Swings:

AWT is used for creating GUI in Java. However, the AWT components are internally depends on native methods like C functions and operating system equivalent and hence problems related to portability arise (look and feel. Ex. Windows window and MAC window). And, also AWT components are heavy weight. It means AWT components take more system resources like memory and processor time.

Due to this, Java soft people felt it is better to redevelop AWT package without internally taking the help of native methods. Hence all the classes of AWT are extended to form new classes and a new class library is created. This library is called JFC (Java Foundation Classes).

Java Foundation Classes (JFC):

JFC is an extension of original AWT. It contains classes that are completely portable, since the entire JFC is developed in pure Java. Some of the features of JFC are:

1. JFC components are light-weight: Means they utilize minimum resources.
2. JFC components have same look and feel on all platforms. Once a component is created, it looks same on any OS.
3. JFC offers "pluggable look and feel" feature, which allows the programmer to change look and feel as suited for platform. For, ex if the programmer wants to display window-style button on Windows OS, and Unix style buttons on Unix, it is possible.
4. JFC does not replace AWT, but JFC is an extension to AWT. All the classes of JFC are derived from AWT and hence all the methods in AWT are also applicable in JFC.

So, JFC represents class library developed in pure Java which is an extension to AWT and swing is one package in JFC, which helps to develop GUIs and the name of the package is

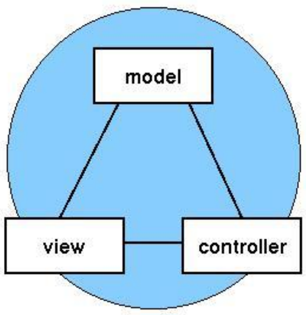
```
import javax.swing.*;
```

Here x represents that it is an 'extended package' whose classes are derived from AWT package.

MVC Architecture:

In MVC terminology,

Model corresponds to the state information associated with the component (data).
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 visual appearance of the component based upon model data.



The controller acts as an interface between view and model. It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.

Although the MVC architecture and the principles behind it are conceptually sound, the high level of separation between the view and the controller is not beneficial for Swing components. Instead, Swing uses a modified version of MVC that combines the view and the controller into a single logical entity called the UI delegate. For this reason, Swing's approach is called either the Model-Delegate architecture or the Separable Model architecture.

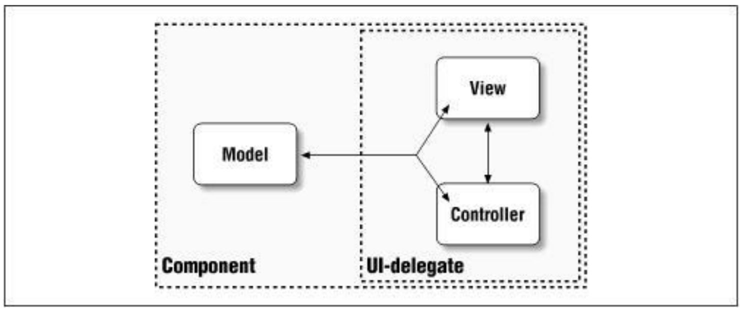


Figure : With Swing, the view and the controller are combined into a UI-delegate object

So let's review: each Swing component contains a model and a UI delegate. The model is responsible for maintaining information about the component's state. The UI delegate is responsible for maintaining information about how to draw the component on the screen. In addition, the UI delegate reacts to various events.

Difference between AWT and Swings:

AWT	**Swing**
Heavy weight	Light weight
Look and feel is OS based	Look and feel is OS independent.
Not pure Java based	Pure Java based
Applet portability: Web-browser is support	Applet portability: A plug-in is required
Do not support features like icon and tool tip.	It supports.
The default layout manager for applet: flow and frame is border layout.	The default layout manger for content pane is border layout.

Components and Containers:

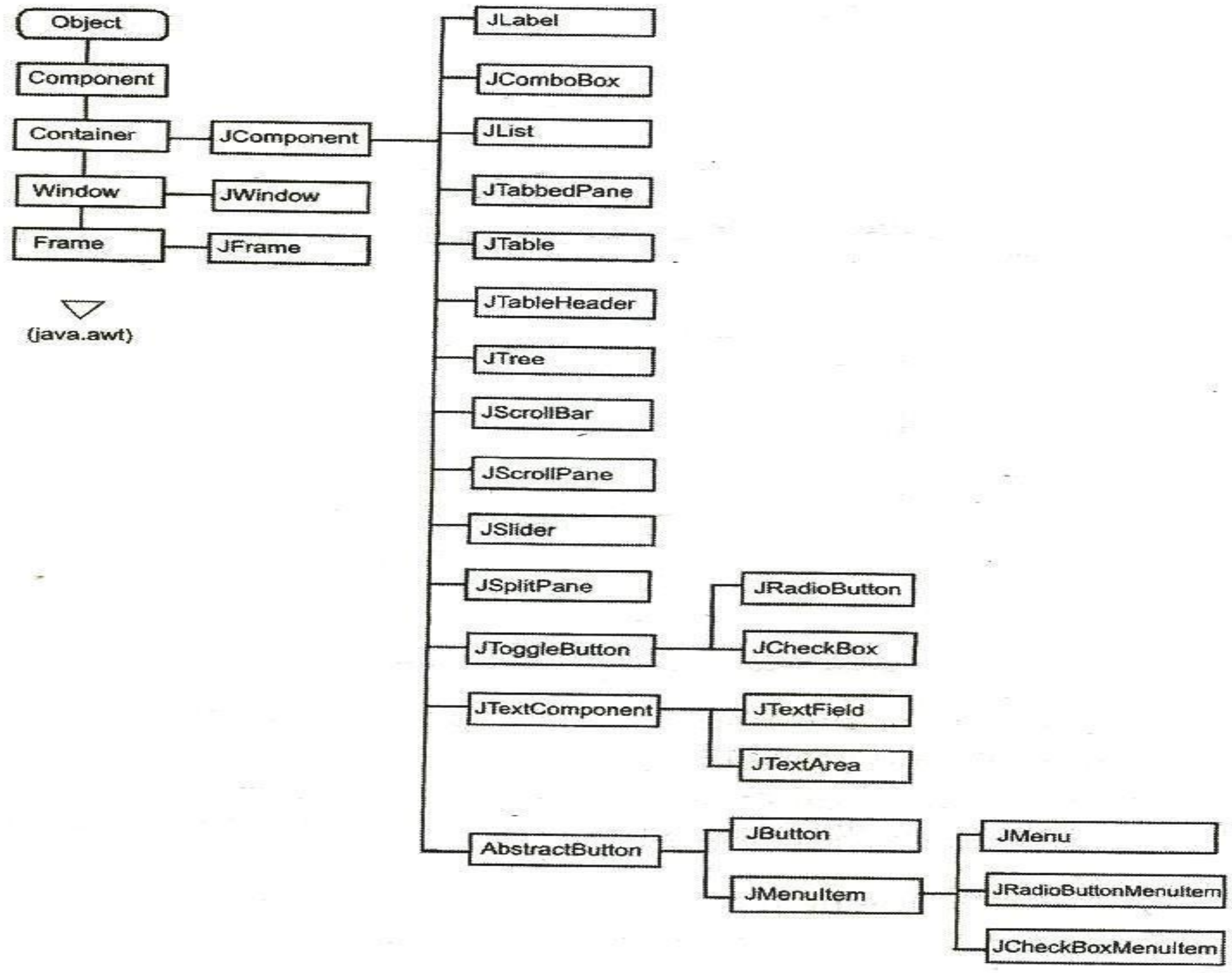
A Swing GUI consists of two key items: ***components*** and ***containers.***

However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components.

Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, **a container can also hold other containers**. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

Components:

In general, Swing components are derived from the **JComponent** class. **JComponent** provides the functionality that is common to all components. For example, **JComponent** supports the pluggable look and feel. **JComponent** inherits the AWT classes **Container** and **Component.** All of Swing's components are represented by classes defined within the package **javax.swing**. The following figure shows hierarchy of classes of javax.swing.



Containers:

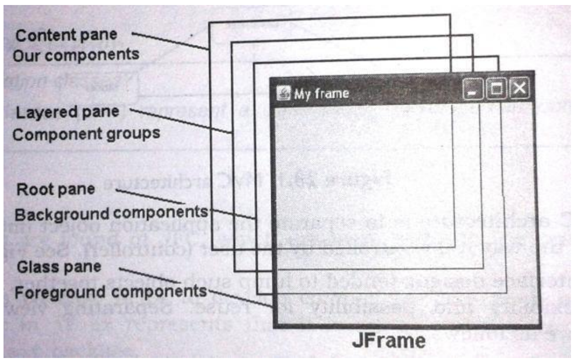
Swing defines two types of containers.

1. **Top-level containers/ Root containers:** JFrame, JApplet,JWindow, and JDialog.
   As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container. Furthermore, every containment hierarchy must begin with a top-level container. The one most commonly used for applications are JFrame and JApplet.

Unlike Swing's other components , the top-level containers are heavyweight. Because they inherit AWT classes Component and Container.

Whenever we create a top level container four sub-level containers are automatically created:

Glass pane (JGlass)
Root pane (JRootPane)
Layered pane(JLayeredPane)
Content pane



Glass pane: This is the first pane and is very close to the monitor's screen. Any components to be displayed in the foreground are attached to this glass pane. To reach this glass pane we use getGlassPane() method of JFrame class, which return Component class object.

Root Pane: This pane is below the glass pane. Any components to be displayed in the background are displayed in this frame. To go to the root pane, we can use getRootPane() method of JFrame class, which returns JRootPane object.

Layered pane: This pane is below the root pane. When we want to take several components as a group, we attach them in the layered pane. We can reach this pane by calling getLayeredPane() method of JFrame class which returns JLayeredPane class object.

Conent pane: This is bottom most of all, Individual components are attached to this pane. To reach this pane, we can call getContentPane() method of JFrame class which returns Container class object.

