvent	The adjustment event emitted by Adjustable objects.
ComponentEvent	A low-level event which indicates that a component moved, changed size, or changed visibility (also, the root class for the other component-level events).
ContainerEvent	A low-level event which indicates that a container's contents changed because a component was added or removed.
FocusEvent	A low-level event which indicates that a Component has gained or lost the input focus.
ItemEvent	A semantic event which indicates that an item was selected or deselected.
KeyEvent	An event which indicates that a keystroke occurred in a component.
MouseEvent	An event which indicates that a mouse action occurred in a component.
MouseWheelEvent	An event which indicates that the mouse wheel was rotated in a component.
WindowEvent	A low-level event that indicates that a window has changed its status.

We see in detail of some of the event classes.

The ComponentEvent class

A low-level event which indicates that a component moved, changed size, or changed visibility (also, the root class for the other component-level events).

Component events are provided for notification purposes only; The **AWT** will automatically handle component moves and resizes internally so that GUI layout works properly regardless of whether a program is receiving these events or not.

In addition to serving as the base class for other component-related events (**InputEvent**, **FocusEvent**, **WindowEvent**, **ContainerEvent**), this class defines the events that indicate changes in a component's size, position, or visibility.

Constructors	Meaning
ComponentEvent (Component source, int id)	Constructs a ComponentEvent object.

Methods	Meaning
Component **getComponent**()	Returns the originator of the event.
String **paramString**()	Returns a parameter string identifying this event.

The MouseEvent class

An event which indicates that a mouse action occurred in a component. A mouse action is considered to occur in a particular component if and only if the mouse cursor is over the unobscured part of the component's bounds when the action happens.

This low-level event is generated by a component object for:

- Mouse Events
  - → a mouse button is pressed
  - → a mouse button is released
  - → a mouse button is clicked (pressed and released)
  - → the mouse cursor enters the component
  - → the mouse cursor exits the component
- Mouse Motion Events
  - → the mouse is moved
  - → the mouse is dragged

The following table shows the fields in the **MouseEvent** class.

Fields	Meaning
static int **MOUSE_CLICKED**	The "mouse clicked" event.
static int **MOUSE_DRAGGED**	The "mouse dragged" event.

static int **MOUSE_ENTERED**	The "mouse entered" event.
static int **MOUSE_EXITED**	The "mouse exited" event.
static int **MOUSE_MOVED**	The "mouse moved" event.
static int **MOUSE_PRESSED**	The "mouse pressed" event.
static int **MOUSE_RELEASE**	The "mouse released" event.
static int **MOUSE_WHEEL**	The "mouse wheel" event.

The following table shows the constructors in the **MouseEvent** class.

Constructors	Meaning
MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount, boolean popupTrigger)	Constructs a MouseEvent object with the specified source component, type, modifiers, coordinates, and click count.
MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int clickCount,boolean popupTrigger,int button)	Constructs a MouseEvent object with the specified source component, type, modifiers, coordinates, and click count.
MouseEvent(Component source, int id, long when, int modifiers, int x, int y, int xAbs, int yAbs, int clickCount, boolean popupTrigger, int button)	Constructs a MouseEvent object with the specified source component, type, modifiers, coordinates, absolute coordinates, and click count.

The following table shows the methods in the **MouseEvent** class.

Methods	Meaning
int **getButton**()	Returns which, if any, of the mouse buttons has changed state.
int **getClickCount**()	Returns the number of mouse clicks associated with this event.
Point **getPoint**()	Returns the x,y position of the event relative to the source component.

int **getX**()	Returns the horizontal x position of the event relative to the source component.
int **getY**()	Returns the vertical y position of the event relative to the source component.
String **paramString**()	Returns a parameter string identifying this event.

Listeners

When an event occurs from any component(event source[**mouse**, **button** etc.,]), it is received by the listener. The listener processes the event and calls the appropriate handler(method) to handle the event. Listeners are the interfaces that contain abstract methods. Any applet that implements these listeners should override these abstract methods otherwise the applet does not execute. All listener interfaces super interface is **EventListener**.

