Unit 3

Event Handling

Two Event Handling Mechanisms

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

The Delegation Event Model

- The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events.
- Its concept is quite simple: a source generates an event and sends it to one or more listeners.
- In this scheme, the listener simply waits until it receives an event. Once an event is received, the listener processes the event and then returns.
- The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events.
- A user interface element is able to “delegate” the

- 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.
- 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.
- An event can be generated as a consequence of a person interacting with the elements in a graphical user interface.
- Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.
- Events may also occur that are not directly caused by interactions with a user interface.
- For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed.

Event Sources

- A source is an object that generates an event. This occurs when the internal state of that object changes in some way.
- Sources may generate more than one type of event.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.
- Each type of event has its own registration method.
- Here is the general form:
  - public void addTypeListener (TypeListener el)
- Here, Type is the name of the event, and el is a reference to the event listener.
- For example, the method that registers a keyboard event listener is called **addKeyListener( ).** The

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

- 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 about specific types of events.
- Second, it must implement methods to receive and process these notifications.
- The methods that receive and process events are defined in a set of interfaces, such as those found in java.awt.event.
- For example, the MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved.
- Any object may receive and process one or both of these events if it provides an implementation

Event Classes

- The classes that represent events are at the core of Java’s event handling mechanism.
- Thus, a discussion of event handling must begin with the event classes.
- The most widely used events at the time of this writing are those defined by the AWT and those defined by Swing.
- This chapter focuses on the AWT events. (Most of these events also apply to Swing.)
- Several Swing-specific events are described later, when Swing is covered.

The KeyEvent Class

- A KeyEvent is generated when keyboard input occurs.
- There are three types of key events, which are identified by these integer constants:
- **KEY_PRESSED, KEY_RELEASED**, and **KEY_TYPED**.
- The first two events are generated when any key is pressed or released.
- The last event occurs only when a character is generated.
- Remember, not all keypresses result in characters. For example, pressing shift does not generate a character.
- There are many other integer constants that are defined by KeyEvent.
- For example, VK_0 through VK_9 and VK_A through VK_Z define the ASCII equivalents of the numbers and letters.
- Here are some others:

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

- 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 is specified by type.
- The system time at which the key was pressed is passed in when.
- The modifiers argument indicates which modifiers were pressed when this key event occurred.
- The virtual key code, such as **VK_UP**, **VK_A**, and so forth, is passed in code.
- The character equivalent (if one exists) is passed in ch. If no valid character exists, then ch contains **CHAR_UNDEFINED**. For **KEY_TYPED** events, code will contain **VK_UNDEFINED.**
- The **KeyEvent** class defines several methods, but

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

```
import java.awt.*;
import java.awt.event.*;
public class KeyEventDemo {

    private Frame f;
    private Label h;
    private Label s;
    private Panel p;
    private TextField t;

    public KeyEventDemo() {
        f = new Frame("Java AWT Examples");
        f.setSize(400, 400);
        f.setLayout(new GridLayout(3, 1));
```

```
        h = new Label("Key Event Demo Program", Label.CENTER);
        s = new Label("", Label.CENTER);
        p = new Panel();
         p.setLayout(new FlowLayout());

         f.add(h);
         f.add(p);
         f.add(s);

         t = new TextField(10);

         t.addKeyListener(new KeyListener() {
             public void keyTyped(KeyEvent e) {
}
```

```
 public void keyPressed(KeyEvent e) {
        s.setText("Entered text: " + t.getText());
           }
 public void keyReleased(KeyEvent e) {
       s.setText(Character.toString(e.getKeyChar()));
}
       });
       p.add(t);
       f.setLocationRelativeTo(null);
       f.setVisible(true);
    }
    public static void main(String[] args) {
       new KeyEventDemo();
    }
}
```

The MouseEvent Class

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

MOUSE_CLICKED	The user clicked the mouse.
MOUSE_DRAGGED	The user dragged the mouse.
MOUSE_ENTERED	The mouse entered a component.
MOUSE_EXITED	The mouse exited from a component.
MOUSE_MOVED	The mouse moved.

