Unit V

JAVA PROGRAMMING

APPLETS

Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document.

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

The difference between a Java applet and a Java application is that an applet runs in the context of a web browser, being typically embedded within an html page, while a Java application runs standalone, outside the browser.

```
import java.awt.*;
import java.applet.*;
public class SimpleApplet extends Applet
{
         public void paint(Graphics g)
        {
                g.drawString("A Simple Applet", 20, 20);
        }
}
```

This applet begins with two **import** statements. The first imports the Abstract Window Toolkit (AWT) classes. Applets interact with the user through the AWT, not through the console-based I/O classes. The AWT contains support for a window-based, graphical interface.

The second **import** statement imports the **applet** package, which contains the class **Applet**. Every applet that we create must be a subclass of **Applet**.

SimpleApplet class must be declared as **public**, so appletviewer can access it. **paint()** is called each time that the applet must redisplay its output. The **paint()** method has one parameter of type **Graphics**. This parameter contains the graphics context. Inside **paint()**

is a call to **drawString()**, which is a member of the **Graphics** class. This method outputs a string beginning at the specified X,Y location.

It has the following general form:

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

The applet does not have a **main()** method. Unlike java programs, applets do not begin execution at **main()**.

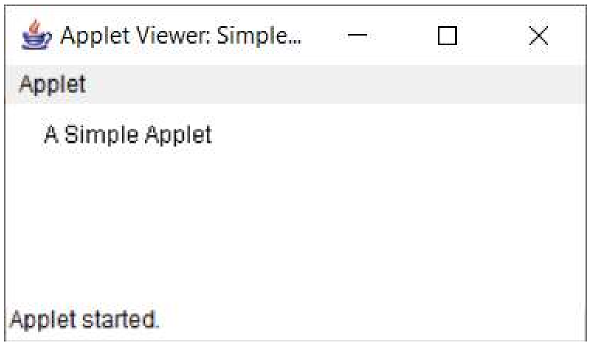
To execute an applet in a web browser, we need to write a sort HTML text file that contains the appropriate APPLET tag. Here is the **RunApp.html** file that executes **<u>SimpleApplet:</u>**

```
<applet code="SimpleApplet" width=300 height=600>
</applet>
```

The **width** and **height** statements specify the dimensions of the display area used by the applet.

1. Edit a java source file. → SimpleApplet.java
2. Compile your program. →Javac SimpleApplet.java
3. Execute the applet viewer →appletviewer RunApp.html

The window produced by SimpleApplet, as displayed by the appletviewer



```
A Simple Applet
```

Applet started.

->Applets do not need a **main()** method.

->Applets must be run under an appletviewer or a java-compatible browser.

->User I/O is not accomplished with java's stream I/O classes. Instead, applets use the interface provided by the AWT**.**

<u>Applet Architecture:</u>

An applet is a window-based program. As such, its architecture is different from the normal, console-based programs. First, applets are event driven. Second, the user initiates interaction with an applet. For example, when the user clicks a mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated. Applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

APPLET LIFE CYCLE

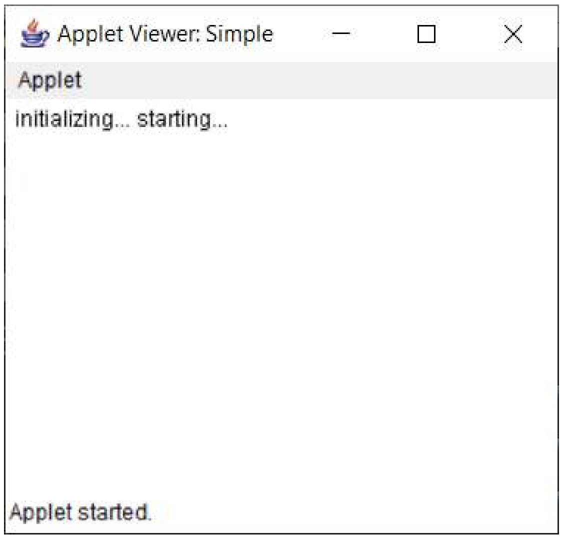
Applet Initialization:

1. **init()** : The **init()** method is the first method the be called. This is where we should initialize variables. This method is called only once during the run time of our applet.
2. **start()** : The **start()** method is called after **init().** It is also called to restart an applet after it has been stopped. Whereas **init()** is called once – the first time an applet is loaded – **start()** is called each time as applet's HTML document is displayed onscreen.
3. **paint()** : The **paint()** method is called each time out applet's output must be redrawn. The **paint()** method has one parameter of type Graphics. This parameter contains the graphics context.

