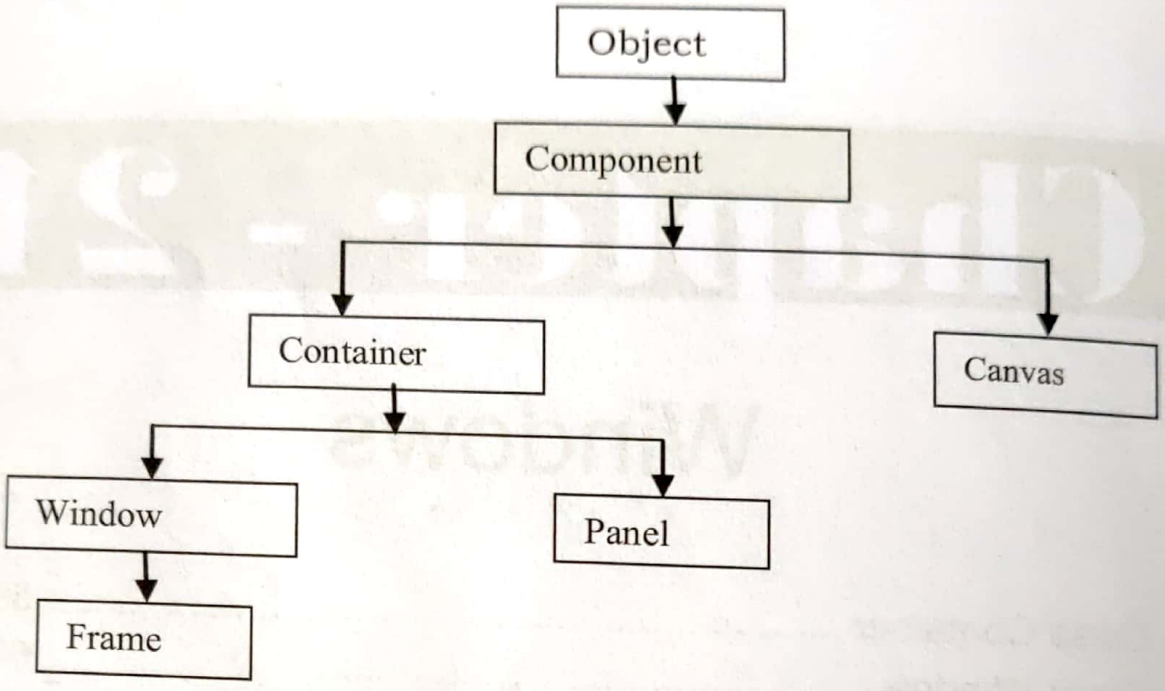
Chapter - 21

Windows

Windows

The AWT of Java allows to develop stand alone windows which are also called as applications. A window is a rectangular area on the screen where we can place components to design GUI application. In Java, window application can be developed using Window class or Frame class or Panel class. The hierarchy of these classes are:



Hierarchy of Frame, Panel and Canvas classes.

Class: Container

Container class is a subclass of **Component**. **Container** is an object that can contain other AWT components.

Components added to a container are tracked in a list. The order of the list will define the components' front-to-back stacking order within the container. If no index is specified when adding a component to a container, it will be added to the end of the list (and hence to the bottom of the stacking order).

Class: Window

Window class is a subclass of **Container** class. A **Window** object is a top-level window with no borders and no menubar. The default layout for a window is **BorderLayout**.

A window must have either a frame, dialog, or another window defined as its owner when it's constructed.

Class: Frame

Frame class is a subclass of **Window**. A **Frame** is a top-level window with a title bar, borders and re-sizing corners. Frame is a stand alone window which executes on the desktop of the computer. The size of the frame includes any area designated for the border. The default layout for a window is **BorderLayout**.

Class: Panel

Panel class is a subclass of **Container** class. Panel is the simplest container class. **Panel** is similar to **Frame** but panel does not contain borders, title bar, re-sizing corners. Panel requires other container to expose itself i.e. panel is a child window which opens in another window. A panel provides space in which an application can attach any other component, including other panels.

The default layout manager for a panel is the **FlowLayout** layout manager.

Applet class inherits from **Panel** class that is which applet window does not have border, title bar. Applet opens in browser which acts as a container for applets.

Class: Canvas

The **Canvas** class is a subclass of **Component** class. A **Canvas** component represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user.

An application must subclass the **Canvas** class in order to get useful functionality such as creating a custom component. The **paint()** method must be overridden in order to perform custom graphics on the canvas.

Creation of User-defined Windows

Up to now we have created applets which are web-applications that opens only in web-browsers(Internet Explorer, Netscape Navigator, etc.) and requires client-server architecture.

We can also create window application that executes on the desktop of PC. A window is a stand alone application which is a parent window that does not require client-server architecture. A window is a rectangular area on the screen which acts as an interfacer between user and application.

Window is a container on which we can place other components to design GUI application.

In Java, user-define windows can be created extending **Frame** class. A Frame window is one which have title bar, borders, re-sizing corners. A Frame window is a top-level window. The default layout for a frame window is **BorderLayout**.

The following are the constructors of **Frame** class.

Constructors	Meaning
Frame()	This default constructor creates a new instance of Frame window that is initially invisible. The title of the Frame is empty.
Frame(String title)	This one-argument constructor creates a new instance of Frame window that is initially invisible with the specified title.

The following table shows methods of **Frame** class.

Methods	Meaning
Image **getIconImage**()	Returns the image to be displayed as the icon for this frame.
MenuBar **getMenuBar**()	Gets the menu bar for this frame.
int **getState**()	Gets the state of this frame (obsolete).
String **getTitle**()	Gets the title of the frame.
boolean **isResizable**()	Indicates whether this frame is resizable by the user.
void **setIconImage**(Image image)	Sets the image to be displayed as the icon for this window.
void **setMaximizedBounds**(Rectangle bounds)	Sets the maximized bounds for this frame.
void **setMenuBar**(MenuBar mb)	Sets the menu bar for this frame to the specified menu bar.
void **setResizable**(boolean resizable)	Sets whether this frame is resizable by the user.
void getTitle()	Sets the state of this frame (obsolete).

void **setTitle**(String title)	Sets the title for this frame to the specified string.
viod setSize(int width,int height)	Sets width and height of the window.
void setVisible(boolean flag)	If flag is true then window is visible on the screen otherwise hidden
void dispose()	closes frame window and de-allocates memory of window.
void paint(Graphics g)	paint on the window container

getTitle()

This method of Frame class can be used to get the title of the frame. The title is displayed in the frame's border.

public String **getTitle**()

setTitle()

This method of Frame class can be used to set the title for the frame window to the specified title string.

public void **setTitle**(String title)

setMenuBar()

This method of Frame class can be used to set the menu bar to the frame window of given menu bar.

public void **setMenuBar**(MenuBar mb)

Few methods are shown below that are inherited from Window class into Frame.

setSize()

This method of Window class can be used to set the size of the frame window to the specified width and height. The width and height values are automatically changed as the window size is re-sized at runtime.

public void **setSize**(int width,int height)

setVisible()

This method of Window class can be used to show or hide the Window.

public void **setVisible**(boolean flag)

If flag is **true** then window is shown on the screen and hides if the flag is **false**.

paint()

This method of Window class can be used to paint on the window container.

public void **paint**(Graphics g)

dispose()

This method of Window class can be used to close the window and releases all of the native screen resources used by this Window, its subcomponents, and all of its owned children. That is, the resources for these Components will be destroyed, any memory they consume will be returned to the OS, and they will be marked as undisplayable.

public void **dispose**()

Frames are capable of generating the following types of WindowEvents:

WINDOW_OPENED
WINDOW_CLOSING
WINDOW_CLOSED
WINDOW_ICONIFIED
WINDOW_DEICONIFIED
WINDOW_ACTIVATED
WINDOW_DEACTIVATED
WINDOW_GAINED_FOCUS
WINDOW_LOST_FOCUS
WINDOW_STATE_CHANGED

Class WindowEvent

A low-level event that indicates that a window has changed its status. This low-level event is generated by a Window object when it is opened, closed, activated, deactivated, iconified, or deiconified, or when focus is transferred into or out of the Window.

The event is passed to every WindowListener or WindowAdapter object which registered to receive such events using the window's addWindowListener method. (WindowAdapter objects implement the WindowListener interface.) Each such listener object gets this WindowEvent when the event occurs.

The following table shows fields defined in the **WindowEvent** class.

Fields	Meaning
static int **WINDOW_ACTIVATED**	The window-activated event type.
static int **WINDOW_CLOSED**	The window closed event.
static int **WINDOW_CLOSING**	The "window is closing" event.
static int **WINDOW_DEACTIVATED**	The window-deactivated event type.
static int **WINDOW_DEICONIFIED**	The window deiconified event type.
static int **WINDOW_GAINED_FOCUS**	The window-gained-focus event type.
static int **WINDOW_ICONIFIED**	The window iconified event.
static int **WINDOW_LOST_FOCUS**	The window-lost-focus event type.
static int **WINDOW_OPENED**	The window opened event.
static int **WINDOW_STATE_CHANGED**	The window-state-changed event type.

The following table shows methods defined in the **WindowEvent** class.

Methods	Meaning
int **getNewState**()	For WINDOW_STATE_CHANGED events returns the new state of the window.
int **getOldState**()	For WINDOW_STATE_CHANGED events returns the previous state of the window.
Window **getOppositeWindow**()	Returns the other Window involved in this focus or activation change.
Window **getWindow**()	Returns the originator of the event.
String **paramString**()	Returns a parameter string identifying this event.

Interface WindowListener

The **WindowListener** interface can be used for receiving window events. The class that is interested in processing a window event either implements this interface (i.e. all the methods it contains) or extends the abstract **WindowAdapter** class (overriding only the methods on interest). The listener object created from that class is then registered with a Window using the window's **addWindowListener()** method. When the window's status changes because of being opened, closed, activated or deactivated, iconified or deiconified, the relevant method in the listener object is invoked, and the **WindowEvent** is passed to it.

The following table shows abstract methods present in **WindowListener** interface.

Methods	Meaning
void **windowActivated**(WindowEvent e)	Invoked when the Window is set to be the active Window.
void **windowClosed**(WindowEvent e)	Invoked when a window has been closed as the result of calling dispose on the window.
void **windowClosing**(WindowEvent e)	Invoked when the user attempts to close the window from the window's system menu or by clicking on close button.
void **windowDeactivated**(WindowEvent e)	Invoked when a Window is no longer the active Window.
void **windowDeiconified**(WindowEvent e)	Invoked when a window is changed from a minimized to a normal state.
void **windowIconified**(WindowEvent e)	Invoked when a window is changed from a normal to a minimized state.
void **windowOpened**(WindowEvent e)	Invoked the first time a window is made visible.

Class WindowAdapter

The **WindowAdapter** class is implemented from **WindowListener** interface which can be used for receiving window events. The **WindowAdapter** class overrides all methods of interface as dummy(empty) methods. This class exists as convenience for creating listener objects.

Any class that extends this class to create a **WindowEvent** listener can override only those methods for events that have to be handled.

Note: The methods of this class are same as **WindowListener**.

A program to create a user-defined window

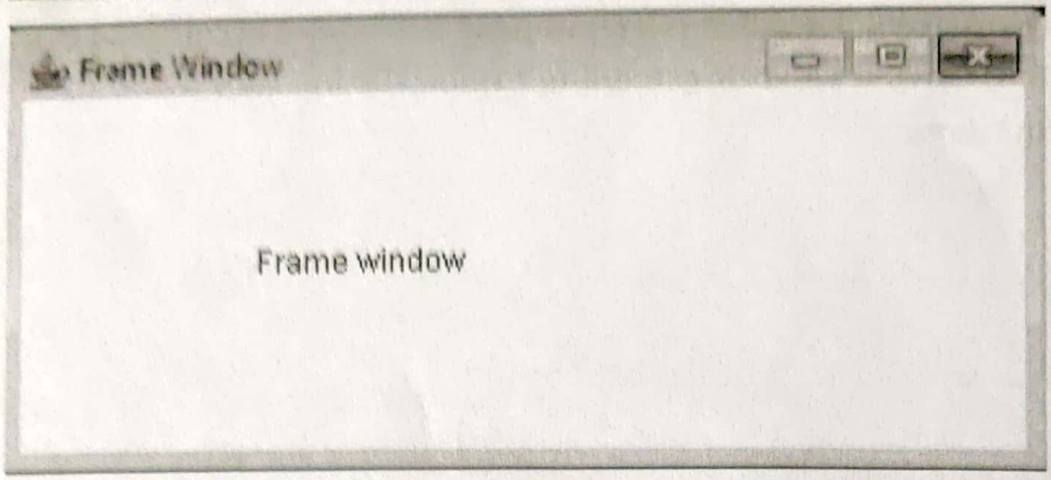
This program creates a window extending Frame class.

Bin> edit win1.java

```
import java.awt.*;
import java.awt.event.*;
class mywindow extends Frame
{
    public mywindow(String title)
    {
        super(title);
        addWindowListener(new mywindowhandler());
    }
    public void paint(Graphics g)
    {
        g.drawString("Frame window", 100,100);
    }
    class mywindowhandler extends WindowAdapter
    {
        public void windowClosing(WindowEvent e)
        {
            dispose();
        }
    }
}

class win1
{
    public static void main(String argv[])
    {
        mywindow w=new mywindow("Frame Window");
        w.setSize(400,400);
        w.setVisible(true);
    }
}
```

Bin>javac win1.java

Bin>java win1



Explanation:

In the above program, the frame window class **mywindow** is created by extending from **Frame** class. In the **mywindow (String title)** constructor, **super(title)** calls the one-argument constructor of **Frame** class which actually creates a frame window object with the given title.

The **addWindowListener(new mywindowhandler());** method registers the window to receive window events and these events are directed to the object of **mywindowhandler** class.

The **mywindowhandler** class extends from **WindowAdapter** class and overrides **windowClosing(WindowEvent w)** method which is executed when close button is clicked and the **dispose()** method closes the window.

The **paint()** is executed when the window is displayed and draws the string in window.

In the main(), an object of **mywindow** class is instantiated by passing **"Frame Window"** as title. the **w.setSize(400,400);** method sets the size of the window to the specified dimensions. The **w.setVisible(true);** method makes the window on the screen.

A program to draw Lines and rectangles on frame window.