- MouseEvent is a subclass of InputEvent. Here is one of its constructors:
- **MouseEvent(Component src, int type, long when,int modifier, int x, int y, int clickCount, boolean triggersPopup)**
- Here, src is a reference to the component that generated the event.
- The type of the event is specified by type.
- The system time at which the mouse event occurred is passed in when.
- The modifiers argument indicates which modifiers were pressed when a mouse event occurred.
- The coordinates of the mouse are passed in x and y.
- The click count is passed in clickCount.
- The triggersPopup flag indicates if this event

- Two commonly used methods in this class are **getX( )** and **getY( )**.
- These return the X and Y coordinates of the mouse within the component when the event occurred.
- Their forms are shown here:
  - int getX( )
  - int getY( )
- Alternatively, you can use the **getPoint( )** method to obtain the coordinates of the mouse. It is shown here:
  - Point getPoint( )
- It returns a Point object that contains the X,Y coordinates in its integer members: x and y.

- The **translatePoint( )** method changes the location of the event. Its form is shown here:
  - void translatePoint(int x, int y)
- Here, the arguments x and y are added to the coordinates of the event.
- The **getClickCount( )** method obtains the number of mouse clicks for this event. Its signature is shown here:
  - int getClickCount( )
- The **isPopupTrigger( )** method tests if this event causes a pop-up menu to appear on this platform. Its form is shown here:
  - boolean isPopupTrigger( )
- Also available is the **getButton( )** method, shown here:
  - int getButton( )
- It returns a value that represents the button that caused the event. For most cases, the return value will b

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

- The NOBUTTON value indicates that no button was pressed or released.
- Also available are three methods that obtain the coordinates of the mouse relative to the screen rather than the component. They are shown here:
  - Point getLocationOnScreen( )
  - int getXOnScreen( )
  - int getYOnScreen( )
- The **getLocationOnScreen( )** method returns a **Point** object that contains both the X and Y coordinate. The other two methods return the indicated coordinate.

The TextEvent Class

- Instances of this class describe text events. These are generated by text fields and text areas when characters are entered by a user or program.
- **TextEvent** defines the integer constant **TEXT_VALUE_CHANGED**.
- The one constructor for this class is shown here:
  - TextEvent(Object src, int type)
- Here, src is a reference to the object that generated this event.
- The type of the event is specified by type.
- The TextEvent object does not include the characters currently in the text component that generated the event. Instead, your program must use other methods associated with the text component to retrieve that information.

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

public class TextEventHandling {

    private Frame f;
    private Label hl,sl;
    private Panel p;
    private TextField t;

    public TextEventHandling() {
        f = new Frame("Java TextEvent Handling Examples");
        f.setSize(400, 400);
        f.setLayout(new GridLayout(3, 1));
```

```
hl = new Label("TextListener in Action", Label.CENTER);
sl = new Label("", Label.CENTER);

p = new Panel();
p.setLayout(new FlowLayout());

f.add(hl);
f.add(p);
f.add(sl);
f.setVisible(true);

t = new TextField(10);
```

```
 t.addTextListener(new TextListener() {
            @Override
            public void textValueChanged(TextEvent e) {
                sl.setText("Entered text: " + t.getText());
            }
        });
        p.add(t);
        f.setVisible(true);
    }

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

The WindowEvent Class

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

WINDOW_ACTIVATED	The window was activated.
WINDOW_CLOSED	The window has been closed.
WINDOW_CLOSING	The user requested that the window be closed.
WINDOW_DEACTIVATED	The window was deactivated.
WINDOW_DEICONIFIED	The window was deiconified.
WINDOW_ICONIFIED	The window was iconified.

- WindowEvent is a subclass of ComponentEvent. It defines several constructors. The first is
  - WindowEvent(Window src, int type)
- Here, src is a reference to the object that generated this event. The type of the event is type. The next three constructors offer more detailed control:
- WindowEvent(Window src, int type, Window other)
- WindowEvent(Window src, int type, int fromState, int toState)
- WindowEvent(Window src, int type, Window other, int fromState, int toState)
- Here, other specifies the opposite window when a focus or activation event occurs. The fromState specifies the prior state of the window, and toState specifies the new state that the window will have

- A commonly used method in this class is **getWindow( )**. It returns the Window object that generated the event. Its general form is shown here:
  - Window getWindow( )
- WindowEvent also defines methods that return the opposite window (when a focus or activation event has occurred), the previous window state, and the current window state. These methods are shown here:
  - Window getOppositeWindow( )
  - int getOldState( )
  - int getNewState( )

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.
- Table below lists several commonly used listener interfaces and provides a brief description of the methods that they define.
- The following sections examine the specific methods that are contained in each interface.

