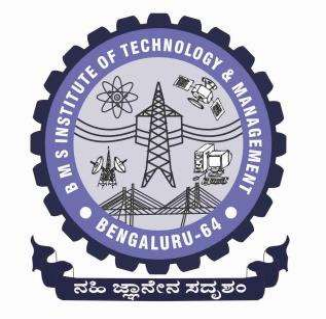
B M S INSTITUTE OF TECHNOLOGY & MANAGEMENT

YELAHANKA, BENGALURU – 560064.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

MODULE -5 NOTES OF

OBJECT ORIENTED CONCEPTS -18CS45

[As per Choice Based Credit System (CBCS) scheme]

(Effective from the academic year 2018 -2019)

SEMESTER – IV

Prepared by,

Mr. Muneshwara M S

Asst. Prof, Dept. of CSE

VISION AND MISSION OF THE CS&E DEPARTMENT

Vision

To develop technical professionals acquainted with recent trends and technologies of computer science to serve as valuable resource for the nation/society.

Mission:

Facilitating and exposing the students to various learning opportunities through dedicated academic teaching, guidance and monitoring.

VISION AND MISSION OF THE INSTITUTE

Vision

To emerge as one of the finest technical institutions of higher learning, to develop engineering professionals who are technically competent, ethical and environment friendly for betterment of the society.

Mission

Accomplish stimulating learning environment through high quality academic instruction, innovation and industry-institute interface

MODULE 5 EVENT HANDLING AND SWINGS

Two event handling mechanisms; The delegation event model; Event classes; Sources of events; Event listener interfaces; Using the delegation event model; Adapter classes; Inner classes. **Swings:** Swings:

The origins of Swing; Two key Swing features; Components and Containers; The Swing Packages; A simple Swing Application; Create a Swing Applet; Jlabel and ImageIcon; JTextField;The Swing Buttons; JTabbedpane; JScrollPane; JList; JComboBox; JTable.

Text book 2: Ch 22: Ch: 29 Ch: 30, RBT: L1, L2, L3

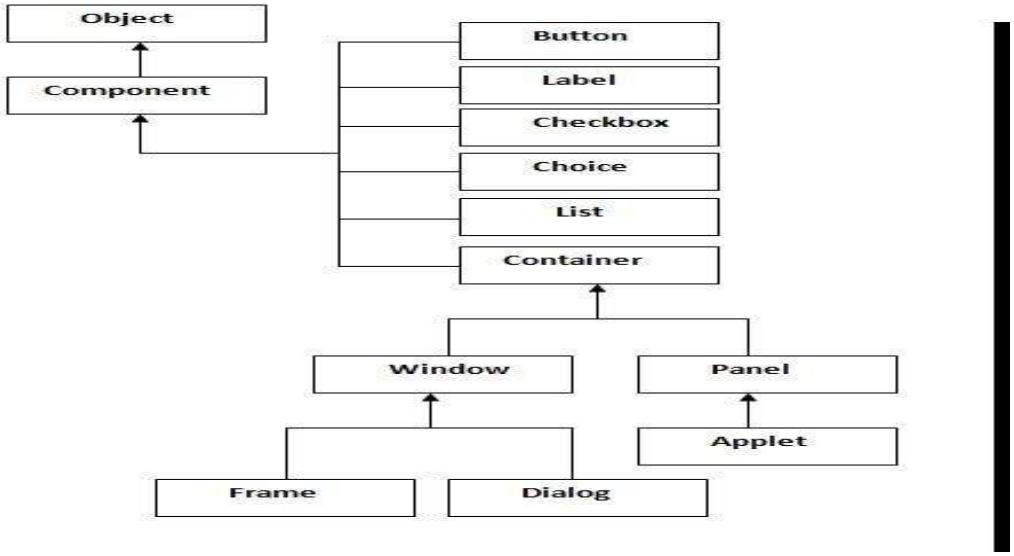
EVENT HANDLING

AWT

Java AWT (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.

- Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system.
- The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.
- In AWT components, (except Panel and Label) generates events when interacted by the user like clicking over a button or pressing enter key in a text field etc. Listeners handle the events.

JAVA AWT HIERARCHY



COMPONENTS OF EVENT HANDLING

Event handling has three main components

1. **Events :** An event is a change of state of an object. *Ex: Button presses, Text field is changed*
2. **Events Source :** Event source is an object that generates an event. *Ex : Button, Checkbox, List, menu Item, window*
3. **Event Listeners :** A listener is an object that listens to the event. A listener gets notified when an event occurs.

Special Group of interfaces

Ex: AdjustmentListener handles the events for scrollbar

ActionListener(Button)

TWO TYPES OF EVENT HANDLING MECHANISMS

The way in which events are handled by an applet changed significantly between the original version of Java(1.0) and modern version of Java, beginning with version 1.1 . The 1.0 method or event handling mechanism is still support the old 1.0 event model have been deprecated.

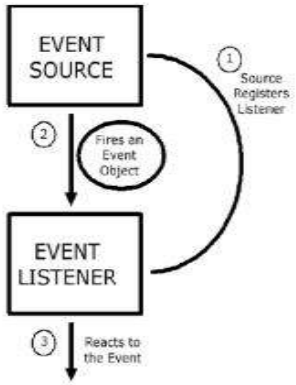
The modern approach is the way that events should be handled by all new programs, including those written for Java2. *Delegation Event Model.*

1. THE DELEGATION EVENT MODEL

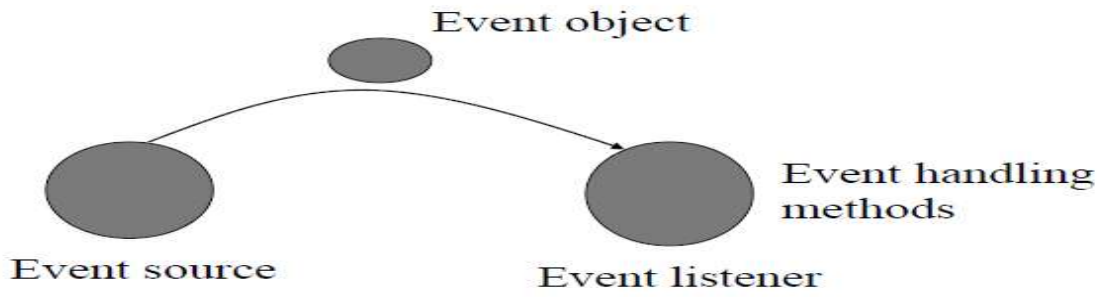
Defines standard and consistent mechanisms to generate and process events.

Listeners must register with a source in order to receive an event notification.

- 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 reacts to the event.



