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

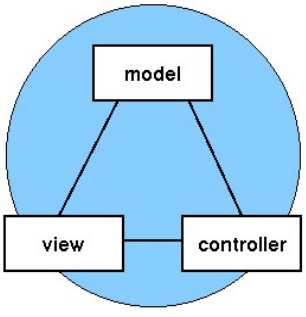
```
import javax.swing.*;
```

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.

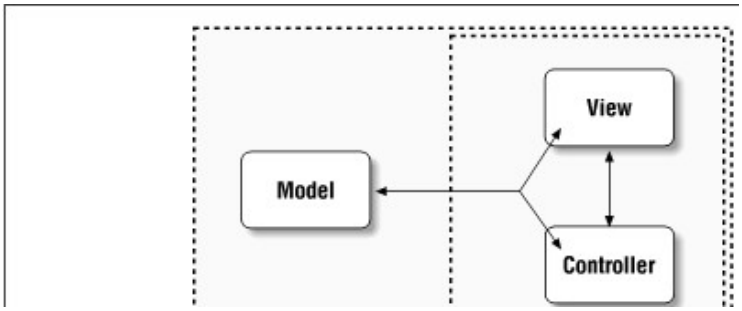


Figure : With Swing, the view and the controller are combined into a UI-delegate object

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 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:

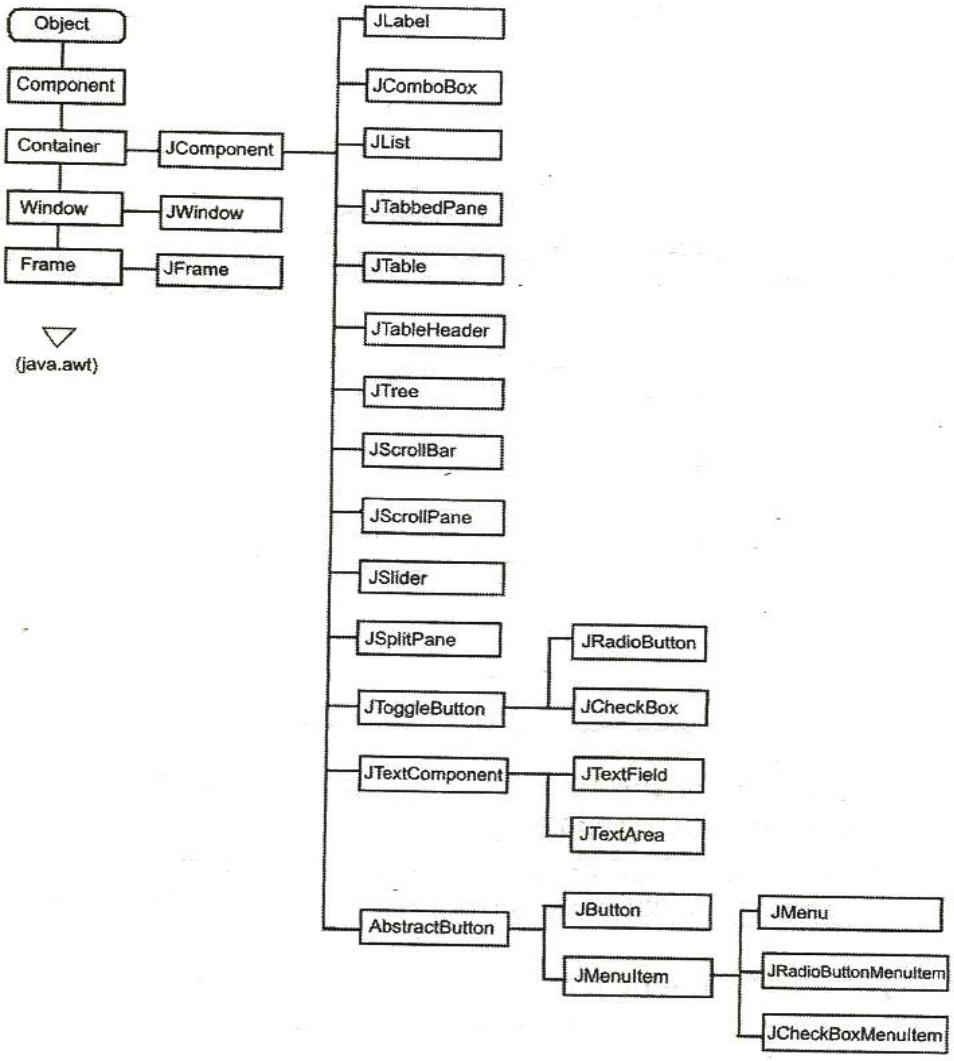
A Swing GUI consists of two key items: ***components*** and ***containers.***

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.