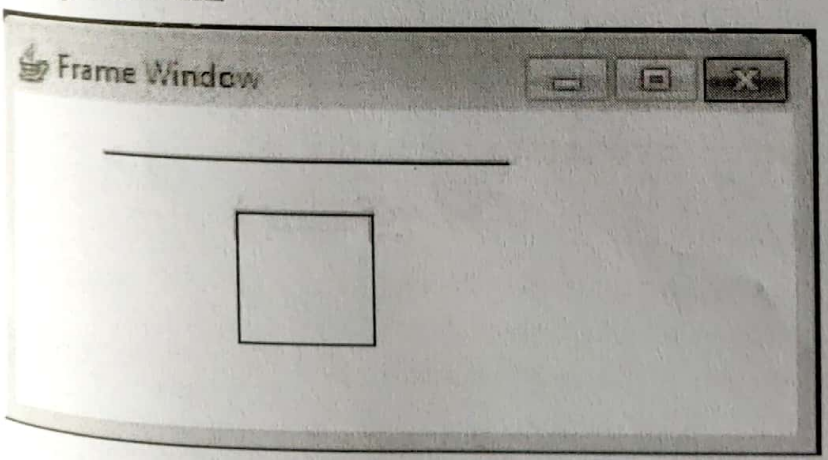
Bin>edit win2.java

```
import java.awt.*;
import java.awt.event.*;
class mywindow extends Frame
{
    public mywindow(String title)
    {
        super(title);
```

```
        addWindowListener(new mywindowhandler());
    }
    public void paint(Graphics g)
    {
        g.drawLine(50,50,200,50);
        g.drawRect(100,70,50,50);
    }

    class mywindowhandler extends WindowAdapter
    {
        public void windowClosing(WindowEvent e)
        {
            dispose();
        }
    }
}

class win2
{
    public static void main(String argv[])
    {
        mywindow w=new mywindow("Frame Window");
        w.setSize(400,400);
        w.setVisible(true);
    }
}
```

Bin>javac win2.java

Bin>java win2



A program on simple calculator to perform addition, subtraction using components.

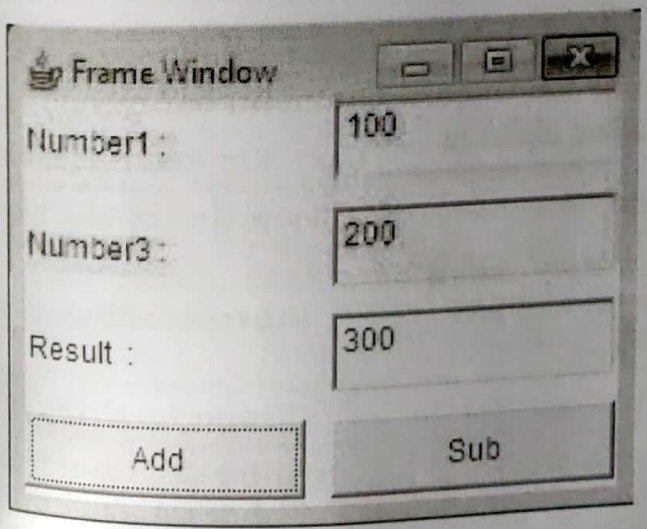
Bin>edit win3.java

```
import java.awt.*;
import java.awt.event.*;
class mywindow extends Frame implements ActionListener
{
    Label l1,l2,l3;
    TextField t1,t2,t3;
    Button b1,b2;
    public mywindow(String title)
    {
          super(title);
          GridLayout gl=new GridLayout(4,2,10,10);
          setLayout(gl);
          l1=new Label("Number1 :");
          l2=new Label("Number3 :");
          l3=new Label("Result  :");
          t1=new TextField(10);
          t2=new TextField(10);
          t3=new TextField(10);
          b1=new Button("Add");
          b2=new Button("Sub");
          add(l1);add(t1);
          add(l2);add(t2);
          add(l3);add(t3);
          add(b1);add(b2);
          b1.addActionListener(this);
          b2.addActionListener(this);
          addWindowListener(new mywindowhandler());
    }

    public void actionPerformed(ActionEvent e)
    {
          int a=0,b=0,c=0;
          a=Integer.parseInt(t1.getText());
          b=Integer.parseInt(t2.getText());
          String str=e.getActionCommand();
          if(str.equals("Add"))
                c=a+b;
```

```
            else if(str.equals("Sub"))
                  c=a-b;

            t3.setText(""+c);
      }

      class mywindowhandler extends WindowAdapter
      {
            public void windowClosing(WindowEvent e)
            {
                  dispose();
            }
      }
}

class win3
{
      public static void main(String argv[])
      {
            mywindow w=new mywindow("Frame Window");
            w.setSize(400,400);
            w.setVisible(true);
      }
}
```

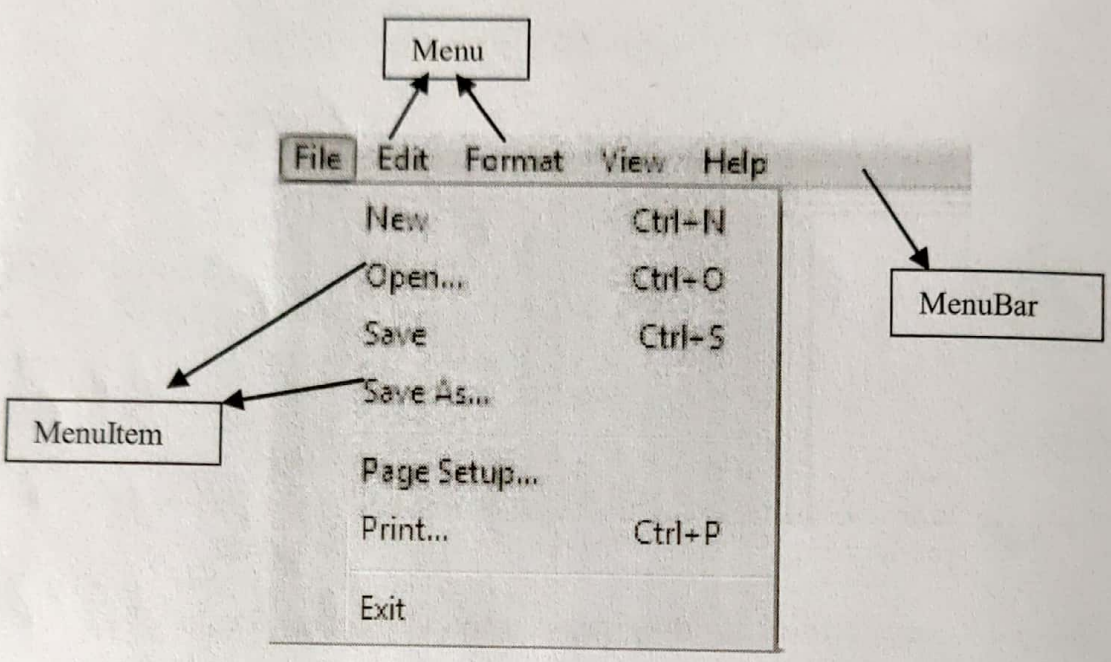
Bin>javac win3.java

Bin>java win3



Creation of Menu

Menu is a set of commands that is displayed at the top of window. Menu can be used to perform operations in the window. One window can have one menu. The following picture shows the menu and its various properties.



Menu bar can be created using **MenuBar** class. Menus are created using **Menu** class. Menu items are created using **MenuItem** class.

The **setMenuBar()** method of **Frame** class can be used to set the menu bar to the window. The signature of this method is:

public void setMenuBar(MenuBar mb);

Class MenuBar

This class can be used to create menu bar. On menu bar menus can be placed. The constructor of MenuBar class is

MenuBar()

This constructor creates an empty menu bar.

Some of the methods of **MenuBar** class.

Methods	Meaning
Menu **add**(Menu m)	Adds the specified menu to the menu bar.
Menu **getMenu**(int i)	Gets the specified menu.
int **getMenuCount**()	Gets the number of menus on the menu bar.
void **remove**(int index)	Removes the menu located at the specified index from this menu bar.
void **remove**(MenuComponent m)	Removes the specified menu component from this menu bar.
Enumeration< MenuShortcut> **shortcuts**()	Gets an enumeration of all menu shortcuts this menu bar is managing.

Class Menu

Menu class can be used to create menu which can be a collection of menu items. A Menu object is a pull-down menu component that is deployed from a menu bar.

The following table shows constructors of **Menu** class.

Constructors	Meaning
Menu()	Creates a new menu with an empty label.
Menu(String label)	Creates a new menu with the specified label.
Menu(String label, boolean tearOff)	Creates a new menu with the specified label, indicating whether the menu can be torn off.

The following table shows some of methods of **Menu** class.

Methods	Meaning
MenuItem **add**(MenuItem mi)	Adds the specified menu item to this menu.
void **add**(String label)	Adds an item with the specified label to this menu.
void **addSeparator**()	Adds a separator line, or a hypen, to the menu at the current position.
int **getItemCount**()	Get the number of items in this menu.
void **insert**(MenuItem menuitem, int index)	Inserts a menu item into this menu at the specified position.

void **insert**(String label, int index)	Inserts a menu item with the specified label into this menu at the specified position.
void **remove**(int index)	Removes the menu item at the specified index from this menu.
void **remove**(MenuComponent item)	Removes the specified menu item from this menu.
void **removeAll**()	Removes all items from this menu.

Class MenuItem

This class can be used to create menu items. These items are the commands that perform operations. These items generate ActionEvent therefore these items should be registered with listener to receive events.

The following table shows constructors of **MenuItem** class.

Constructors	Meaning
MenuItem()	Constructs a new MenuItem with an empty label and no keyboard shortcut.
MenuItem(String label)	Constructs a new MenuItem with the specified label and no keyboard shortcut.
MenuItem(String label, MenuShortcut s)	Create a menu item with an associated keyboard shortcut.

The following table shows some of the methods of **MenuItem** class.

Methods	Meaning
void **addActionListener**(ActionListener l)	Adds the specified action listener to receive action events from this menu item.
String **getActionCommand**()	Gets the command name of the action event that is fired by this menu item.
String **getLabel**()	Gets the label for this menu item.
boolean **isEnabled**()	Checks whether this menu item is enabled.
void **setEnabled**(boolean b)	Sets whether or not this menu item can be chosen.
void **setLabel**(String label)	Sets the label for this menu item to the specified label.
void **setShortcut**(MenuShortcut s)	Set the MenuShortcut object associated with this menu item.

A program to create menu and adds it to the frame window.

In this program, a menu bar is created with two menus(Draw and Colors). Draw menu is added with three menu items (Line, Rect and Exit) and Colors menu is added with two menu items(Yellow and Green). These menu items are registered with **ActionListener** to receive events. When these menu items are selected, accordingly the operations are performed.

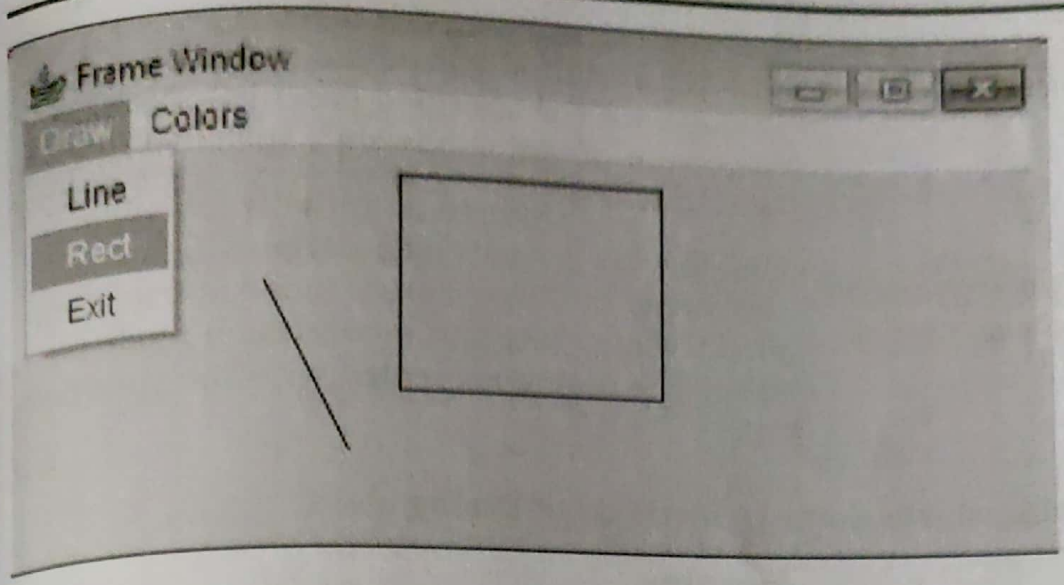
Bin>edit menu1.java

```
import java.awt.*;
import java.awt.event.*;
class mywindow extends Frame implements ActionListener
{
    MenuBar mb;
    public mywindow(String title)
    {
        super(title);
        mb=new MenuBar();
        setMenuBar(mb);

        Menu m1,m2;
        m1=new Menu("Draw");
        m2=new Menu("Colors");

        MenuItem i1,i2,i3,i4,i5;
        i1=new MenuItem("Line");
        i2=new MenuItem("Rect");
        i3=new MenuItem("Exit");
        i4=new MenuItem("Yellow");
        i5=new MenuItem("Green");

        m1.add(i1);m1.add(i2);m1.add(i3);
        m2.add(i4);m2.add(i5);

        mb.add(m1);mb.add(m2);

        i1.addActionListener(this);
        i2.addActionListener(this);
        i3.addActionListener(this);
        i4.addActionListener(this);
        i5.addActionListener(this);
        addWindowListener(new mywindowhandler());
    }
```

```
    public void actionPerformed(ActionEvent  e)
    {
          String str=e.getActionCommand();
          Graphics g=getGraphics();
          if(str.equals("Line"))
                g.drawLine(100,100,200,300);
          else if(str.equals("Rect"))
                g.drawRect(150,60,100,80);
          else if(str.equals("Yellow"))
                setBackground(Color.yellow);
          else if(str.equals("Green"))
                setBackground(Color.green);
          else if(str.equals("Exit"))
                System.exit(1);
    }

    class mywindowhandler extends WindowAdapter
    {
          public void windowClosing(WindowEvent e)
          {
                dispose();
          }
    }
}

class menu1
{
    public static void main(String argv[])
    {
          mywindow w=new mywindow("Frame Window");
          w.setSize(400,400);
          w.setVisible(true);
    }
}
```

Bin>javac menu1.java

Bin>java menu1



Creating Dialogs

The Dialog class of java.awt package can be used to create dialog window. A dialog window is a top-level window with a title and a border that is typically used to take some form of input from the user. Dialog window does not have the facility of re-sizing the dimensions at runtime. It is often child window of a top level-window. Dialog window does not have menu bar and is similar to frame window.

A dialog can be either modeless (the default) or modal. A modal dialog is one when it is active all the input is focused to it and other windows can not be activated until it is closed. A mode-less dialog is one when it is active we can activate other windows.

Dialog class is a subclass of Window class therefore methods of Window class also available to Dialog.

User-defined dialog box window class can be defined extending Dialog class.

The following are the constructors of **Dialog** class.

Constructors	Meaning
Dialog(Frame parentwindow)	Creates an initially invisible, modeless Dialog with the specified parentwindow as the owner of dialog box with empty title.
Dialog(Frame parentwindow, boolean mode)	Creates an initially invisible Dialog with the specified parentwindow as the owner of dialog box. If mode is true then modal dialog box and mode-less if mode is false. The dialog is created with empty title.

Dialog(Frame parentwindow, String title)	Creates an initially invisible, modeless Dialog with the specified parentwindow as the owner of dialog box with specified title.
Dialog(Frame parentwindow, String title, boolean mode)	Creates an initially invisible Dialog with the specified parentwindow as the owner of dialog box. If mode is true then modal dialog box and mode-less if mode is false. The dialog is created with the specified title.

The following table shows some of methods of **Dialog** class.

Methods	Meaning
String **getTitle**()	Gets the title of the dialog.
boolean **isModal**()	Indicates whether the dialog is modal.
void **setModal**(oolean modal)	Specifies whether this dialog should be modal.
void **setTitle**(String title)	Sets the title of the Dialog.
void **setVisible**(boolean flag)	Shows or hides this Dialog depending on the value of parameter flag. If true shows and hides if false.