EVENT HANDLING MODEL OF AWT



EXAMPLE

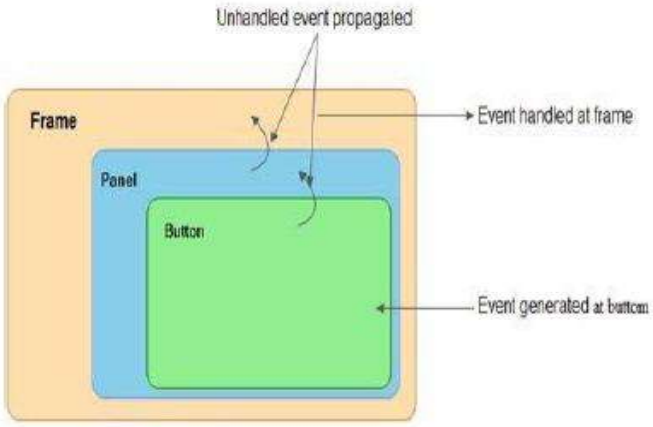
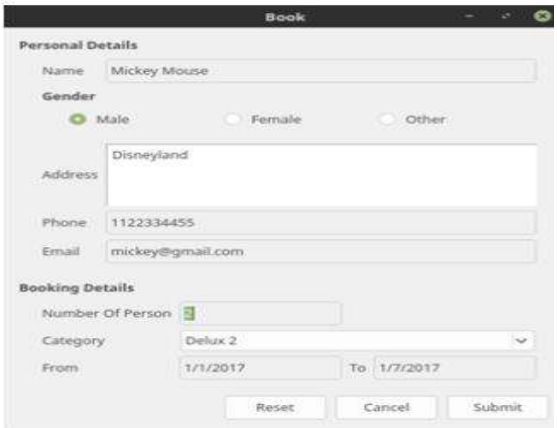


Figure Java 1.0 Event Handling Mechanism

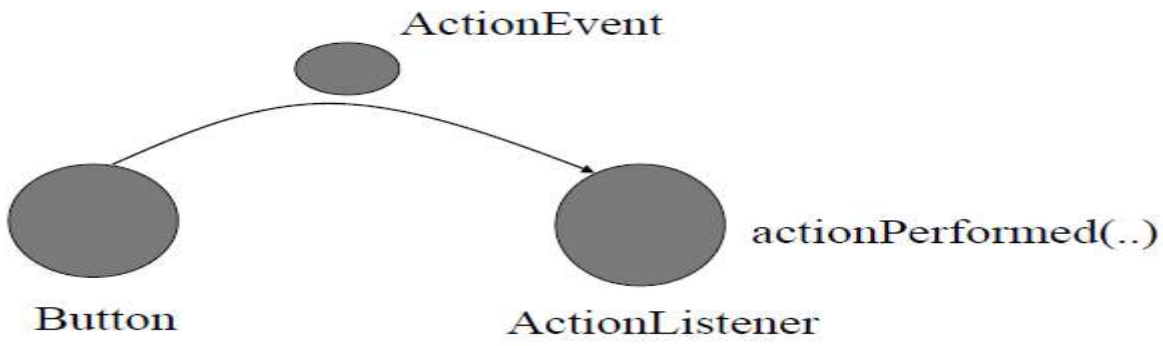


IMPORT

To implement event listener we need to import package : **Import java.awt.event.*;**

And event source in : **Import java.awt.*;**

ACTION EVENTS ON BUTTONS



ADVANTAGES OF EVENT DELEGATION MODEL

- Notifications are sent only to listeners that want to receive them which is more efficient way to handle event.
- 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.
- Accelerates the performance of the application which has multiple events.

DELEGATION EVENT MODEL

It has four main components/classes

1) Event Sources

2) Event classes

3) Event Listeners

4) Event Adapters

1. EVENT SOURCES

- A source is an object that generates an event.
- Event sources are components, subclasses of java.awt.Component, capable to generate events. The event source can be a button, TextField or a Frame etc.
- A source must register listeners in order for the listeners to receive notifications about a specific type of event.

2. EVENT CLASSES

The classes that are responsible for handling events in event handling mechanism are event classes.

- Java event classes are present in
- java.util and java.awt.event
    - EventObject is a superclass of all events. [java.util]
    - AWTEvent is a superclass of all AWT events that are handled by the delegation event model. [java.awt.event]

3. EVENT LISTENERS

- A listener is an object that is notified when an event occurs.
- It has two major requirements.
- First, it must have been registered with one or more sources to receive notifications about specific types of events.
- Second, it must implement methods to receive and process these notifications.

EVENT CLASSES

Event Classes	Description	Listener Interface
ActionEvent	generated when button is pressed, menu-item is selected, list-item is double clicked	ActionListener
MouseEvent	generated when mouse is dragged, moved,clicked,pressed or released also when the enters or exit a component	MouseListener
KeyEvent	generated when input is received from keyboard	KeyListener
ItemEvent	generated when check-box or list item is clicked	ItemListener
TextEvent	generated when value of textarea or textfield is changed	TextListener
MouseWheelEvent	generated when mouse wheel is moved	MouseWheelListener
WindowEvent	generated when window is activated, deactivated, deiconified, iconified, opened or closed	WindowListener
ComponentEvent	generated when component is hidden, moved, resized or set visible	ComponentEventListener
ContainerEvent	generated when component is added or removed from container	ContainerListener
AdjustmentEvent	generated when scroll bar is manipulated	AdjustmentListener
FocusEvent	generated when component gains or loses keyboard focus	FocusListener

EVENT ADAPTERS

Java adapterclasses provide the default implementation of listener interfaces. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.

Adapters are abstract classes for receiving various events. The methods in these classes are empty. These classes exists as convenience for creating listener objects.

SIGNIFICANCE OF ADAPTER CLASS

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.

E.g. Suppose you want to use MouseClicked Event or method from MouseListener, if you do not use adapter class then unnecessarily you have to define all other methods from Mouse

But If you use adapter class then you can only define MouseClicked method and don't worry about other method definition because class provides an empty implementation of all methods in an event

listener interfaceseListener such as MouseReleased, MousePressed etc.

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

EXAMPLE FOR ADAPTER CLASS

```
public class sample
{
   Frame f;
   sample()
   {
      f=new Frame("Window Adapter");
      f.addWindowListener(new WindowAdapter()
      {
         public void windowClosing(WindowEvent e)
         {
            f.dispose();
         }
      }
      );
```

```
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
public static void main(String[] args)
 {
    new sample();

}
}
```

CREATE A FRAME

```
 import java.awt.*;
public class event1
  {
   event1()
   {
    Frame fm=new Frame();
    Label lb = new Label("welcome to java graphics");
    fm.add(lb);
    fm.setSize(300, 300);
    fm.setVisible(true);
   }
   public static void main(String args[])
   {
    event1 ta = new event1();
   }
  }
```

CLOSING FRAME

```
 fm.addWindowListener(new WindowAdapter()
{
public void windowClosing(WindowEvent we)
{
System.exit(0);
}
}
);
```

ADDING LAYOUT

Note: Class should be extended with – extends Frame

FlowLayout fm = new FlowLayout(); setLayout(fm);

ADD BUTTON IN LAYOUT

Button btn=new Button("Hello World"); add(btn

ADD BUTTON

Button btn=new Button("Hello World"); fm.add(btn);

OTHER

setSize(400, 500); //setting size.

setTitle(“Welcome"); //setting title.

setLayout(new FlowLayout()); //set default layout for frame.