Applet Termination :

1. **stop()** : The **stop()** method is called when a web browser leaves the HTML document containing the applet – when it goes to another page.
2. **destroy()** : The **destroy()** method is called when the environment determines that out applet needs to be removed completely from memory. At this point, we should free up any resources the applet may be using. The **stop()** method is always called before **destroy()**.

```
import java.applet.Applet;
import java.awt.Graphics;
public class Simple extends Applet
{       StringBuffer buffer;
         public void init()
        {
                buffer = new StringBuffer();
                addItem("Initialising");
        }
        public void start()
        {
                addItem("starting....");
        }
        public void stop()
        {
                addItem("stopping....");
        }
        public void destroy()
        {
                addItem("preparing for unloading....");
        }
        private void addItem(String newWord)
        {
                System.out.println(newWord);
                buffer.append(newWord);
                repaint();
        }
        public void paint(Graphics g)
        {
                  // Draw a Rectangle around the applet's display area.
                  g.drawRect(0,0,getWidth()-1,getHeight()-1);
                  // Draw the current string inside the rectangle
                  g.drawString(buffer.toString(),5,15);
        }
}
```



```
initializing... starting...
```

```
Applet started.
```

Applet Display Methods:

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

message is the string to be output beginning at x,y

To set the background color of an applet's window, use **setBackground().**

void setBackground(Color *newColor*)

To set the foreground color use **setForeground().**

void setForeground(Color *newColor*)

newColor specifies the new color. The class **Color** defines the constants

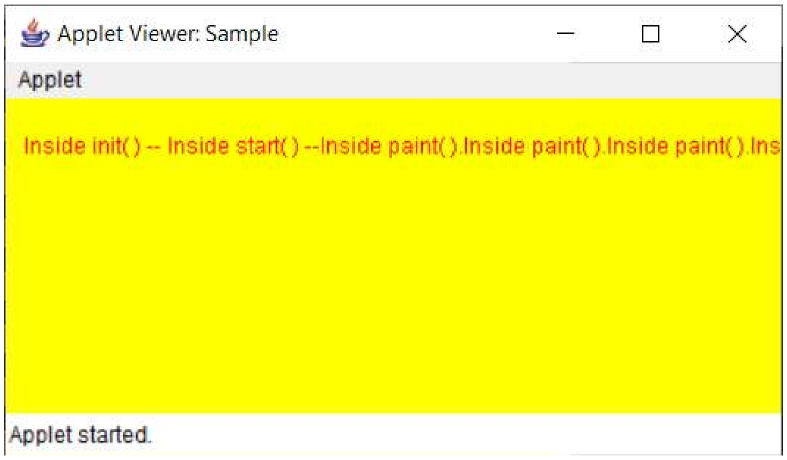
Color.black	Color.magenta
Color.blue	Color.orange
Color.cyan	Color.pink
Color.darkGray	Color.red
Color.gray	Color.white
Color.green	Color.yellow
Color.lightGray	

Ex:
```
setBackground(Color.green);
setForeground(Color.red);
```

Set the foreground and background colors is in the **init()** method. We can obtain the current settings for the background and foreground colors by calling **getBackground()** and **getForeground()**.

```
Color getBackground()
Color getForeground()
```

```
import java.awt.*;
import java.applet.*;
/*
<applet code="Sample" width=300 height=50>
</applet>
*/
public class Sample extends Applet
{
        String msg;
              // set the foreground and background colors.
        public void init()
        {
                setBackground(Color.yellow);
                setForeground(Color.red);
                msg="Inside init() --";
        }
             // Initialise the string to be displayed.
        public void start()
        {
                msg+="Inside start() --";
        }
        public void paint(Graphics g)
        {
                msg+="Inside paint().";
                g.drawString(msg,10,30);
        }
}
```

