Unit-5

Swings

1. Introduction
2. Limitations of AWT
3. MVC Architecture
4. Components
5. Containers
6. Exploring Swing:-
    a. JApplet
    b. JFrame
    c. JComponent
    d. ImageIcon
    e. JLabel
    f. JTextfield
    g. JButton
    h. JCheckBox
    i. JList
    j. JRadiobutton
    k. JComboBox
    l. JTabbedPane
    m. JScrollPane

1.Introduction

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.

Swing Defination:-

Swing is a framework or API that is used to create GUI (or) window-based applications.It is an advanced version of AWT(Abstract Window Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

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

Swing is a Java Foundation Classes [JFC] library and an extension of the Abstract Window Toolkit [AWT]. Swing offers much-improved functionality over AWT, new components, expanded components features, and excellent event handling with drag-and-drop support.

2.Limitations of AWT

- AWT supports limited number of GUI components.
- AWT components are heavy weight components.
- AWT components are platform-dependent.
- AWT components are developed by using platform specific code.
- AWT components behaves differently in different operating systems.
- AWT components is converted by the native code of the operating system.
- AWT components do not support features like icons and tool-tips.
- AWT doesn’t follows MVC(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.

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

3.MVC Architecture

The Model-View-Controller Architecture

Swing uses the *model-view-controller architecture* (MVC) as the fundamental design behind each of its components. Essentially, MVC breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves.

Model

The model encompasses the state data for each component. There are different models for different types of components. For example, the model of a scrollbar component might contain information about the current position of its adjustable “thumb,” its minimum and maximum values, and the thumb’s width (relative to the range of values). A menu, on the other hand, may simply contain a list of the menu items the user can select from. This information remains the same no matter how the component is painted on the screen; model data is always independent of the component’s visual representation.

View

The view refers to how you see the component on the screen. For a good example of how views can differ, look at an application window on two different GUI platforms. Almost all window frames have a title bar spanning the top of the window. However, the title bar may have a close box on the left side (like the Mac OS platform), or it may have the

close box on the right side (as in the Windows platform). These are examples of different types of views for the same window object.

Controller

The controller is the portion of the user interface that dictates how the component interacts with events. Events come in many forms — e.g., a mouse click, gaining or losing focus, a keyboard event that triggers a specific menu command, or even a directive to repaint part of the screen. The controller decides how each component reacts to the event—if it reacts at all.

Figure 1-6 shows how the model, view, and controller work together to create a scrollbar component. The scrollbar uses the information in the model to determine how far into the scrollbar to render the thumb and how wide the thumb should be. Note that the model specifies this information relative to the minimum and the maximum. It does not give the position or width of the thumb in screen pixels—the view calculates that. The view determines exactly where and how to draw the scrollbar, given the proportions offered by the model. The view knows whether it is a horizontal or vertical scrollbar, and it knows exactly how to shadow the end buttons and the thumb. Finally, the controller is responsible for handling mouse events on the component. The controller knows, for example, that dragging the thumb is a legitimate action for a scrollbar, within the limits defined by the endpoints, and that pushing on the end buttons is acceptable as well. The result is a fully functional MVC scrollbar.

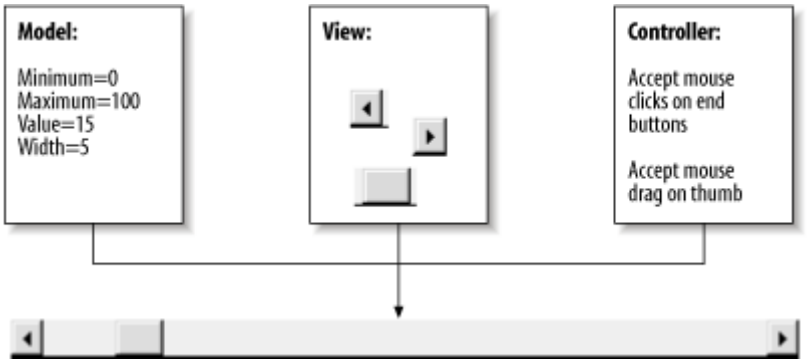


Figure 1-6. The three elements of a model-view-controller architecture

MVC Interaction

With MVC, each of the three elements—the model, the view, and the controller—requires the services of another element to keep itself continually updated. Let’s continue discussing the scrollbar component.

We already know that the view cannot render the scrollbar correctly without obtaining information from the model first. In this case, the scrollbar does not know where to draw its “thumb” unless it can obtain its current position and width relative to the minimum and

maximum. Likewise, the view determines if the component is the recipient of user events, such as mouse clicks. (For example, the view knows the exact width of the thumb; it can tell whether a click occurred over the thumb or just outside of it.) The view passes these events on to the controller, which decides how to handle them. Based on the controller's decisions, the values in the model may need to be altered. If the user drags the scrollbar thumb, the controller reacts by incrementing the thumb's position in the model. At that point, the whole cycle repeats. The three elements, therefore, communicate their data as shown in Figure 1-7.