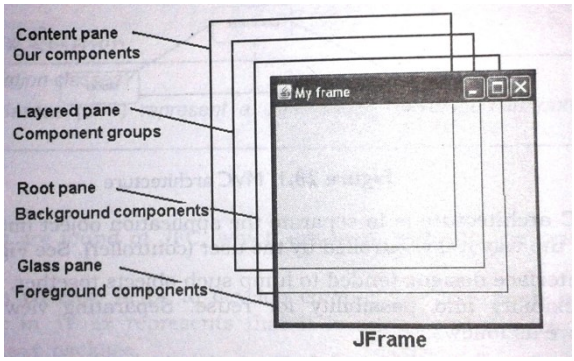
As the name implies, a top-level container must be at the top of a containment hierarchy. A top-level container is not contained within any other container.

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:

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 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
```

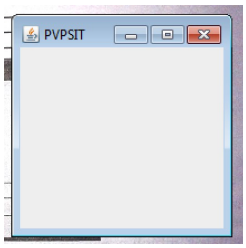
To close the frame, use setDefaultCloseOperation() method of JFrame class

```
setDefaultCloseOperation(constant)
```

where constant values are

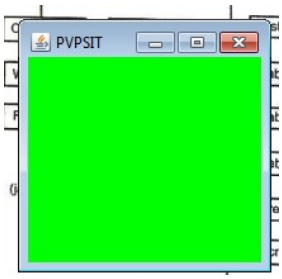
JFrame.EXIT_ON_CLOSE	This closes the application upon clicking the close button
JFrame.DISPOSE_ON_CLOSE	This closes the application upon clicking the close button
JFrame.DO_NOTHING_ON_CLOSE	This will not perform any operation upon clicking close button
JFrame.HIDE_ON_CLOSE	This hides the frame upon clicking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

JLabel:

- Jlabel is used to display a text
  - JLabel(string str)
  - JLabel(Icon i)
  - JLabel(String s, Icon i, int align)
    - CENTER, LEFT, RIGHT, LEADING, TRAILING
- Icon – is an interface
  - The easiest way to obtain icon is to use ImageIcon class. ImageIcon class implements Icon interface.

Important Methods:

Icon getIcon()

String getText()

void setIcon(Icon icon)

void setText(String s)

JText 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 javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class MyFrame extends JFrame implements ActionListener
{
   JLabel jl, jl2;
   JTextField jtf;
   MyFrame()
   {
```

```
        setLayout(new FlowLayout());
        jl=new JLabel("Enter your name");
        jl2=new JLabel();
        jtf=new JTextField("PVPSIT",15);


        add(jl);
        add(jtf);
        add(jl2);
        jtf.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        jl2.setText(jtf.getText());
    }
}
class FrameDemo
{
    public static void main(String arg[])
    {
        MyFrame f=new MyFrame();
        f.setTitle("Welcome to Swings");
        f.setSize(500,500);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

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 javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class MyFrame extends JFrame implements ActionListener
{
    JButton jb,jb1,jb2;
    JLabel jl;
    MyFrame()
    {
        setLayout(new FlowLayout());
        jl=new JLabel();
```

```
        jb=new JButton("VRSEC");
        ImageIcon ii=new ImageIcon("pvp.JPG");
        jb1=new JButton("PVPSIT",ii);

        ImageIcon ii2=new ImageIcon("bec.JPG");
        jb2=new JButton("BEC", ii2);

        add(jb);   add(jb1); add(jb2); add(jl);

        jb.addActionListener(this);
        jb1.addActionListener(this);
        jb2.addActionListener(this);
    }
    public void actionPerformed(ActionEvent ae)
    {
        jl.setText("You Pressed:  "+ae.getActionCommand());
    }
}
class FrameDemo
{
    public static void main(String arg[])
    {
        MyFrame f=new MyFrame();
        f.setTitle("Welcome to Swings");
        f.setSize(500,500);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

JCheckBox:

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 javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class MyFrame extends JFrame  implements ItemListener
{
    JCheckBox jcb,jcb1,jcb2;
    JLabel jl;
    MyFrame()
    {
        setLayout(new FlowLayout());
        jl=new JLabel();

        jcb=new JCheckBox("VRSEC");
        jcb1=new JCheckBox("PVPSIT");
        jcb2=new JCheckBox("BEC" );

        add(jcb); add(jcb1);  add(jcb2);  add(jl);

        jcb.addItemListener(this);
        jcb1.addItemListener(this);
        jcb2.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie)
    {
        JCheckBox jc=(JCheckBox)ie.getItem();
        jl.setText("You Selected  :"+jc.getText() );
    }
}
class FrameDemo
{
    public static void main(String arg[])
    {
        MyFrame f=new MyFrame();
        f.setTitle("Welcome to Swings");
        f.setSize(500,500);
        f.setVisible(true);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

JRadioButton:

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 returns the text that is associated with a radio button and uses it to set the text field.

Example:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class MyFrame extends JFrame implements ActionListener
{
    JRadioButton jrb,jrb1,jrb2;
    JLabel jl;
    MyFrame()
    {
        setLayout(new FlowLayout());
        jl=new JLabel();

        jrb=new JRadioButton("VRSEC");
        jrb1=new JRadioButton("PVPSIT");
        jrb2=new JRadioButton("BEC" );

        add(jrb); add(jrb1);  add(jrb2);  add(jl);

        ButtonGroup bg=new ButtonGroup();
        bg.add(jrb);   bg.add(jrb1);   bg.add(jrb2);
```

```
                jrb.addActionListener(this);
                jrb1.addActionListener(this);
                jrb2.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae)
        {
            jl.setText("You Selected  :"+ae.getActionCommand());
        }
    }
    class FrameDemo
    {
        public static void main(String arg[])
        {
            MyFrame f=new MyFrame();
            f.setTitle("Welcome to Swings");
            f.setSize(500,500);
            f.setVisible(true);
            f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
    }
```