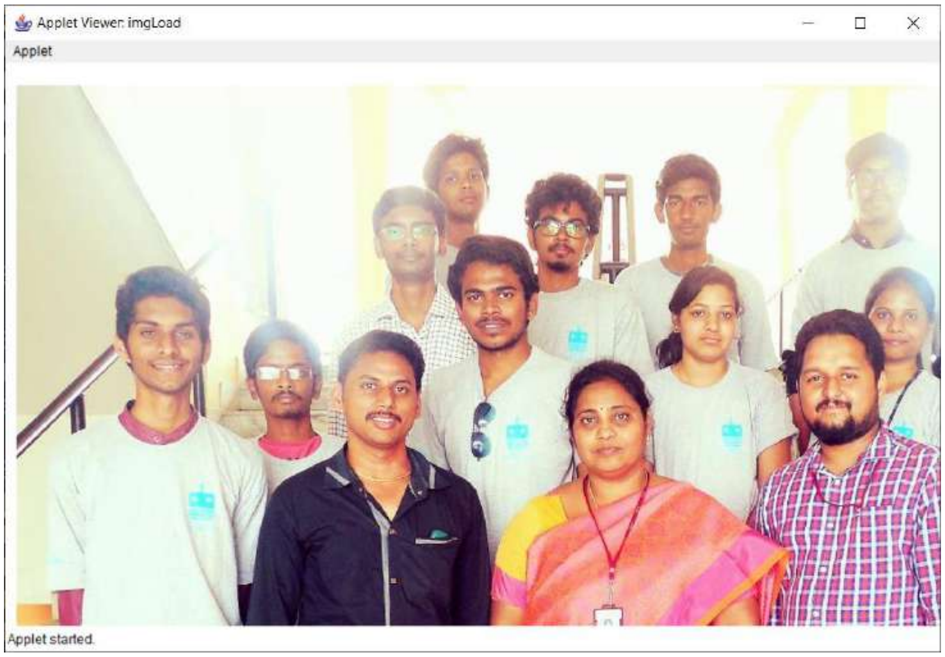


```
Inside init() -- Inside start() --Inside paint().Inside paint().Inside paint().Ins
```

Applet started.

Loading Images in Background of an Applet:

```
import java.awt.*;
import java.applet.Applet;
public class imgLoad extends Applet
{
        Image i;
        public void init()
        {
                System.out.println("In init");
                i=getImage(getDocumentBase(),"a.jpg");
        }
        public void paint(Graphics g)
        {
                System.out.println("In paint");
                g.drawImage(i,10,20,this);
        }
}
```



The repaint, update, and paint methods. These methods define graphics output:

- Use the repaint(), paint(), and update() methods to generate graphic output. The default repaint indirectly invokes update.
- The default update method clears the background (potential source of flicker) then calls paint.
- The paint method draws the output. The paint method alone is called when old areas are re-exposed (for example, moving an obscured window).

Requesting Repainting:

The **repaint()** method has four forms.

1. void repaint() : This version causes the entire window to be repainted.
2. void repaint(int *left,* int *top,* int *width*, int *height*) : This version specifies a region that will be repainted. The coordinates of the upper-left corner of the region are specified by *left* and *top*, and the width and height of the region are passed in *width* and *height*. These dimensions are specified in pixels.
3. void repaint(long *maxDelay*) : *maxDelay* specifies the maximum number of milliseconds that can elapse before **update()** is called.
4. void repaint(long *maxDelay*, int *x*, int *y*, int *width*, int *height*)

The HTML APPLET Tag

The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

< APPLET

[CODEBASE = *codebaseURL*]

CODE = *appletFile*

[ALT = *alternateText*]

[NAME = *appletInstanceName*]

WIDTH = *pixels* HEIGHT = *pixels*

[ALIGN = alignment]

[VSPACE = *pixels*][HSPACE = *pixels*]

>

[<PARAM NAME = *AttributeName* VALUE = *AttributeName*>]

[<PARAM NAME = *AttributeName2* VALUE = *AttributeName*>]

…

[*HTML Displayed in the absence of Java*]