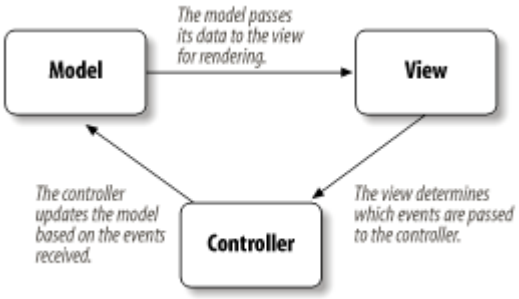


Figure 1-7. Communication through the model-view-controller architecture

MVC in Swing

Swing actually uses a simplified variant of the MVC design called the *model-delegate* . This design combines the view and the controller object into a single element, the *UI delegate* , which draws the component to the screen and handles GUI events. Bundling graphics capabilities and event handling is somewhat easy in Java, since much of the event handling is taken care of in AWT. As you might expect, the communication between the model and the UI delegate then becomes a two-way street, as shown in Figure 1-8.

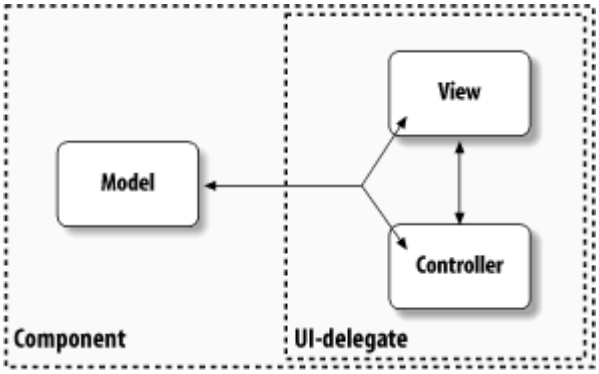


Figure 1-8. With Swing, the view and the controller are combined into a UI-delegate object

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 (in conjunction with AWT) reacts to various events that propagate through the component.

Note that the separation of the model and the UI delegate in the MVC design is extremely advantageous. One unique aspect of the MVC architecture is the ability to tie multiple views to a single model. For example, if you want to display the same data in a pie chart and in a table, you can base the views of two components on a single data model. That way, if the data needs to be changed, you can do so in only one place—the views update themselves accordingly In the same manner, separating the delegate from the model gives the user the added benefit of choosing what a component looks like without affecting any of its data. By using this approach, in conjunction with the lightweight design, Swing can provide each component with its own pluggable L&F.

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.

4.Components:

Defination:-

A component is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of Jcomponents are the JButtons, JCheckBoxes, and JScrollbars of a typical graphical user interface.

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.

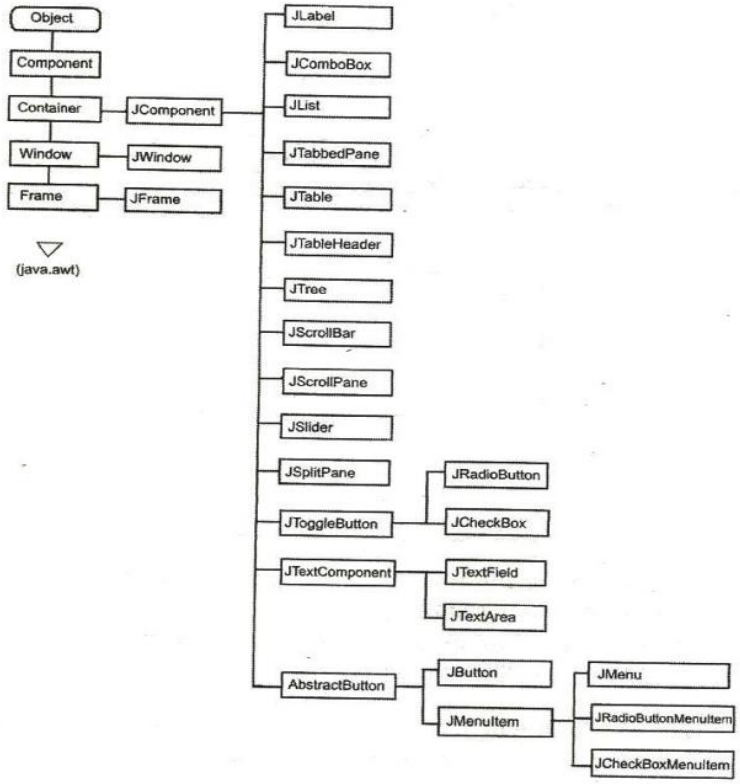


Fig. Swing Class Hierarchy

5.Containers:

Swing defines two types of containers.

