**3. Event Handling**

**Introduction**

Event handling is fundamental to Java programming because it is integral to the creation of applets andother types of GUI-based programs. Applets are event-driven programs that use a graphical user interface to interact with the user. Furthermore, any program that uses a graphical user interface, such as a Java application written for Windows, is event driven. Thus, you cannot write these types of programs without a solid command of event handling. Events are supported by a number of packages, including **java.util**, **java.awt**, and**java.awt.event**.

**Two Event Handling Mechanisms**

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

**The Delegation Event Model**

The modern approach to handling events is based on the *delegation event model,* which definesstandard and consistent mechanisms to generate and process events.

**Concept**a *source* generates an event and sends it to one or more *listeners.* In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to “delegate” the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them. This is a more efficient way to handle events than the design used by the 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.

**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.

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,a software or hardware failure occurs, or an operation is completed. You are free to defineevents that are appropriate for your application.

**Event Sources**

A*source* is an object that generates an event. This occurs when the internal state of that objectchanges in some way. Sources may generate more than one type of event.

Asource must register listeners in order for the listeners to receive notifications abouta specific type of event. Each type of event has its own registration method. Here is thegeneral 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 methodthat registers a mouse motion listener is called **addMouseMotionListener( )**. When an eventoccurs, all registered listeners are notified and receive a copy of the event object. This is knownas *multicasting* the event. In all cases, notifications are sent only to listeners that register toreceive them.

Some sources may allow only one listener to register. The general form of such a methodIs 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 suchan 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( )**.

**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 aboutspecific types of events. Second, it must implement methods to receive and process thesenotifications.

The methods that receive and process events are defined in a set of interfaces found in**java.awt.event**.

**Event Classes**

The most widely used events are those defined by the AWT and those defined by

Swing.

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

EventObject(Object *src*)

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

**EventObject** contains two methods: **getSource( )** and **toString( )**. The **getSource( )** methodreturns the source of the event. Its general form is shown here:

Object getSource( )

As expected, **toString( )** returns the string equivalent of the event.

The class **AWTEvent**, defined within the **java.awt** package, is a subclass of **EventObject**.

It is the superclass (either directly or indirectly) of all AWT-based events used by the delegationevent model. Its **getID( )** method can be used to determine the type of the event. The signatureof this method is shown here:

int getID( )

The package **java.awt.event** defines many types of events that are generated by varioususer interface elements.

Following are the event classes.

**The ActionEvent Class**

An **ActionEvent** is generated when a button is pressed, a list item is double-clicked, or amenu item is selected. The **ActionEvent** class defines four integer constants that can beused 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.

**ActionEvent** has these three constructors:

ActionEvent(Object *src*, int *type*, String *cmd*)

ActionEvent(Object *src*, int *type*, String *cmd*, int *modifiers*)

ActionEvent(Object *src*, int *type*, String *cmd*, long *when*, int *modifiers*)

Here, *src* is a reference to the object that generated this event. The type of the event is specifiedby *type,* and its command string is *cmd.* The argument *modifiers* indicates which modifier keys(ALT, CTRL, META, and/or SHIFT) were pressed when the event was generated. The *when*parameter specifies when the event occurred.

You can obtain the command name for the invoking **ActionEvent** object by using the**getActionCommand( )** method, shown here:

String getActionCommand( )

For example, when a button is pressed, an action event is generated that has a commandname 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. Its form is shown here:

int getModifiers( )

The method **getWhen( )** returns the time at which the event took place. This is called theevent’s *timestamp.* The **getWhen( )** method is shown here:

long getWhen( )

**The AdjustmentEvent Class**

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. Theconstants 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.

In addition, there is an integer constant, **ADJUSTMENT\_VALUE\_CHANGED**, thatindicates that a change has occurred.

Here is one **AdjustmentEvent** constructor:

AdjustmentEvent(Adjustable *src*, int *id*, int *type*, int *data*)

Here, *src* is a reference to the object that generated this event. The *id* specifies the event. Thetype of the adjustment is specified by *type,* and its associated data is *data.*The **getAdjustable( )** method returns the object that generated the event. Its form isshown here:

Adjustable getAdjustable( )

The type of the adjustment event may be obtained by the **getAdjustmentType( )** method. Itreturns one of the constants defined by **AdjustmentEvent**. The general form is shown here:

int getAdjustmentType( )

The amount of the adjustment can be obtained from the **getValue( )** method, shown here:

int getValue( )

For example, when a scroll bar is manipulated, this method returns the value representedby that change.