TEXT FIELD & CHECKBOX

TextField tf= new TextField();

fm.add(tf);

CheckboxGroup cbg = new CheckboxGroup();

Checkbox checkBox1 = new Checkbox("C++", cbg, true); checkBox1.setBounds(100,100, 50,50);

Checkbox checkBox2 = new Checkbox("Java", cbg, false);

checkBox2.setBounds(100,150, 50,50);

STEPS TO PERFORM EVENT HANDLING

Following steps are required to perform event handling:

- Implement the Listener interface and overrides its methods
- Register the component with the Listener

SYNTAX TO HANDLE THE EVENT

```
class className implements XXXListener
{
  .....
}
addcomponentobject.addXXXListener(this);
.....
```

- override abstract method of given interface and write proper logic public void methodName(XXXEvent e)

```
{
.....
}
.....
}
```

EVENT HANDLING FOR BUTTON COMPONENT

GUI Component (Event Source)	Event class	Listener Interface	Method (abstract method)
Button	ActionEvent	ActionListener	public void actionPerformed(ActionEvent e)

- The ActionEvent is generated when button is clicked or the item of a list is double clicked.
- The object that implements the ActionListener interface gets this ActionEvent when the event occurs.
- ActionEvent(java.lang.Object source, int id, java.lang.String command) Constructs an ActionEvent
- Class methods
- String getActionCommand() [Returns the command string associated with this action.
- int getModifiers() [Returns the modifier keys held down during this action event.]
- long getWhen() [Returns the timestamp of when this event occurred.] java.lang.String paramString() [Returns a parameter string identifying this action event.]

```
import java.awt.*;
import java.awt.event.*;
class A implements ActionListener
{
Frame f;
Button b1,b2,b3,b4;
A()
        {
        f=new Frame();
        f.setSize(500,500);
        f.setLayout(new BorderLayout());
        Panel p=new Panel();
        p.setBackground(Color.yellow);
        b1=new Button("Red");
        b2=new Button("green");
        b3=new Button("Blue");
        b4=new Button("Exit");
        p.add(b1);
        p.add(b2);
        p.add(b3);
        p.add(b4);
        f.add("North",p);
        f.add("North",p);
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        b4.addActionListener(this);
        f.setVisible(true);
        }
public void actionPerformed(ActionEvent e)
{
if(e.getSource().equals(b1))

        {
        f.setBackground(Color.red);
        }
else if(e.getSource().equals(b2))

{
f.setBackground(Color.green);
}
else if(e.getSource().equals(b3)

{
f.setBackground(Color.blue);
}
else if(e.getSource().equals(b4))

{
System.exit(0);
}
```

```
}

class ActionEventEx
{
public static void main(String[] args)
        {
        A a1=new A();
        }
}
```

EVENT HANDLING FOR TEXTFIELD

GUI Component	Event class	Listener Interface	Method (abstract method)
TextField	TextEvent	TextListener	public void textValueChanged(TextEvent e)

```
class AEvent extends Frame implements ActionListener
{
TextField tf;
Button b1,b;
AEvent(){
tf=new TextField();
tf.setBounds(60,50,170,20);
b=new Button("click me");
b.setBounds(100,120,80,30);
b1=new Button("Exit");
b1.setBounds(200,320,80,30);
//register listener
b.addActionListener(this); //passing current instance b1.addActionListener(this);
//add components and set size, layout and visibility
add(b);add(tf); add(b1);
```

```
setSize(300,300);
setVisible(true);
}
public void actionPerformed(ActionEvent e)
{ if(e.getSource().equals(b)) { tf.setText("Welcome");
}
Else if(e.getSource().equals(b1))
{
System.exit(0);
}
public static void main(String args[])
{
new AEvent();
}
}
```

CODE TO PERFORM ADDITION

```
public void actionPerformed(ActionEvent e)
{
String s1=tf1.getText();
String s2=tf2.getText();
int a=Integer.parseInt(s1);
int b=Integer.parseInt(s2);
int c=0;
if(e.getSource()==b1)
        {
        c=a+b;
        }
else if(e.getSource()==b2)
{
c=a-b;
}
String result=String.valueOf(c);
```

```
tf3.setText(result);
}
```

EVENT HANDLING FOR MOUSE

GUI Component	Event class	Listener Interface
Mouse	MouseEvent	MouseListener

Methods:

mousePressed(MouseEvent e): This method will execute whenever mouse button is pressed (not released).

mouseReleased(MouseEvent e): This method will execute whenever mouse button is only released (if already it is pressed).

mouseClicked(MouseEvent e): This method will execute whenever mouse button is both pressed and released.

mouseEntered(MouseEvent e): This method will execute whenever mouse cursor position is placed on specific location or component.

mouseExited(MouseEvent e): This method will execute whenever mouse cursor position is taken back from any location or component.

GUI Component	Event class	Listener Interface	Method (abstract method)
RadioButton, Checkbox, Lis Choice	ItemEvent	ItemListener	public void itemStateChanged(ItemEvent e

Mouse	MouseEvent	MouseListener	Void mousePressed(MouseEvent e) Void mouseReleased(MouseEvent e) Void mouseClicked (MouseEvent e) Void mouseEnetered(MouseEvent e) Void mouseExited(MouseEvent e)
TextField	TextEvent	TextListener	public void textValueChanged(TextEvent e
Button	ActionEvent	ActionListener	public void actionPerformed(ActionEvent e
Scrollbar	AdjustmentEver	AdjustmentListene	public void adjustmentValueChanged(ItemEvent e)
Keyboard	KeyEvent	KeyListener	Void keyPressed(keyEvent k) Void keyReleased(keyEvent k) Void keyTyped(keyEvent k)
Window	WindowEvent	WindowListener	void windowClosing(WindowEvent we)

ADJUSTMENTEVENT

The Class AdjustmentEvent represents adjustment event emitted by Adjustable objects like scroll bar.

Syntax :

- AdjustmentEvent(Adjustable source, int id, int type, int value)
- Constructs an AdjustmentEvent object with the specified Adjustable source, event type, adjustment type, and value.

Methods

1) **Adjustable getAdjustable()** [Returns the Adjustable object where this event originated.]
2) **int getAdjustmentType()** [Returns the type of adjustment which caused the value changed event.]
3) **int getValue()** [Returns the current value in the adjustment event.]
4) **boolean getValueIsAdjusting()** [Returns true if this is one of multiple adjustment events.]
5) **String paramString() [**Returns a string representing the state of this Event.]

PROGRAMS

1. WAP in java to implement all methods available to handle Mouse Events . Change background color for every event.
2. WAP in java to handle Keyboard. Implement the methods for Keyboard handling events. Display the characters
3. pressed, in Textfield . *[Hint : Override KeyTyped() and call e.getKeyChar(); and display it in Text fied]*
4. WAP to create a Registration Form that contains Username, Password, Languages Known. Display it on Frame
5. WAP in java to simulate simple calculator using event handling mechanisms.

MOUSE EVENT

```
import java.awt.*;
import java.awt.event.*;
class MouseListenerExample extends Frame implements MouseListener

{
Label l;
MouseListenerExample()

        {
        addMouseListener(this);
        l=new Label();
        l.setBounds(20,50,100,20);
        add(l);
        setSize(300,300);
        setLayout(null);
        setVisible(true);
        addWindowListener(new WindowAdapter()

                {
                public void windowClosing(WindowEvent we)
                        {System.exit(0);
                        }
                });
        }
public void mouseClicked(MouseEvent e) { l.setText("Mouse Clicked");
}
public void mouseEntered(MouseEvent e) { l.setText("Mouse Entered");
}
public void mouseExited(MouseEvent e) { l.setText("Mouse Exited");
}
public void mousePressed(MouseEvent e) { l.setText("Mouse Pressed");
}
public void mouseReleased(MouseEvent e) { l.setText("Mouse Released");
}
}
public class mouse

{
public static void main(String[] args)

        {
        new MouseListenerExample();
        }
}
```

KEYBOARD EVENT

```
import java.awt.*;
import java.awt.event.*;
class KeyListenerExample extends Frame implements KeyListener

{ Label l;
TextArea area;
KeyListenerExample()

        {
        l=new Label();
        l.setBounds(20,50,100,20);
        area=new TextArea();
        area.setBounds(20,80,300, 300);
        area.addKeyListener(this);
        add(l);add(area);
        setSize(400,400);
        setLayout(null);
        setVisible(true);
        }

public void keyPressed(KeyEvent e)

{
l.setText("Key Pressed");
}
public void keyReleased(KeyEvent e)

{
l.setText("Key Released");
}
public void keyTyped(KeyEvent e)

{
l.setText("Key Typed");
}
}
public class keyboard{
public static void main(String[] args)

{
new KeyListenerExample();
}
}
```