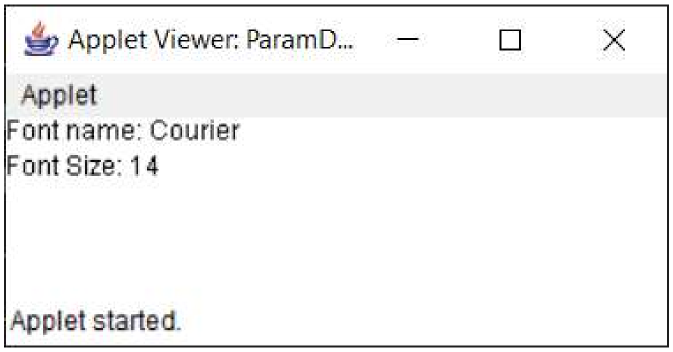
</APPLET>

Passing Parameters to Applets: The APPLET tag in HTML allows you to pass parameters to out applet. To retrieve a parameter, use the **getParameter()** method. It returns the value of the specified parameter in the form of a **String** object.

// Use Parameters

```
import java.awt.*;
import java.applet.*;
/* <applet code="ParamDemo" width=300 height=80>
    <param name=fontName value=Courier>
          <param name=fontSize value=14>
          </applet>   */
public class ParamDemo extends Applet
{
          String fontName;
          int fontSize;
          float leading;
          boolean active;
          // Initialise the string to be displayed.

          public void start()
          {
                    String param;
                    fontName = getParameter("fontName");
                    if(fontName==null)
                              fontName="Not Found";
                    param = getParameter("fontSize");
```

```
        try
        {       if(param!=null)// if not found
                    fontSize=Integer.parseInt(param);
                else
                        fontSize=0;
        }
        catch (NumberFormatException e)
        {           fontSize=-1;
        }
    }
    // Display parameters
    public void paint(Graphics g)
    {
        g.drawString("Font name: "+fontName,0,10);
        g.drawString("Font Size: "+fontSize,0,26);
    }
}
```



```
Font name: Courier
Font Size: 14
```

Applet started.

EVENT HANDLING

Event: An *event* is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface.
Ex: Pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse.

Event may also occur that are not directly caused by interactions with a user interface. Ex: A counter exceeds a value, software or hardware failure occurs, or an operation is completed.

In java language massages are known as Events

Event Sources: A *source* is an object that generates an event. A source must register listeners in order for the listeners to receive notifications about a specific type of event.
<u>General form:</u> public void *Type* Listener(*Type*Listener el)
Here, *Type* is the name of the event and *el* is a reference to the event listener.
Ex : The method that registers a keyboard event listener is called **addKeyListener().** The method that registers a mouse motion listener is called **addMouseMotionListener().**

When an event occurs, all registered listeners are notified and receive a copy of the event Object. This is known as **multicasting** the event.
A source must also provide a method that allows a listener to unregister. The general form of such a method is this:

Public void remove *Type*Listener(*Type*Listener *el*)

Ex: removeKeyListener

Event Listeners: A *listener* is an object is notified when an event occurs. It has two major requirements. First, it must have been registered with one more sources to receive notification about specific types of events. Second, it must implement methods to receive and process these notifications.
Ex: MouseMotionListener interface defines two methods to receive notifications when the mouse is dragged or moved

Event Classes: At the root of the java event hierarchy is **EventObject,** which is in **java.util.** It is the superclass for all events. Its one constructor is shown here:

EventObject(Object *src*)

Here, *src* is the object that generates this event.
EventObject contains two methods: **getSource()** and **toString().**