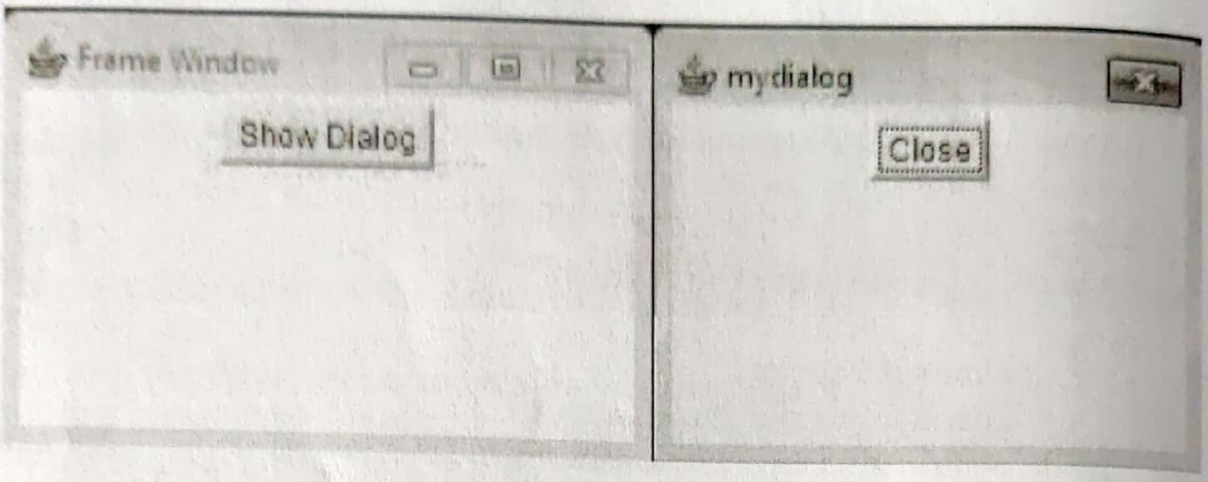
A program to demonstrate Dialog class.

Bin>edit dilaog1.java

```
import java.awt.*;
import java.awt.event.*;
class mydialog extends Dialog implements ActionListener
{
    Button b;
    public mydialog(Frame parent,String title)
    {
        super(parent,title,true);
        setSize(300,300);
        setLayout(new FlowLayout());
        b=new Button("Close");
        add(b);
        b.addActionListener(this);
    }
```

```
    public void actionPerformed(ActionEvent e)
    {
          dispose();
    }
}
class mywindow extends Frame implements ActionListener
{
    Button b;
    public mywindow(String title)
    {
          super(title);
          setLayout(new FlowLayout());
          b=new Button("Show Dialog");
          add(b);
          b.addActionListener(this);
          addWindowListener(new mywindowhandler());
    }
    public void actionPerformed(ActionEvent e)
    {
          mydialog d=new mydialog(this,"mydialog");
                    // this references to object of mywindow
          d.setVisible(true);
    }
    class mywindowhandler extends WindowAdapter
    {
          public void windowClosing(WindowEvent e)
          {
                dispose();
          }
    }
}

class dialog1
{
    public static void main(String argv[])
    {
          mywindow w=new mywindow("Frame Window");
          w.setSize(400,400);
          w.setVisible(true);
    }
}
```

```
Bin>javac dialog1.java
Bin>java dialog1
```



Class FileDialog

The **Filedialog** class extends from **Dialog** class which can be used to display built-in dialog boxes such as **open file dialog** or **save file dialog**. These dialog boxes can be used to specify or select a file name. These dialog boxes are the standard dialog boxes provided by the operating system to give a new name for saving or to select existing filename to open. These dialog boxes are modal dialog boxes, therefore when they are active other applications can not be activated.

The following table shows fields of **FileDialog** class that specifies which dialog box to show.

Fields	Meaning
static int **LOAD**	This constant value indicates that the purpose of the file dialog window is to locate a file from which to read.
static int **SAVE**	This constant value indicates that the purpose of the file dialog window is to locate a file to which to write.

The following are the constructors of **OpenFileDialog** class that can create open or save dialog boxes object.

Constructors	Meaning
FileDialog(Frame parent)	Creates a file dialog for loading a file

FileDialog(Frame parent, String title)	Creates a file dialog window with the specified title for loading a file. The files shown are those in the current directory. This is a convenience method for FileDialog(parent, title, LOAD).
FileDialog(Frame parent, String title, int mode)	Creates a file dialog window with the specified title for loading or saving a file. If the value of mode is **OpenFileDialog.LOAD**, then the file dialog is finding a file to read, and the files shown are those in the current directory. If the value of mode is **OpenFileDialog.SAVE**, the file dialog is finding a place to write a file

The following table shows methods of **OpenFileDialog** class.

Methods	Meaning
String **getDirectory**()	Gets the directory of this file dialog.
String **getFile**()	Gets the selected file of this file dialog.
int **getMode**()	Indicates whether this file dialog box is for loading from a file or for saving to a file.
void **setDirectory**(String dir)	Sets the directory of this file dialog window to be the specified directory.
void **setFile**(String file)	Sets the selected file for this file dialog window to be the specified file.
void **setMode**(int mode)	Sets the mode of the file dialog

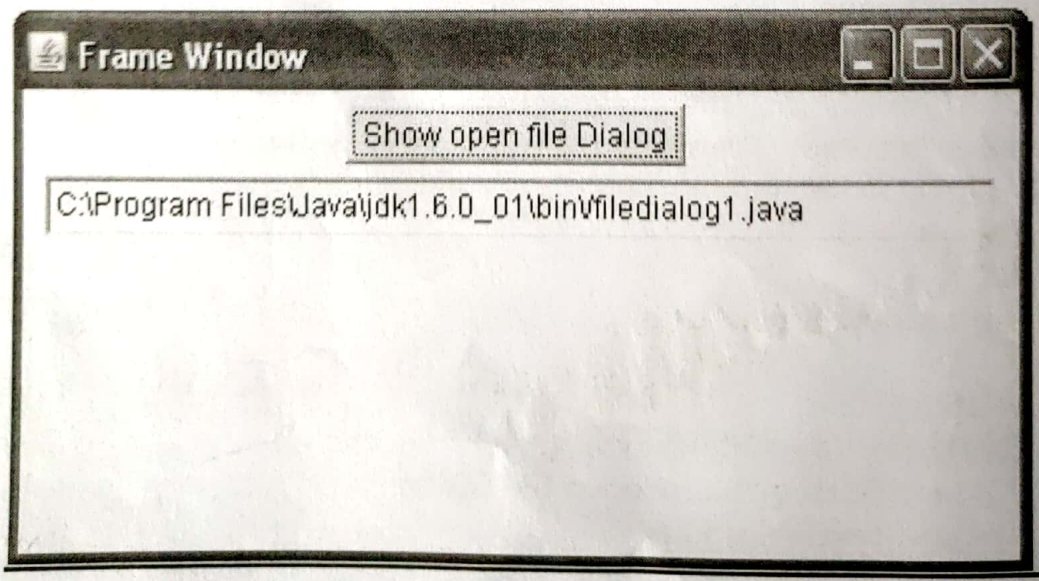
Bin>edit filedialog1.java

```
import java.awt.*;
import java.awt.event.*;
class mywindow extends Frame implements ActionListener
{
    Button b;
    TextField t;
    public mywindow(String title)
    {
        super(title);
        setLayout(new FlowLayout());
        b=new Button("Show open file Dialog");
        t=new TextField(50);
```

```
            add(b);
            add(t);
            b.addActionListener(this);
            addWindowListener(new mywindowhandler());
    }
    public void actionPerformed(ActionEvent e)
    {
            FileDialog f=new FileDialog(this,"Open",FileDialog.LOAD);
            f.setVisible(true);
            t.setText(""+f.getDirectory()+"/"+f.getFile());
    }
    class mywindowhandler extends WindowAdapter
    {
            public void windowClosing(WindowEvent e)
            {
                    dispose();
            }
    }
}
class filedialog1
{
    public static void main(String argv[])
    {
            mywindow w=new mywindow("Frame Window");
            w.setSize(400,400);
            w.setVisible(true);
    }
}
```

Bin>javac filedialog1.java

Bin>java filedialog1



Chapter - 22

SWINGS

Swings

The **AWT** contains limited GUI components(**Label**, **Button**, **TextField**, **Checkbox**, **Choice**, **List**, **TextArea** and **Scrollbar**) which we have studied in previous chapters of this book. The **AWT** components can not be used to develop complete and powerful GUI applications. So, to develop complete and powerful GUI application Java introduced an extension of **AWT** and it is know as **Swings**. Swing is an extension of AWT. Swing is a set of classes that can be used to develop more powerful and flexible GUI applications than the AWT.

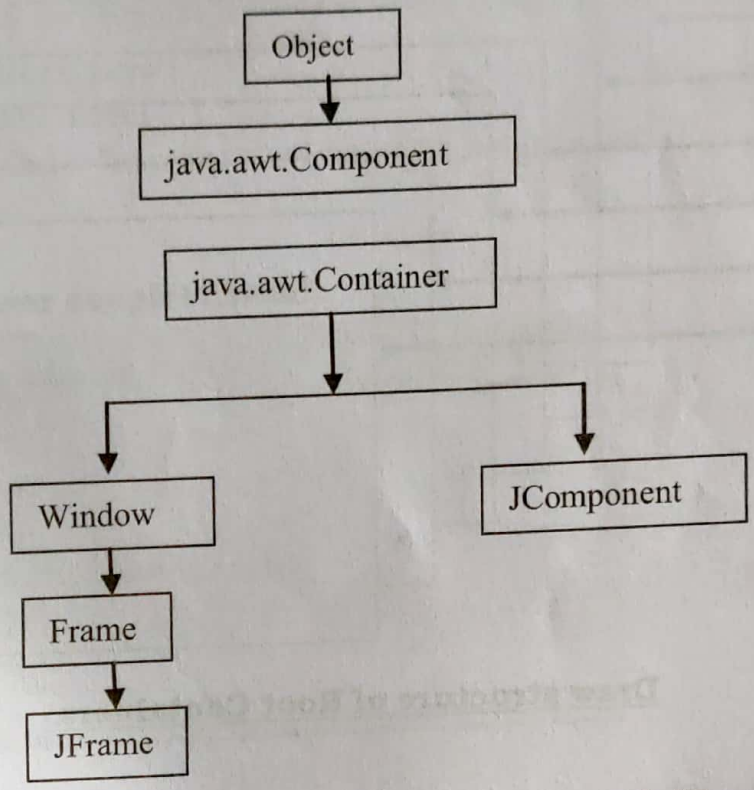
AWT components when executed in different operating systems have different appearance because AWT components use operating system's user interface (i.e. graphics). Therefore AWT components are called as ***heavy weight components***.

Swing components when executed in different operating systems have same look and feel i.e. appearance is same because Swing components does not use operating system's user interface. Swings are developed purely in java and use its own user interface to produce the graphics for the components. Therefore swing components are called as ***light weight components***.

The differences between **AWT** and **Swing** are:

AWT	Swing
1. AWT was introduced in Java before Swings. Therefore it contains old style of components.	1. Swing was introduced after AWT and therefore it contains latest style of components.
2. AWT contains limited components.	2. Swing contains extended components.
3. AWT uses operating systems user interface to produce graphics for the components.	3. Swings have its own user interface classes to produce graphics for the components.
4. AWT application appearance changes from operating system to operating system.	4. Swing application appearance is same in all operating systems.
5. AWT components are not portable components.	5. Swing components are portable components.
6. AWT components are heavy weight components.	6. Swing components are light weight components.
7. AWT contains limited components.	7. Swing contains extended components.

The Swing classes are subclasses of **java.awt.Container** and **java.awt.Component**. The name of the Swing class starts with the letter **J**. The super most class of Swing is **JComponent**. The **JComponent** is both a container and a component. GUI components like **JButton**, **JLabel** , **JCheckBox** , **JTextField**, **JPanel** ect., are sub classes of **JComponent** class. GUI components can be added on a panel window or a frame window. The panel window can be defined using **JFrame** class and frame class can be defined using **JFrame** class. The **JComponent** class and AWT class hierarchy as shown below.



Class Hierarchy of AWT, JComponent and JFrame

Class JComponent

The **JComponent** class is a subclass of **Container** class. The components such as **JButton**, **JLabel**, **JTextField**, etc., are the subclasses of **JComponent**. These classes can be used to develop GUI applications.

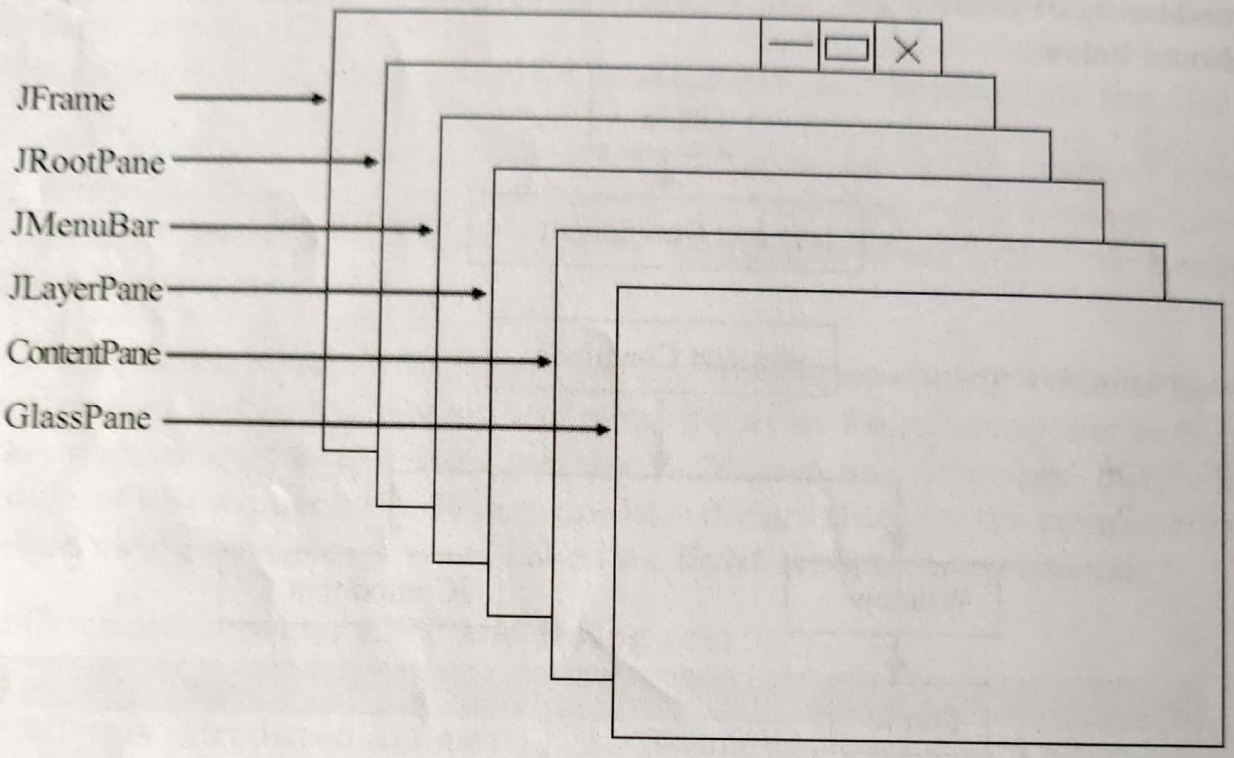
The different component classes present in swings are:

JLabel	JButton	JTextField	JTextArea
JCheckBox	JRadioButton	JList	JScrollPane
JTree	JComboBox	JMenuBar	JTable
JTabbedPane	JPanel	JFrame	JOptionPane

Containers

In Swings top-level window is created by **JFrame**, **JApplet**, **JDialog** and **JWindow** classes. These are called root containers. These root containers have several panes such as

JRootPane, **JMenuBar**, **JLayeredPane**, **ContentPane** and **GlassPane** as shown in following:



Draw structure of Root Containers

Class JApplet

JApplet is one of the top-level containers because it is not inherited from **JComponent**. **JApplet** extends from Applet class of AWT and all the features of Applet are inherited into **JApplet** and it also contains the features of Swing. Swing applets can be developed extending **JApplet** class of **javax.swing** package.