BUTTON & TEXT FIELD

```
import java.awt.*;
import java.awt.event.*;
public class test implements ActionListener
{
Frame fm;
Button b1,b2;
TextField tf;
test()
        {
        fm=new Frame();
        fm.setSize(500, 500); //Creating a frame.
        fm.setBackground(Color.cyan);
        fm.setLayout(new FlowLayout());
        Label lb = new Label("welcome to java graphics");        //Creating a label
        fm.add(lb);                                              //adding label to the frame.
        fm.setVisible(true);
        m.setResizable(true);//set frame visibilty true.
        fm.addWindowListener(new WindowAdapter()

                {
                public void windowClosing(WindowEvent we)
                        {
                        System.exit(0);
                        }
                });
        b1=new Button("Red");
        fm.add(b1);
        b2=new Button("Green");
        fm.add(b2);
        tf= new TextField();
        tf.setBounds(30, 250, 300, 500);
        tf.setSize(350, 30);
        fm.add(tf);
        b1.addActionListener(this);
        b2.addActionListener(this);
        }
public void actionPerformed(ActionEvent e)
{
if(e.getSource().equals(b1))

        {
        fm.setBackground(Color.red);
        tf.setText("Hurray Its red");
        }
else if(e.getSource().equals(b2))

{
fm.setBackground(Color.green);
tf.setText("Hurray Its green");
```

```
}
}
public static void main(String args[])
{
test ta = new test();
}
}
```

Event

Change in the state of an object is known as event i.e. change in state of source. The activities that is carried out between user and application is called an event.

For example,

1. clicking on a button,
2. moving the mouse,
3. entering a character through keyboard,
4. selecting an item from list,
5. scrolling the page are the activities that causes an event to happen

Types of Event

The events can be broadly classified into two categories:

Foreground Events -

Those events which require the direct interaction of user.

They are generated as consequences of a person interacting with the graphical components in Graphical User Interface.

For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page etc.

2. Background Events -

Those events that require the interaction of end user are known as background events.

Example of background events.

Operating system interrupts, hardware or software failure, timer expires

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs.

Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.

Steps involved in event handling

1. The User clicks the button and the event is generated.
2. The object of concerned event class is created automatically and information about the source and the event get populated with in same object.
3. Event object is forwarded to the method of registered listener class.
4. the method is now get executed and returns.

Delegation Event Model

The Delegation Event Model has the following key participants namely:

Source -

- The source is an object on which event occurs.
- Source is responsible for providing information of the occurred event to it's handler.
- Java provide as with classes for source object.

Listener -

- It is also known as event handler.
- Listener is responsible for generating response to an event.
- From java implementation point of view the listener is also an object.

- Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Event and Listener

Changing the state of an object is known as an event.

For example, click on button, dragging mouse etc.

The java.awt.event package provides many event classes and Listener interfaces for event handling.

1. Event Sources

Event sources are components, subclasses of java.awt.Component, capable to generate events. The event source can be a button, TextField or a Frame etc.

2. Event classes

- Almost every event source generates an event and is named by some Java class.
- For example, the event generated by button is known as ActionEvent and that of Checkbox is known as ItemEvent.
- All the events are listed in java.awt.event package.

3. Event Listeners interface

- The events generated by the GUI components are handled by a special group of interfaces known as "listeners".
- Listener is an interface.
- Every component has its own listener, say, AdjustmentListener handles the events of scrollbar.
- Some listeners handle the events of multiple components.

For example, ActionListener handles the events of Button, TextField, List and Menus. Listeners are from java.awt.event package.

4. Event Adapter Classes

- When a listener includes many abstract methods to override, the coding becomes heavy to the programmer.
- For example, to close the frame, you override seven abstract methods of WindowListener, in which, infact you are using only one method.
- To avoid this heavy coding, the designers come with another group of classes known as "adapters".
- Adapters are abstract classes defined in java.awt.event package.
- Every listener that has more than one abstract method has got a corresponding adapter class.

EVENT CLASSES	LISTENER INTERFACES
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Java Event Handling Code

We can put the event handling code into one of the following places:

1. Inner/Within class
2. Outer class
3. Anonymous class

SWINGS

Swing is a part of Java Foundation classes (JFC), the other parts of JFC are java2D and Abstract window toolkit (AWT). AWT, Swing & Java 2D are used for building graphical user interfaces (GUIs) in java. Unlike AWT, Java Swing provides platform-independent and lightweight components.

Main Features of Swing Toolkit

6. Platform Independent
7. Customizable
8. Extensible
9. Configurable
10. **Light Weight** – Swing components are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
11. **Rich Controls** – Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, and table controls.
12. **Highly Customizable** – Swing controls can be customized in a very easy way as visual appearance is independent of internal representation.
13. **Pluggable look-and-feel** – SWING based GUI Application look and feel can be changed at run-time, based on available values.

Difference between AWT and Swing

No	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes,colorchooser,tabbedpane etc.
5)	AWT **doesn't follow MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Origins of Swing

Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.

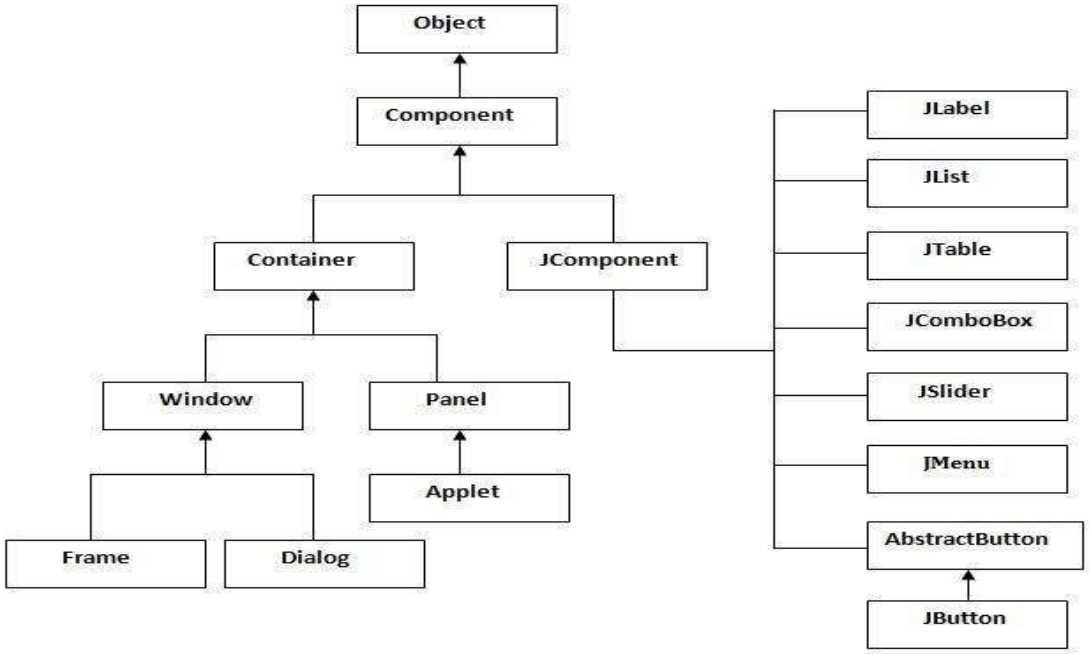
One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or peers. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as **heavyweight**.

The use of native peers led to several problems.

- First, because of variations between operating systems, a component might look, or even act, differently on different platforms, which affected the potential variability for write once, run anywhere.
- Second, the look and feel of each component was fixed and could not be easily changed.
- Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component was always opaque.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc

Two Key Swing Features

- Swing Components Are Lightweight
- Swing Supports a Pluggable Look and Feel

Swing Components Are Lightweight

With very few exceptions, Swing components are **lightweight**. They are written entirely in Java and do not map directly to platform-specific peers. Thus, lightweight components are more efficient and more flexible.