**The ComponentEvent Class**

A**ComponentEvent** is generated when the size, position, or visibility of a component ischanged. There are four types of component events. The **ComponentEvent** class definesinteger constants that can be used to identify them. The constants and their meanings areshown 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.

**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 isspecified by *type.*

**ComponentEvent** is the superclass either directly or indirectly of **ContainerEvent**,**FocusEvent**, **KeyEvent**, **MouseEvent**, and **WindowEvent**.

The **getComponent( )** method returns the component that generated the event. It is as

Component getComponent( )

**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. The **ContainerEvent** class defines **int** constants thatcan 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 specifiedby *type,* and the component that has been added to or removed from the container is *comp.*You can obtain a reference to the container that generated this event by using the**getContainer( )** method, shown here:

Container getContainer( )

The **getChild( )** method returns a reference to the component that was added to or removedfrom the container. Its general form is shown here:

Component getChild( )

**The FocusEvent Class**

A**FocusEvent** is generated when a component gains or loses input focus. These events areidentified by the integer constants **FOCUS\_GAINED** and **FOCUS\_LOST**.**FocusEvent** is a subclass of **ComponentEvent** and has these constructors:

FocusEvent(Component *src*, int *type*)

FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*)

FocusEvent(Component *src*, int *type*, boolean *temporaryFlag*, Component *other*)

Here, *src* is a reference to the component that generated this event. The type of the event isspecified by *type.* The argument *temporaryFlag* is set to **true** if the focus event is temporary.Otherwise, it is set to **false**. (A temporary focus event occurs as a result of another userinterface operation. For example, assume that the focus is in a text field. If the user movesthe mouse to adjust a scroll bar, the focus is temporarily lost.)

The other component involved in the focus change, called the *opposite component,* is passedin *other.* Therefore, if a **FOCUS\_GAINED** event occurred, *other* will refer to the component thatlost focus. Conversely, if a **FOCUS\_LOST** event occurred, *other* will refer to the componentthat gains focus.

You can determine the other component by calling **getOppositeComponent( )**,

shown here:

Component getOppositeComponent( )

The opposite component is returned.

The **isTemporary( )** method indicates if this focus change is temporary. Its form

is shown here:

boolean isTemporary( )

The method returns **true** if the change is temporary. Otherwise, it returns **false**.

**The InputEvent Class**

The abstract class **InputEvent** is a subclass of **ComponentEvent** and is the superclass forcomponent input events. Its subclasses are **KeyEvent** and **MouseEvent**.**InputEvent** defines several integer constants that represent any modifiers, such as thecontrol key being pressed, that might be associated with the event. Originally, the **InputEvent**class defined the following eight values to represent the modifiers:

ALT\_MASK BUTTON2\_MASK META\_MASK

ALT\_GRAPH\_MASK BUTTON3\_MASK SHIFT\_MASK

BUTTON1\_MASK CTRL\_MASK

However, because of possible conflicts between the modifiers used by keyboard events andmouse events, and other issues, the following extended modifier values were added:

ALT\_DOWN\_MASK BUTTON2\_DOWN\_MASK META\_DOWN\_MASKALT\_GRAPH\_DOWN\_MASK BUTTON3\_DOWN\_MASK SHIFT\_DOWN\_MASKBUTTON1\_DOWN\_MASK CTRL\_DOWN\_MASK

When writing new code, it is recommended that you use the new, extended modifiers ratherthan the original modifiers.

To test if a modifier was pressed at the time an event is generated, use the **isAltDown( )**,**isAltGraphDown( )**, **isControlDown( )**, **isMetaDown( )**, and **isShiftDown( )** methods.Theforms of these methods are shown here:

boolean isAltDown( )

boolean isAltGraphDown( )

boolean isControlDown( )

boolean isMetaDown( )

boolean isShiftDown( )

You can obtain a value that contains all of the original modifier flags by calling the

**getModifiers( )** method. It is shown here:

int getModifiers( )

You can obtain the extended modifiers by calling **getModifiersEx( )**, which is shown here:

int getModifiersEx( )

**The ItemEvent Class**

An **ItemEvent** is generated when a check box or a list item is clicked or when a checkable menuitem is selected or deselected. There are two types of item events, which are identified by the following integer constants:

DESELECTED The user deselected an item.