Swing applets uses the same life-cycle as that of an AWT applet. The life-cycle methods of applet are : **init()**, **start()**, **stop()** and **destroy()**. However **paint()** method can be used to perform drawing in applet.

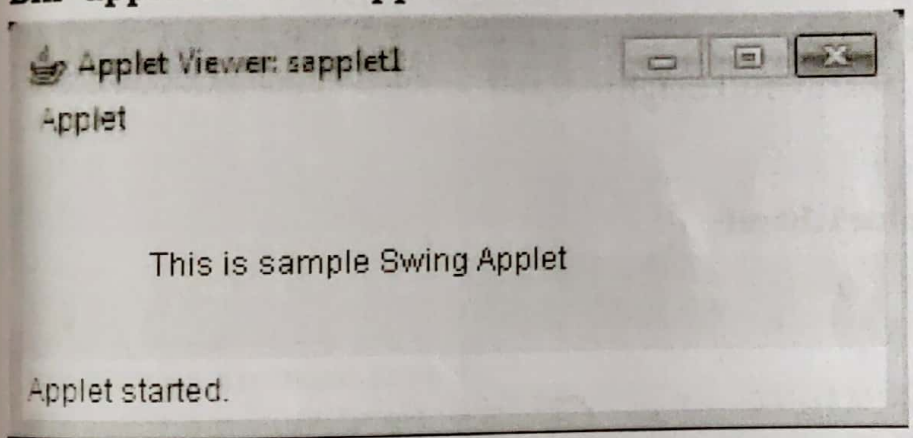
A program on Swing Applet
Bin>edit sapplet1.java

```
import java.awt.*;
import javax.swing.*;
public class sapplet1 extends JApplet
{
    public void paint(Graphics g)
    {
        g.drawString("This is sample Swing Applet", 50, 50);
    }
}
```

Bin>javac sapplet1.java
Bin>edit sapplet1.html

```
<applet code="sapplet1" width=400 height=400>
</applet>
```

Bin>appletviewer sapplet1.html



A swing applet program that sets back and fore colors and also prints width and height of an applet.

Bin>edit scolor1.java

```
import java.awt.*;
import javax.swing.*;
public class scolor1 extends JApplet
{
    private Container Panel;
    public scolor1 ()
    {
        super ();
        Panel = getContentPane();
        Panel.setBackground (Color.cyan);
```

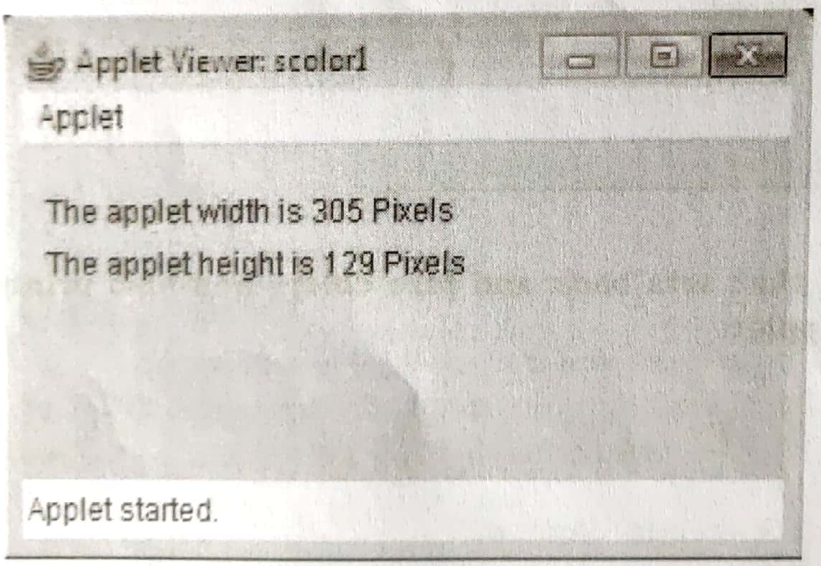
```
    }
    public void paint (Graphics g)
    {
        int Width;
        int Height;
        super.paint (g);
        Width = getWidth();
        Height = getHeight();
        g.drawString ("The applet width is " + Width +
                      " Pixels", 10, 30);
        g.drawString ("The applet height is " + Height +
                      " Pixels", 10, 50);
    }
}
```

Bin>javac scolor1.java

Bin>edit scolor1.html

```
<applet code="scolor1" width=400 height=400>
</applet>
```

Bin>appletviewer scrolor1.html



Note: Execute all AWT Applet programs using Swing **JApplet**.

A program to demonstrate mouse events.

Bin>edit smouse.java

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

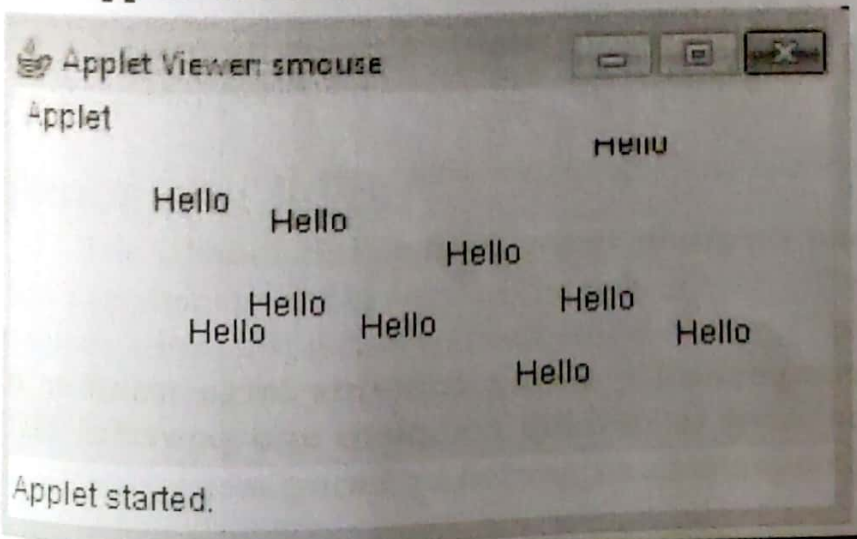
```
import javax.swing.*;
public class smouse extends JApplet
{
    String str="Hello";
    int x=0,y=0;
    public void init()
    {
        addMouseListener(new mymousehandler());
    }
    public void paint(Graphics g)
    {
        g.drawString(str,x,y);
    }
    class mymousehandler extends MouseAdapter
    {
        public void mouseClicked(MouseEvent m)
        {
            x=m.getX();
            y=m.getY();
            repaint();
        }
    }
}
```

Bin>javac smouse.java

Bin>edit smouse.html

<applet code="smouse" width=400 height=400>
</applet>

Bin>appletviewer smouse.html



A program to draw lines

Bin>edit sline1.java

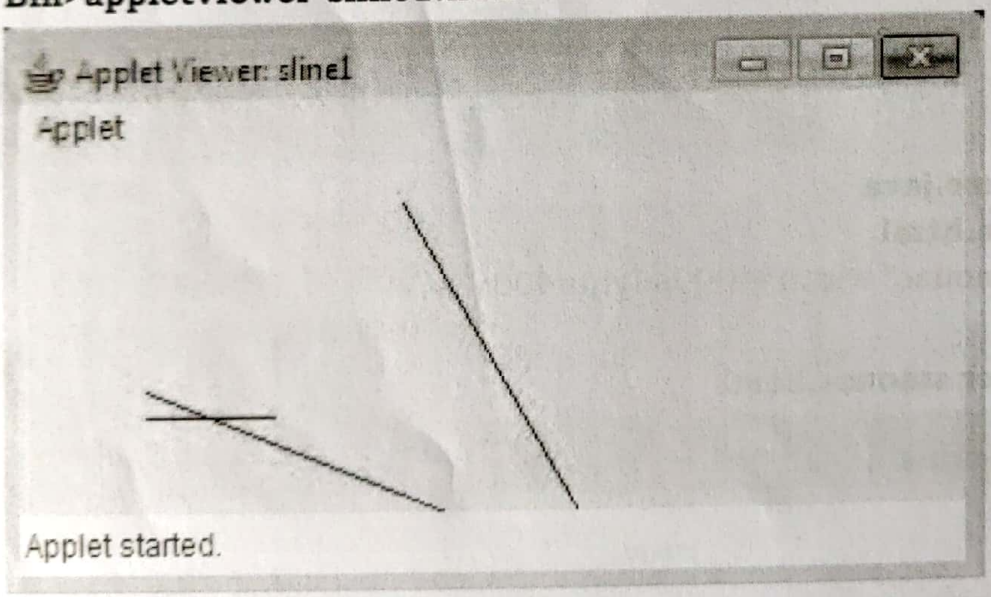
```
import java.awt.*;
import javax.swing.*;
public class sline1 extends JApplet
{
    public void paint(Graphics g)
    {
        g.drawLine(100,100,50,100);
        g.drawLine(50,90,200,150);
        g.drawLine(150,20,250,190);
    }
}
```

Bin>javac sline1.java

Bin>edit sline1.html

```
<applet code="sline1" width=400 height=400>
</applet>
```

Bin>appletviewer sline1.html



Note: Execute AWT graphics programs using Swings.

Swing Components

Swing is rich in components. Swing contains large number of components which can be used to develop complete and powerful GUI applications. Some of the components supported by Swing are:

JLabel	JButton	JTextField	JTextArea
JCheckBox	JRadioButton	JPasswordField	JList
JComboBox	JScrollPane	JTabbedPane	JTable
JTree	JToggleButton	JMenu	

All the above component classes are derived from **JComponent** class therefore all these components are lightweight.

Class ImageIcon

The **ImageIcon** class is implemented from Icon interface. This class can be used to load images from the given path and the image can be painted on the components by passing the **ImageIcon** object as an argument to the component.

The following are the constructors that loads images.

Constructors	Meaning
ImageIcon(Image image)	Creates an ImageIcon from an image object.
ImageIcon(String filename)	Creates an ImageIcon from the specified file.
ImageIcon(URL location)	Creates an ImageIcon from the specified URL.

Example:

```
ImageIcon i=new ImageIcon("c:/hand.gif");
```

Creates an object **i** of **ImageIcon** and is loaded with **hand.gif** image from the given path.

Component : JLabel

The **JLabel** class can be used to create label component. A **JLabel** object can display either text, an image, or both. The text can be changed by the application, but a user cannot edit it directly. Label controls do not raise events therefore they need not be registered with any events.

The following constructors are used to create object of labels.

Constructors	Meaning
JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(Icon image)	Creates a JLabel instance with the specified image.
JLabel(Icon image, int alignment)	Creates a JLabel instance with the specified image and alignment.
JLabel(String text)	Creates a JLabel instance with the specified text.
JLabel(String text, Icon icon, int alignment)	Creates a JLabel instance with the specified text, image, and alignment.
JLabel(String text, int alignment)	Creates a JLabel instance with the specified text and alignment.

The alignment can be **JLabel.LEFT**, **JLabel.RIGHT**, **JLabel.CENTER**, **JLabel.LEADING or JLabel.TRAILING**

Some of the methods of **JLabel** class are:

Methods	Meaning
String getText()	returns text from the label
void setText(String text)	sets given text on the label
Icon getIcon()	returns Icon object associated with label.
void setIcon(Icon i)	sets an image to the label of given Icon object i

A program to demonstrate JLabel class.

Bin>edit slabel1.java

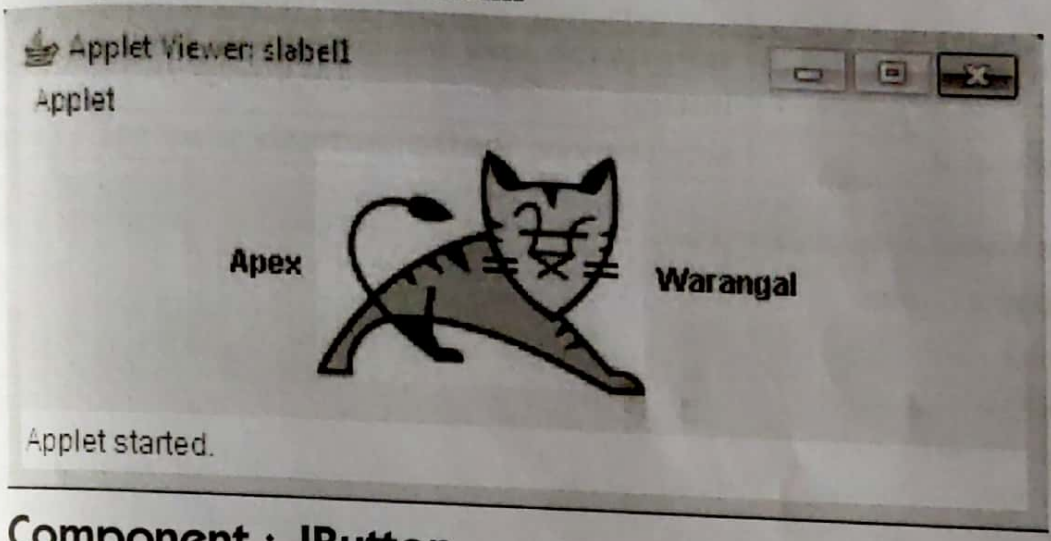
```
import javax.swing.*;
import java.awt.*;
public class slabel1 extends JApplet
{
    JLabel l1,l2,l3;
    public void init()
    {
        setLayout(new FlowLayout(FlowLayout.CENTER));
        l1=new JLabel("Apex");
        ImageIcon i=new ImageIcon("tomcat.gif");
```

```
        l2=new JLabel(i);
        l3=new JLabel("Warangal");
        add(l1);
        add(l2);
        add(l3);
    }
}
```

Bin>javac slabel1.java
Bin>edit slabel1.html

```
<applet code="slabel1" width=400 height=400>
</applet>
```

Bin>appletviewer slabel1.html



Component : JButton

The Button class can be used to create push button. When button is selected it fires an operation. The button component fires an **ActionEvent** when it is selected using mouse or keyboard. The button component should be registered with **ActionListener** using **addActionListener()** method so that it receives the **ActionEvent** raised on button.

The following are the constructors of **JButton** class which can be used to create push button objects.