Swing Supports a Pluggable Look and Feel

Swing supports a pluggable look and feel (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. SWING based GUI Application look and feel can be changed at run-time, based on available values.

Pluggable look-and-feels offer several important advantages.

- It is possible to define a look and feel that is consistent across all platforms.
- Conversely, it is possible to create a look and feel that acts like a specific platform. For example, if you know that an application will be running only in a Windows environment, it is possible to specify the Windows look and feel.
- It is also possible to design a custom look and feel.
- Finally, the look and feel can be changed dynamically at run time.

MVC Architecture [Model-View-Controller]

In MVC terminology,

- **Model** corresponds to the state information associated with the component.

  For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked.

- The **view** determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model.
- The **controller** determines how the component reacts to the user.

  For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated.

Swing API architecture follows loosely based MVC architecture in the following manner.

- Model represents component's data.
- View represents visual representation of the component's data.
- Controller takes the input from the user on the view and reflects the changes in Component's data.

Swing component has Model as a separate element, while the View and Controller part are clubbed in the User Interface elements. Because of which, Swing has a pluggable look-and-feel architecture.

Components and Containers

A Swing GUI consists of two key items:

- components and
- containers.

In Java, a component is the basic user interface object and is found in all Java applications.

Components include lists, buttons, panels, and windows.

To use components, you need to place them in a container. A container is a component that holds and manages other components. Containers display components using a layout manager.

Components

Swing components inherit from the javax.Swing.JComponent class.

JPanel : JPanel is Swing's version of AWT class Panel and uses the same default layout, FlowLayout. JPanel is descended directly from JComponent.

JFrame : JFrame is Swing's version of Frame and is descended directly from Frame class.

The component which is added to the Frame, is refered as its Content.

JWindow : This is Swing's version of Window and hass descended directly from Window class. Like Window it uses BorderLayout by default.

JLabel : JLabel descended from Jcomponent, and is used to create text labels.

JButton : JButton class provides the functioning of push button. JButton allows an icon, string or both associated with a button.

JTextField : JTextFields allow editing of a single line of text.

JRadioButton is similar to JCheckbox, except for the default icon for each class. A set of radio buttons can be associated as a group in which only one button at a time can be selected. **JCheckBox** is not a member of a checkbox group. A checkbox can be selected and deselected, and it also displays its current state.

JComboBox is like a drop down box. You can click a drop-down arrow and select an option from a list. For example, when the component has focus, pressing a key that corresponds to the first character in some entry's name selects that entry. A vertical scrollbar is used for longer lists.

JList provides a scrollable set of items from which one or more may be selected. JList can

be populated from an Array or Vector.

JMenuBar An implementation of a menu bar.

JMenuItem An implementation of an item in a menu.

JOptionPane JOptionPane makes it easy to pop up a standard dialog box that prompts users for a value or informs them of something.

JPanel JPanel is a generic lightweight container.

JPasswordField JPasswordField is a lightweight component that allows the editing of a single line of text where the view indicates something was typed, but does not show the original characters.

JPopupMenu :An implementation of a popup menu -- a small window that pops up and displays a series of choices.

Containers

There are two types of containers namely,

- top-level containers and
- lightweight containers

Top-Level Containers

- The first are top-level containers: JFrame, JApplet, JWindow, and JDialog.
- These containers do not inherit **JComponent**.
- They do, however, inherit the AWT classes **Component** and **Container**.
- The top-level containers are heavyweight.

Each top-level container defines a set of **panes**. At the top of the hierarchy is an instance of **JRootPane**.

- **JRootPane** is a special container which extends JComponent and manages the appearance of JApplet and JFrame objects. It contains a fixed set of panes, namely,
  - *glass pane,*

- *content pane, and*
- *layered pane.*

• **Glass pane:** A glass pane is a top-level pane which covers all other panes. By default, it is a transparent instance of JPanel class. It is used to handle the mouse events affecting the entire container.

• **Layered pane:** A layered pane is an instance of JLayeredPane class. It holds a container called the content pane and an optional menu bar.

• **Content pane:** A content pane is a pane which is used to hold the components. All the visual components like buttons, labels are added to content pane. By default, it is an opaque instance of JPanel class and uses border layout. The content pane is accessed via getContentPane () method of JApplet and JFrame classes.

Lightweight Containers

- Lightweight containers lie next to the top-level containers in the containment hierarchy.
- They inherit **JComponent**.
- One of the examples of lightweight container is JPanel.
- As lightweight container can be contained within another container, they can be used to organize and manage groups of related components.

Swing Packages

Some of the packages of swing components that are used most are the following:

• *Javax.swing*
• *javax.swing.event*
• *javax.swing.plaf.basic*
• *javax.swing.table*
• *javax.swing.border*
• *javax.swing.tree*

Packages	Description
javax.swing	Provides a set of "lightweight" (all-Java language) components to the maximum degree possible, work the same on all platforms.
javax.swing.border	Provides classes and interface for drawing specialized borders around a Swing component.
javax.swi ng.colorchooser	Contains classes and interfaces used by the JcolorChooser component.
javax.swing.event	Provides for events fired by Swing components
javax.swing.filechooser	Contains classes and interfaces used by the JfileChooser component.
javax.swing.plaf	Provides one interface and many abstract classes that Swing uses to provide its pluggable look-and-feel capabilities.
javax.swing.plaf.basic	Provides user interface objects built according to the Basic look and feel.
javax.swing.plaf.metal	Provides user interface objects built according to the Java look and feel (once condenamed Metal), which is the default look and feel.
javax.swi ng.plaf.mult	Provides user interface objects that combine two or more look and feels.
javax.swing.table	Provides classes and interfaces for dealing with javax.swing.jtable
javax.swing.text	Provides classes and interfaces that deal with editable and noneditable text components
javax.swing.text.html	Provides the class HTML Editor Kit and supporting classes for creating HTML text editors.
javax.swing.text.rtf	Provides a class RTF Editor Kit for creating Rich- Text-Format text editors.
javax.swing.tree	Provides classes and interfaces for dealing with javax.swing.jtree
javax.swing.undo	Allows developers to provide support for undo/redo in applications such as text editors.

A Simple Swing Application

There are 2 ways of creating Java Swing Programs

1) Creating Swing Application 2) Creating Swing Applet

1)

```
import javax.swing.*;

class SwingDemo
{
SwingDemo()
{
    JFrame jfrm = new JFrame("A Simple Swing Application"); // Create a new frame

    jfrm.setSize(275, 100); // Set the size of a frame

    jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //To close the frame

JLabel jlab = new JLabel(" Swing means powerful GUIs.");

jfrm.add(jlab);

jfrm.setVisible(true);

}

public static void main(String args[]) {

SwingUtilities.invokeLater ( new Runnable() { public void run() {new SwingDemo();

} });

}

}
```

Note:

jfrm.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

After this call executes, closing the window causes the entire application to terminate.