1.Top-level Container

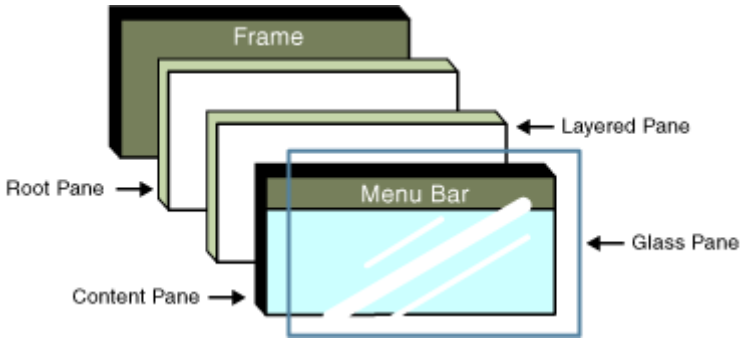
2.Lightweight Container

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.

Conent pane: This pane is below the glass pane. 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.

Layered pane: This pane is below the content 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.

Root Pane: This is bottom most 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.

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.

7. Exploring Swing:-

1.JApplet

1. Swing Based applets are similar to AWT-based applets, but with an important difference:A swing applet extends JApplet rather than Applet.
2. JApplet is derived from Applet.Thus,JApplet includes all of the functionality found in Applet and adds support for swing.
3. JApplet is a top-level Swing Container, which means that it is not derived from JComponent.
4. Swing applets also follow same applet life cycle methods init(), start(), paint(),stop() and destroy().
5. 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.
6. 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.
7. The content pane can be obtained via the method shown here:

   Container getContentPane( )

8. The add( ) method of Container can be used to add a component to a content pane.

Its form is shown here:

void add(comp)

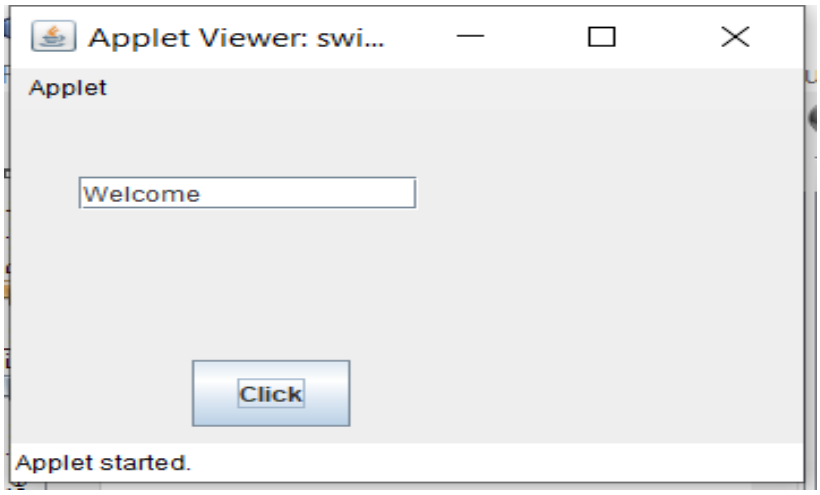
Here, comp is the component to be added to the content pane.

Example Program Executing appletviewer:-

```
1. import java.applet.*;
2. import javax.swing.*;
3. import java.awt.event.*;
4. /*<applet code="EventJApplet.class" width="300" height="300">  </applet> */
5. public class EventJApplet extends JApplet implements ActionListener{
6. JButton b;
7. JTextField tf;
8. public void init(){
9.
10.tf=new JTextField();
11.tf.setBounds(30,40,150,20);
12.
13.b=new JButton("Click");
14.b.setBounds(80,150,70,40);
15.
16.add(b);add(tf);
17.b.addActionListener(this);
18.
19.setLayout(null);
20.}
21.
22.public void actionPerformed(ActionEvent e){
23.tf.setText("Welcome");
24.}
25.}
```

In the above example, we have created all the controls in init() method because it is invoked only once.

Output:-



2.JFrame

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. The Frame in Java Swing is defined in class javax.swing.JFrame. JFrame class inherits the java.awt.Frame class. JFrame is like the main window of the GUI application using swing.

To create a frame, 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

Constant	Description
JFrame.EXIT_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.DISPOSE_ON_CLOSE	This closes the application upon clicking the close button

JFrame.HIDE_ON_CLOSE	This hides the frame upon clicking close button

We can create a JFrame window object using two approaches:

#1) By Extending The JFrame Class

The first approach is creating a new class to construct a Frame. This class inherits from the JFrame class of the javax.swing package.

The following program implements this approach.

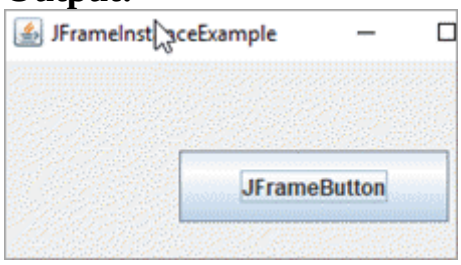
```
import javax.swing.*;

class FrameInherited extends JFrame{    //inherit from JFrame class

   JFrame f;

   FrameInherited(){

      JButton b=new JButton("JFrame_Button");//create button object

      b.setBounds(100,50,150, 40);


      add(b);//add button on frame

      setSize(300,200);

      setLayout(null);

      setVisible(true);

   }

}

public class Main {

   public static void main(String[] args) {

     new FrameInherited(); //create an object of FrameInherited class

   }

}
```