The following table shows various listeners of AWT.

Interfaces	Meaning
ActionListener	The listener interface for receiving action events.
AdjustmentListener	The listener interface for receiving adjustment events.
ComponentListener	The listener interface for receiving component events.
ContainerListener	The listener interface for receiving container events.
FocusListener	The listener interface for receiving keyboard focus events on a component.
ItemListener	The listener interface for receiving item events.
KeyListener	The listener interface for receiving keyboard events (keystrokes).
MouseListener	The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component.
MouseMotionListener	The listener interface for receiving mouse motion events on a component.
MouseWheelListener	The listener interface for receiving mouse wheel events on a component.
WindowListener	The listener interface for receiving window events.

The components have to register with the listener to listen or receive the events that comes from component. This can be achieved using the following statement.

public void add *Type*Listener(*Type*Listener listobj)

In the above statement, ***Type*** represent name of the event(Ex: **Mouse**, **Key**, **Action** etc,.).

Examples:

```
addMouseListener(this);
addKeyListener(this);
button1.addActionListener(this);
```

In above examples, ***this*** represents the handlers object i.e. the handlers are present in this(same) class. Therefore the Java looks for the handlers to execute in the same class.

When required, we can also remove the registration of listeners from the objects using remove method as:

public void remove *Type*Listener(*Type*Listener listobj)

Examples:

```
removeMouseListener(this);
removeKeyListener(this);
button1.removeActionListener(this);
```

Interface MouseListener

The **MouseListener** interface is used for receiving mouse events (press, release, click, enter, and exit) on a component. This interface contains five abstract methods as shown in below table.

Methods	Meaning
void **mouseClicked**(MouseEvent e)	Invoked when the mouse button has been clicked (pressed and released) on a component.
void **mouseEntered**(MouseEvent e)	Invoked when the mouse enters a component.
void **mouseExited**(MouseEvent e)	Invoked when the mouse exits a component.
void **mousePressed**(MouseEvent e)	Invoked when a mouse button has been pressed on a component.

void **mouseReleased**(MouseEvent e)	Invoked when a mouse button has been released on a component.

Any class that is interested in processing mouse events should implement this interface and have to override all abstract method, otherwise the class becomes an abstract class and program does no execute.

Any class that implements **MousetListener** interface should then register with a component using the component's **addMouseListener()** method. A mouse event is generated when the mouse is pressed, released clicked (pressed and released). A mouse event is also generated when the mouse cursor enters or leaves a component. When a mouse event occurs, the relevant method in the listener object is invoked, and the **MouseEvent** is passed to it.

A program to demonstrate mouse events

Bin>edit mouse1.java

```
import java.awt.*;
import java.awt.events.*;
import java.applet.*;
public class mouse1 extends Applet implements MouseListener
{
     String str="";
     public void init()
     {
          addMouseListener(this);
     }
     public void paint(Graphics g)
     {
          g.drawString(str,100,100);
     }
     public void mouseClicked(MouseEvent m)
     {
          str="MouseClicked…";
          repaint();
     }
     public void mouseExited(MouseEvent m)
     {
          str="MouseExited…";
          repaint();
     }
```

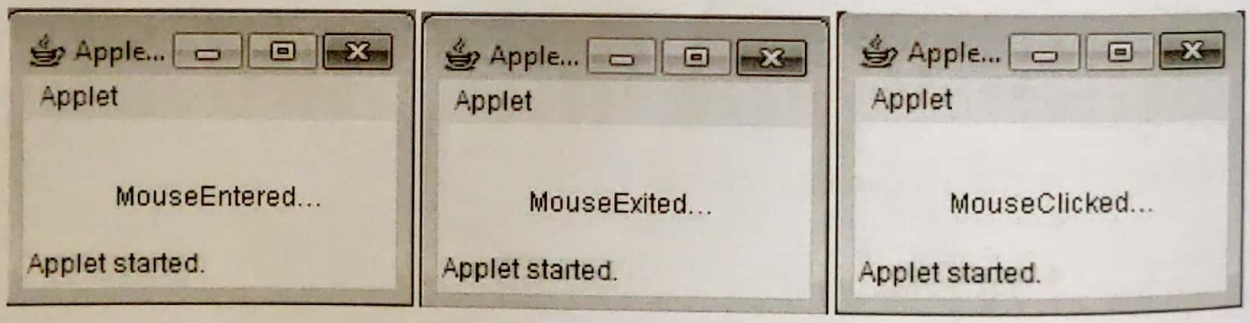
```
    public void mouseReleased(MouseEvent m)
    {
        str="MouseReleased...";
        repaint();
    }
    public void mousePressed(MouseEvent m)
    {
        str="MousePressed...";
        repaint();
    }
    public void mouseEntered(MouseEvent m)
    {
        str="MouseEntered...";
        repaint();
    }
}
```