SELECTED The user selected an item.

In addition, **ItemEvent** defines one integer constant, **ITEM\_STATE\_CHANGED**, thatsignifies a change of state.

**ItemEvent** has this constructor:

ItemEvent(ItemSelectable *src*, int *type*, Object *entry*, int *state*)

Here, *src* is a reference to the component that generated this event. For example, this mightbe a list or choice element. The type of the event is specified by *type.* The specific item thatgenerated the item event is passed in *entry.* The current state of that item is in *state.*

The **getItem( )** method can be used to obtain a reference to the item that generated anevent. Its signature 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 is shown here:

ItemSelectable getItemSelectable( )

Lists and choices are examples of user interface elements that implement the **ItemSelectable**interface.

The **getStateChange( )** method returns the state change (that is, **SELECTED** or

**DESELECTED**) for the event. It is shown here:

int getStateChange( )

**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. Thelast event occurs only when a character is generated. Remember, not all keypresses resultin characters. For example, pressing SHIFT does not generate a character.

There are many other integer constants that are defined by **KeyEvent**. For example, **VK\_0**through **VK\_9** and **VK\_A** through **VK\_Z** define the ASCII equivalents of the numbers andletters. Here are some others:

VK\_ALT VK\_DOWN VK\_LEFT

VK\_RIGHT VK\_CANCEL VK\_ENTER VK\_PAGE\_DOWN VK\_SHIFT VK\_CONTROL

VK\_ESCAPE VK\_PAGE\_UP VK\_UP

The **VK** constants specify *virtual key codes* and are independent of any modifiers, such ascontrol, shift, or alt.**KeyEvent** is a subclass of **InputEvent**. Here is one of its constructors:

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

Here, *src* is a reference to the component that generated the event. The type of the event isspecified by *type.* The system time at which the key was pressed is passed in *when.* The *modifiers*argument indicates which modifiers were pressed when this key event occurred. The virtualkey code, such as **VK\_UP**, **VK\_A**, and so forth, is passed in *code.* The character equivalent(if one exists) is passed in *ch.* If no valid character exists, then *ch* contains **CHAR\_UNDEFINED**.For **KEY\_TYPED** events, *code* will contain **VK\_UNDEFINED**.

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

char getKeyChar( )

int getKeyCode( )

If no valid character is available, then **getKeyChar( )** returns **CHAR\_UNDEFINED**. Whena **KEY\_TYPED** event occurs, **getKeyCode( )** returns **VK\_UNDEFINED**.

**The MouseEvent Class**

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

MOUSE\_CLICKED The user clicked the mouse.

MOUSE\_DRAGGED The user dragged the mouse.

MOUSE\_ENTERED The mouse entered a component.

MOUSE\_EXITED The mouse exited from a component.

MOUSE\_MOVED The mouse moved.

MOUSE\_PRESSED The mouse was pressed.

MOUSE\_RELEASED The mouse was released.

MOUSE\_WHEEL The mouse wheel was moved.

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

MouseEvent(Component *src*, int *type*, long *when*, int *modifiers*,

int *x*, int *y*, int *clicks*, boolean *triggersPopup*)

Here, *src* is a reference to the component that generated the event. The type of the event isspecified by *type.* The system time at which the mouse event occurred is passed in *when.* The*modifiers* argument indicates which modifiers were pressed when a mouse event occurred.

The coordinates of the mouse are passed in *x* and *y.* The click count is passed in *clicks.* The*triggersPopup* flag indicates if this event causes a pop-up menu to appear on this platform.Two commonly used methods in this class are **getX( )** and **getY( )**. These return the X andY coordinates of the mouse within the component when the event occurred. Their forms areshown here:

int getX( )

int getY( )

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

Point getPoint( )

It returns a **Point** object that contains the X,Y coordinates in its integer members: **x** and **y**.The **translatePoint( )** method changes the location of the event. Its form is shown here:

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

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

The **getClickCount( )** method obtains the number of mouse clicks for this event.

Its signature is shown here:

int getClickCount( )

The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on thisplatform. Its form is shown here:

boolean isPopupTrigger( )

Also available is the **getButton( )** method, shown here:

int getButton( )

It returns a value that represents the button that caused the event. The return value will beone of these constants defined by **MouseEvent**:

NOBUTTON BUTTON1 BUTTON2 BUTTON3

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

Java SE 6 added three methods to **MouseEvent** that obtain the coordinates of the mouserelative to the screen rather than the component. They are shown here:

Point getLocationOnScreen( )

int getXOnScreen( )

int getYOnScreen( )

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

**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 rightbuttons. Mouse wheels are used for scrolling. **MouseWheelEvent** defines these two integerconstants:

WHEEL\_BLOCK\_SCROLL A page-up or page-down scroll event occurred.