Output:

#2) By Instantiating The JFrame Class

```
import javax.swing.*;

public class Main {


   public static void main(String[] args) {
      JFrame f=new JFrame("JFrameInstanceExample");//create a JFrame object


      JButton b=new JButton("JFrameButton");//create instance of JButton
      b.setBounds(100,50,150, 40);//dimensions of JButton object


      f.add(b);//add button in JFrame


      f.setSize(300,200);//set frame width = 300 and height = 200
      f.setLayout(null);//no layout manager specified
      f.setVisible(true);//make the frame visible
   }
}
```

Output:



c. JComponent

The JComponent class is the base class of all Swing components except top-level containers. Swing components whose names begin with "J" are descendants of the JComponent class. For example, JButton, JScrollPane, JPanel, JTable etc. But, JFrame and JDialog don't inherit JComponent class because they are the child of top-level containers.

The JComponent class extends the Container class which itself extends Component. The Container class has support for adding components to the container.

Constructor: JComponent();

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

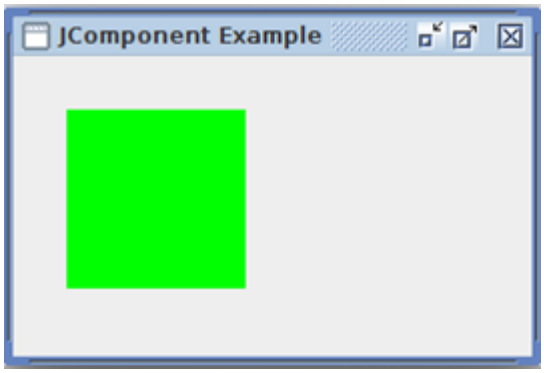
public int getWidth ()	Returns the current width of this component in pixel.
public int getHeight ()	Returns the current height of this component in pixel.
public int getX()	Returns the current x coordinate of the component's top-left corner.
public int getY ()	Returns the current y coordinate of the 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.

Example program:-

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;
class MyJComponent extends JComponent {
    public void paint(Graphics g) {
      g.setColor(Color.green);
      g.fillRect(30, 30, 100, 100);
    }
}
public class JComponentEx {
    public static void main(String[] arguments) {
```

```
        MyJComponent com = new MyJComponent();
        // create a basic JFrame
        JFrame.setDefaultLookAndFeelDecorated(true);
        JFrame frame = new JFrame("JComponent Example");
        frame.setSize(300,200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        // add the JComponent to main frame
        frame.add(com);
        frame.setVisible(true);
      }
}
```

Output:-



d. ImageIcon

The class **ImageIcon** is an implementation of the Icon interface that paints Icons from Images.

Class Declaration

Following is the declaration for **javax.swing.ImageIcon** class −

```
public class ImageIcon
   extends Object
      implements Icon, Serializable, Accessible
```

An implementation of the Icon interface that paints Icons from Images. Images that are created from a URL, filename or byte array are preloaded using MediaTracker to monitor the loaded state of the image. For further information and examples of using image icons, see How to Use Icons in The Java Tutorial.

Class Constructors

Sr.No.	Constructor & Description

1	**ImageIcon()** Creates an uninitialized image icon.
2	**ImageIcon(byte[] imageData)** Creates an ImageIcon from an array of bytes which were read from an image file containing a supported image format, such as GIF, JPEG, or (as of 1.3) PNG.
3	**ImageIcon(byte[] imageData, String description)** Creates an ImageIcon from an array of bytes which were read from an image file containing a supported image format, such as GIF, JPEG, or (as of 1.3) PNG.
4	**ImageIcon(Image image)** Creates an ImageIcon from an image object.
5	**ImageIcon(Image image, String description)** Creates an ImageIcon from the image.
6	**ImageIcon(String filename)** Creates an ImageIcon from the specified file.
7	**ImageIcon(String filename, String description)** Creates an ImageIcon from the specified file.
8	**ImageIcon(URL location)** Creates an ImageIcon from the specified URL.
9	**ImageIcon(URL location, String description)** Creates an ImageIcon from the specified URL.

Class Methods

Sr.No.	Method & Description
1	**String getDescription()** Gets the description of the image.
2	**int getIconHeight()**

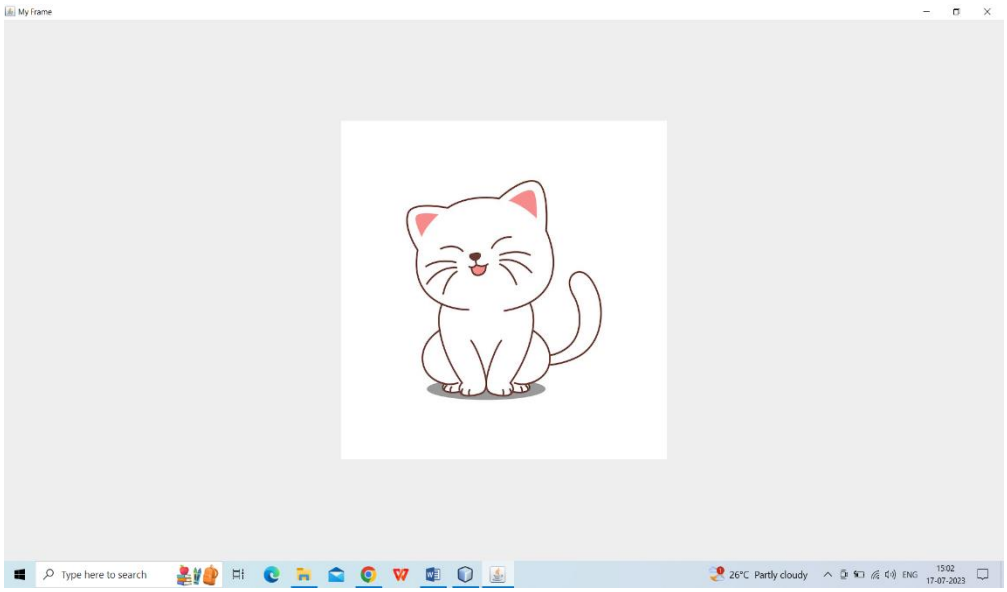
	Gets the height of the icon.
3	**int getIconWidth()** Gets the width of the icon.
4	**Image getImage()** Returns this icon's Image.
5	**void paintIcon(Component c, Graphics g, int x, int y)** Paints the icon.
6	**void setDescription(String description)** Sets the description of the image.
7	**void setImage(Image image)** Sets the image displayed by this icon.

Example program:-