Bin>javac mouse1.java

Bin>edit mouse1.html

```
<applet code="mouse1" width=400 height=400>
</applet>
```

Bin>appletviewer mouse1.html



Explanation:

When the mouse pointer is moved in to the applet window, the **MOUSE_ENTERED** event is raised. This event is received by the **addMouseListener(this);** statement and calls the **mouseEntered(MouseEvent m)** method of this(same) class and passes an argument of **MouseEvent** to it. In the **mouseEntered()** method **str** is assigned with "**MouseEntered**" string and calls the **repaint()** method, which calls **paint()** where it prints the **str** string on the applet "**MouseEntered...**" as shown in the first output window.

Same procedure is applied for other events (**exit**, **click**, **released**, **pressed**).

In the statement, **addMouseListener(this);** *this* represents the handlers object i.e. the handlers are present in this(same) class. Therefore the Java looks for the handlers to execute in the same class.

A program to print "Hello" message on the applet, at the mouse pointer co-ordinates(X and Y) where ever the mouse is clicked.

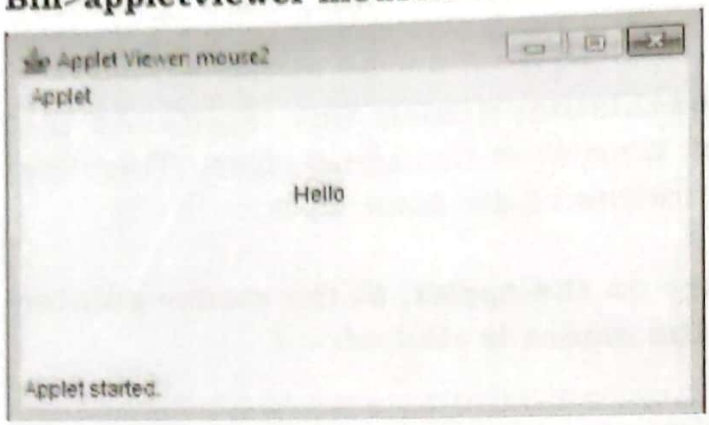
Bin>edit mouse2.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class mouse2 extends Applet implements  MouseListener
{
    String str="Hello";
    int x=0,y=0;
    public void init()
    {
          addMouseListener(this);
    }
    public void paint(Graphics g)
    {
          g.drawString(str,x,y);
    }
    public void mouseClicked(MouseEvent m)
    {
          x=m.getX();
          y=m.getY();
          repaint();
    }
    public void mousePressed(MouseEvent m){ }
    public void mouseEntered(MouseEvent m){ }
    public void mouseExited(MouseEvent m) { }
    public void mouseReleased(MouseEvent m){ }
}
```

Bin>javac mouse2.java
Bin>edit mouse2.html