Constructors	Meaning
JButton()	Creates a button with no set text or icon.
JButton(Icon icon)	Creates a button with an icon.
JButton(String text)	Creates a button with text.
JButton(String text, Icon icon)	Creates a button with initial text and an icon.

The following are the methods of **JButton** class.

Methods	Meaning
void setText(String text)	sets the given text on the button.
String getText()	returns text present on the button.
String getActionCommand()	returns button text or command that is set using setActionCommand().
void SetActionCommand(String command)	sets the command to the button which returns the same when getActionCommand() is called.
void addActionListener(ActionListener al)	Adds the specified action listener to receive action from this button.
Icon getIcon()	returns the icon associated with the button.
void setIcon(Icon i)	sets the icon for the button.

A program to demonstate JButton.

Bin>edit sbutton1.java

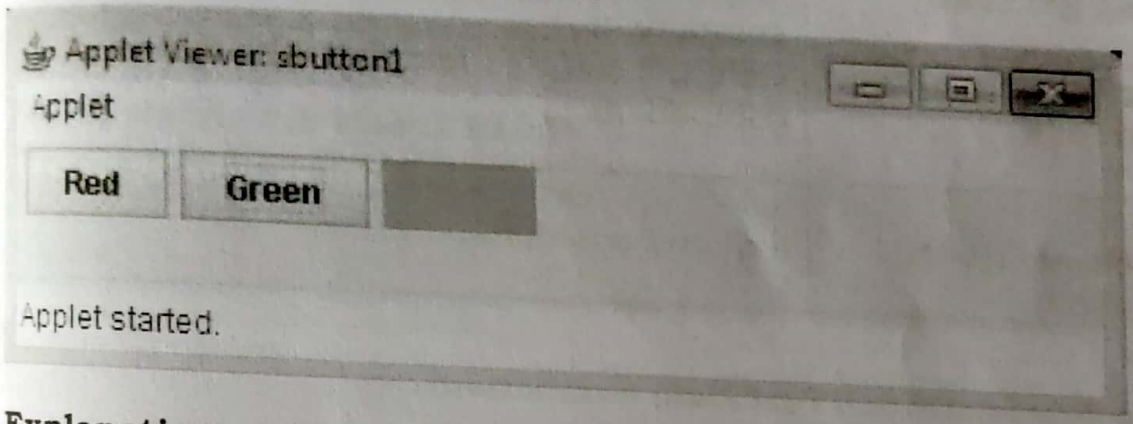
```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
public class sbutton1 extends JApplet implements ActionListener
{
    JButton b1,b2;
    JLabel l1;
    JPanel p;
    public void init()
    {
          setLayout(new FlowLayout(FlowLayout.LEFT));
          b1=new JButton("Red");
          b2=new JButton("Green");
          p=new JPanel();
          l1=new JLabel("                    ");
          p.add(l1);
          p.setBackground(Color.white);
          add(b1);add(b2);add(p);
          b1.addActionListener(this);
          b2.addActionListener(this);
    }
```

```
    public void actionPerformed(ActionEvent e)
    {
          String str=e.getActionCommand();
          if(str.equals("Red"))
               p.setBackground(Color.red);
          else if(str.equals("Green"))
               p.setBackground(Color.green);
    }
}
```

Bin>javac sbutton1.java
Bin>edit sbutton1.html

```
<applet code="sbutton1" width=400 height=400>
</applet
```

Bin>appletviewer sbutton1.html



Explanation:

The **JLabel** component is transparent and there is not effect of **setBackground()** method on it. To apply the background color, take a **JPanel** and add a label to it and then apply back color to the panel using the **setBackground()** method.

Component : JTextField

The **JTextField** class can be used to create textbox component. The **JTextField** component takes the input from the user and gives the output. A text component allows for the editing of a single line of text.

Constructors of **JTextField** that allows to create textbox component.

Constructors	Meaning
JTextField()	Constructs a new TextField.
JTextField(int columns)	Constructs a new empty TextField with the specified number of columns.
JTextField(String text)	Constructs a new TextField initialized with the specified text.
JTextField(String text, int columns)	Constructs a new TextField initialized with the specified text and columns.

The following are the methods supported by **JTextField** class.

Methods	Meaning
public void **setText**(**String** text)	Sets the given text to the text field component.
public **String getText**()	Gets the text present in the text field component.
public **String getSelectedText**()	Gets the selected text from the text component.
public void **select**(int selectionStart, int selectionEnd)	select the text between the specified start and end positions.

A program to demonstrate JTextField component.

Bin>edit stext1.java

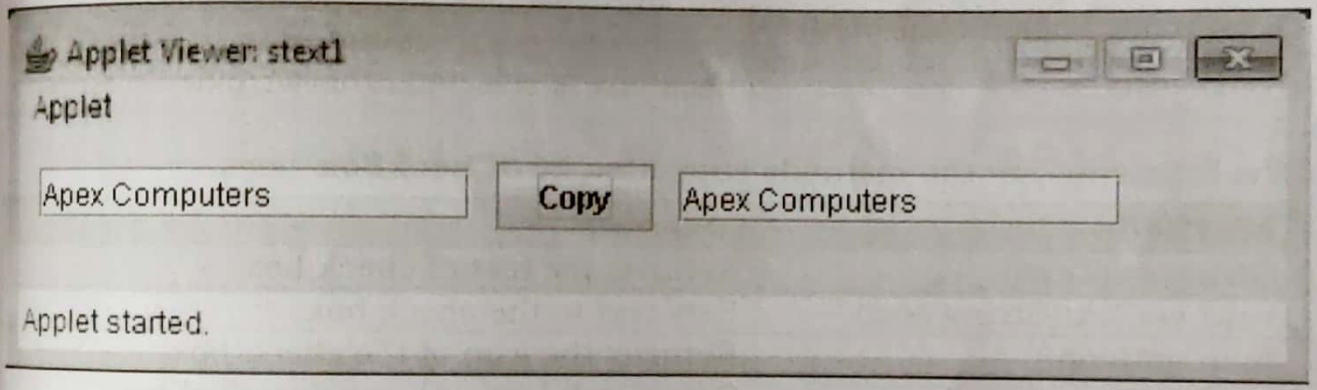
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class stext1 extends JApplet implements ActionListener
{
    JTextField t1,t2;
    JButton b1;
    public void init()
    {
        setLayout(new FlowLayout(FlowLayout.LEFT,10,10));
        t1=new JTextField(15);
        t2=new JTextField(15);
        b1=new JButton("Copy");
        add(t1);add(b1);add(t2);
```

```
        b1.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        String str=e.getActionCommand();
        if(str.equals("Copy"))
        {
            String s=t1.getText();
            t2.setText(s);
        }
    }
}
```

Bin>javac stext1.java
Bin>edit stext1.html

```
<applet code="stext1" width=400 height=400>
</applet>
```

Bin>appletviewer stext1.html



Component : JCheckBox

The **JCheckbox** class of **javax.swing** package can be used to create checkbox component. This component allows to select or de-select an option. More than one checkbox can be selected or de-selected. Clicking on a checkbox changes its state from "on" (true) to "off" (false), or from "off" to "on."

When a **Checkbox** component is selected or de-selected either using keyboard or mouse, the checkbox object generates an **ItemEvent**. Therefore checkbox component object should be registered to receive **ItemEvent** and this can be done using the **addItemListener()** method. When the **ItemEvent** is received by the **ItemListener** it immediately calls **itemStateChanged()** method by passing an argument of **ItemEvent** class.

Constructors of **JCheckBox** that allows to create checkbox component.

Constructors	Meaning
JCheckBox(Icon icon)	Creates an initially unselected check box with an icon.
JCheckBox(Icon icon, boolean selected)	Creates a check box with an icon and specifies whether or not it is initially selected. If **selected** is *true* then check box is selected otherwise de-selected.
JCheckBox(String text)	Creates an initially unselected check box with text.
JCheckBox(String text, boolean selected)	Creates a check box with text and specifies whether or not it is initially selected. If **selected** is *true* then check box is selected otherwise de-selected.
JCheckBox(String text, Icon icon)	Creates an initially unselected check box with the specified text and icon.
JCheckBox(String text, Icon icon, boolean selected)	Creates a check box with text and icon, and specifies whether or not it is initially selected. If **selected** is *true* then check box is selected otherwise de-selected.

The following are the methods supported by **JCheckBox** class.

Methods	Meaning
String getText()	Returns the text of check box.
void setText(String text)	Sets text to the check box.
Icon getIcon()	Returns the icon of the check box.
void setIcon(Icon i)	Sets check box with the specified icon object i
boolean isSelected()	Returns *true* if check box is selected otherwise *false*.
void setSelected(boolean state)	If the state is *true* then check box is selected otherwise de-selected.

A program to demonstrate JCheckBox component.

Bin>edit scheck1.java

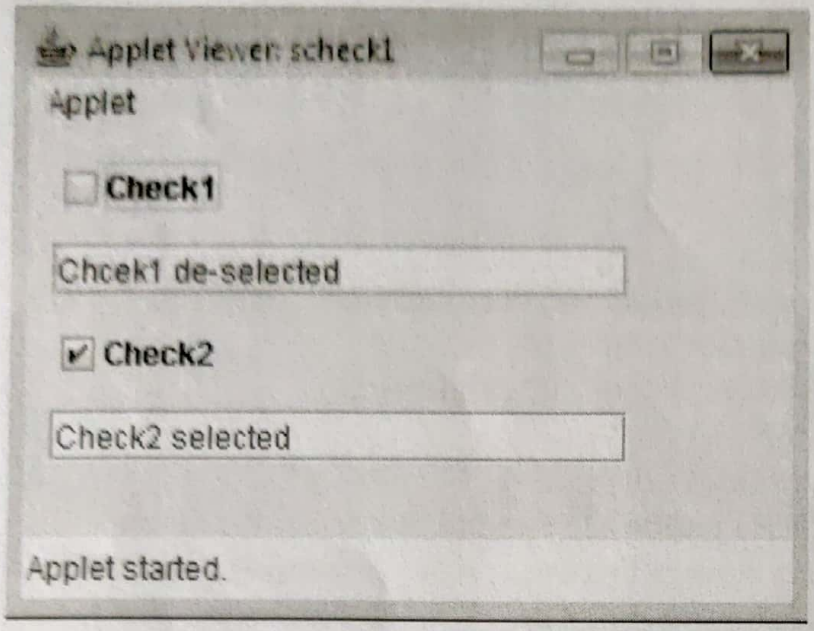
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

```
public class scheck1 extends JApplet implements ItemListener
{
    JCheckBox c1,c2;
    JTextField t1,t2;
    public void init()
    {
        setLayout(new FlowLayout(FlowLayout.LEFT,10,10));
        c1=new JCheckBox("Check1",true);
        c2=new JCheckBox("Check2",false);
        t1=new JTextField(20);
        t2=new JTextField(20);
        add(c1);add(t1);add(c2);add(t2);
        c1.addItemListener(this);
        c2.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent i)
    {
        if(c1.isSelected()==true)
            t1.setText("Check1 selected");
        else
            t1.setText("Chcek1 de-selected");

        if(c2.isSelected()==true)
            t2.setText("Check2 selected");
        else
            t2.setText("Chcek2 de-selected");
    }
}
```

Bin>javac scheck1.java
Bin>edit scheck1.html

```
<applet code="scheck1" width=400 height=400>
</applet>
```

Bin>appletviewer scheck1.html



Component : JRadioButton

The **RadioButton** class can be used to create radio button component. This component is similar to checkbox component with a difference that at any given time only one radio button can be selected. Radio buttons should be grouped so that from the group at any given time only one radio button can be selected. An application can have any number of radio buttons group. The radio buttons can be grouped using the **ButtonGroup** class. The **add()** method of **ButtonGroup** class can be used to add radio buttons to **ButtonGroup** object.

Radio buttons generates **ActionEvent**, **ItemEvent** or **ChangeEvent**, therefore radio buttons should be registered with the listeners.

A program to demonstrate JRadioButton component.

Bin>edit sradio1.java

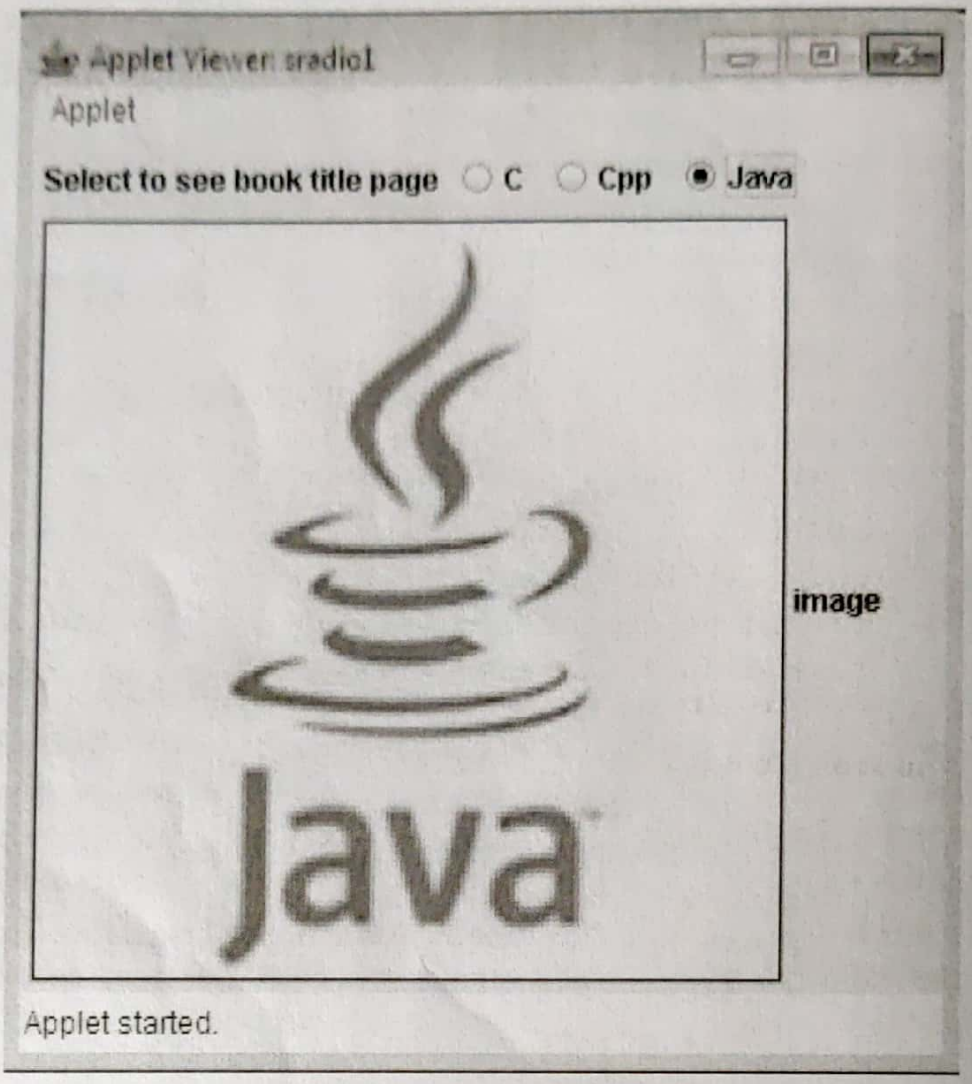
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class sradio1 extends JApplet implements ActionListener
{
    JRadioButton r1,r2,r3;
    JLabel l1,l2;
    ButtonGroup bg;
    public void init()
    {
        setLayout(new FlowLayout(FlowLayout.LEFT));
```

```
        bg=new ButtonGroup();
        l1=new JLabel("Select to see book title page");
        l2=new JLabel("image");
        r1=new JRadioButton("C",true);
        r2=new JRadioButton("Cpp",true);
        r3=new JRadioButton("Java",true);
        bg.add(r1);
        bg.add(r2);
        bg.add(r3);
        add(l1);
        add(r1);add(r2);add(r3);
        add(l2);
        r1.addActionListener(this);
        r2.addActionListener(this);
        r3.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        String str=e.getActionCommand();
        if(str.equals("C"))
          l2.setIcon(new ImageIcon("c.jpg"));
        else if(str.equals("cpp"))
          l2.setIcon(new ImageIcon("Cpp.jpg"));
        else if(str.equals("Java"))
          l2.setIcon(new ImageIcon("java-logo.jpg"));
    }
}
```

Bin>javac sradio1.java
Bin>edit sradio1.html

```
<applet code="sradio1" width=400 height=400>
</applet>
```

Bin>appletviewer sradio1.html



Component: JComboBox

The **JComboBox** class can be used to create a drop-down list. It is a collection of items and at any given time only one item can be selected. When clicked on this component, it pops up a menu where it shows all items in it. This component shows large number of items in minimum space.