```
import java.awt.BorderLayout;

import javax.swing.ImageIcon;

import javax.swing.JFrame;

import javax.swing.JLabel;

   public class ImageIconEx {

      public static void main(String args[])

      {

         TimeFrame frame = new TimeFrame();
```

```
    }

}

class TimeFrame extends JFrame

{

    //Image icon = Toolkit.getDefaultToolkit().getImage("me.jpg");

    ImageIcon icon = new
ImageIcon("C:\\Users\\Admin\\Pictures\\pictures\\cat.jpg","Anusha");

    JLabel label = new JLabel(icon);

    public TimeFrame(){

        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        setTitle("My Frame");

        setSize(500,400);

        //this.setIconImage(icon);

        add(label);

        setVisible(true);

    }

}
```

Output:-



e. 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.

Fields:-

Public static final int LEFT

Public static final int RIGHT

Public static final int CENTER

Public static final int LEADING

Public static final int TRAILING

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

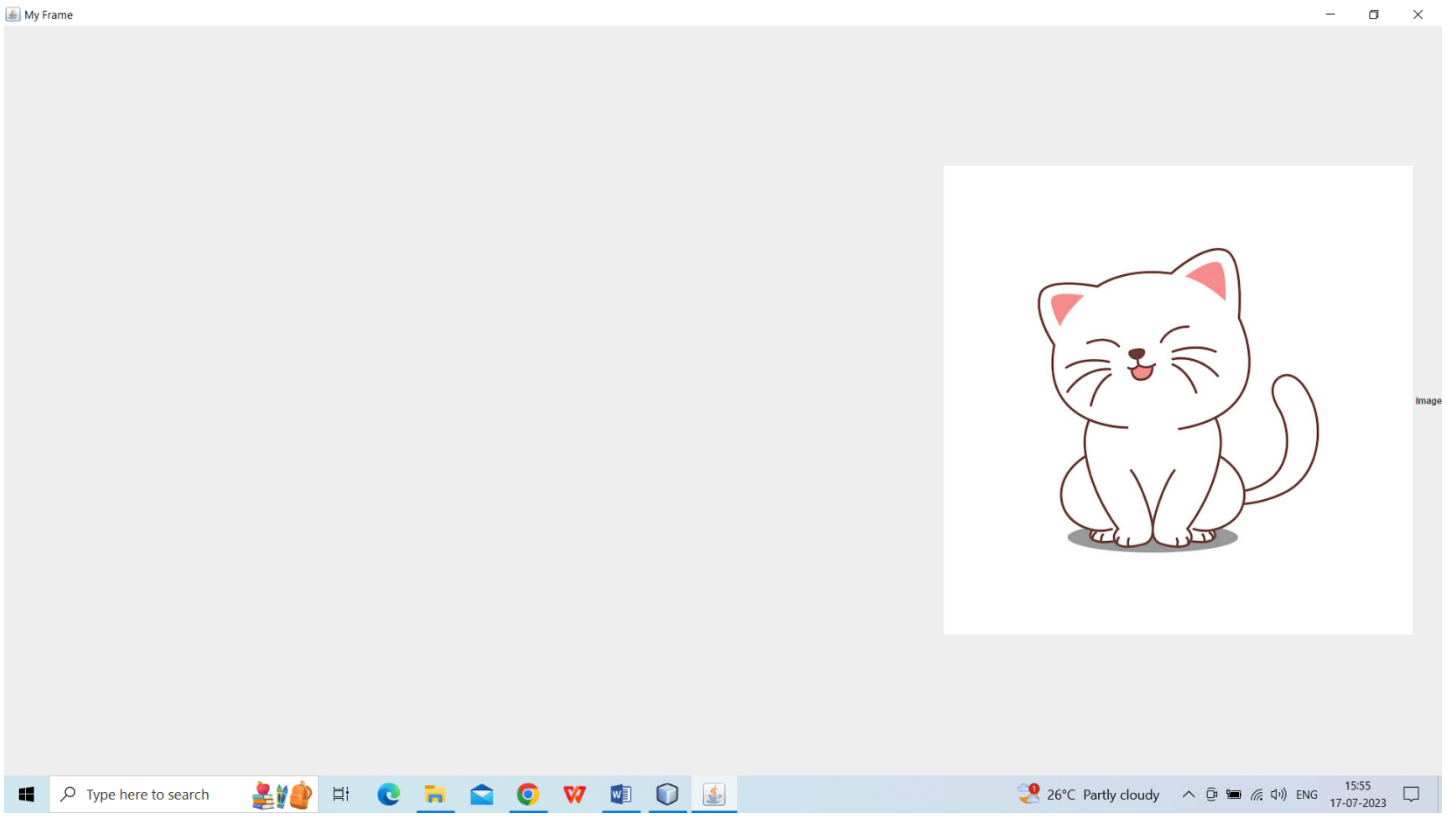
getText() : returns the text that the label will display

setText(String s) : sets the text that the label will display to string s

Example program:-