```
<applet code="mouse2" width=400 height=400>
</applet>
```

Bin>appletviewer mouse2.html



Explanation:

When the mouse is clicked on the applet, the **mouseClicked(MouseEvent m)** method is called and passes an argument of **MouseEvent** object. In the method, the **getX()** and **getY()** methods of **MouseEvent** returns **X** and **Y** co-ordinates of Applet where the mouse is clicked and assigns to **x** and **y** variables. Then the **repaint()** method calls the **paint()** method which prints the **str**("Hello") at **x** and **y** co-ordinates of applet.

Click the mouse pointer at different locations on the applet as see the output. But, we can see only one output i.e. the last click event output. This is because, **paint()** method refresh the applet before it perform any drawings as a result old drawings are erased.

Without calling the **paint()** method we can perform the drawings by obtaining the **Graphics** object using **getGraphics()** method of **Component** class. The signature of **getGraphics()** method is

public Graphics getGraphics()

Example:

```
Graphics g=getGraphics();
g.drawString("applet",100,100);
```

The above program is written using getGraphics() method.

Bin>edit mouse3.java

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

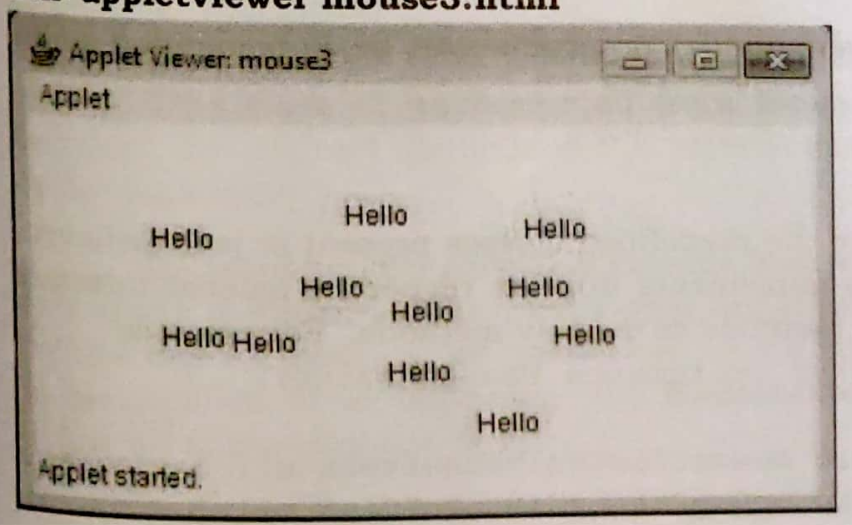
```
public class mouse3 extends Applet implements  MouseListener
{
    String str="Hello";
    int x=0,y=0;
    public void init()
    {
          addMouseListener(this);
    }
    public void mouseClicked(MouseEvent m)
    {
          x=m.getX();
          y=m.getY();
          Graphics g=getGraphics();
          g.drawString(str,x,y);
    }
    public void mousePressed(MouseEvent m){ }
    public void mouseEntered(MouseEvent m){ }
    public void mouseExited(MouseEvent m) { }
    public void mouseReleased(MouseEvent m){ }
}
```

Bin>javac mouse3.java

Bin>edit mouse2.html

```
<applet code="mouse3" width=400 height=400>
</applet>
```

Bin>appletviewer mouse3.html



In the above program, only mouse click event is handling therefore the applet class should contain **mouseClicked()** handler and other handlers are not required. But, in the program we are overriding other handlers as dummy methods, the methods that do not have code. These methods has to overridden because the **mouse3** class implements from **MouseListener** which have five abstract methods as explained above.

To overcome the above problem of overriding the un-necessary methods, we can implement **Delegation Event Model**.

Delegation Event Model

In the above mouse events programs, it can be seen that the applet code and event handling code(mouseClick(), mouseEntered(), etc..) is written in the same class i.e. applet class. As the program size grows this type of programming may become inconvenience.

Therefore we separate the applet user interface code with the event handling code. This can be achieved by writing the event handling code in a separate listener class. When an event is raised in the applet, it is received by the listener calls the handler present in the listener class. After executing the handler the control again goes back to applet. The advantage of this technique of delegating the event handling code to other classes is, the applet user interfacer code is separated with the application logic that process the events.

This technique of delegating the even handling mechanism to a separate class is known as **Delegation Event Model.** This model of writing the windows programming is the modern approach to handle events.

The delegation event model programs can be implemented using **Adapter classes**.

Adapter classes

Adapter classes are the predefined classes present in java.awt.event package. Adapter classes implements from its respective listener interface and overrides all abstract methods as dummy methods. For example