Interface	Description
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.
InputMethodListener	The listener interface for receiving input method events.
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.
TextListener	The listener interface for receiving text events.
WindowFocusListener	The listener interface for receiving WindowEvents, including WINDOW_GAINED_FOCUS and WINDOW_LOST_FOCUS events.

The ActionListener Interface

- The listener interface for receiving action events.
- The class that is interested in processing an action event implements this interface, and the object created with that class is registered with a component, using the component's addActionListener method.
- When the action event occurs, that object's actionPerformed method is invoked.
- This interface defines the **actionPerformed( )** method that is invoked when an action event occurs. Its general form is shown here:
- void actionPerformed(ActionEvent ae)

The ContainerListener Interface

- The listener interface for receiving container events.
- The class that is interested in processing a container event either implements this interface (and all the methods it contains) or extends the abstract ContainerAdapter class (overriding only the methods of interest).
- The listener object created from that class is then registered with a component using the component's addContainerListener method.
- 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:

The KeyListener Interface

- The listener interface for receiving keyboard events (keystrokes).
- The class that is interested in processing a keyboard event either implements this interface (and all the methods it contains) or extends the abstract KeyAdapter class (overriding only the methods of interest).
- 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,

Contd..

- 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

- The listener interface for receiving "interesting" mouse events (press, release, click, enter, and exit) on a component. (To track mouse moves and mouse drags, use the MouseMotionListener.)
- The class that is interested in processing a mouse event either implements this interface (and all the methods it contains) or extends the abstract MouseAdapter class (overriding only the methods of interest).
- 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.

Contd..

- 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

- The Java MouseMotionListener is notified whenever you move or drag mouse. It is notified against MouseEvent.
- The MouseMotionListener interface is found in java.awt.event package. It has two methods.
  - void mouseDragged(MouseEvent e);
  - void mouseMoved(MouseEvent e);

The MouseWheelListener

- The listener interface for receiving mouse wheel events on a component.
- The class that is interested in processing a mouse wheel event implements this interface
- It has only one method
  - **void** mouseWheelMoved(MouseWheelEvent e)

The WindowListener Interface

- The listener interface for receiving window events.
- The class that is interested in processing a window event either implements this interface (and all the methods it contains) or extends the abstract WindowAdapter class (overriding only the methods of interest).
- 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

Contd..

- 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)

The WindowFocusListener Interface

- The listener interface for receiving WindowEvents, including WINDOW_GAINED_FOCUS and WINDOW_LOST_FOCUS events.
- The class that is interested in processing a WindowEvent either implements this interface (and all the methods it contains) or extends the abstract WindowAdapter class.
- It has two methods
  - void windowGainedFocus(WindowEvent e)
  - void windowLostFocus(WindowEvent e)

The WindowStateListener Interface

- The listener interface for receiving window state events.
- It has only one method

The FocusListener Interface

- The listener interface for receiving keyboard focus events on a component.
- The class that is interested in processing a focus event either implements this interface or extends the abstract FocusAdapter class
- It has two methods
  - void focusGained(FocusEvent e)Invoked when a component gains the keyboard focus.
  - void focusLost(FocusEvent e)Invoked when a component loses the keyboard focus.

The ItemListener Interface

- The listener interface for receiving item events. The class that is interested in processing an item event implements this interface.
  - void itemStateChanged(ItemEvent e)Invoked when an item has been selected or deselected by the user.

The AdjustmentListener Interface

- The listener interface for receiving adjustment events.
  - void adjustmentValueChanged(AdjustmentEvent e)Invoked when the value of the adjustable has changed.

Using the Delegation Event Model

- Using the delegation event model is actually quite easy. Just follow these two steps:
- Implement the appropriate interface in the listener so that it can receive the type of event desired.
- Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.
- Remember that a source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.
- To see how the delegation model works in practice, we will look at examples that handle two commonly used event generators: the mouse and keyboard.