The general form of setDefaultCloseOperation( ) is shown here:

void setDefaultCloseOperation(int what)

The value passed in what determines what happens when the window is closed. There are several other options in addition to JFrame.EXIT_ON_CLOSE. They are shown here:

- DISPOSE_ON_CLOSE
- HIDE_ON_CLOSE
- DO_NOTHING_ON_CLOSE

6) Swing Applet

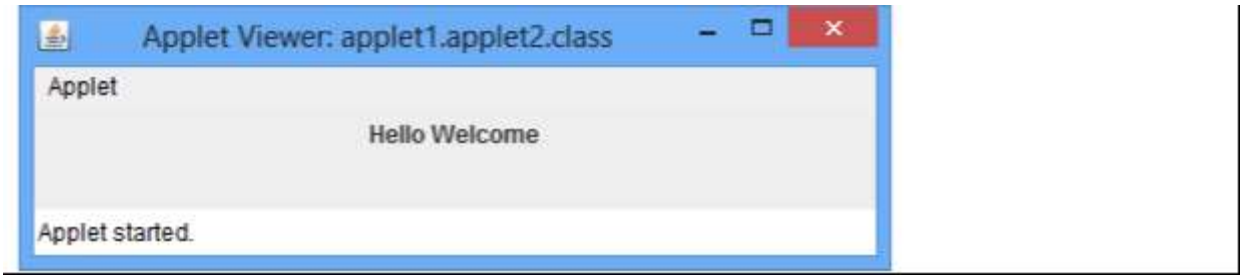
A Swing applet extends JApplet rather than Applet. JApplet is derived from Applet. Thus, JApplet includes all of the functionality found in Applet and adds support for Swing. JApplet is a top-level Swing container, which means that it is not derived from JComponent. Because JApplet is a toplevel container, it includes the various panes described earlier. This means that all components are added to JApplet's content pane in the same way that components are added to JFrame's content pane.

Swing applets use the same four life-cycle methods

- **init( ),**
- **start( ),**
- **stop( ), and**
- **destroy( ).**

```
import java.awt.*;

import javax.swing.*;

/*<APPLET CODE=sample.class WIDTH=510 HEIGHT=210></APPLET>*/

public class sample extends JApplet

{

JLabel jlab;


public void init(){

try

{

SwingUtilities.invokeAndWait(new Runnable(){

public void run()

{

sample();

}

});

}


catch(Exception e){

System.out.println(e);
```

```
}
}
private void sample()
{
jlab=new JLabel("Hello Welcome");

      setSize(300,300);

 setLayout(new FlowLayout());
add(jlab);
}
}
```



Event Handling in Swings

Event: An event is a signal to the program that something has happened. It can be triggered by typing in a text field, selecting an item from the menu etc. The action is initiated outside the scope of the program and it is handled by a piece of code inside the program. Events may also be triggered when timer expires, hardware or software failure occurs, operation completes, counter is increased or decreased by a value etc.

Event handler: The code that performs a task in response to an event. is called event handler.

Event handling: It is process of responding to events that can occur at any time during execution of a program.

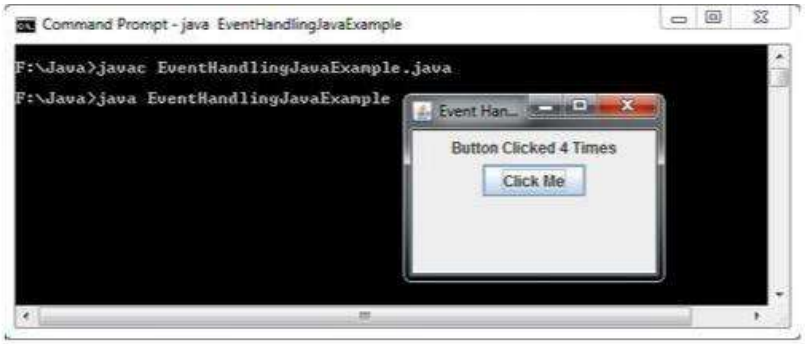
Event Source: It is an object that generates the event(s). Usually the event source is a button or the other component that the user can click but any Swing component can be an event source. The job of the event source is to accept registrations, get events from the user and call the listener's event handling method.

Event Listener: It is an object that watch for (i.e. listen for) events and handles them when they occur. It is basically a consumer that receives events from the source. To sum up, the job of an event listener is to implement the interface, register with the source and provide the eventhandling.

Listener interface: It is an interface which contains methods that the listener must implement and the source of the event invokes when the event occurs.

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

class EventExample extends JFrame implements ActionListener {

   private int count =0;
   JLabel lb;

   EventExample()
   {
    setLayout(new FlowLayout());
    lb = new JLabel("Button Clicked 0 Times");
```

```
        JButton btn=new JButton("Click Me");
        btn.addActionListener(this);
        add(lb);
        add(btn);
      }
      public void actionPerformed(ActionEvent e)
      {
        count++;
        lb.setText("Button Clicked " + count +" Times");
      }
}
  class sample
 {
    public static void main(String args[])
    {
       EventExample frame = new EventExample();
       frame.setTitle("Event Handling Java Example");
       frame.setBounds(200,150,180,150);
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setVisible(true);
    }
 }
```


Command Prompt - java EventHandlingJavaExample
F:\Java>javac EventHandlingJavaExample.java
F:\Java>java EventHandlingJavaExample
Event Han...
Button Clicked 4 Times
Click Me

JLabel and ImageIcon within JApplet

A label is a simple control which is used to display text (non-editable) on the window. Some of the constructors defined by

JLabel class are as follows.

JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	Creates a JLabel instance with the specified text.
JLabel(Icon i)	Creates a JLabel instance with the specified image.
JLabel(String s, Icon i, int horizontalAlignment)	Creates a JLabel instance with the specified text, image, and horizontal alignment

ImageIcon(String filename);	It obtains the image in the file named filename.

```
import java.awt.*;

import javax.swing.*;

/*<APPLET CODE=JavaExampleLabelImageInJApplet.class
WIDTH=510 HEIGHT=210></APPLET>*/

public class sample extends JApplet
```

```
{

  public void init()
  {
    Container c = getContentPane();

    ImageIcon img = new ImageIcon("C:/Users/sri p/workspace/applet1/Koala.jpg");

    JLabel Lbl = new JLabel("Good Evening ",img,
    JLabel.CENTER);
    Lbl.setVerticalTextPosition(JLabel.BOTTOM);
    Lbl.setHorizontalTextPosition(JLabel.CENTER);
    c.add(Lbl);

  }
```


```
}
```

JTextField : Is extended from the JComponent class. This allows us to add a single line of text.

Syntax :

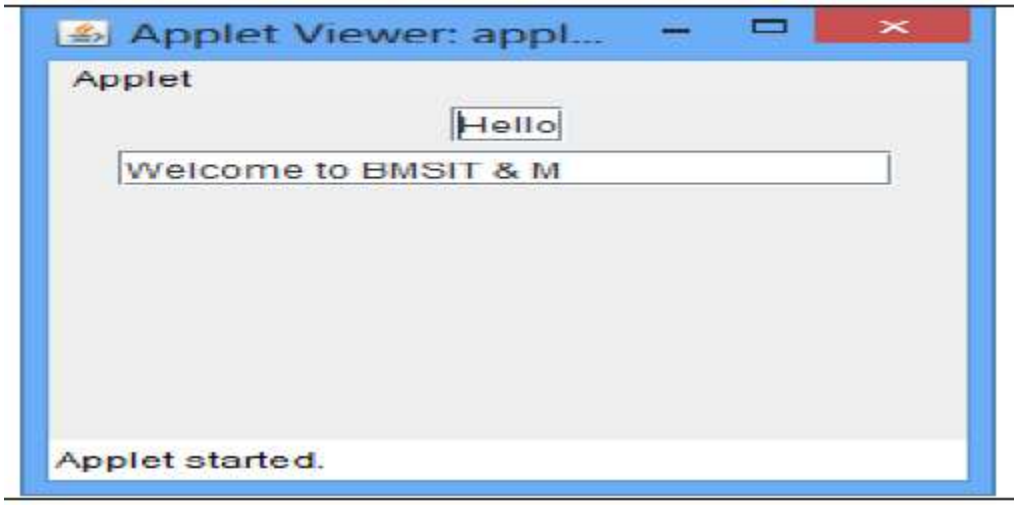
JTextField(int cols)

JTextField(String str, int cols)

JTextField(String str)

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="CreateNewJTextFieldExample" width=200 height=200>
</applet>
*/

public class applet2 extends JApplet{

   public void init(){

        Container c = getContentPane();
        c.setLayout(new FlowLayout());

          JTextField field2 = new JTextField("Hello");
          JTextField field4 = new JTextField("Welcome to BMSIT & M", 20);
```

```
        c.add(field2);
        c.add(field4);

    }
}
```