```
class MouseAdapter implements MouseListener
{
        public void mouseClicked(MouseEvent m){ }
        public void mouseEntered(MouseEvent m){ }
        public void mouseExited(MouseEvent m){ }
```

```
        public void mousePressed(MouseEvent m){ }
        public void mouseReleased(MouseEvent m){ }
}
```

The following table shows the adapter classes defined in **java.awt.event** package.

Classes	Meaning
ComponentAdapter	An abstract adapter class for receiving component events.
ContainerAdapter	An abstract adapter class for receiving container events.
FocusAdapter	An abstract adapter class for receiving keyboard focus events.
KeyAdapter	An abstract adapter class for receiving keyboard events.
MouseAdapter	An abstract adapter class for receiving mouse events.
MouseMotionAdapter	An abstract adapter class for receiving mouse motion events.
WindowAdapter	An abstract adapter class for receiving window events.

Using the adapter classes we override only those methods to handle required events and need not override dummy methods.

Class MouseAdapter

MouseAdapter is an abstract adapter class that receives mouse events. This class is implemented from **MouseListener** interface and overrides the abstract methods of this interface as dummy methods (which do not have code).

Any event handler class that extends **MouseAdapter** class can override only those methods for the events that are required. This abstract class defines null methods, so we ourself can define methods for events required to be handled. If we implement the **MouseListener** interface, we have to define all of the methods in it.

Create an object to the extended class of **MouseAdapter** and then register it with a component using the component's **addMouseListener()**

methods. The relevant method in the listener object is invoked and the **MouseEvent** is passed to it.

The following table shows methods overridden by **MouseAdapter** class.

Methods	Meaning
void **mouseClicked**(MouseEvent e)	Invoked when the mouse button has been clicked (pressed and released) on a component.
void **mouseDragged** (MouseEvent e)	Invoked when a mouse button is pressed on a component and then dragged.
void **mouseEntered**(MouseEvent e)	Invoked when the mouse enters a component.
void **mouseExited**(MouseEvent e)	Invoked when the mouse exits a component.
void **mouseMoved**(MouseEvent e)	Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.
void **mousePressed**(MouseEvent e)	Invoked when a mouse button has been pressed on a component.
void **mouseReleased**(MouseEvent e)	Invoked when a mouse button has been released on a component.
void **mouseWheelMoved**(MouseWheelEvent e)	Invoked when the mouse wheel is rotated.

A program to implement Delegation Event Model approach.
The above program we rewrite using the **MouseAdapter** class

Bin>edit mouse4.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class mouse4 extends Applet
{
    String str="Hello";
    int x=0,y=0;
    public void init()
    {
        mymousehandler mhobj=new mymousehandler(this);
```

```
            addMouseListener(mhobj);
      }
      public void paint(Graphics g)
      {
            g.drawString(str,x,y);
      }
}
class mymousehandler extends MouseAdapter
{
      mouse4 mobj;
      public mymousehandler(mouse4 mob)
      {
            mobj=mob;
      }
      public void mouseClicked(MouseEvent m)
      {
            mobj.x=m.getX();
            mobj.y=m.getY();
            mobj.repaint();
      }
}
```

Bin>javac mouse4.java

Bin>edit mouse4.html

```
<applet code="mouse4" width=400 height=400>
</applet>
```

Bin>appletviewer mouse4.html

We get the same output as mouse2.java program.

Explanation:

In the above program, the event handling code is delegated to **mymousehandler** class. The **mymousehandler** class is extended from **MouseAdapter** class and is overriding only **mouseClicked()** method and other methods are inherited from **MouseAdapter** class.

The **mymousehandler** class contain the event handlers that is why an object of this class is passed as an argument to **addMouseListener()** method. This means, when listener receives the event it call the handler from the **mymousehandler** object.