```
import java.awt.*;
import javax.swing.ImageIcon;
import javax.swing.*;
public class JLabelEx extends JFrame{
  public JLabelEx(){
     ImageIcon icon = new ImageIcon("C:\\Users\\Admin\\Pictures\\pictures\\cat.jpg");
JLabel label = new JLabel("Image",icon,JLabel.RIGHT);
this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
setTitle("My Frame");
setSize(500,400);
add(label);
setVisible(true);

       }
  public static void main(String args[])
{
 JLabelEx frame = new JLabelEx();
}

   }
```

Output:-



f.JTextfield

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
 jtf = new JTextField(10);
contentPane.add(jtf);
}
}
```

Output:-

g.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)

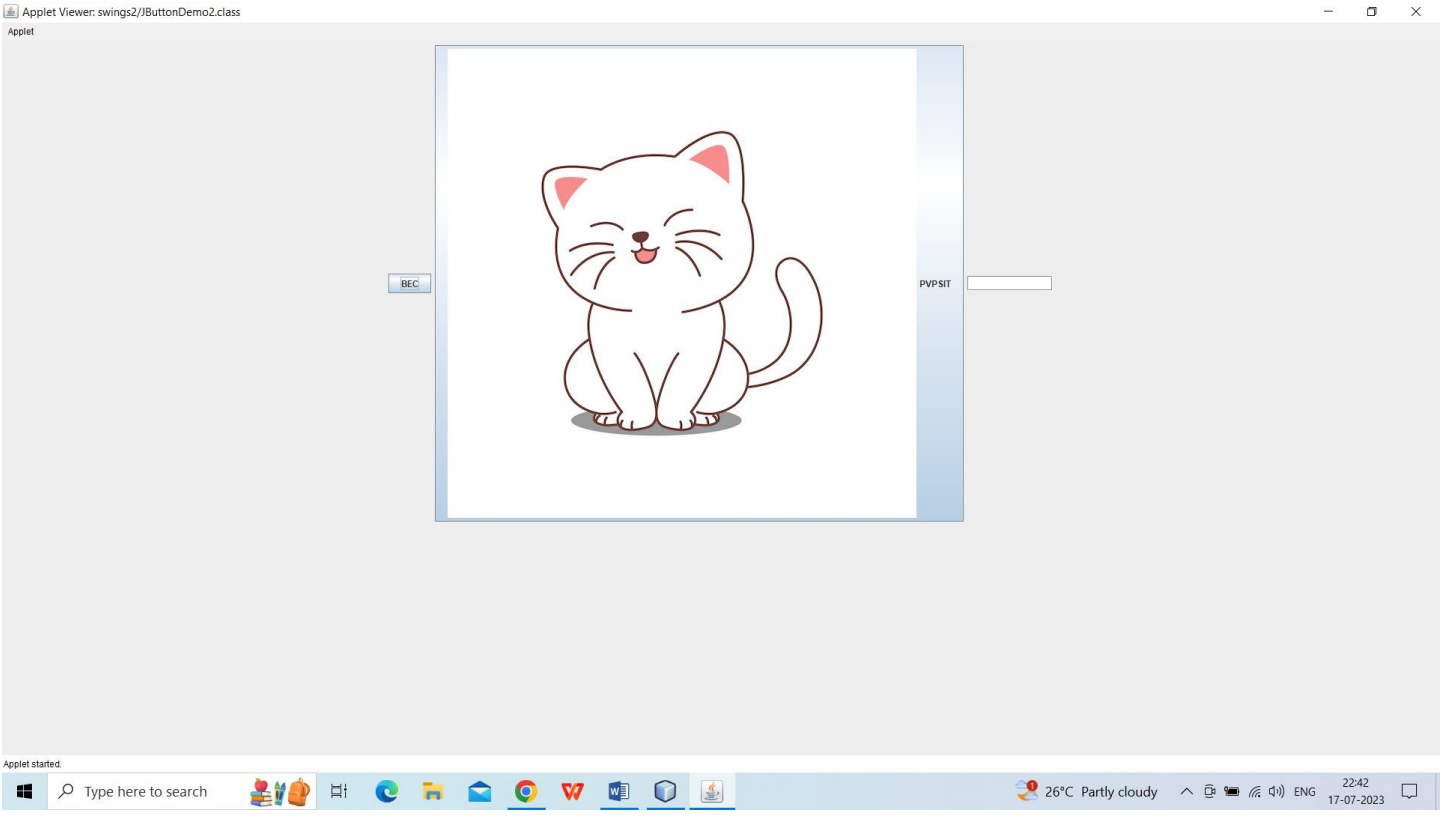
Here, s and i are the string and icon used for the button.