The **JComboBox** component should be registered with **ItemListener** using **addItemListener()** method so that it receives the events raised on combo box.

The following are the constructors of **JComboBox** class

Constructors	Meaning
JComboBox()	Creates a JComboBox with a default data model.
JComboBox(Object items[])	Creates a JComboBox that contains the elements in the specified array.

The following are some of the methods of **JComboBox** class.

Methods	Meaning
void **addActionListener**(ActionListener l)	Adds an ActionListener.
void **addItem**(Object anObject)	Adds an item to the item list.
void **addItemListener**(ItemListener aListener)	Adds an ItemListener.
Object **getItemAt**(int index)	Returns the list item at the specified index.
int **getItemCount**()	Returns the number of items in the list.
int **getSelectedIndex**()	Returns the first item in the list that matches the given item.
Object **getSelectedItem**()	Returns the current selected item.
void **removeAllItems**()	Removes all items from the item list.
void **removeItem**(Object anObject)	Removes an item from the item list.
void **removeItemAt**(int anIndex)	Removes the item at anIndex This method works only if the JComboBox uses a mutable data model.

A program to demonstrate JComboBox component.

Bin>edit schoice1.java

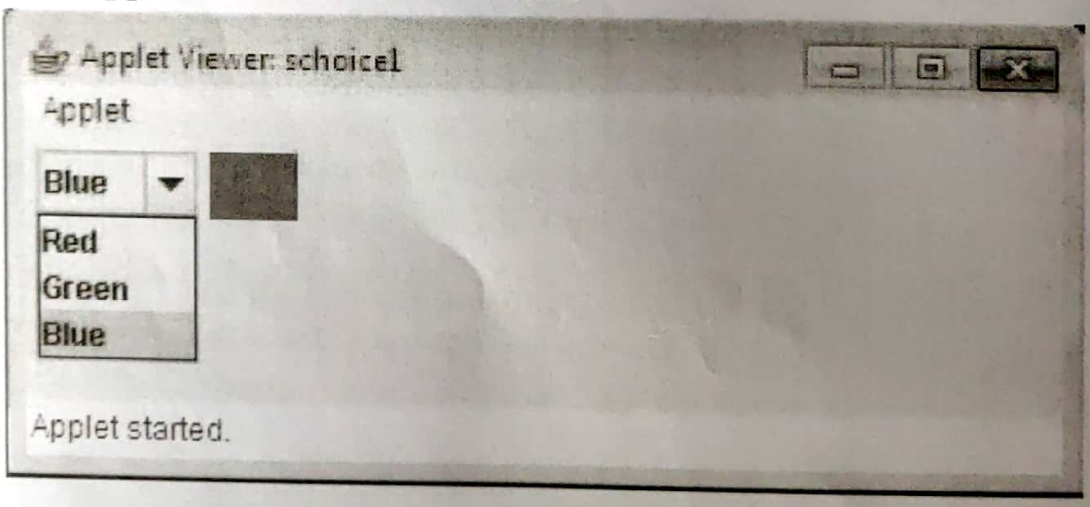
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class schoice1 extends JApplet implements ItemListener
{
    JComboBox c;
    JPanel p;
    public void init()
    {
        setLayout(new FlowLayout(FlowLayout.LEFT));
        c=new JComboBox();
        c.addItem("Red");
        c.addItem("Green");
        c.addItem("Blue");
```

```
            p=new JPanel();
            p.add(new JLabel("          "));
            add(c);
            add(p);
            c.addItemListener(this);
      }
      public void itemStateChanged(ItemEvent i)
      {
            String str=(String)c.getSelectedItem();
            if(str.equals("Red"))
                  p.setBackground(Color.red);
            else if(str.equals("Green"))
                  p.setBackground(Color.green);
            else if(str.equals("Blue"))
                  p.setBackground(Color.blue);
      }
}
```

Bin>javac schoice1.java
Bin>edit schoice1.html

```
<applet code="schoice1" width=400 height=400>
</applet>
```

Bin>appletviewer schoice1.html



Component : JScrollPane

The **JScrollPane** class can be used to create a rectangular area and into which a component can be added and viewed. The **JScrollPane** contains scrollbars which allows to scroll a big component.

The **JScrollPane** class is a sub class of **JComponent**.

Following are the constructors of **JScrollPane** class.

Constructors	Meaning
JScrollPane()	Creates an empty (no viewport view) JScrollPane where both horizontal and vertical scrollbars appear when needed.
JScrollPane(**Component** view)	Creates a JScrollPane that displays the contents of the specified component, where both horizontal and vertical scrollbars appear whenever the component's contents are larger than the view.
JScrollPane(**Component** view, int vsbPolicy, int hsbPolicy)	Creates a JScrollPane that displays the view component in a viewport whose view position can be controlled with a pair of scrollbars.
JScrollPane(int vsbPolicy, int hsbPolicy)	Creates an empty (no viewport view) JScrollPane with specified scrollbar policies.

Following are the public static final variables defined in **ScrollPaneConstants** interface. These constant variables can be specified for **vsbpolicy** and **hsbpolicy** in the above constructors.

Fields	Meaning
static **String**	HORIZONTAL_SCROLLBAR Identifies a horizonal scrollbar.
static int	HORIZONTAL_SCROLLBAR_ALWAYS Used to set the horizontal scroll bar policy so that horizontal scrollbars are always displayed.
static int	HORIZONTAL_SCROLLBAR_AS_NEEDED Used to set the horizontal scroll bar policy so that horizontal scrollbars are displayed only when needed.
static int	HORIZONTAL_SCROLLBAR_NEVER Used to set the horizontal scroll bar policy so that horizontal scrollbars are never displayed.
static **String**	VERTICAL_SCROLLBAR Identifies a vertical scrollbar.

static int	VERTICAL_SCROLLBAR_ALWAYS Used to set the vertical scroll bar policy so that vertical scrollbars are always displayed.
static int	VERTICAL_SCROLLBAR_AS_NEEDED Used to set the vertical scroll bar policy so that vertical scrollbars are displayed only when needed.
static int	VERTICAL_SCROLLBAR_NEVER Used to set the vertical scroll bar policy so that vertical scrollbars are never displayed.

A program to demonstrate JScrollPane.

In this program, we add a big image to the scroll pane so that the image can be scrolled.

Bin>edit scrollpane1.java

```
import java.awt.*;
import javax.swing.*;
public class scrollpane1 extends JFrame
{
  public scrollpane1 (  )
  {
       super ( "JScrollPane Demo" ) ;
       ImageIcon ii = new ImageIcon ( "sunset.jpg" ) ;
      JScrollPane jsp = new JScrollPane(new JLabel(ii));
      add ( jsp ) ;
      setSize ( 300,250 ) ;
      setVisible ( true ) ;
  }
  public static void main ( String args[ ] )
  {
     scrollpane1 obj=new scrollpane1 (  ) ;
  }
}
```

Bin>javac scrollpane1.java

Bin>java scrollpane1

Another example on JScrollPane.

In this program, to the panel we add large number of buttons and we add scroll bar to the panel using scroll pane.

Bin>edit scrollpane2.java

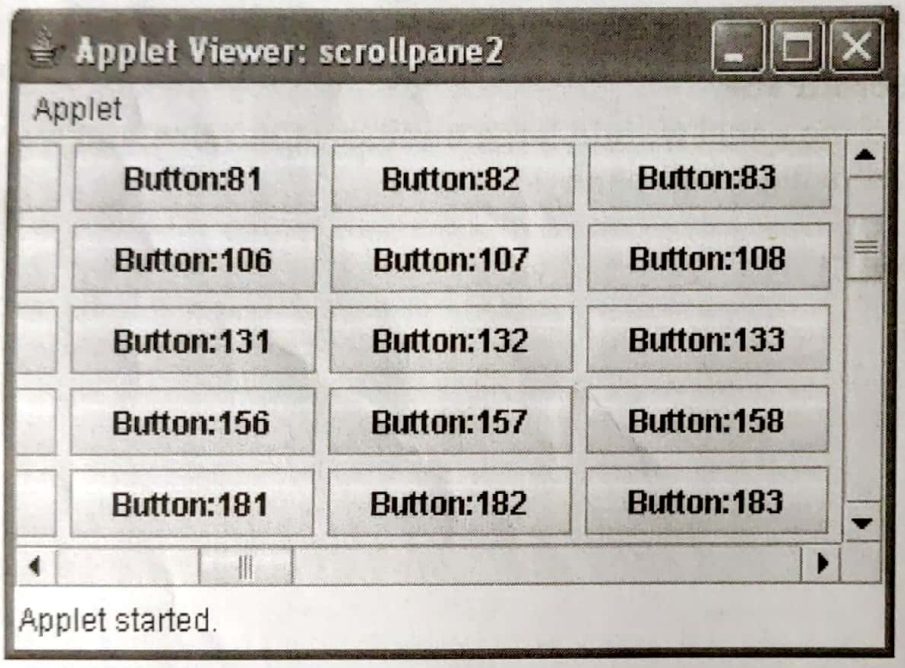
```
import java.awt.*;
import javax.swing.*;
public class scrollpane2 extends JApplet
{
    JScrollPane sp;
    JButton b;
    JPanel p;
    int n=1;
    public void init()
    {
        p=new JPanel();
        p.setLayout(new GridLayout(25,25,5,5));
        for(int i=1;i<=25;i++)
        {
            for(int j=1; j<=25; j++)
            {
                b=new JButton("Button:"+n++);
                p.add(b);
            }
        }
```

```
sp=new JScrollPane(p,ScrollPaneConstants.VERTICAL_SCROLLBAR_
AS_NEEDED, ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED);
add(sp);
    }
}
```

Bin>javac scrollpane2.java
Bin>edit scrollpane2.html

```
<applet code="scrollpane2" width=400 height=400>\
</applet>
```

Bin>appletviewer scrollpane2.html



Component: JList

The **JList** class of **javax.swing** can be used to create a list component. The list component is a collection of items similar to combo box. The list component occupies more space than choice. List component can show more than one item and remaining items can be selected by scrolling items. List allows to select one or more items at a time.

The list component should be registered with **ItemListener** or **ActionListener** using **addItemListener()** or **addActionListener()** method so that they receives the events raised on list.

Following are the constructors of **JList** class.

Constructors	Meaning
JList()	Creates a JList with an empty, read-only, model.
JList(Object[] listData)	Creates a JList that displays the elements in the specified array.

The following are some of the methods of **JList** class.

Methods	Meaning
void **addActionListener**(ActionListener l)	Adds an ActionListener.
void	**addItem**(Object anObject) Adds an item to the item list.
void	**addItemListener**(ItemListener aListener) Adds an ItemListener.
Object	**getItemAt**(int index) Returns the list item at the specified index.
int	**getItemCount**() Returns the number of items in the list.
int	**getSelectedIndex**() Returns the first item in the list that matches the given item.
Object	**getSelectedValue**() Returns the current selected item.
int[]	**getSelectedIndices()** Returns an array of all of the selected indices, in increasing order.
Object[]	**getSelectedValues()** Returns an array of all the selected values, in increasing order based on their indices in the list
void	**removeAllItems**() Removes all items from the item list.
void	**removeItem**(Object anObject) Removes an item from the item list.

void	**removeItemAt**(int anIndex) Removes the item at anIndex This method works only if the JComboBox uses a mutable data model.

A program to demonstate JList component.

Bin>edit slist1.java

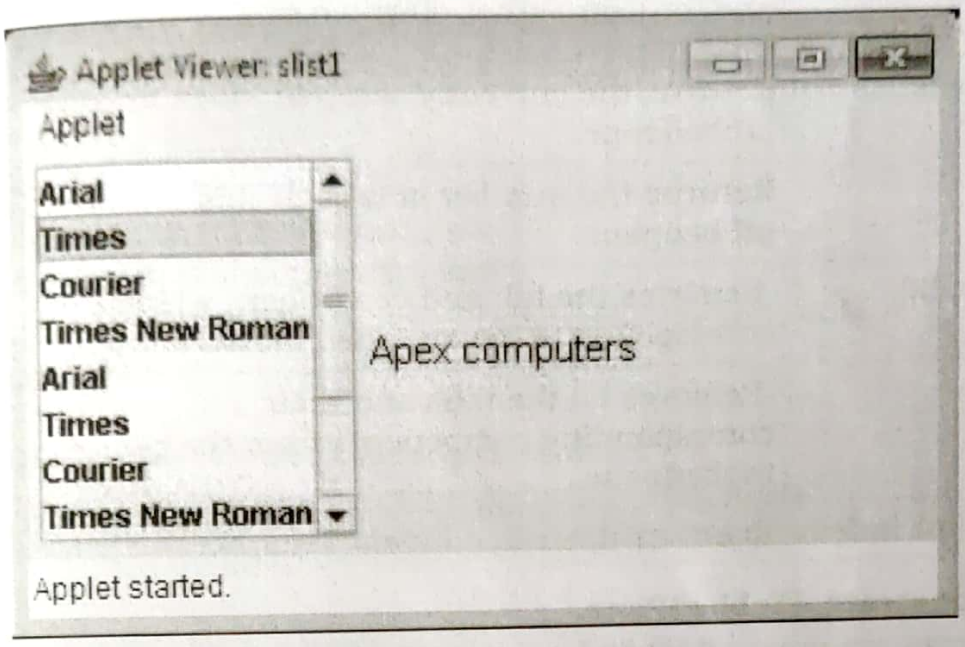
```
import java.awt.*;
import javax.swing.event.*;
import javax.swing.*;
public class slist1 extends JApplet implements
ListSelectionListener
{
    JList list;
    JLabel l1;
    String fontnames[]={ "Arial","Times","Courier",
          "Times New Roman","Arial","Times","Courier",
          "Times New Roman","Arial","Times","Courier",
          "Times New Roman" };
    public void init()
    {
          setLayout(new FlowLayout(FlowLayout.LEFT));
          list=new JList(fontnames);
          l1=new JLabel("Apex computers");
          JScrollPane sp=new JScrollPane(list);
          add(sp);
          add(l1);
          list.addListSelectionListener(this);
    }
    public void valueChanged(ListSelectionEvent i)
    {
          String str=(String)list.getSelectedValue();
          if(str.equals("Arial"))
                l1.setFont(new Font("Arial",Font.PLAIN,15));
          else if(str.equals("Times"))
                l1.setFont(new Font("Times",Font.PLAIN,15));
          else if(str.equals("Courier"))
                l1.setFont(new Font("Courier",Font.PLAIN,15));
          else if(str.equals("Times New Roman"))
    l1.setFont(new Font("Times New Roman",Font.PLAIN,15));
    }
}
```

Bin>javac slist1.java

Bin>edit slist1.html