The **mymousehandler** class can't access the members of mouse4 class directly that is why to the constructor of **mymousehandler** we are passing an argument of **mouse4** object(**this**). The constructor is assigning the **mob** reference of **mouse4** to **mobj**. Using the **mobj** object **x**, **y** and **repaint()** are accessed.

Delegation event model using Inner classes

The above program is little complex to understand. To make it easy, we implement inner classes concept. A class defined inside another class is said to be inner class and such inner class can access the members of outer class directly with out using object name.

The above program is implemented using inner class.

Bin>edit mouse5.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class mouse5 extends Applet
{
    String str="Hello";
    int x=0,y=0;
    public void init()
    {
        addMouseListener(new mymousehandler());
    }
    public void paint(Graphics g)
    {
        g.drawString(str,x,y);
    }
    class mymousehandler extends MouseAdapter
    {
        public void mouseClicked(MouseEvent m)
        {
            x=m.getX();
            y=m.getY();
            repaint();
        }
    }
}
```

Bin>javac mouse5.java

Bin>edit mouse5.html

```
<applet code="mouse5" width=400 height=400>
</applet>
```

Bin>appletviewer mouse5.html

We get the same output as mouse4.java program.

Anonymous Inner classes

Anonymous inner class is a class which does not have a name. This class can be used where it is created and can't use else where. Using this class programming reduces the code. The following program demonstrate the use of anonymous inner class.

Bin>edit mouse7.java

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class mouse7 extends Applet
{
    String str="Hello";
    int x=0,y=0;
    public void init()
    {
          addMouseListener ( new MouseAdapter()
          {
                public void mouseClicked(MouseEvent m)
                {
                      x=m.getX();
                      y=m.getY();
                      Graphics g=getGraphics();
                      g.drawString(str,x,y);
                }
          } );
    }
}
```

Bin>javac mouse7.java

Bin>edit mouse7.html

```
<applet code="mouse7" width=400 height=400>
</applet>
```

Bin>appletviewer mouse7.html

Interface MouseMotionListener

The **MouseMotionListener** interface can be used for receiving mouse motion events(move and drag) on a component. This interface contains two abstract methods as shown in below table.

Methods	Meaning
void **mouseDragged**(MouseEvent e)	Invoked when a mouse button is pressed on a component and then dragged.
void **mouseMoved**(MouseEvent e)	Invoked when the mouse cursor has been moved onto a component but no buttons have been pushed.

Any class that wants to handle mouse motion events(move and drag) either implement **MouseMotionListener** interface or extend **MouseMotionAdapter** class.

Any class that implements **MouseMotionListener** should then registered with a component using the component's **addMouseMotionListener()** method. A mouse motion event is generated when the mouse is moved or dragged. (Many such events will be generated). When a mouse motion event occurs, the relevant method in the listener object is invoked, and the **MouseEvent** is passed to it.

Class MouseMotionAdapter

MouseMotionAdapter is an abstract adapter class is used for receiving mouse motion events. This class is implemented from **MouseMotionListener** interface and overrides the abstract methods of this interface as dummy methods.

Any event handler class that extends **MouseMotionAdapter** class can override only those methods for the events that are required. This abstract class defines null methods, so we ourself can define methods for events required to be handled. If we implement the **MouseMotionListener** interface, we have to define all of the methods in it.

Create an object to the extended class of **MouseMotionAdapter** and then register it with a component using the component's **addMouseMotionListener()** method. When the mouse is moved or dragged, the relevant method in the listener object is invoked and the **MouseEvent** is passed to it.

The following table shows methods overridden by **MouseMotionAdapter** class.

Methods	Meaning
void **mouseDragged**(MouseEvent e)	Invoked when a mouse button is pressed on a component and then dragged.
void **mouseMoved**(MouseEvent e)	Invoked when the mouse button has been moved on a component (with no buttons no down).