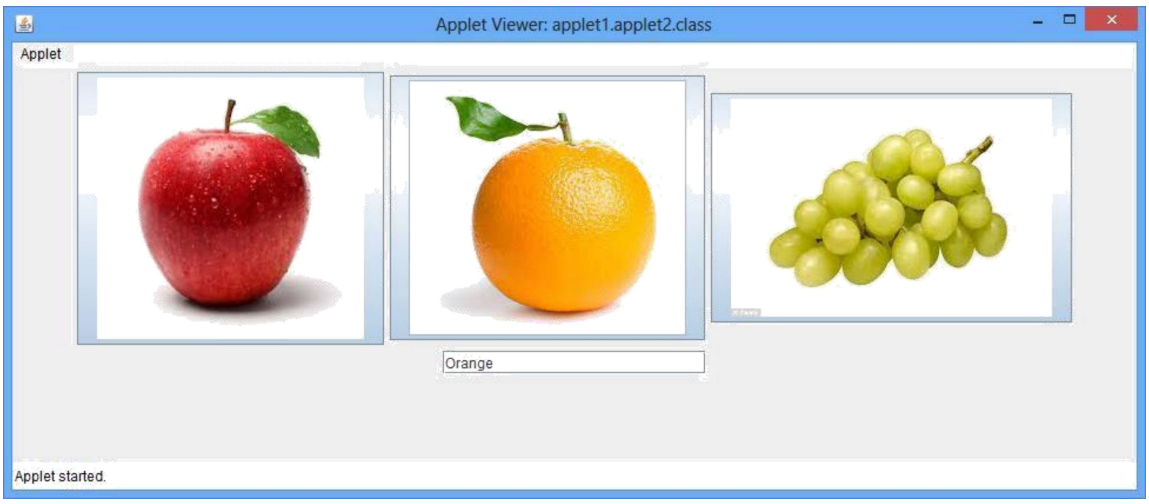
JButton [VERY VERY IMPORTANT TOPIC]

The Buttons component in Swing is similar to the Button component in AWT except that it can contain text, image or both. It can be created by instantiating the **JButton** class. A JButton can display Icon objects.

Method	Description
JButton()	Constructs a button with no text
JButton(String)	Constructs a button with the text entered
JButton(String, Icon)	Constructs abutton with thetext andinformedimage

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

import javax.swing.*;

public class applet2 extends JApplet implements ActionListener
{
 JTextField T;

   public void init()
{
       Container c = getContentPane();
        c.setLayout(new FlowLayout());

           ImageIcon appi = new ImageIcon("C:/Users/sri p/workspace/applet1/apple.jpg");
           JButton btn1 = new JButton(appi);
           btn1.setActionCommand("Apple");
           btn1.addActionListener(this);
           c.add(btn1);

           ImageIcon org = new ImageIcon("C:/Users/sri p/workspace/applet1/orange.jpg");
           JButton btn2 = new JButton(org);
           btn2.setActionCommand("Orange");
```

```
        btn2.addActionListener(this);
        c.add(btn2);

        ImageIcon grp = new ImageIcon("C:/Users/sri p/workspace/applet1/grapes.jpg");
       JButton btn3= new JButton(grp);
        btn3.setActionCommand("Grapes");
        btn3.addActionListener(this);

        c.add(btn3);

        T = new JTextField(100);
        c.add(T);
  }
  public void actionPerformed(ActionEvent e)
  {
  T.setText(e.getActionCommand());
}
  }
```



JTabbedpane

Tabbed pane is a common user interface component that provides an easy access to more than one panel. Each tab is associated with a single component that will be displayed when the tab is selected.

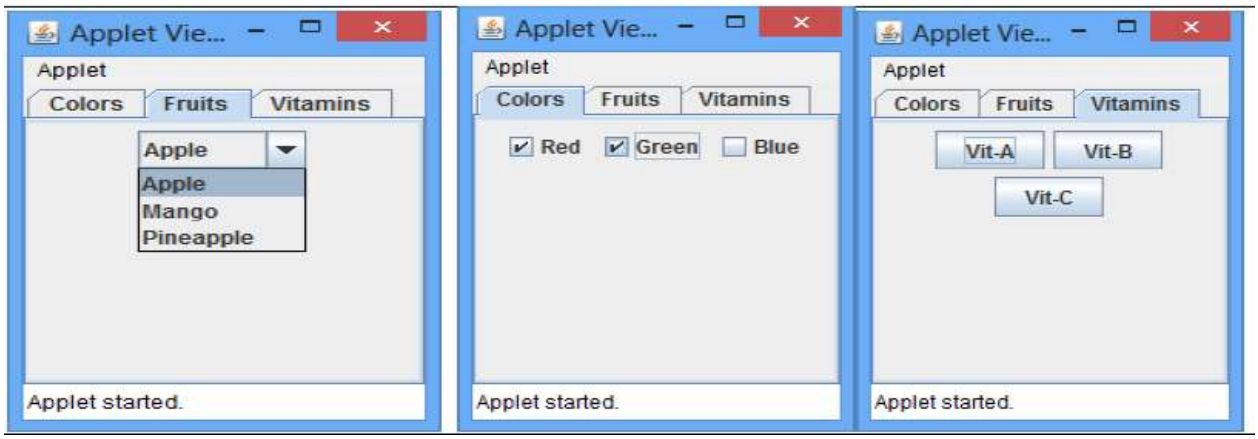
Syntax :

Void addTab(String str, Component comp);

```
import java.awt.*;
import javax.swing.*;

public class JTPDemo extends JApplet
{
  public void init()
  {
    JTabbedPane jt = new JTabbedPane();

    jt.add("Colors", new CPanel());
    jt.add( "Fruits", new FPanel());
    jt.add("Vitamins", new VPanel( ) ) ;
```

```
    getContentPane().add(jt);
  }
}
class CPanel extends JPanel
{
  public CPanel()
  {
    JCheckBox cb1 = new JCheckBox("Red");
    JCheckBox cb2 = new JCheckBox("Green");
    JCheckBox cb3 = new JCheckBox("Blue");
    add(cb1);    add(cb2);    add(cb3) ;
  }
}
class FPanel extends JPanel
{
  public FPanel()
  {
    JComboBox cb = new JComboBox();
    cb.addItem("Apple");
    cb.addItem("Mango");
    cb.addItem("Pineapple");
    add(cb);
  }
}
class VPanel extends JPanel
```