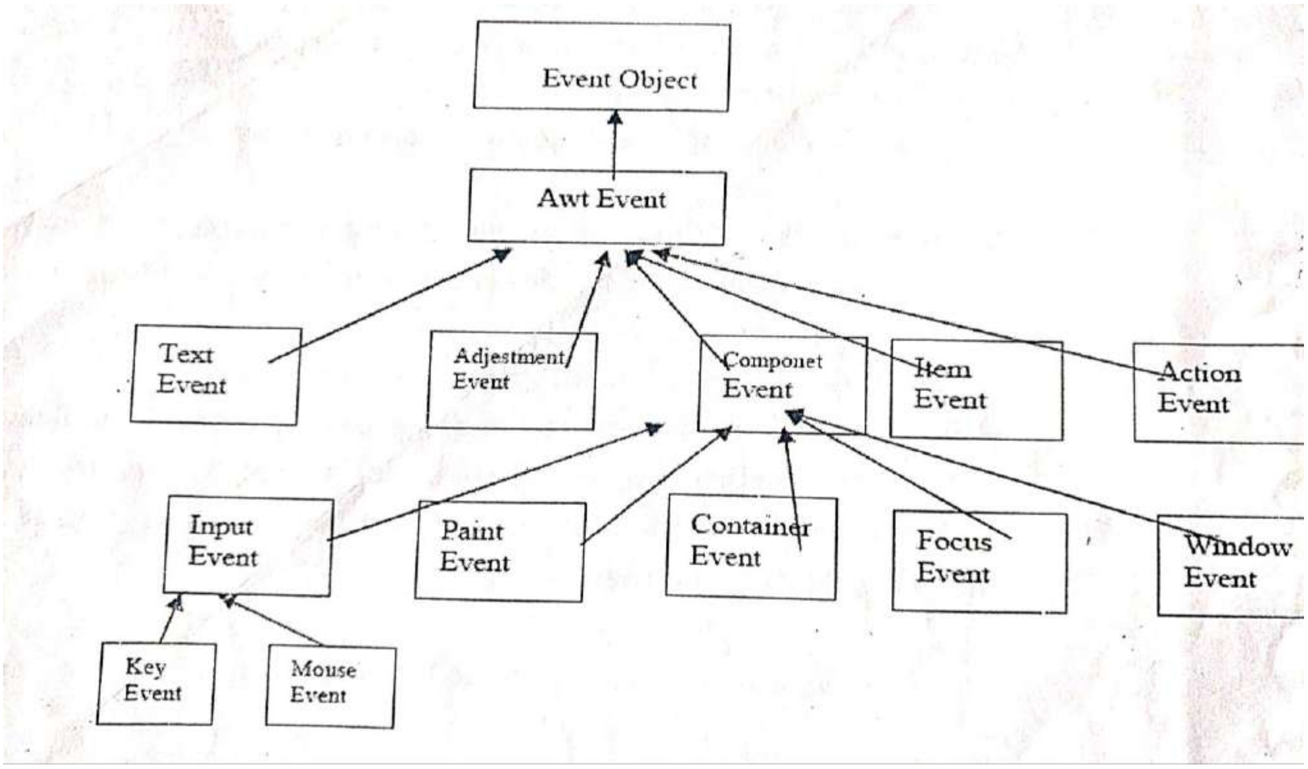
The **getSource()** method returns the source of the event. Its general form is showing here:

Object getSource()

As expected, **toString()** returns the string equivalent of the event.

The package **java.awt.event** defines several types of events

Events Hierarchy:



Event Classes in java.awt.event

Event Class	Description
ActionEvent	Generated when a button is pressed, a list item is double-clicked, or a menu item is selected.
AdjustmentEvent	Generated when a scroll bar is manipulated.
ComponentEvent	Generated when a component is hidden, moved, resized, or becomes visible.
ContainerEvent	Generated when a component is added to or removed from a container.
FocusEvent	Generated when a component gains or loses keyboard focus.
InputEvent	Abstract super class for all component input event classes.
ItemEvent	Generated when a check box or list item is clicked; also occurs when a choice selection is made or a checkable menu item is selected or deselected.