2. **Lightweight containers** – containers do inherit JComponent. An example of a lightweight container is JPanel, which is a general-purpose container. Lightweight containers are often used to organize and manage groups of related components.

JFrame:

We know frame represents a window with a title bar and borders. Frame becomes the basis for creating the GUIs for an application because all the components go into the frame. To create a fram, we have to create an object to JFrame class in swing as

```
JFrame jf=new JFrame(); // create a frame without title
JFrame jf=new JFrame("title"); // create a frame with title
```

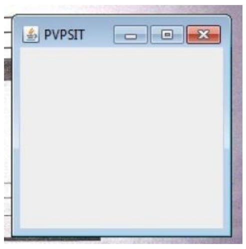
To close the frame, use setDefaultCloseOperation() method of JFrame class

```
setDefaultCloseOperation(constant)
```

where constant values are

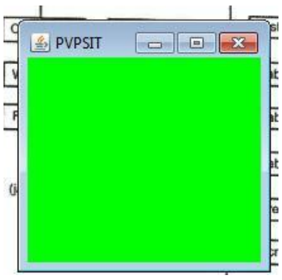
JFrame.EXIT_ON_CLOSE	This closes the application upon clicking the close button
JFrame.DISPOSE_ON_CLOSE	This closes the application upon clicking the close button
JFrame.DO_NOTHING_ON_CLOSE	This will not perform any operation upon clicking close button
JFrame.HIDE_ON_CLOSE	This hides the frame upon clicking close button

Example:



```
import javax.swing.*;
class FrameDemo
{
        public static void main(String arg[])
        {
                JFrame jf=new JFrame("PVPSIT");
                jf.setSize(200,200);
                jf.setVisible(true);
                jf.setDefaultCloseOperation(JFrame.HIDE_ON_CLOSE );
        }
}
```

Example: To set the background



```
import javax.swing.*;
 import java.awt.*;
class FrameDemo
{
        public static void main(String arg[])
        {
                JFrame jf=new JFrame("PVPSIT");
                jf.setSize(200,200);
                jf.setVisible(true);
                Container c=jf.getContentPane();
                c.setBackground(Color.green);
        }
}
```

JApplet:

Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that use Swing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is not found in **Applet**. For example, **JApplet** supports various "panes," such as the content pane, the glass pane, and the root pane.

One difference between **Applet** and **JApplet** is, When adding a component to an instance of **JApplet**, do not invoke the **add( )** method of the applet. Instead, call **add( )** for the *content pane* of the **JApplet** object.

The content pane can be obtained via the method shown here:

Container getContentPane( )

The **add( )** method of **Container** can be used to add a component to a content pane. Its form is shown here:

void add(*comp*)

Here, *comp* is the component to be added to the content pane.

JComponent:

The class **JComponent** is the base class for all Swing components except top-level containers. To use a component that inherits from JComponent, you must place the component in a containment hierarchy whose root is a top-level SWING container.

Constructor: JComponent();

The following are the JComponent class's methods to manipulate the appearance of the component.

public int getWidth ()	Returns the current width of this component in pixel.
public int getHeight ()	Returns the current height of this component in pixel.
public int getX()	Returns the current x coordinate of the component's top-left corner.
public int getY ()	Returns the current y coordinate of the component's top-left corner.
public java.awt.Graphics getGraphics()	Returns this component's Graphics object you can draw on. This is useful if you want to change the appearance of a component.
public void setBackground (java.awt.Color bg)	Sets this component's background color.
public void setEnabled (boolean enabled)	Sets whether or not this component is enabled.
public void setFont (java.awt.Font font)	Set the font used to print text on this component.
public void setForeground (java.awt.Color fg)	Set this component's foreground color.
public void setToolTipText(java.lang.String text)	Sets the tool tip text.
public void setVisible (boolean visible)	Sets whether or not this component is visible.

Text Fields

The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**. It provides functionality that is common to Swing text components. One of its subclasses is **JTextField**, which allows you to edit one line of text. Some of its constructors are shown here:

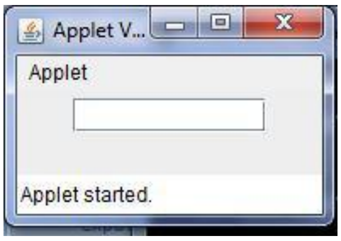
JTextField( )
JTextField(int *cols*)
JTextField(String *s*, int *cols*)
JTextField(String *s*)

Here, *s* is the string to be presented, and *cols* is the number of columns in the text field.

The following example illustrates how to create a text field. The applet begins by getting its content pane, and then a flow layout is assigned as its layout manager. Next, a **JTextField** object is created and is added to the content pane.