WHEEL\_UNIT\_SCROLL A line-up or line-down scroll event occurred.

Here is one of the constructors defined by **MouseWheelEvent**:

MouseWheelEvent(Component *src*, int *type*, long *when*, int *modifiers*,

int *x*, int *y*, int *clicks*, boolean *triggersPopup*,int *scrollHow*, int *amount*, int *count*)

Here, *src* is a reference to the object that generated the event. The type of the event is specifiedby *type.* The system time at which the mouse event occurred is passed in *when.* The *modifiers*argument indicates which modifiers were pressed when the event occurred. The coordinatesof the mouse are passed in *x* and *y.* The number of clicks the wheel has rotated is passed in*clicks.* The *triggersPopup* flag indicates if this event causes a pop-up menu to appear on thisplatform. The *scrollHow* value must be either **WHEEL\_UNIT\_SCROLL** or **WHEEL\_BLOCK\_SCROLL**. The number of units to scroll is passed in *amount.* The *count* parameter indicates

the number of rotational units that the wheel moved.

**MouseWheelEvent** defines methods that give you access to the wheel event. To obtainthe 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 scrollby calling **getScrollAmount( )**. It is shown here:

int getScrollAmount( )

**The TextEvent Class**

Instances of this class describe text events. These are generated by text fields and text areaswhen 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 isspecified by *type.*

The **TextEvent** object does not include the characters currently in the text component thatgenerated the event. Instead, your program must use other methods associated with the textcomponent to retrieve that information. This operation differs from other event objectsdiscussed in this section. For this reason, no methods are discussed here for the **TextEvent**class. Think of a text event notification as a signal to a listener that it should retrieve informationfrom a specific text component.

**The WindowEvent Class**

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

WINDOW\_ACTIVATED The window was activated.

WINDOW\_CLOSED The window has been closed.

WINDOW\_CLOSING The user requested that the window be closed.

WINDOW\_DEACTIVATED The window was deactivated.

WINDOW\_DEICONIFIED The window was deiconified.

WINDOW\_GAINED\_FOCUS The window gained input focus.

WINDOW\_ICONIFIED The window was iconified.

WINDOW\_LOST\_FOCUS The window lost input focus.

WINDOW\_OPENED The window was opened.

WINDOW\_STATE\_CHANGED The state of the window changed.

**WindowEvent** is a subclass of **ComponentEvent**. It defines several constructors. Thefirst is

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

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

The next three constructors offer more detailed control:

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

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

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

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

Acommonly used method in this class is **getWindow( )**. It returns the **Window** objectthat generated the event. Its general form is shown here:

Window getWindow( )

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

Window getOppositeWindow( )

int getOldState( )

int getNewState( )

**Sources of Events**

Following table shows some of the user interface components that can generate the events.

**Event Source Description**

Button Generates action events when the button is pressed.

Checkbox Generates item events when the check box is selected ordeselected.

Choice Generates item events when the choice is changed.

List Generates action events when an item is double- clicked; generatesitem events when an item is selected or deselected.

Menu Item Generates action events when a menu item is selected; generatesitem events when a checkable menu item is selected or deselected.

Scrollbar 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**

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 methoddefined by the listener and provides an event object as its argument.

Following are the listeners.

**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)

**Examples on ActionListener**

1)//program to accept two numbers through text boxes and display its addition in third //text box when the Button captioned result is pressed.

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

/\*<applet code=addition width=300 height=300>

</applet>\*/

public class addition extends Applet implements ActionListener

{

TextField t1,t2,t3;

Button res=new Button("Result");

Label l1,l2,l3;

public void init()

{

l1=new Label("NO1");

l2=new Label("NO2");

l3=new Label("Result of addition");

t1=new TextField();

t2=new TextField();

t3=new TextField();

setLayout(new GridLayout(4,2));

add(l1);add(t1);

add(l2);add(t2);

add(l3);add(t3);

add(res);

res.addActionListener(this);

}

public void actionPerformed(ActionEvent e)

{

t3.setText(Integer.parseInt(t1.getText())+Integer.parseInt(t2.getText())+" ");

}

}

2)//program to change the background color of applet when we press //RED,GREEN,BLUE captioned buttons.