```
<applet code="slist1" width=400 height=400>
</applet>
```

Bin>appletviewer slist1.html



Component: JTabbedPane

The **JTabbedPane** class can be used to create tabbed pane component, which is a collection of tabs. Each tab has a title and a tab can be added with any number of components. At any given time only one tab can be selected.

The **JTabbedPane** class is subclass of **JComponent**.

Constructors of **JTabbedPane** class.

Constructors	Meaning
JTabbedPane()	Creates an empty TabbedPane with a default tab placement of JTabbedPane.TOP.
JTabbedPane(int tabPlacement)	Creates an empty TabbedPane with the specified tab placement of either: JTabbedPane.TOP, JTabbedPane.BOTTOM, JTabbedPane.LEFT, or JTabbedPane.RIGHT.
JTabbedPane(int tabPlacement, int tabLayoutPolicy)	Creates an empty TabbedPane with the specified tab placement and tab layout policy.

Methods of **JTabbedPane** class.

Methods	Meaning
void addTab(**String** title, **Component** component)	Adds a component represented by a title and no icon.
void addTab(**String** title, **Icon** icon, **Component** component)	Adds a component represented by a title and/or icon, either of which can be null.
int getSelectedIndex()	Returns the currently selected index for this tabbedpane.
int getTabCount()	Returns the number of tabs in this tabbedpane.
void remove(int index)	Removes the tab and component which corresponds to the specified index.
void removeAll()	Removes all the tabs and their corresponding components from the tabbedpane.
void removeTabAt(int index)	Removes the tab at index.

A program to demonstrate JTabbedPane.

In this program, we create a tabbed pane to which we add three tabs. To each tab we add some controls.

Bin>edit tabbedpane1.java

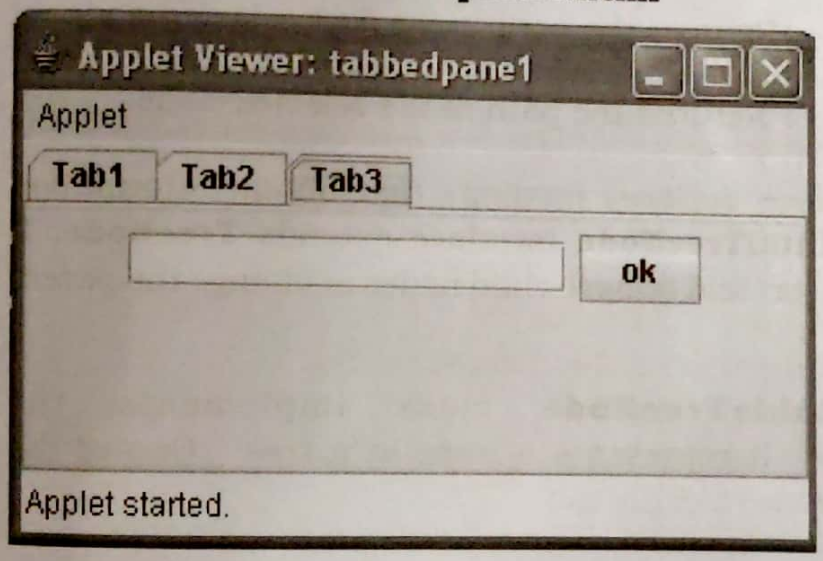
```
import java.awt.*;
import javax.swing.*;
public class tabbedpane1 extends JApplet
{
    JTabbedPane tb;
    public void init()
    {
        tb=new JTabbedPane();
        JPanel p1,p2,p3;
        p1=new JPanel();
        p1.add(new JButton("mybutton"));
        p1.add(new JLabel("mylabel"));
        p2=new JPanel();
        p2.add(new JCheckBox("mycheckbox",true));
        p2.add(new JRadioButton("myradio",true));
```

```
            p3=new JPanel();
            p3.add(new JTextField(15));
            p3.add(new JButton("ok"));

            tb.addTab("Tab1",p1);
            tb.addTab("Tab2",p2);
            tb.addTab("Tab3",p3);

            add(tb);
    }
}
```

Bin>javac tabbedpane1.java
Bin>edit tabbedpane1.html

```
<applet code="tabbedpane1" width=400 height=400>
</applet>
```

Bin>appletviewer tabbedpane1.html



Component: JTree

The **JTree** class can be used to create tree component which is used to represent the data in hierarchical order. There will be a root node and contains sub trees where each sub tree can contain sub trees or nodes. The individual sub trees can be expanded or collapsed.

The **JTree** class is a subclass of **JComponent**

Following are the constructors of **JTree** class

Constructors	Meaning
JTree()	Returns a JTree with a sample model.
JTree(Object value[])	Returns a JTree with each element of the specified array as the child of a new root node which is not displayed.
JTree(Vector value)	Returns a JTree with each element of the specified Vector as the child of a new root node which is not displayed.

Following are the methods of **JTree** class.

Methods	Meaning
int **getRowCount**()	Returns the number of rows that are currently being displayed.
int **getSelectionCount()**	Returns the number of nodes selected.
TreePath **getSelectionPath()**	Returns the path to the first selected node.
TreePath **getPath()**	Returns the path of the selected node.

The **TreeNode** interface declares methods that obtain information about a tree node. The **MutalTreeNode** interface extends **TreeNode**. It declares methods that can insert and remove child nodes or change the parent node.

The .**DefaultMutableTreeNode** class implements the **MutableTreeNode** interface. It represents a node in a tree. One of the constructors is shown below:

DefaultMutableTreenode (Object obj)

obj is the object to be enclosed in this tree node. The new tree node doesn't have a parent or children. To create a hierarchy of tree nodes, the **add()** method of **DefaultMutableTreeNode** can be used. Its signature is:

void add(MutableTreeNode child)

child is the node that should be added as a child to the current node.

The tree component should be register with **TreeSelectionListener** to receive the events. This listener calls **valueChanged()** method in which we can obtain the path of the selection node.

A program to demonstrate JTree component.

Bin>edit jtree1.java

```
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;
public class jtree1 extends JApplet implements
TreeSelectionListener
{
    JTree tree;
    JTextField t;
    public void init()
    {
          setLayout(new BorderLayout());
          DefaultMutableTreeNode root=new
                    DefaultMutableTreeNode("Root");
          DefaultMutableTreeNode courses=new
                    DefaultMutableTreeNode("Courses");
          DefaultMutableTreeNode dca=new
                    DefaultMutableTreeNode("Dca");
          DefaultMutableTreeNode pgdca=new
                    DefaultMutableTreeNode("Pgdca");
          DefaultMutableTreeNode java=new
                    DefaultMutableTreeNode("Java");
          DefaultMutableTreeNode colors=new
                    DefaultMutableTreeNode("Colors");
          DefaultMutableTreeNode red=new
                    DefaultMutableTreeNode("Red");
          DefaultMutableTreeNode green=new
                    DefaultMutableTreeNode("Green");
          DefaultMutableTreeNode blue=new
                    DefaultMutableTreeNode("Blue");
          root.add(courses);root.add(colors);
    courses.add(dca);courses.add(pgdca);courses.add(java);
    colors.add(red);colors.add(green);colors.add(blue);
          tree=new JTree(root);
          tree.addTreeSelectionListener(this);
          JScrollPane sp=new JScrollPane(tree);
          add(sp,BorderLayout.NORTH);
          t=new JTextField(30);
```

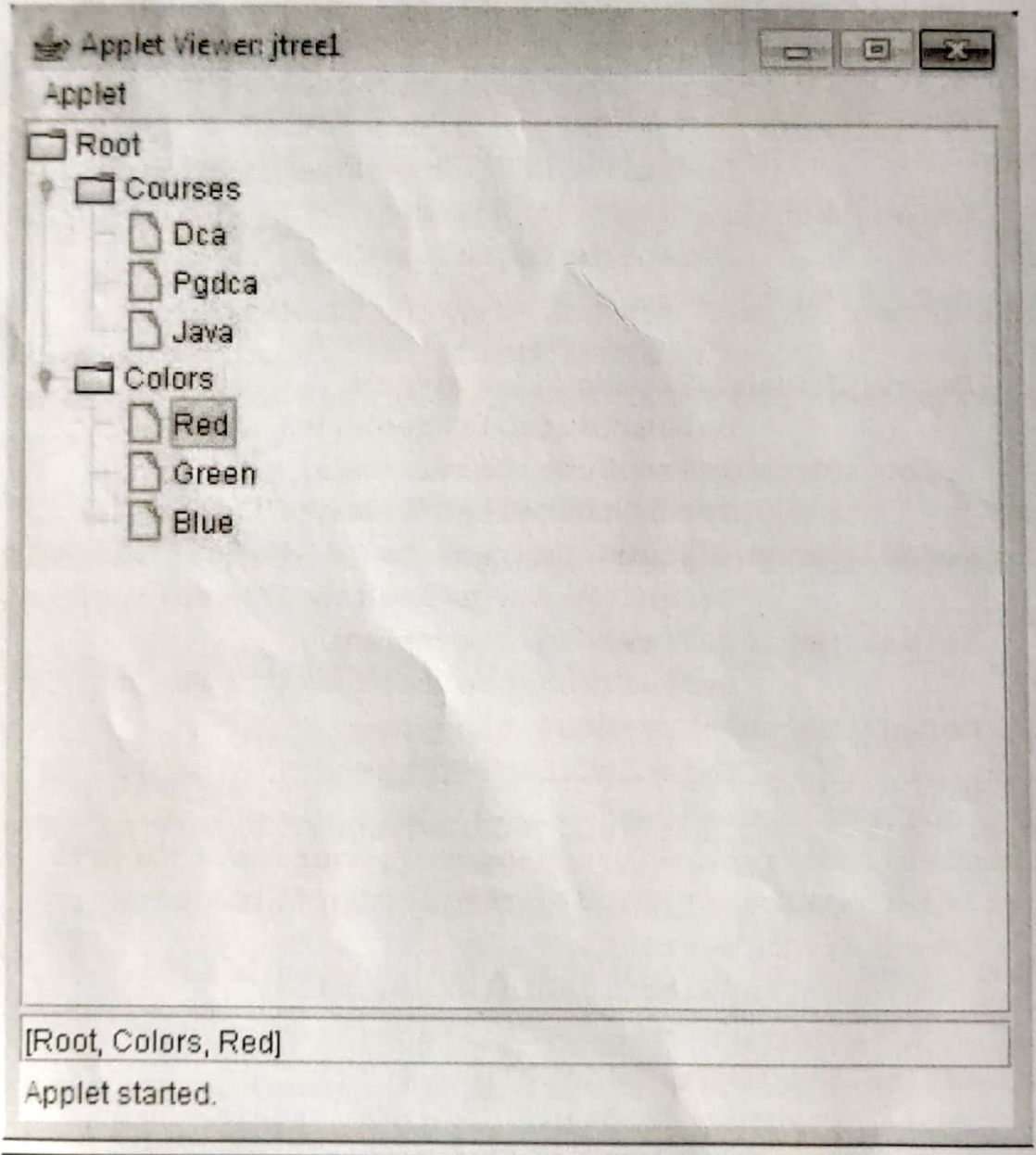
```
            add(t,BorderLayout.SOUTH);
        }
        public void valueChanged(TreeSelectionEvent e)
        {
            t.setText(""+e.getPath());
          //getpath() returns string from root to selected node
        }
}
```

Bin>javac jtree1.java

Bin>edit jtree1.html

```
<applet code="jtree1" width=400 height=400>
</applet>
```

Bin>appletviewer jtree1.html



Component : JTable

The **JTable** class can be used to create a table component which can be used to display data in tabular format i.e. rows and columns. The table component contains column heading and data below each column heading. The data present in the table can be edited.

The **JTable** component generates two types of events, they are **ListSelectionEvent** and **TableModelEvent**. A **ListSelectionEvent** is generated when the user selects something in the table. By default, **JTable** allows us to select one or more complete rows, but we can change this behavior to allow one ore more columns, or one or more individual cells to be selected. A **TableModelEvent** is fired when that table's data changes in some way.

Following are the constructors of **JTable** class.

Constructors	Meaning
JTable()	Constructs a default JTable that is initialized with a default data model, a default column model, and a default selection model.
JTable(int numRows, int numColumns) DefaultTableModel.	Constructs a JTable with numRows and numColumns of empty cells using
JTable(Object[][] rowData, Object[] columnNames)	Constructs a JTable to display the values in the two dimensional array, rowData, with column names, columnNames.

A program to demonstrate JTable component

Bin>edit table1.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
public class table1 extends JApplet
{
    public void init()
    {
        String headings[]={"Name","Empno","Esal"};
        Object data[][]={
                {"Kumar","101","1000"},
```

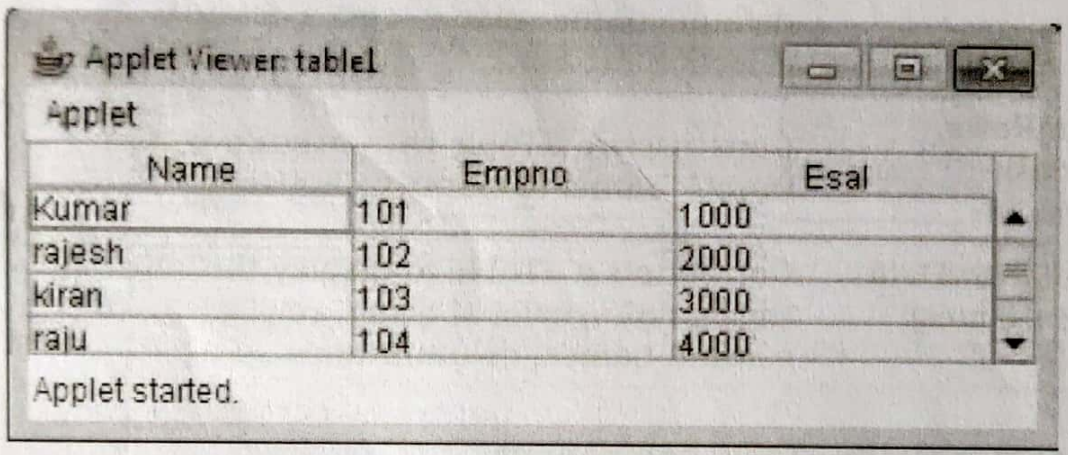
```
                        {"rajesh","102","2000"},
                        {"kiran","103","3000"},
                        {"raju","104","4000"},
                        {"sridhar","105","5000"}
                    };
            JTable t=new JTable(data,headings);
            JScrollPane sp=new JScrollPane(t);
            add(sp);
    }
}
```

Bin>javac table1.java

Bin>edit table1.html

```
<applet code="table1" width=400 height=400>
</applet>
```

Bin>appletviewer table1.html



Component : JScrollPane

The **JScrollPane** class can be used to create a rectangular area and into which a component can be added and viewed. The **JScrollPane** contains scrollbars which allows to scroll a big component.