Handling Mouse Events

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

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

public class MouseEventHandlingDemo {

    private Frame f;
    private Label hl, ml, sl;
    private Panel p;

    public MouseEventHandlingDemo() {
        f = new Frame("Java MouseEvent Examples");
        f.setSize(400, 400);
        f.setLayout(new GridLayout(3, 1));
```

```
hl = new Label("MouseListener", Label.CENTER);
sl = new Label("Result", Label.CENTER);
ml = new Label("Interact here to see event");
p = new Panel();
p.setLayout(new FlowLayout());
f.add(hl);
f.add(p);
f.add(sl);

ml.setAlignment(Label.CENTER);
```

```
        ml.addMouseListener(new MouseListener() {
            public void mouseClicked(MouseEvent e) {
                sl.setText("Mouse Clicked: (" + e.getX() + ", " +
e.getY() + ")");
            }
            public void mousePressed(MouseEvent e) {
            }
            public void mouseReleased(MouseEvent e) {
            }
public void mouseEntered(MouseEvent e) {
            }
            public void mouseExited(MouseEvent e) {
            }
        });
```

```
p.add(ml);
        f.add(p);
        f.setVisible(true);
    }

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

Program 2

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

public class MouseEventDemo extends Frame implements MouseListener
{
    int x = 0, y = 0;
    String strEvent = "";
    public MouseEventDemo(String title) {
        super(title);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                System.exit(0);
            }
        });
        addMouseListener(this);
        setSize(300, 300);
        setVisible(true);
    }
```

```
public void mouseClicked(MouseEvent e) {
        strEvent = "MouseClicked";
        x = e.getX();
        y = e.getY();
        repaint();
    }

    public void mousePressed(MouseEvent e) {
        strEvent = "MousePressed";
        x = e.getX();
        y = e.getY();
        repaint();

    }

    public void mouseReleased(MouseEvent e) {
        strEvent = "MouseReleased";
        x = e.getX();
        y = e.getY();
        repaint();

    }
```

```
public void mouseEntered(MouseEvent e) {
        strEvent = "MouseEntered";
        x = e.getX();
        y = e.getY();
        repaint();

    }
    public void mouseExited(MouseEvent e) {
        strEvent = "MouseExited";
        x = e.getX();
        y = e.getY();
        repaint();
    }
    public void paint(Graphics g) {
        g.drawString(strEvent + " at " + x + "," + y, 50, 50);
    }
    public static void main(String[] args) {
        new MouseEventDemo("Window With Mouse Events Example");
    }

}
```

Handling Keyboard Events

- To handle keyboard events, you use the same general architecture as that shown in the mouse event example in the preceding section. The difference, of course, is that you will be implementing the **KeyListener** interface.
- When a key is pressed, a **KEY_PRESSED** event is generated. This results in a call to the **keyPressed( )** event handler.
- When the key is released, a **KEY_RELEASED** event is generated and the **keyReleased( )** handler is executed.
- If a character is generated by the keystroke, then a KEY_TYPED event is sent and the keyTyped( ) handler is invoked.
- Thus, each time the user presses a key, at least two and often three events are generated. If all you care about are actual characters, then you can ignore the information passed by the key press and release events.
- However, if your program needs to handle special keys, such as the arrow or function keys, then it must watch for them through the **keyPressed( )** handler.

Window Event Handling

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

public class WindowEventHandling {

    private Frame mf,af;
    private Label hl;
    private Label sl;
    private Panel p;

    public WindowEventHandling() {
        mf = new Frame("Java AWT Examples");
        mf.setSize(400, 400);
        mf.setLayout(new GridLayout(3, 1));
```

```
hl = new Label();
hl.setAlignment(Label.CENTER);
sl = new Label();
sl.setAlignment(Label.CENTER);
sl.setSize(350, 100);

p = new Panel();
p.setLayout(new FlowLayout());

mf.add(hl);
mf.add(p);
mf.add(sl);
mf.setVisible(true);
hl.setText("Listener in action: WindowListener");
```

```
        af = new Frame("WindowListener Demo");
        af.setSize(300, 200);;
        af.addWindowListener(new
CustomWindowListener());

        Label msgLabel = new Label("Welcome to
tutorialspoint.");
        msgLabel.setAlignment(Label.CENTER);
        msgLabel.setSize(100, 100);
        af.add(msgLabel);
        af.setVisible(true);

    }
```