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*<applet code=rgb width=400 height=400>

</applet>\*/

public class rgb extends Applet implements ActionListener

{

Button r=new Button("RED");

Button g=new Button("GREEN");

Button b=new Button("BLUE");

public void init()

{

add(r);add(g);add(b);

r.addActionListener(this);

g.addActionListener(this);

b.addActionListener(this);

}

public void actionPerformed(ActionEvent e)

{

String msg=e.getActionCommand();

if(msg.equals("RED"))

{

setBackground(Color.RED);

}

else

if(msg.equals("GREEN"))

{

setBackground(Color.GREEN);

}

else

setBackground(Color.BLUE);

}

}

3)//Program on list

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

/\*<applet code=listevent width=400 height=400>

</applet>\*/

public class listevent extends Applet implements ActionListener

{

List l=new List(10);

String msg=" ";

public void init()

{

l.add("AJP");

l.add("OMD");

l.add("MAN");

add(l);

l.addActionListener(this);

}

public void actionPerformed(ActionEvent e)

{

msg=e.getActionCommand();

repaint();

}

public void paint(Graphics g)

{

g.drawString(msg,100,100);

}

}

**The AdjustmentListener Interface**

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

void adjustmentValueChanged(AdjustmentEvent ae)

**Example:**

1. //program to move string according to scrolling of scrollbar.

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

/\*<applet code=scrolldemo width=400 height=400>

</applet>\*/

public class scrolldemo extends Applet implements AdjustmentListener

{

Scrollbar h,v;

String msg="Hello";

int x=50,y=50;

public void init()

{

setLayout(new BorderLayout());

h=new Scrollbar(Scrollbar.HORIZONTAL,10,10,0,200);

v=new Scrollbar(Scrollbar.VERTICAL,10,10,0,200);

add(h,BorderLayout.SOUTH);

add(v,BorderLayout.WEST);

h.addAdjustmentListener(this);

v.addAdjustmentListener(this);

}

public void adjustmentValueChanged(AdjustmentEvent e)

{

Scrollbar s=(Scrollbar)e.getAdjustable();

if(s==h)

{

if(e.getAdjustmentType()==1)

x=x+h.getValue();

else

x=x-h.getValue();

}

else

{

if(e.getAdjustmentType()==1)

y=y+v.getValue();

else

y=y-v.getValue();

}

repaint();

}

public void paint(Graphics g)

{

g.drawString(msg,x,y);

}

}

**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)

**Example:**

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*<applet code=compevent width=400 height=400>

</applet>\*/

public class compevent extends Applet implements ActionListener,ComponentListener

{

Button b=new Button("OK");

public void init()

{

setLayout(null);

add(b);

b.setBounds(100,100,50,50);

b.addActionListener(this);

b.addComponentListener(this);

}

public void actionPerformed(ActionEvent e)

{

b.setBounds(100,100,70,70);

}

public void componentResized(ComponentEvent e)

{

showStatus("Button resized");

}

public void componentMoved(ComponentEvent e){}

public void componentShown(ComponentEvent e){}

public void componentHidden(ComponentEvent e){}

}

**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 ofan item changes. Its general form is shown here:

void itemStateChanged(ItemEvent ie)

**Example:**

**//Program to handle choice control itemevent.**

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*<applet code=itemevent width=400 height=400>

</applet>\*/

public class itemevent extends Applet implements ItemListener

{

Choice subj=new Choice();

Choice teacher=new Choice();

public void init()

{

subj.add("AJP");

subj.add("OMD");

subj.add("MAN");

subj.add("STE");

teacher.add("DMJ");

teacher.add("SSK");

teacher.add("MCP");

teacher.add("MMK");

add(subj);

add(teacher);

subj.addItemListener(this);

teacher.addItemListener(this);

}

public void itemStateChanged(ItemEvent e)

{

int i,j;

if(e.getSource()==subj)

{

i=subj.getSelectedIndex();

teacher.select(i);

}

else

{

i=teacher.getSelectedIndex();

subj.select(i);

}

}

}

**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)

**Example:**

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

/\*<applet code=keyeventdemo width=400 height=400>

</applet>\*/

public class keyeventdemo extends Applet implements KeyListener

{

String str=" ";

public void init()