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.

Example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class MyFrame extends JFrame implements ItemListener
{
	JComboBox jcb;
	MyFrame()
	{
		setLayout(new FlowLayout());
		String cities[]={"Amaravati","Guntur","Vijayawada","Vizag","Kurnool"};

		jcb=new JComboBox(cities);
		jcb.addItem("Tirupati");
		jcb.setEditable(true);
		add(jcb);
		jcb.addItemListener(this);
	}
	public void itemStateChanged(ItemEvent ie)
	{
		JOptionPane.showMessageDialog(null,jcb.getSelectedItem());
	}
}
public class JComboBoxDemo
{
	public static void main(String[] args)
	{
		MyFrame jf = new MyFrame();
		jf.setSize(500,500);
		jf.setVisible(true);
		jf.setTitle("Frame Example");
		jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
	 }
}
```

JList:

- JList class is useful to create a list which displays a list of items and allows the user to select one or more items.
  - Constructors
    - JList()
    - JList(Object arr[])
    - JList(Vector v)
  - Methods
    - getSelectedIndex() – returns selected item index
    - getSelectedValue() – to know which item is selected in the list
    - getSelectedIndices() – returns selected items into an array
    - getSelectedValues() – returns selected items names into an array

- JList generates **ListSelectionEvent**
  - ListSelectionListener
    - void valueChanged(ListSelectionEvent)
  - Package is javax.swing.event.*;

Example:

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

class MyFrame extends JFrame  implements ListSelectionListener
{
        JLabel jl;
        JList j;
        MyFrame()
        {
                setLayout(new FlowLayout());
                jl=new JLabel("Choose one college..");

                String arr[]={"BEC", "PVPSIT","RVR&JC", "VRSEC"};

                j=new JList(arr);
                add(jl);
                add(j);
                j.setToolTipText("I am PVPSIT");
                j.addListSelectionListener(this);
        }
        public void valueChanged(ListSelectionEvent le)
        {
                JOptionPane.showMessageDialog(null, j.getSelectedValue());
        }
}
class FrameDemo2
{
        public static void main(String arg[])
        {
                MyFrame f=new MyFrame();
                f.setTitle("Welcome to Swings");
                f.setSize(500,500);
                f.setVisible(true);
                f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
}
```