A program to handle mouse motion events

Bin>edit mouse6.java

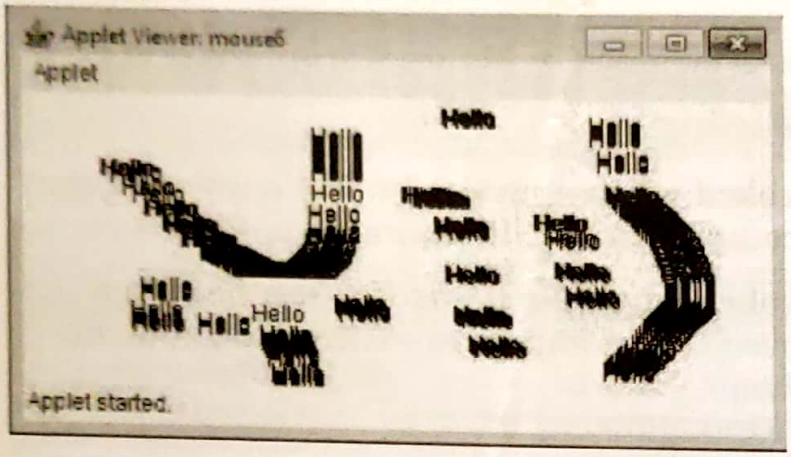
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class mouse6 extends Applet
{
    String str="Hello";
    int x=0,y=0;
    public void init()
    {
        addMouseMotionListener(new mymousemotionhandler());
    }
    class mymousemotionhandler extends MouseMotionAdapter
    {
        public void mouseDragged(MouseEvent m)
        {
            x=m.getX();
            y=m.getY();
            Graphics g=getGraphics();
            g.drawString(str,x,y);
        }
    }
}
```

Bin>javac mouse6.java

Bin>edit mouse6.html

```
<applet code="mouse6" width=400 height=400>
</applet>
```

Bin>appletviewer mouse6.html



The KeyEvent class

The **KeyEvent** class can be used to indicate the keystroke occurred in a component.

This low-level event is generated by a component object (such as a text field) when a key is pressed, released, or typed. The event is passed to every **KeyListener** or **KeyAdapter** object which registered to receive such events using the component's **addKeyListener()** method.

Pressing and releasing a key on the keyboard results in the generating the following key events

KEY_PRESSED
KEY_TYPED (is only generated if a valid Unicode character could begenerated.)
KEY_RELEASED

The following table shows public static final variables present in **KeyEvent** class.

VK_0 to VK_9
VK_A to VK_Z
VK_F1 to VK_F12

KEY_PRESSED	KEY_RELEASED	KEY_TYPED	VK_ALT
VK_CANCEL	VK_CAPS_LOCK	VK_CONTROL	VK_DELETE
VK_DOWN	VK_END	VK_ENTER	VK_UP
VK_ESCAPE	VK_HOME	VK_INSERT	VK_LEFT
VK_RIGHT	VK_SHIFT	VK_SPACE	VK_TAB

The following table shows constructors of **KeyEvent** class.

Constructors	Meaning
KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar)	Constructs a KeyEvent object.
KeyEvent(Component source, int id, long when, int modifiers, int keyCode, char keyChar, int keyLocation)	Constructs a KeyEvent object.

The following table shows methods of **KeyEvent** class.

Methods	Meaning
char **getKeyChar**()	Returns the character associated with the key in this event.
int **getKeyCode**()	Returns the integer keyCode associated with the key in this event.
int **getKeyLocation**()	Returns the location of the key that originated this key event.
static String **getKeyText**(int keyCode)	Returns a String describing the keyCode, such as "HOME", "F1" or "A".
boolean **isActionKey**()	Returns whether the key in this event is an "action" key.
String **paramString**()	Returns a parameter string identifying this event.
void **setKeyChar**(char keyChar)	Set the keyChar value to indicate a logical character.
void **setKeyCode**(int keyCode)	Set the keyCode value to indicate a physical key.

The two main methods of the **KeyEvent** class are **getKeyChar()** and **getKeyCode()**. The **getKeyChar()** method returns the character(alphabets, digits, special symbols) typed on the keyboard, where as **getKeyCode()** returns code of the any key on the keyboard(including special keys : **Enter**, **Arrow keys** etc..). These key codes can be compared with the fields of **KeyEvent** class (shown in the above table) to determine which key was typed on keyboard.

class KeyAdapter

KeyAdapter is an abstract adapter class is used for receiving key events. This class is implemented from **KeyListener** interface and overrides abstract methods of this interface as dummy methods.

Any event handler class that extends **KeyAdapter** class can override only those methods for the events that are required. This abstract class defines null methods, so we can only have to define methods for events required to handle. If we implement the **KeyListener** interface, we have to define all of the methods in it.

Create an object to the extended class of **KeyAdapter** and then register it with a component using the component's **addKeyListener()** method. When a key is pressed, released, or typed, the relevant method in the listener object is invoked, and the **KeyEvent** is passed to it.

The following table shows methods overridden by **KeyAdapter** class.

Methods	Meaning
void **keyPressed**(KeyEvent e)	Invoked when a key has been pressed.
void **keyReleased**(KeyEvent e)	Invoked when a key has been released.
void **keyTyped**(KeyEvent e)	Invoked when a key has been typed.

A program to display characters on the applet for which keys are typed on the keyboard.

Bin>edit key1.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class key1 extends Applet
{
    String str="";
    public void init()
    {
        addKeyListener(new mykeyhandler());
    }
    public void paint(Graphics g)
    {
        g.drawString(str,50,50);
    }
```