Example:

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
</applet>
*/
public class JTextFieldDemo extends JApplet
{
        JTextField jtf;
        public void init()
        {
                // Get content pane
                Container contentPane = getContentPane();
                contentPane.setLayout(new FlowLayout());
                // Add text field to content
                pane jtf = new JTextField(10);
                contentPane.add(jtf);
        }
}
```



Applet started.

The JButton Class

The **JButton** class provides the functionality of a push button. **JButton** allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here:

JButton(Icon *i*)
JButton(String *s*)
JButton(String *s*, Icon *i*)
Here, *s* and *i* are the string and icon used for the button.

Example:

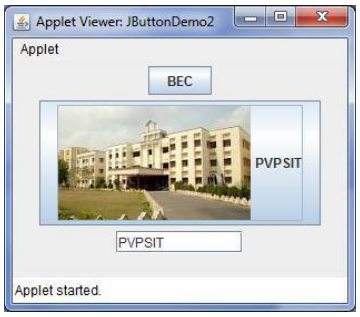
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
/*
<applet code="JButtonDemo2" width=250 height=300>
</applet>
*/
public class JButtonDemo2 extends JApplet implements ActionListener
{
        JTextField jtf;
        public void init()
        {
                // Get content pane
                Container contentPane = getContentPane();
                contentPane.setLayout(new FlowLayout());

                JButton jb1 = new JButton("BEC");
                jb1.addActionListener(this);
                contentPane.add(jb1);

                // Add buttons to content pane
                ImageIcon pvp = new ImageIcon("pvp.jpg");
                JButton jb2 = new JButton("PVPSIT",pvp);
                jb2.setActionCommand("PVPSIT");
                jb2.addActionListener(this);
                contentPane.add(jb2);

                jtf = new JTextField(10);
                contentPane.add(jtf);
        }
        public void actionPerformed(ActionEvent ae)
        {
                jtf.setText(ae.getActionCommand());
        }
}
```



Check Boxes:

The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of **AbstractButton**. Its immediate super class is **JToggleButton**, which provides support for two-state buttons (true or false). Some of its constructors are shown here:

JCheckBox(Icon *i*)
JCheckBox(Icon *i*, boolean *state*)
JCheckBox(String *s*)
JCheckBox(String *s*, boolean *state*)
JCheckBox(String *s*, Icon *i*)
JCheckBox(String *s*, Icon *i*, boolean *state*)

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the check box is initially selected. Otherwise, it is not.
The state of the check box can be changed via the following method:

void setSelected(boolean *state*)

Here, *state* is **true** if the check box should be checked.

When a check box is selected or deselected, an item event is generated. This is handled by **itemStateChanged( )**. Inside **itemStateChanged( )**, the **getItem( )** method gets the **JCheckBox** object that generated the event. The **getText( )** method gets the text for that check box and uses it to set the text inside the text field.

Example:

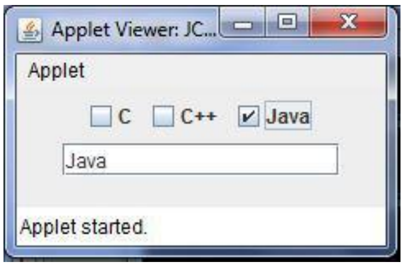
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo2" width=400 height=50>
</applet>
*/
public class JCheckBoxDemo2 extends JApplet implements ItemListener
{
        JTextField jtf;
        public void init() {
                // Get content pane
                Container contentPane = getContentPane();
                contentPane.setLayout(new FlowLayout());

                JCheckBox cb = new JCheckBox("C", true);
                cb.addItemListener(this);
                contentPane.add(cb);

                cb = new JCheckBox("C++");
                cb.addItemListener(this);
                contentPane.add(cb);

                cb = new JCheckBox("Java");
                cb.addItemListener(this);
                contentPane.add(cb);

                // Add text field to the content
                pane jtf = new JTextField(15);
                contentPane.add(jtf);
        }
        public void itemStateChanged(ItemEvent ie) {
                JCheckBox cb = (JCheckBox)ie.getItem();
                jtf.setText(cb.getText());
        }
}
```



Java

Applet started.

Radio Buttons:

Radio buttons are supported by the **JRadioButton** class, which is a concrete implementation of **AbstractButton**. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons. Some of its constructors are shown here:

JRadioButton(Icon *i*)
JRadioButton(Icon *i*, boolean *state*)
JRadioButton(String *s*)
JRadioButton(String *s*, boolean *state*)
JRadioButton(String *s*, Icon *i*)
JRadioButton(String *s*, Icon *i*, boolean *state*)

Here, *i* is the icon for the button. The text is specified by *s*. If *state* is **true**, the button is initially selected. Otherwise, it is not.

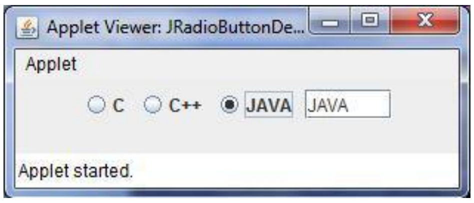
Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time. For example, if a user presses a radio button that is in a group, any previously selected button in that group is automatically deselected. The **ButtonGroup** class is instantiated to create a button group. Its default constructor is invoked for this purpose. Elements are then added to the button group via the following method:

void add(AbstractButton *ab*)

Here, *ab* is a reference to the button to be added to the group.

Radio button presses generate action events that are handled by **actionPerformed( )**. The **getActionCommand( )** method gets the text that is associated with a radio button and uses it to set the text field.

Example:



```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet implements ActionListener
{
        JTextField tf;
        public void init()
        {
                // Get content pane
                Container contentPane = getContentPane();
                contentPane.setLayout(new FlowLayout());

                // Add radio buttons to content pane
                JRadioButton c = new JRadioButton("C");
                c.addActionListener(this);
                contentPane.add(c);
                JRadioButton cpp = new JRadioButton("C++");
                cpp.addActionListener(this);
                contentPane.add(cpp);

                JRadioButton java = new JRadioButton("JAVA");
                java.addActionListener(this);
                contentPane.add(java);

                // Define a button group
                ButtonGroup bg = new ButtonGroup();
                bg.add(c);
                bg.add(cpp);
                bg.add(java);

                // Create a text field and add it to the content pane
                tf = new JTextField(5); contentPane.add(tf);

        }
        public void actionPerformed(ActionEvent ae)
                { tf.setText(ae.getActionCommand());
        }
 }
```

Combo boxes:

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**.

A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field.

Two of **JComboBox**'s constructors are shown here:

```
JComboBox( )
JComboBox(Vector v)
```

Here, *v* is a vector that initializes the combo box. Items are added to the list of choices via the **addItem( )** method, whose signature is shown here:

```
void addItem(Object obj)
```

Here, *obj* is the object to be added to the combo box.

By default, a JComboBox component is created in read-only mode, which means the user can only pick one item from the fixed options in the drop-down list. If we want to allow the user to provide his own option, we can simply use the setEditable() method to make the combo box editable.

The following example contains a combo box and a label. The label displays an icon. The combo box contains entries for “PVPSIT”, “BEC”, and “VRSEC”. When a college is selected, the label is updated to display the flag for that country

Example:

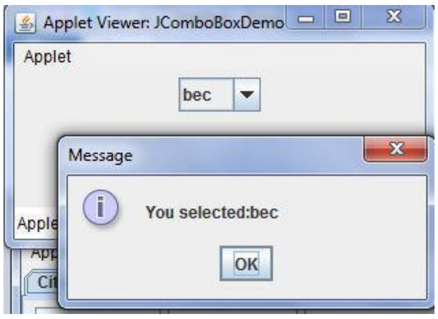
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/* <applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JApplet implements ItemListener
{
        Container contentPane;

        public void init()
        {
                // Get content pane
                contentPane = getContentPane();
                contentPane.setLayout(new FlowLayout());

                // Create a combo box and add it to the
                panel JComboBox jc = new JComboBox();
                jc.addItem("pvp");
                jc.addItem("bec");
                jc.addItem("vrsec");

                jc.addItemListener(this);
                contentPane.add(jc);

         }
        public void itemStateChanged(ItemEvent ie)
        {
                String s = (String)ie.getItem();
                JOptionPane.showMessageDialog(null,"You selected:"+s);
        }
}
```



You selected:bec

Tabbed Panes:

A *tabbed pane* is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.

Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor. Tabs are defined via the following method:

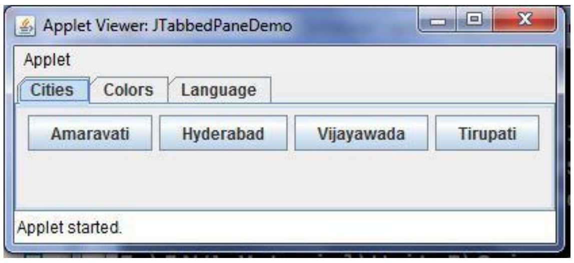
void addTab(String *str*, Component *comp*)

Here, *str* is the title for the tab, and *comp* is the component that should be added to the tab. Typically, a **JPanel** or a subclass of it is added.

The general procedure to use a tabbed pane in an applet is outlined here:

1. Create a **JTabbedPane** object.
2. Call **addTab( )** to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.

The following example illustrates how to create a tabbed pane. The first tab is titled“Cities” and contains four buttons. Each button displays the name of a city. The second tab is titled “Colors” and contains three check boxes. Each check box displays the name of a color. The third tab is titled “Language” and contains radio buttons.



Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JTabbedPaneDemo" width=400
height=100>
</applet>
*/
public class JTabbedPaneDemo extends JApplet
{
        public void init()
        {
                Container contentPane = getContentPane();
                JTabbedPane jtp = new JTabbedPane();
                jtp.addTab("Cities", new CitiesPanel());
                jtp.addTab("Colors", new ColorsPanel());
                jtp.addTab("Language", new LanguagesPanel());
                contentPane.add(jtp);
        }
}
class CitiesPanel extends JPanel
{
        public CitiesPanel()
        {
                JButton b1 = new JButton("Amaravati");
                add(b1);
                JButton b2 = new JButton("Hyderabad");
                add(b2);
                JButton b3 = new JButton("Vijayawada");
                add(b3);
                JButton b4 = new JButton("Tirupati");
                add(b4);
        }
}
class ColorsPanel extends JPanel
{
        public ColorsPanel()
        {
                JCheckBox cb1 = new JCheckBox("Red");
                add(cb1);
                JCheckBox cb2 = new JCheckBox("Green");
                add(cb2);
                JCheckBox cb3 = new JCheckBox("Blue");
                add(cb3);
        }
}
class LanguagesPanel extends JPanel
{
        public LanguagesPanel()
        {
                JRadioButton rb1 = new JRadioButton("Telugu");
                add(rb1);
                JRadioButton rb2 = new JRadioButton("Hindi");
                add(rb2);
                JRadioButton rb3 = new JRadioButton("English");
                add(rb3);
        }
}
```

JLabel :

JLabel is a class of java Swing . JLabel is used to display a short string or an image icon. JLabel can display text, image or both . JLabel is only a display of text or image and it cannot get focus . JLabel is inactive to input events such a mouse focus or keyboard focus. By default labels are vertically centered but the user can change the alignment of label.

Constructor of the class are :

JLabel() : creates a blank label with no text or image in it.
JLabel(String s) : creates a new label with the string specified.
JLabel(Icon i) : creates a new label with a image on it.
JLabel(String s, Icon i, int align) : creates a new label with a string, an image and a specified horizontal alignment

Commonly used methods of the class are :

getIcon() : returns the image that the label displays

setIcon(Icon i) : sets the icon that the label will display to image i
getText() : returns the text that the label will display
setText(String s) : sets the text that the label will display to string s

JList :

JList is part of Java Swing package . JList is a component that displays a set of Objects and allows the user to select one or more items . JList inherits JComponent class. JList is a easy way to display an array of Vectors .

Constructor for JList are :

JList(): creates an empty blank list
JList(E [ ] l) : creates an new list with the elements of the array.
JList(ListModel d): creates a new list with the specified List Model
JList(Vector l) : creates a new list with the elements of the vector

Commonly used methods are :

method	explanation
getSelectedIndex()	returns the index of selected item of the list
getSelectedValue()	returns the selected value of the element of the list
setSelectedIndex(int i)	sets the selected index of the list to i
getSelectedValuesList()	returns a list of all the selected items.
getSelectedIndices()	returns an array of all of the selected indices, in increasing order

```
// java Program to create a simple JList

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


class solve extends JFrame
{

    //frame
    static JFrame f;

    //lists
    static JList b;
    //main class
    public static void main(String[] args)
    {
        //create a new frame
        f = new JFrame("frame");

        //create a object
        solve s=new solve();

        //create a panel
        JPanel p =new JPanel();

        //create a new label
        JLabel l= new JLabel("select the day of the week");

        //String array to store weekdays
        String week[]= { "Monday","Tuesday","Wednesday",
                        "Thursday","Friday","Saturday","Sunday"};

        //create list
        b= new JList(week);

        //set a selected index
        b.setSelectedIndex(2);

        //add list to panel
        p.add(b);

        f.add(p);

        //set the size of frame
        f.setSize(400,400);

        f.show();
    }


}
```

A Java applet is a special kind of Java program that a browser enabled with Java technology can download from the internet and run. An applet is typically embedded inside a web page and runs in the context of a browser. An applet must be a subclass of the **java.applet.Applet** class. The Applet class provides the standard interface between the applet and the browser environment.

The **Applet** class is contained in the **java.applet** package.**Applet** contains several methods that give you detailed control over the execution of your applet.

In addition,**java.applet** package also defines three interfaces: **AppletContext**, **AudioClip**, and **AppletStub**.

Applet Basics:

All applets are subclasses of **Applet**. Thus, all applets must import **java.applet**. Applets must also import **java.awt**. **AWT** stands for the Abstract Window Toolkit. Since all applets run in a window, it is necessary to include support for that window by importing java.awt package.

Applets are not executed by the console-based Java run-time interpreter. Rather, they are executed by either a Web browser or an applet viewer.

Execution of an applet does not begin at **main( ).** Output to your applet's window is not performed by **System.out.println( )**. Rather, it is handled with various AWT methods, such as **drawString( )**, which outputs a string to a specified X,Y location. Input is also handled differently than in an application.

Once an applet has been compiled, it is included in an HTML file using theAPPLET tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file.
To view and test an applet more conveniently, simply include a comment at the head of your Java source code file that contains the APPLET tag.

Here is an example of such a comment:
```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```
This comment contains an APPLET tag that will run an applet called **MyApplet** in a window that is 200 pixels wide and 60 pixels high. Since the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this tutorial will contain the appropriate APPLET tag embedded in a comment.

The Applet Class:

Applet extends the AWT class **Panel**. In turn, **Panel** extends **Container**, which extends **Component**. These classes provide support for Java's window-based, graphical interface. Thus, **Applet** provides all of the necessary support for window-based activities

Applet Architecture:

An applet is a window-based program. As such, its architecture is different from the so-called normal, console-based programs .
First, applets are event driven. it is important to understand in a general way how the event-driven architecture impacts the design of an applet.
Here is how the process works. An applet waits until an event occurs. The AWT notifies the applet about an event by calling an event handler that has been provided by the applet. Once this happens, the applet must take appropriate action and then quickly return control to the AWT.