```
{
  public VPanel()
  {
    JButton b1 = new JButton("Vit-A");
    JButton b2 = new JButton("Vit-B");
    JButton b3 = new JButton("Vit-C");
    add(b1);    add(b2);    add(b3);
  }
}
```



JRadioButton

Syntax: ***JRadioButton(String str)***

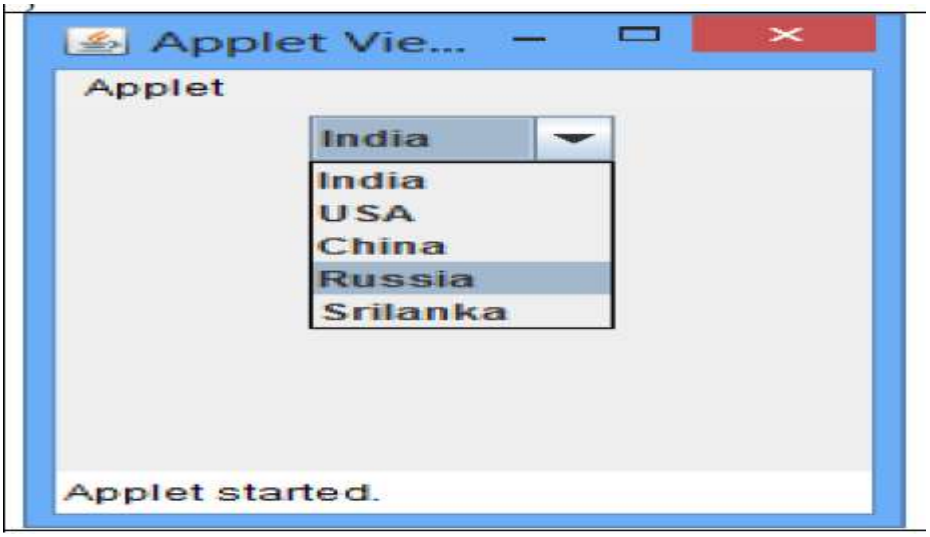
JComboBox

A combobox is a combination of textfield and dropdown list. JComboBox is a subclass of JComponent.

JList allows you to select multiple items in the list but Jcombobox selects only one item from the list

```
import java.awt.*;

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

```
import javax.swing.*;

public class sample extends JApplet

{

public void init()

{

Container c = getContentPane();

c.setLayout(new FlowLayout());

JComboBox cb=new JComboBox();

 cb.addItem("India"); cb.addItem("USA"); cb.addItem("China"); cb.addItem("Russia");

cb.addItem("Srilanka"); c.add(cb);



}

}
```



JScrollPane

JScrollPanes are used in GUI development to restrict a widget to a certain size on the screen, then provide a way of scrolling up and down, left and right if the widget becomes any bigger than the 'Viewport' size. This is used primarily in JTextArea, JTable and JList where the data displayed by the widget is changed dynamically and can really affect the size and dimensions of the widget.

Some of the constructors defined by JScrollPane class are as follows.

- JSrollPane ()
- JScrollPane(Component component)
- JScrollPane(int ver, int hor)
- JScrollPane(Component component, int ver, int hor)

where,

- `component` is the component to be added to the scroll pane
- `ver` and `hor` specify the policies to display the vertical and horizontal scroll bar, respectively.

Some of the standard policies are:

- `HORIZONTAL_SCROLLBAR_ALWAYS`
- `HORIZONTAL_SCROLLBAR_AS_NEEDED`
- `VERTICAL_SCROLLBAR_ALWAYS`
- `VERTICAL_SCROLLBAR_AS_NEEDED`

```
import javax.swing.*;

import java.awt.event.*;

import java.awt.*;

public class sample extends JApplet

{
```

```
public void init()

{

Container cp = getContentPane();


JPanel jp = new JPanel( ) ;

jp.setLayout( new GridLayout( 20, 20 ) ) ;

Icon image = new ImageIcon("C:/Users/sri p/workspace/applet1/flower.jpg");

JLabel label = new JLabel(image);

jp.add(label);


int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED ;

int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED ;


JScrollPane js = new JScrollPane( jp, v, h ) ;

cp.add(js, BorderLayout.CENTER) ;

}}
```

JList

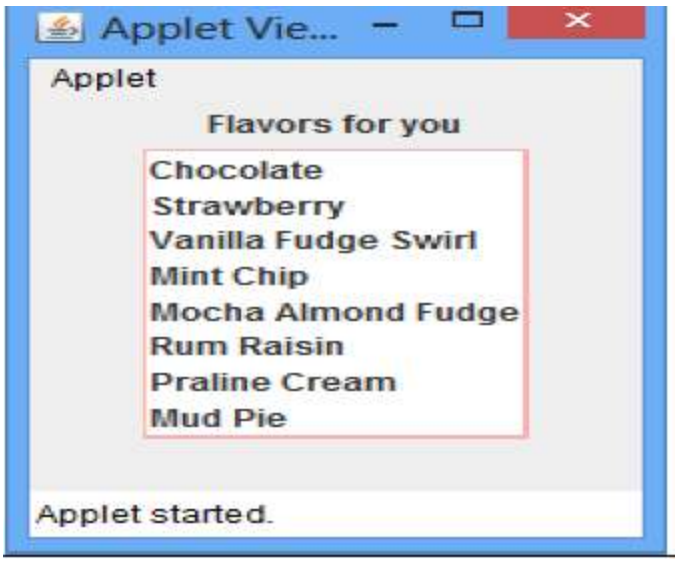
A list components allows user to select a single or multiple items from a given list of items by clicking on each. By default, a user can select multiple items.

A list can be created by instantiating a JList class. Relevant methods:

Method	Description
Public JList()	Creates an empty JList;
Public JList(Object [] listItems)	created with the items contained in the array of objects;
Public JList(VectorlistItems)	created with theVector elements;

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
public class sample extends JApplet
{
JLabel l;
public void init()
{
        String[] flavors = { "Chocolate", "Strawberry",
```

```
                "Vanilla Fudge Swirl", "Mint Chip", "Mocha Almond Fudge",
                "Rum Raisin", "Praline Cream", "Mud Pie" };
        Container cp = getContentPane();
          cp.setLayout(new FlowLayout());

     l= new JLabel("Flavors for you");
     cp.add(l);

   JList lst = new JList(flavors);
   cp.add(lst);

  Border brd = BorderFactory.createMatteBorder( 1, 1, 2, 2, Color.pink);
  lst.setBorder(brd);
  }
}
```



Jtable

The JTable class is another Swing component that has no equivalent in AWT. JTable provides a very flexible possibility to create and display tables. It extends JComponent class. The JTable component of swing represents a two dimensional grid of objects. A JTable displays data in rows and column format. You can edit data values in a table as well. The JTable class of swing represents swing tables.

Constructors and methods of the class JTable.

Constructors and Methods	Description
JTable()	Constructs a default JTable Initialized to default data model, default column and selection model.
JTable(int nrows, int ncols)	Constructs a new JTable with *nrows* and *ncols* of empty cells using DelaultTableModel.
JTable(Object[][] rowdata, Object[ ] colnames)	Constructs a JTable to display the data contained in the two-dimensional array, row data, having the column names specified in the array, colnames.
JTable(Vector rowvector, Vector colvector)	Constructs a JTable to display the data available in vector of vectors, rowvector, with column names available in colvector.

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.* ;

public class sample extends JApplet

{

  public void init()
```

```
{
  Container c = getContentPane();
  c.setLayout( new BorderLayout()) ;
  String fields[] = { "empid", "empname",  "empsal" } ;
  Object details[][] = {{ "1", "ABC", "4500.50" },          // instead of Object array
                        { "2","XYZ","4567.50" },            // String array can also be used
                        { "3", "PQR", "2246.30" },
                        { "4", "KLM", "3245.75" },
                        { "5", "Jyostna", "2500.25" }
                      };
  JTable jt = new JTable(details, fields);
// if the rows are more than height of the applet, scroll bar is added
  int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
  int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
  JScrollPane jsp = new JScrollPane(jt , v , h); c.add(jsp, "Center");
 }
}
```