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());
}
}
```

Output:-



h.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:

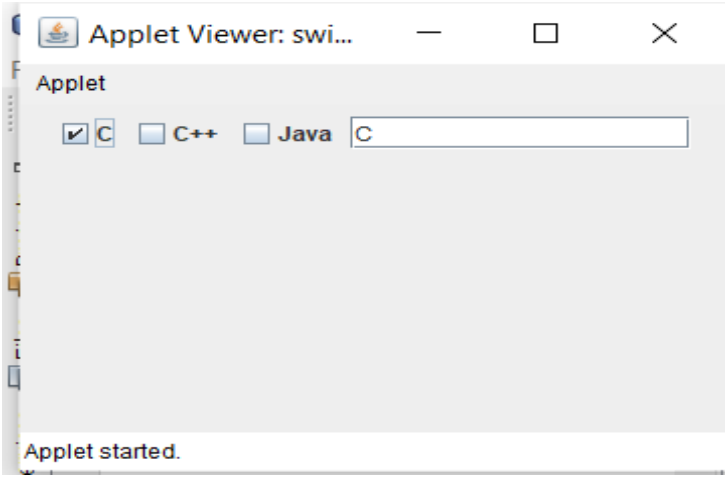
void setSelected(boolean state)

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 contentpane
 jtf = new JTextField(15);
contentPane.add(jtf);
}
public void itemStateChanged(ItemEvent ie) {
JCheckBox cb = (JCheckBox)ie.getItem();
jtf.setText(cb.getText());
}
}
```



i.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.

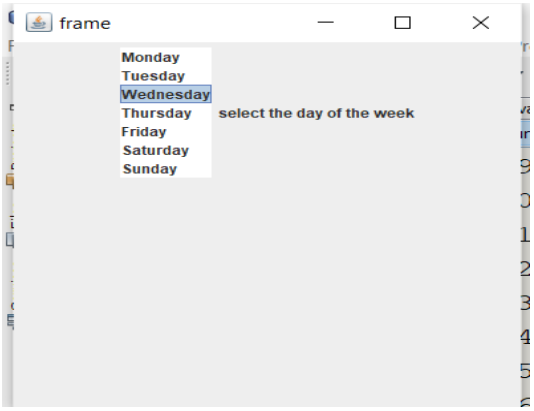
getSelectedIndices() returns an array of all of the selected indices, in increasing order

Example program:-

```
// java Program to create a simple JList
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
class JListEx
{

 //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
 //JListEx s=new JListEx();

 //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);
p.add(l);

f.add(p);

//set the size of frame
f.setSize(400,400);
f.setVisible(true);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
//f.show();
 }
}
```

Output:-



j.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

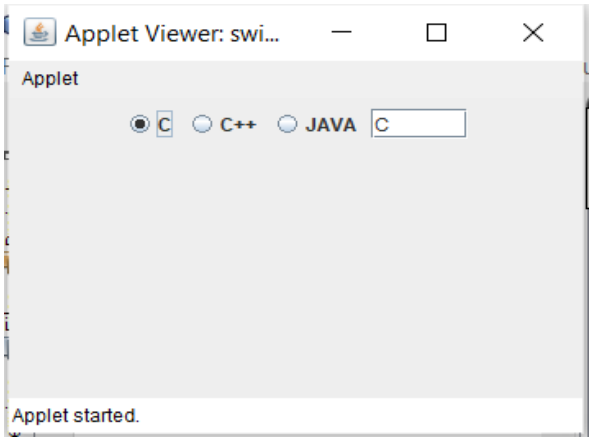
uses it to set the text field.