{

requestFocus();

addKeyListener(this);

}

public void paint(Graphics g)

{

g.drawString(str,100,100);

}

public void keyPressed(KeyEvent e)

{

switch(e.getKeyCode())

{

case KeyEvent.VK\_F1: str="The key pressed is F1";

break;

case KeyEvent.VK\_F2: str="The key pressed is F2";

break;

case KeyEvent.VK\_ENTER: str="The key pressed is enter";

break;

case KeyEvent.VK\_UP:str="The key pressed is up arrow key";

break;

default: str="The key pressed is other than F1,F2,Enter,Up key";

}

repaint();

}

public void keyReleased(KeyEvent e)

{}

public void keyTyped(KeyEvent e)

{

str="The key typed is"+" "+e.getKeyChar();

}

}

**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*)

**Example:**

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

/\*<applet code=mouseevdemo width=400 height=400>

</applet>\*/

public class mouseevdemo extends Applet implements MouseListener

{

public void init()

{

addMouseListener(this);

}

public void mouseEntered(MouseEvent e)

{

showStatus("Mouse entered in applet area");

}

public void mouseExited(MouseEvent e)

{

showStatus("Mouse exited from applet area");

}

public void mousePressed(MouseEvent e)

{

showStatus("Mouse Pressed in applet area at"+e.getX()+" "+e.getY());

}

public void mouseReleased(MouseEvent e)

{

showStatus("Mouse released at current position");

}

public void mouseClicked(MouseEvent e)

{

showStatus("Mouse clicked in applet area for"+" "+e.getClickCount()+" no of times");

}

}

**The MouseMotionListener Interface**

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

void mouseDragged(MouseEvent *me*)

void mouseMoved(MouseEvent *me*)

**Example:**

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

/\*<applet code=mousemotiondemo width=400 height=400>

</applet>\*/

public class mousemotiondemo extends Applet implements MouseMotionListener

{

public void init()

{

addMouseMotionListener(this);

}

public void mouseDragged(MouseEvent e)

{

showStatus("Mouse dragged in applet area");

}

public void mouseMoved(MouseEvent e)

{

showStatus("Mouse Moved in applet area");

}

}

**The MouseWheelListener Interface**

This interface defines the **mouseWheelMoved( )** method that is invoked when the mousewheel is moved. Its general form is shown here:

void mouseWheelMoved(MouseWheelEvent *mwe*)

**Example:**

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

/\*<applet code=mousewheeldemo width=400 height=400>

</applet>\*/

public class mousewheeldemo extends Applet implements MouseWheelListener

{

public void init()

{

addMouseWheelListener(this);

}

public void mouseWheelMoved(MouseWheelEvent e)

{

showStatus("Mouse wheel moved in applet area");

}

}

**The TextListener Interface**

This interface defines the **textValueChanged( )** 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*)

**Example:**

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

/\*<applet code=textevdemo width=400 height=400>

</applet>\*/

public class textevdemo extends Applet implements TextListener

{

TextField f=new TextField("Hello");

public void init()

{

add(f);

f.addTextListener(this);

}

public void textValueChanged(TextEvent e)

{

showStatus("you are changeing text from textfield");

}

}

**The WindowFocusListener Interface**

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

void windowGainedFocus(WindowEvent *we*)

void windowLostFocus(WindowEvent *we*)

**Example:**

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

public class windowfocusdemo extends Frame implements WindowFocusListener

{

String msg=" ";

windowfocusdemo(String title)

{

super(title);

setSize(400,400);

setVisible(true);

addWindowFocusListener(this);

}

public void windowGainedFocus(WindowEvent e)

{

msg="Frame got the focus";

repaint();

}

public void windowLostFocus(WindowEvent e)

{

msg="Frame lost focus";

repaint();

}

public void paint(Graphics g)

{

g.drawString(msg,100,100);

}

public static void main(String ar[])

{

windowfocusdemo d=new windowfocusdemo("WindowEvent demo for focus");

}

}

**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 windowis 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*)

**Example:**

import java.awt.\*;

import java.applet.\*;

import java.awt.event.\*;

public class windowevent extends Frame implements WindowListener

{

String msg=" ";

windowevent(String title)

{

super(title);

setSize(400,400);

setVisible(true);

addWindowListener(this);

}

public void windowActivated(WindowEvent e)

{

msg="Activated";

repaint();