The **JScrollPane** class is a sub class of **JComponent**.

Following are the constructors of JScrollPane class.

Constructors	Meaning
JScrollPane()	Creates an empty (no viewport view) JScrollPane where both horizontal and vertical scrollbars appear when needed.
JScrollPane(Component view)	Creates a JScrollPane that displays the contents of the specified component, where both horizontal and vertical scrollbars appear whenever the component's contents are larger than the view.
JScrollPane(Component view, int vsbPolicy, int hsbPolicy)	Creates a JScrollPane that displays the view component in a viewport whose view position can be controlled with a pair of scrollbars.
JScrollPane(int vsbPolicy, int hsbPolicy)	Creates an empty (no viewport view) JScrollPane with specified scrollbar policies.

Following are the public static final variables defined in **ScrollPaneConstants** interface. These constant variables can be specified for vsbpolicy and hsbpolicy in the above constructors.

Fields	Meaning
static String **HORIZONTAL_SCROLLBAR**	Identifies a horizonal scrollbar.
static int **HORIZONTAL_SCROLLBAR _ ALWAYS**	Used to set the horizontal scroll bar policy so that horizontal scrollbars are always displayed.
static int **HORIZONTAL_SCROLLBAR _AS_ NEEDED**	Used to set the horizontal scroll bar policy so that horizontal scrollbars are displayed only when needed.
static int **HORIZONTAL_SCROLLBAR _ NEVER**	Used to set the horizontal scroll bar policy so that horizontal scrollbars are never displayed.
static String **VERTICAL_SCROLLBAR**	Identifies a vertical scrollbar.
static int **VERTICAL_SCROLLBAR_ ALWAYS**	Used to set the vertical scroll bar policy so that vertical scrollbars are always displayed.

static String **VERTICAL_SCROLLBAR**	Identifies a vertical scrollbar.
static int **VERTICAL_SCROLLBAR_ALWAYS**	Used to set the vertical scroll bar policy so that vertical scrollbars are always displayed.

A program to demonstrate JScrollPane.

In this program, we add a big image to the scroll pane so that the image can be scrolled.

Bin>edit scrollpane1.java

```
import java.awt.*;
import javax.swing.*;
public class scrollpane1 extends JFrame
{
  public scrollpane1 (  )
  {
       super ( "JScrollPane Demo" ) ;
       ImageIcon ii = new ImageIcon ( "sunset.jpg" ) ;
      JScrollPane jsp = new JScrollPane (new JLabel(ii));
      add ( jsp ) ;
      setSize ( 300,250 ) ;
      setVisible ( true ) ;
  }
  public static void main ( String args[ ] )
  {
     scrollpane1 obj=new scrollpane1 (  ) ;
  }
}
```

Bin>javac scrollpane1.java
Bin>java scrollpane1

Another example on JScrollPane.

In this program, to the panel we add large number of buttons and we add scroll bar to the panel using scroll pane.

Bin>edit scrollpane2.java

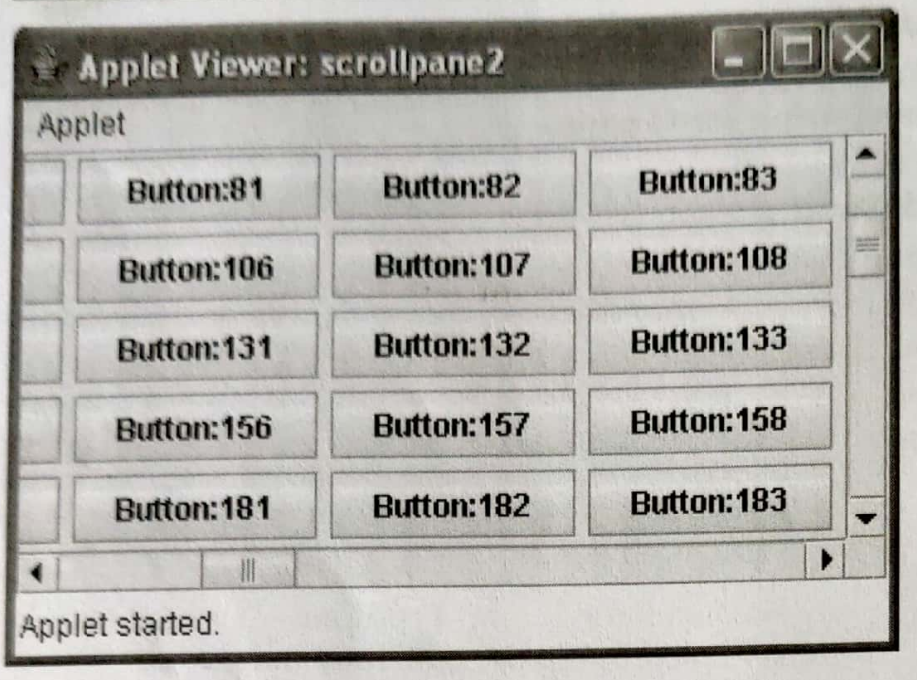
```
import java.awt.*;
import javax.swing.*;
public class scrollpane2 extends JApplet
{
    JScrollPane sp;
    JButton b;
    JPanel p;
    int n=1;
    public void init()
    {
          p=new JPanel();
          p.setLayout(new GridLayout(25,25,5,5));
          for(int i=1;i<=25;i++)
          {
                for(int j=1; j<=25; j++)
                {
                      b=new JButton("Button:"+n++);
                      p.add(b);
                }
          }
          sp=new JScrollPane(p,ScrollPaneConstants.VERTICAL_
                SCROLLBAR_AS_NEEDED,
                ScrollPaneConstants.HORIZONTAL_SCROLLBAR_
                AS_NEEDED);
                add(sp);
    }
}
```

Bin>javac scrollpane2.java

Bin>edit scrollpane2.html

```
<applet code="scrollpane2" width=400 height=400>\
</applet>
```

Bin>appletviewer scrollpane2.html



Component: JTabbedPane

The **JTabbedPane** class can be used to create tabbed pane component, which is a collection of tabs. Each tab has a title and a tab can be added with any number of components. At any given time only one tab can be selected.

The **JTabbedPane** class is subclass of **JComponent**.

Constructors of **JTabbedPane** class.

Constructors	Meaning
JTabbedPane()	Creates an empty TabbedPane with a default tab placement of JTabbedPane.TOP.
JTabbedPane(int tabPlacement)	Creates an empty TabbedPane with the specified tab placement of either: JTabbedPane.TOP, JTabbedPane.BOTTOM, JTabbedPane.LEFT, or JTabbedPane.RIGHT.
JTabbedPane(int tabPlacement, int tabLayoutPolicy)	Creates an empty TabbedPane with the specified tab placement and tab layout policy.

Methods of JTabbedPane class.

Methods	Meaning
void **addTab**(String title, Component component)	Adds a component represented by a title and no icon.
void **addTab**(String title, Icon icon, Component component)	Adds a component represented by a title and/or icon, either of which can be null.
int **getSelectedIndex**()	Returns the currently selected index for this tabbedpane.
int **getTabCount**()	Returns the number of tabs in this tabbedpane.
void **remove**(int index)	Removes the tab and component which corresponds to the specified index.
void **removeAll**()	Removes all the tabs and their corresponding components from the tabbedpane.
void **removeTabAt**(int index)	Removes the tab at index.

A program to demonstrate JTabbedPane.

In this program, we create a tabbed pane to which we add three tabs. To each tab we add some controls.

Bin>edit tabbedpane1.java

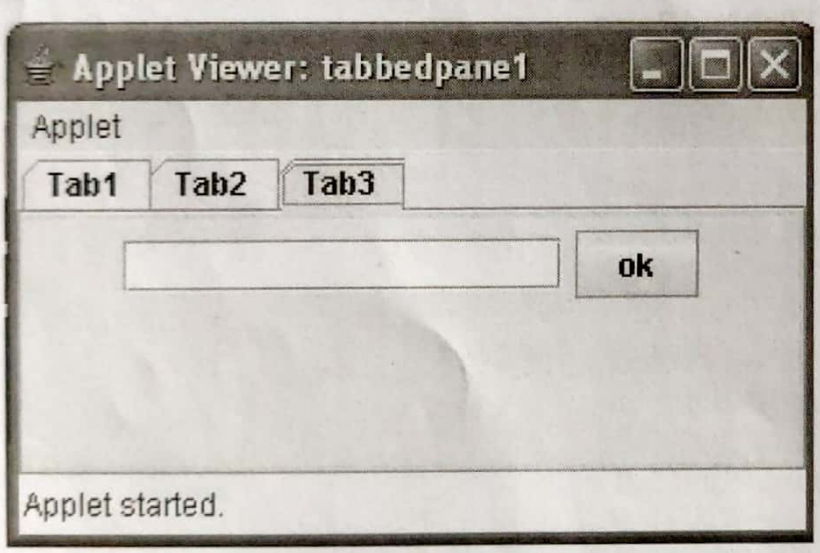
```
import java.awt.*;
import javax.swing.*;
public class tabbedpane1 extends JApplet
{
    JTabbedPane tb;
    public void init()
    {
        tb=new JTabbedPane();
        JPanel p1,p2,p3;
        p1=new JPanel();
        p1.add(new JButton("mybutton"));
        p1.add(new JLabel("mylabel"));
```

```
        p2=new JPanel();
        p2.add(new JCheckBox("mycheckbox",true));
        p2.add(new JRadioButton("myradio",true));

        p3=new JPanel();
        p3.add(new JTextField(15));
        p3.add(new JButton("ok"));

        tb.addTab("Tab1",p1);
        tb.addTab("Tab2",p2);
        tb.addTab("Tab3",p3);

        add(tb);
    }
}
```

Bin>javac tabbedpane1.java
Bin>edit tabbedpane1.html

```
<applet code="tabbedpane1" width=400 height=400>
</applet>
```

Bin>appletviewer tabbedpane1.html



Creation of user-defined windows in swings

A stand-alone application window can be created in swings using JFrame class. This class is a sub class of Frame class of AWT.

The following are the constructors of JFrame class.

Constructors	Meaning
JFrame()	This default constructor creates a new instance of JFrame window that is initially invisible. The title of the Frame is empty.
JFrame(String title)	This one-argument constructor creates a new instance of JFrame window that is initially invisible with the specified title.

The following table shows methods of JFrame class.

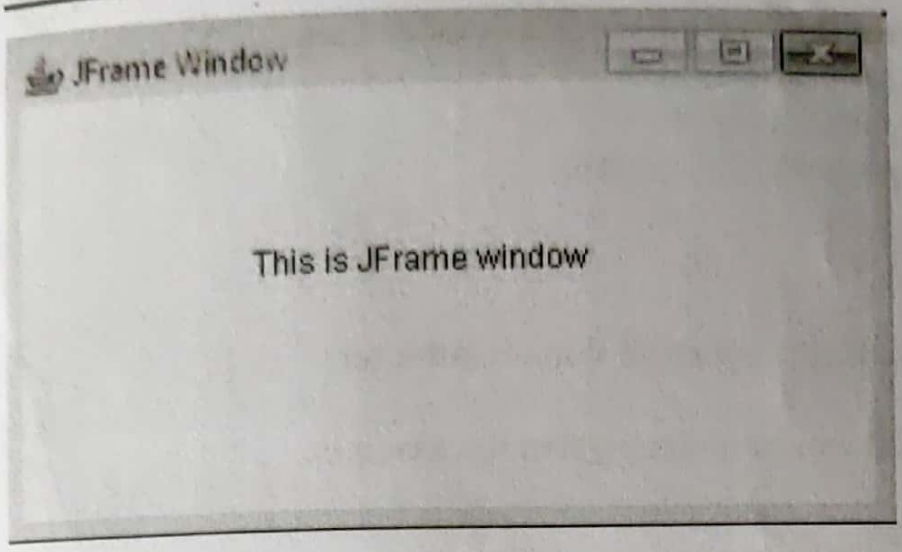
Methods	Meaning
`Container` **`getContentPane`**`()`	Returns the `contentPane` object for this frame.
`Graphics` **`getGraphics`**`()`	Creates a graphics context for this component.
`JMenuBar` **`getJMenuBar`**`()`	Returns the menubar set on this frame.
`void` **`remove`**`(Component comp)`	Removes the specified component from the container.
`void` **`repaint`**`(long time, int x, int y, int width, int height)`	Repaints the specified rectangle of this component within `time` milliseconds.
`void` **`setContentPane`**`(Container contentPane)`	Sets the `contentPane` property.
`void` **`setIconImage`**`(Image image)`	Sets the image to be displayed as the icon for this window.
`void` **`setJMenuBar`**`(JMenuBar menubar)`	Sets the menubar for this frame.
`void` **`setLayout`**`(LayoutManager manager)`	Sets the `LayoutManager`.
`String getTitle()`	returns title of the window
`void setTitle(String title)`	Sets the title of window.
`void setSize(int width, int height)`	Sets the width and height of window.
`void setVisible(boolean flag)`	If flag is true then window is visible on the screen otherwise hidden.

A program to create a user-defined window using JFrame.
This program creates a window extending JFrame class.

Bin> edit swin1.java

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
class mywindow extends JFrame
{
    public mywindow(String title)
    {
        super(title);
        addWindowListener(new mywindowhandler());
    }
    public void paint(Graphics g)
    {
        g.drawString("This is JFrame window", 100,100);
    }

    class mywindowhandler extends WindowAdapter
    {
        public void windowClosing(WindowEvent e)
        {
            dispose();
        }
    }
}

class swin1
{
    public static void main(String argv[])
    {
        mywindow w=new mywindow("JFrame Window");
        w.setSize(400,400);
        w.setVisible(true);
    }
}
```

Bin>javac win1.java
Bin>java win1



A program to demonstrate JFrame window width components

Bin>edit swin2.java

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
class mywindow extends JFrame implements ActionListener
{
   JButton b1,b2;
    JPanel p;
    public mywindow(String title)
      {
         super(title);
        setLayout(new FlowLayout(FlowLayout.LEFT));
        b1=new JButton("Red");
         b2=new JButton("Green");
         p=new JPanel();
         p.add(new JLabel("     "));
        add(b1);
        add(b2);
        add(p);
         b1.addActionListener(this);
         b2.addActionListener(this);
         addWindowListener(new mywindowhandler());
      }
     public void actionPerformed(ActionEvent e)
     {
          String str=e.getActionCommand();
```

```
            if(str.equals("Red"))
              p.setBackground(Color.red);
            else if(str.equals("Green"))
              p.setBackground(Color.green);

        }

        class mywindowhandler extends WindowAdapter
        {
                public void windowClosing(WindowEvent e)
                {
                        dispose();
                }
        }
}

class swin2
{
        public static void main(String argv[])
        {
          mywindow w=new mywindow("JFrame Window");
                w.setSize(400,400);
                w.setVisible(true);
        }
}
```

Bin>javac swin2.java
Bin>java swin2