Applet Initialization and Termination(Applet Life Cycle methods):

It is important to understand the order in which the various methods shown in theskeleton are called. When an applet begins, the AWT calls the following methods, in this sequence:
1. **init( )**
2. **start( )**
3. **paint( )**
When an applet is terminated, the following sequence of method calls takes place:
1. **stop( )**
2. **destroy( )**

init( ):init( ) method is called once—the first time an applet is loaded. The **init( )** method is the first method to be called. This is where you should initialize variables.

start(): The **start( )** method is called after **init( )**. It is also called to restart an applet after it has been stopped(i.e start() method is called every time, the applet resumes execution).

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

stop( ): The stop() method is called when the applet is stopped(i.e for example ,when the applet is minimized the stop method is called).

destroy( ): The **destroy( )** method is called when the environment determines that your applet needs to be removed completely from memory(i.e destroy() method is called when the applet is about to terminate).The **stop( )** method is always called before **destroy( )**.

An Applet Skeleton
Program:

Four methods—**init( )**, **start( )**, **stop( )**, and **destroy( )**—are defined by **Applet**. Another, **paint( )**, is defined by the AWT **Component** class. All applets must import **java.applet**. Applets must also import **java.awt.**

These five methods can be assembled into the skeleton shown
here:

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

Simple Applet programe:

<u>SimpleApplet.java</u>

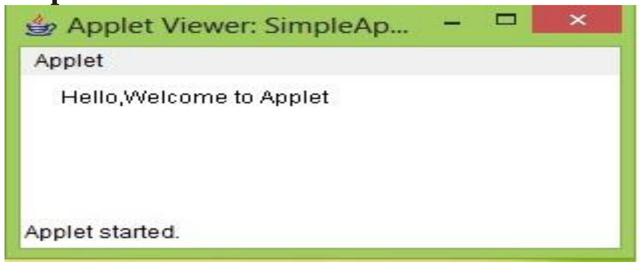
```
import java.awt.*;
import java.applet.*;
/*
<applet code="SimpleApplet" width=300 height=100>
</applet>
*/

public class SimpleApplet extends Applet
{
   String msg="";
   // Called first.
  public void init()
  {
    msg="Hello";
  }
   /* Called second, after init().
   Also called       whenever  the applet is restarted. */
  public void start()
  {
    msg=msg+",Welcome to Applet";
  }

  // whenever the applet must redraw its output, paint( ) is called.
  public void paint(Graphics g)
  {

    g.drawString(msg,20,20);
  }
}
```

Output:



```
Applet Viewer: SimpleAp...
Applet
   Hello,Welcome to Applet

Applet started.
```

How To Run an Applet Programe:

There are two ways in which you can run an applet:

■ Executing the applet within a Java-compatible Web browser.

■ Using an applet viewer, such as the standard SDK tool, **appletviewer**. An applet viewer executes your applet in a window. This is generally the fastestand easiest way to test your applet.

Using an applet viewer to run applet(demonstrates you to run SimpleApplet.java):

Place the applet tag in comments in java source code.
Note:Code attribute value must be equal to name of class which extends Applet class.

```
Compiling:  javac SimpleApplet.java
       Run:  AppletViewer SimpleApplet.java
```

Executing the applet within a Java-compatible Web browser(demonstrates you to run SimpleApplet.java):

Compiling: javac SimpleApplet.java
Create an Html file and embeded Applet tag in html file.

Attributes in applet tag:

Code(attribute):specify name of applet class to load into browser.
Width(attribute):width of an applet.
Height(attribute):height of an applet.

<u>SimpleApplet.html</u>

```
<html>
   <body>
      <applet code="SimpleApplet" width=300 height=100></applet>
    </body>
</html>
```

When you open SimpleApplet.html , SimpleApplet.class applet is loaded into browser.

Note: The Browser must be java enabled to load applet programe.

Simple Applet Display Methods:

As we've mentioned, applets are displayed in a window and they use the AWT to perform input and output.To output a string to an applet, use **drawString( )**, which is a member of the **Graphics** class.Graphics class is defined in **java.awt** package.

void drawString(String *message*, int *x*, int *y*)

Here, *message* is the string to be output and x and y are x-coordinate ,y-coordinate respectively. In a Java window, the upper-left corner is location 0,0.