}

public void windowDeactivated(WindowEvent e)

{

msg="Deactivated";

repaint();

}

public void windowOpened(WindowEvent e)

{

}

public void windowClosed(WindowEvent e)

{

msg="Closed";

repaint();

}

public void windowClosing(WindowEvent e)

{

msg="Requested window to be closed";

repaint();

}

public void windowIconified(WindowEvent e)

{

msg="Iconified";

repaint();

}

public void windowDeiconified(WindowEvent e)

{

msg="Deiconified";

repaint();

}

public void paint(Graphics g)

{

g.drawString(msg,100,100);

}

public static void main(String ar[])

{

windowevent d=new windowevent("WindowEvent demo");

}

}

**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 toreceive 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( )**. The signatures of these empty methods are exactly as defined in the

**MouseMotionListener** interface. If you were interested in only mouse drag events, thenyou could simply extend **MouseMotionAdapter** and implement **mouseDragged( )**. Theempty implementation of **mouseMoved( )** would handle the mouse motion events foryou.

Following Table lists the different adapter classes in **java.awt.event** and notes the interfacethat each implements.

**Listener Interfaces Implemented by Adapter Classes**

**Adapter Class Listener Interface**

ComponentAdapter ComponentListener

ContainerAdapter ContainerListener

FocusAdapter FocusListener

KeyAdapter KeyListener

MouseAdapter M ouseListener

MouseMotionAdapter MouseMotionListener

WindowAdapter W indowListener

**Example:**

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*<applet code=keyadademo width=400 height=400>

</applet>\*/

public class keyadademo extends Applet

{

String msg=" ";

public void init()

{

addKeyListener(new keyada(this));

}

public void paint(Graphics g)

{

g.drawString(msg,100,100);

}

}

class keyada extends KeyAdapter

{

keyadademo d;

keyada(keyadademo d)

{

this.d=d;

}

public void keyTyped(KeyEvent e)

{

d.msg=e.getKeyChar()+" ";

d.repaint();

//d.showStatus("key typed is"+ e.getKeyChar());

}

}

**Inner Classes**

An *inner class* is a class defined within other class, or even withinan expression. Due to use of the inner class we does not need the reference of parent class object .as it is the inner class all the methods and variables of parent class are directly accessible to the inner class.

Let us see the above program using inner class.

**Example:**

import java.awt.\*;

import java.awt.event.\*;

import java.applet.\*;

/\*<applet code=keyadademo width=400 height=400>

</applet>\*/

public class keyadademo extends Applet

{

String msg=" ";

public void init()

{

addKeyListener(new keyada());

}

public void paint(Graphics g)

{

g.drawString(msg,100,100);

}

class keyada extends KeyAdapter

{

public void keyTyped(KeyEvent e)

{

msg=e.getKeyChar()+" ";

repaint();

//d.showStatus("key typed is"+ e.getKeyChar());

}

}

}

**Anonymous Inner Classes**

An *anonymous* inner class is one that is not assigned a name. This section illustrates

how an anonymous inner class can facilitate the writing of event handlers. Consider the

applet shown in the following listing. As before, its goal is to display the string "Mouse

Pressed" in the status bar of the applet viewer or browser when the mouse is pressed.

// Anonymous inner class demo.

import java.applet.\*;

import java.awt.event.\*;

/\*

<applet code="AnonymousInnerClassDemo" width=200 height=100>

</applet>

\*/

public class AnonymousInnerClassDemo extends Applet {

public void init() {

addMouseListener(new MouseAdapter() {

public void mousePressed(MouseEvent me) {

showStatus("Mouse Pressed");

}

});

}

}

There is one top-level class in this program: **AnonymousInnerClassDemo**. The **init( )**

method calls the **addMouseListener( )** method. Its argument is an expression that

defines and instantiates an anonymous inner class. Let's analyze this expression

carefully.

The syntax **new MouseAdapter( ) { ... }** indicates to the compiler that the code between

the braces defines an anonymous inner class. Furthermore, that class extends

**MouseAdapter**. This new class is not named, but it is automatically instantiated when

this expression is executed.

Because this anonymous inner class is defined within the scope of

**AnonymousInnerClassDemo**, it has access to all of the variables and methods within

the scope of that class. Therefore, it can call the **showStatus( )** method directly.

As just illustrated, both named and anonymous inner classes solve some annoying

problems in a simple yet effective way. They also allow you to create more efficient code.