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

    class CustomWindowListener implements
WindowListener {
        public void windowOpened(WindowEvent e) {
            sl.setText("Window Opened");
        }
        public void windowClosing(WindowEvent e) {
          af.dispose();
        }
        public void windowClosed(WindowEvent e) {
            sl.setText("Window closed");
```

```
 public void windowIconified(WindowEvent e) {
          sl.setText("Window iconified");
}

       public void windowDeiconified(WindowEvent e) {
          sl.setText("Window deiconified");
       }

       public void windowActivated(WindowEvent e) {
          sl.setText("Window activated");
       }

       public void windowDeactivated(WindowEvent e) {
          sl.setText("Window deactivated");
       }
   }
}
```

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

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

```
import java.awt.*;
import java.awt.event.*;
public class KeyAdapterDemo extends KeyAdapter{
    private Frame f;
    private Label headerLabel;
    private Label statusLabel;
    private Panel panel;
    private TextField textField;
    public KeyAdapterDemo() {
        f = new Frame("Java AWT Examples");
        f.setSize(400, 400);
        f.setLayout(new GridLayout(3, 1));
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent
windowEvent) {
                System.exit(0);
            }
        });
```

```
        headerLabel = new Label("Key Event Program",
Label.CENTER);
        statusLabel = new Label("", Label.CENTER);
        panel = new Panel();
        panel.setLayout(new FlowLayout());
        f.add(headerLabel);
        f.add(panel);
        f.add(statusLabel);
        TextField textField = new TextField(10);
        textField.addKeyListener(this);

        panel.add(textField);
        f.setLocationRelativeTo(null);
        f.setVisible(true);
    }
```

```
@Override
    public void keyPressed(KeyEvent e){
        System.out.println("hello");
      //
       }

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

Inner Classes

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

public class InnerClassEventHandling {

    private Frame f;
    private Label hl,sl;
    private Panel p;
    private TextField t;

    public InnerClassEventHandling() {
        f = new Frame("Java AWT Examples");
        f.setSize(400, 400);
        f.setLayout(new GridLayout(3, 1));
        hl = new Label();
        hl.setAlignment(Label.CENTER);
        sl = new Label();
        sl.setAlignment(Label.CENTER);
```

```
    sl.setSize(350, 100);
    p = new Panel();
    p.setLayout(new FlowLayout());

    f.add(hl);
    f.add(p);
    f.add(sl);
    f.setVisible(true);
    hl.setText("Listener in action: KeyListener");

    t = new TextField(10);
    t.addKeyListener(new CustomKeyListener());

    p.add(t);
    f.setVisible(true);
}
```

```
 public static void main(String[] args) {
        InnerClassEventHandling a = new InnerClassEventHandling();
    }
    class CustomKeyListener implements KeyListener {

        public void keyTyped(KeyEvent e) {
        }

        public void keyPressed(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_ENTER) {
                sl.setText("Entered text: " + t.getText());
            }
        }

        public void keyReleased(KeyEvent e) {
        }
    }
}
```

Container Event

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

public class EventDemo {
    public EventDemo() {
        Frame f = new Frame();
        f.setLayout(new GridLayout(3, 1));
        Button b1 = new Button("Click Here");
        Button b2 = new Button("Again Click Here");
        f.setSize(200, 200);
        f.add(b1);
```

```
        f.addContainerListener(new ContainerListener() {
        public void componentAdded(ContainerEvent e) {
            System.out.println("COmponent added");
         }
        @Override
         public void componentRemoved(ContainerEvent
e) {
            System.out.println("Component removed");
         }
      });
b1.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
            f.add(b2);
            f.setVisible(true);
         }
       });
```

```
b2.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                f.remove(b2);
                f.setVisible(true);
            }
        });
        f.setVisible(true);
    }
    public static void main(String[] args) {
        new EventDemo();
    }
}
```