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());
}
}
```

Output:-



k.JComboBox:

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

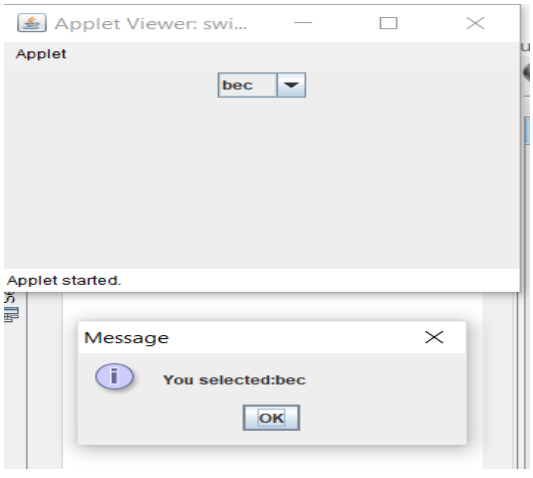
selected, the label is updated to display the flag for that country

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;
JComboBox jc;
public void init()
{
// Get content pane
contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create a combo box and add it to the
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)jc.getItemAt(jc.getSelectedIndex());
JFrame f=new JFrame();
JOptionPane.showMessageDialog(f,"You selected:"+s);
}}
```

Output:-



l.JTabbedPane:

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.

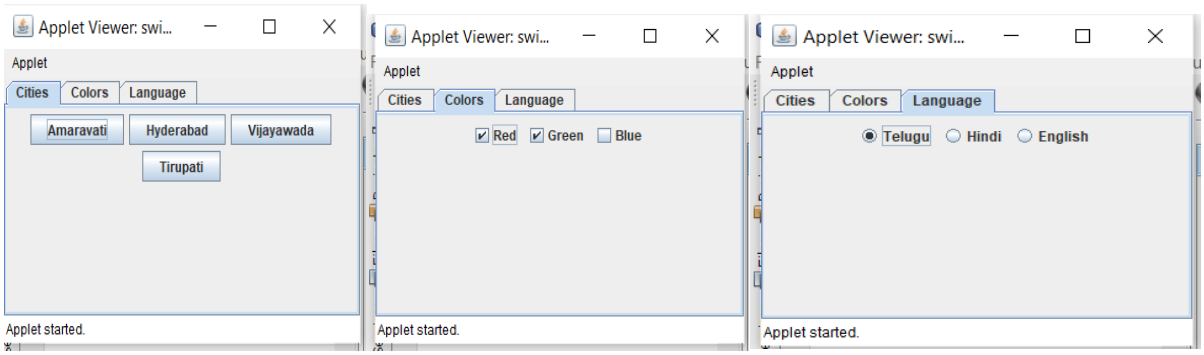
Example:

```
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()
{
    ButtonGroup bg = new ButtonGroup();

JRadioButton rb1 = new JRadioButton("Telugu");
bg.add(rb1);
JRadioButton rb2 = new JRadioButton("Hindi");
bg.add(rb2);
JRadioButton rb3 = new JRadioButton("English");
bg.add(rb3);
add(rb1);
add(rb2);
add(rb3);
}
}
```

Output:-



m.JScrollPane

JScrollPane provides a **scrollable view of a component, where a component maybe an image, table, tree etc**. A JScrollPane can be added to a top-level container like JFrame or a component like JPanel. JScrollPane is another **lightweight component** which extends **JComponent** class.

Constructors of JScrollPane

Constructor	Description
public JScrollPane()	Creates an empty JScrollPane with no viewport, where both the horizontal and vertical scrollbars appearing when required.
public JScrollPane(Component view)	Creates a JScrollPane that displays a component within it. This JScrollPane also shows the horizontal and vertical bar, only when its component's contents are larger than the viewing area.

Methods of JScrollPane class

Methods	Description
public setPreferredSize(Dimension d)	Sets the preferred size of JScrollPane.
public int setLayout(LayoytManager managaer)	Sets the layout manager for JScrollPane.

Example Program:-

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


public class JScrollPaneEx
{
public static void main(String... ar)
{
SwingUtilities.invokeLater(new Runnable() {
public void run()
{
new A();
}
});
```

```
}//Closing the main method
}//Closing the class Combo


class A //implements ActionListener
{

JFrame jf;
JPanel jp;
JLabel label1;

A()
{
jf = new JFrame("JScrollPane");
label1 = new JLabel("Displaying a picture ",JLabel.CENTER);

//Creating an ImageIcon object to create a JLabel with image
ImageIcon image = new ImageIcon("nature.jpg");
JLabel label = new JLabel(image, JLabel.CENTER);

//Creating a JPanel and adding JLabel that contains the image
jp = new JPanel(new BorderLayout());
jp.add( label, BorderLayout.CENTER );

//Adding JPanel to JScrollPane
JScrollPane scrollP = new JScrollPane(jp);

//Adding JLabel and JScrollPane to JFrame
jf.add(label1,BorderLayout.NORTH);
jf.add(scrollP,BorderLayout.CENTER);
```

```
jf.setSize(350,270);
jf.setVisible(true);
}

}
```

Output:-