To set the background color of an applet's window, use **setBackground( )**. To set the foreground color (the color in which text is shown, for example), use **setForeground( )**. These methods are defined by **Component**, and they have the following general forms:

void setBackground(Color *newColor*)
void setForeground(Color *newColor*)

Here, *newColor* specifies the new color. The class **Color** defines the constants shown here that can be used to specify colors:

Color.black Color.magenta
Color.blue Color.orange
Color.cyan Color.pink
Color.darkGray Color.red
Color.gray Color.white
Color.green Color.yellow
Color.lightGray

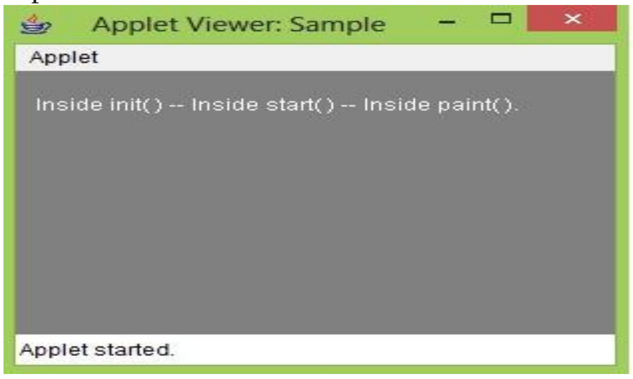
For example, this sets the background color to green and the text color to red:

setBackground(Color.green);
setForeground(Color.red);

<u>Sample.java</u>

```
/* A simple applet that sets the foreground and background colors and outputs a string. */
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=200>
</applet>*/
 public class Sample extends Applet
 {
    String msg;
       public void init()
    {
       setBackground(Color.gray);
       setForeground(Color.white);
       msg = "Inside init( ) --";
    }
    // Initialize the string to be displayed.
    public void start()
    {
      msg += " Inside start( ) --";
    }
    // Display msg in applet window.
   public void paint(Graphics g)
   {   msg += " Inside paint( ).";
        g.drawString(msg, 10, 30);
   }
```

Output :



```
Inside init() -- Inside start() -- Inside paint().
```

```
Applet started.
```

Requesting Repainting:

Whenever your applet needs to update the information displayed in its window, it simply calls **repaint( )**.The **repaint( )** method is defined by the AWT. It causes the AWT run-time system to execute a call to your applet's **update( )** method, which, in its default implementation, calls **paint( )**.

The simplest version of **repaint( )** is shown here:

void repaint( )

This version causes the entire window to be repainted. The following version specifies a region that will be repainted:

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

Here, the coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height.* These dimensions are specified in pixels. You save time by specifying a region to repaint.
The other two versions of repaint():

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

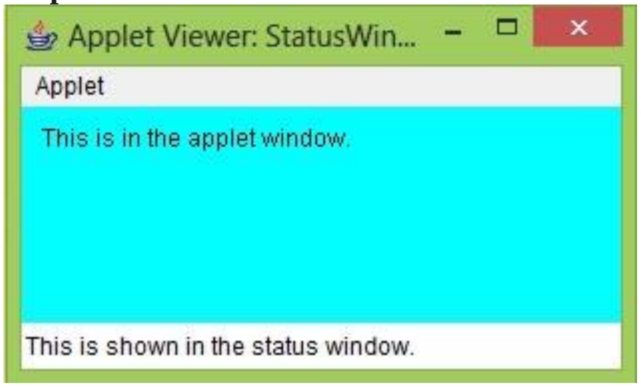
Here, *maxDelay* specifies the maximum number of milliseconds that can elapse before **update( )** is called.

Using the Status window:
In addition to displaying information in its window, an applet can also output a message to the status window of the browser or applet viewer on which it is running. To do so, call
showStatus( ) with the string that you want displayed.

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

/*<applet      code="StatusWindow"      width=300
height=300></applet>*/
public   class   StatusWindow   extends
Applet
{
  public void init()
  {setBackground(Color.cyan);
  }
  // Display msg in applet window.
  public void paint(Graphics g)
  {
    g.drawString("This is in the applet window.", 10, 20);
    showStatus("This is shown in the status window.");
  }
}
```

Output:



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

Types of applets :(Based on look and Feel)

There are two varieties of applets. The first are those based directly on the **Applet** class. These applets use the Abstract Window Toolkit (AWT) to provide the graphic user interface (or use no GUI at all). This style of applet has been available since Java was first created.

The second type of applets are those based on the Swing class **JApplet**. Swing applets use the Swing classes to provide the GUI. Swing offers a richer and often easier-to-use user interface than does the AWT. Thus, Swing-based applets are now the most popular. **JApplet** inherits **Applet**, all the features of **Applet** are also available in **Japplet**.

Types of Applets (In General) :

Web pages can contain two types of applets which are named after the location at which they are stored.

1. Local Applet 2. Remote Applet

Local Applets: A local applet is the one that is stored on our own computer system. When the Web-page has to find a local applet, it doesn't need to retrieve information from the Internet. A local applet is specified by a path name and a file name as shown below in which the codebase attribute specifies a path name, whereas the code attribute specifies the name of the byte-code file that contains the applet's code.

<applet codebase="MyAppPath" code="MyApp.class" width=200 height=200> </applet>

Remote Applets: A remote applet is the one that is located on a remote computer system . This computer system may be located in the building next door or it may be on the other side of the world. No matter where the remote applet is located, it's downloaded onto our computer via the Internet. The browser must be connected to the Internet at the time it needs to display the remote applet. To reference a remote applet in Web page, we must know the applet's URL (where it's located on the Web) and any attributes and parameters that we need to supply. A local applet is specified by a url and a file name as shown below.

<applet codebase="http://www.apoorvacollege.com" code="MyApp.class" width=200 height=200> </applet>