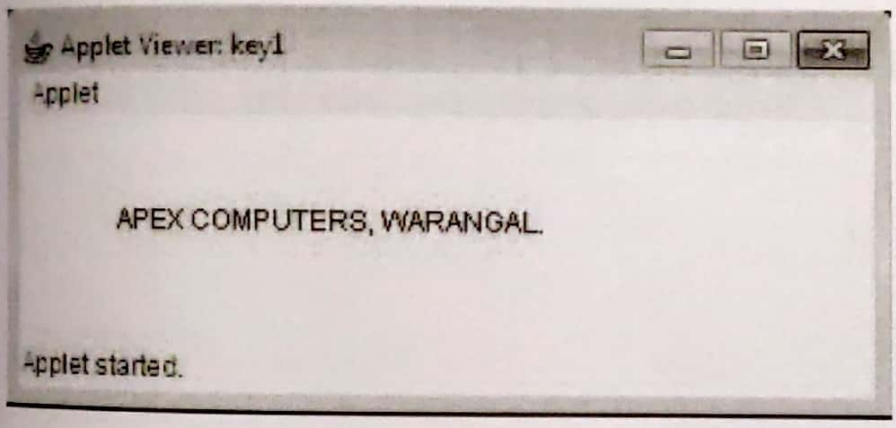
```
    class mykeyhandler extends KeyAdapter
    {
          public void keyPressed(KeyEvent k)
          {
                str=str+k.getKeyChar();
                repaint();
          }
    }
}
```

Bin>javac key1.java

Bin>edit key1.html

```
<applet code="key1" width=400 height=400>
</applet>
```

Bin>appletviewer key1.html



Note: Click the mouse button in the applet window and then type on the keyboard.

A program to display typed text on separate lines in the applet.

Bin>edit key2.java

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class key2 extends Applet
{
    String str="";
    int x=10,y=20;  // x represents x co-ordinate
                    // y represents y co-ordinate
```

```
    public void init()
    {
          addKeyListener(new mykeyhandler());
    }
    class mykeyhandler extends KeyAdapter
    {
          public void keyPressed(KeyEvent k)
          {
                int key=k.getKeyCode();
                if(key==KeyEvent.VK_ENTER)
                                    //when enter key is pressed
                {
                      y=y+15;
                      x=10;
                }
                else                // other than enter key
                {
                      x=x+10;
                      Graphics g=getGraphics();
                      g.drawString(""+(char)key,x,y);
                }
          }
    }
}
```

Bin>javac key2.java

Bin>edit key2.html

```
<applet code="key2" width=400 height=400>
</applet>
```

Bin>appletviewer key2.html