Event Class	Description
KeyEvent	Generated when input is received from the keyboard.
MouseEvent	Generated when the mouse is dragged, moved, clicked, pressed, or released; also generated when the mouse enters or exits a component.
MouseWheelEvent	Generated when the mouse wheel is moved. (Added by Java 2, version 1.4)
TextEvent	Generated when the value of a text area or text field is changed.
WindowEvent	Generated when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Sources:

Event Source	Description
Button	Generates action events when the button is pressed.
Checkbox	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu Item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scrollbar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Listener Interfaces

The ActionListenerInterface: This interface defines the **actionPerformed()** method that is invoked when an action event occurs. Its general form is shown here:

void actionPerformed(ActionEvent *ae*)

The AdjustmentListener Interface: This interface defines the **adjustmentValueChanged()** method that is invoked when an adjustment event occurs. Its general form is shown here.

void adjustmentValueChanged(AdjustmentEvent *ae*)

TheComponentListener Interface: This interface defines four methods that are invoked when a component resized, moved, or hidden. Their general form is shown here:

void componentResized(ComponentEvent *ce*)

void componentMoved(ComponentEvent *ce*)

void componentShown(ComponentEvent *ce*)

void componentHidden(ComponentEvent *ce*)

The ContainerListener Interface : This interface contains two methods. When a component is added to a container, **componentAdded()** is invoked. When a component is removed from a container, **componentRemoved()** is invoked. Their general forms are here:

void componentAdded(ContainerEvent *ce*)

void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface : This interface defines two methods. When a component obtains keyboard focus, **focusGained()** is invoked. When a component loses keyboard focus, **focusLost()** is called. Their general form are shown here:

void focusGained(FocusEvent *fe*)

void focusLost(FocusEvent *fe*)

The ItemListener Interface : This interface defines the **itemStateChanged()** method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface : This interface defines three methods. The **Keypressed()** and **KeyReleased()** method is invoked when a key is pressed and released, respectively. The **KeyTyped()** method is invoked when a character has been entered.

The general form of these methods are shown here:

void Keypressed(KeyEvent *ke*)

void KeyReleased(KeyEvent *ke*)

void KeyTyped(KeyEvent *ke*)

The MouseListener Interface : This interface defines five methods.

The general forms of these methods are shown here :

void mouseClicked(MouseEvent *me*)

void mouseEntered(MouseEvent *me*)

void mouseExited(MouseEvent *me*)

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface : This interface defines two methods.
Their general forms are shown here :

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

The TextListener Interface : This interface defines the **textChanged()** method that is invoked when a changed occurs in a text field**.** Its general form is shown here :

void textChanged(TextEvent *te*)

The WindowListener Interface : This is interface defines the following methods

void windowActivated(WindowEvent *we*)
void windowClosed(WindowEvent *we*)
void windowClosing(WindowEvent *we*)
void windowDeactivated(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

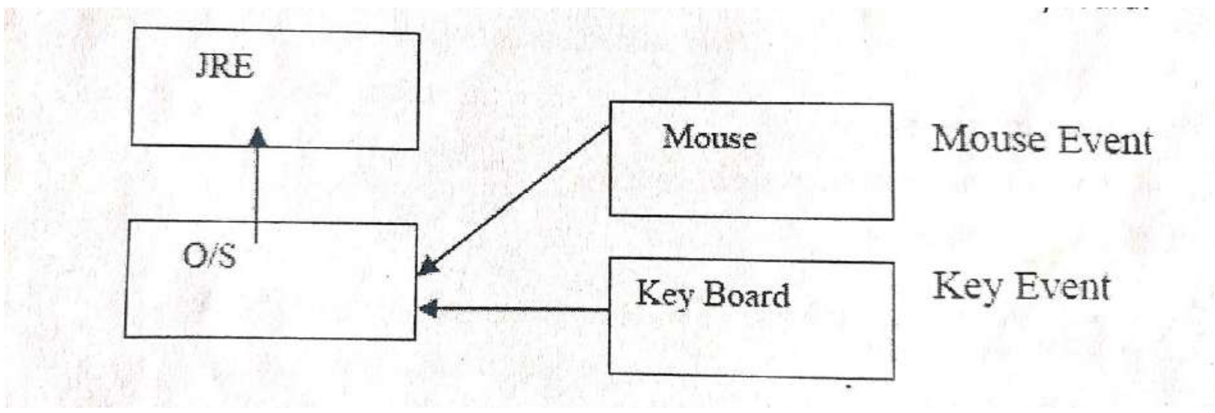
Delegated event model :

A source generates an event and sends it to one or more *listeners.* The listener simply waits until it receives an event. Once received, the listener processes the event and then returns.

Applet programming using the delegation event model is actually quite easy. Just follow these two steps :

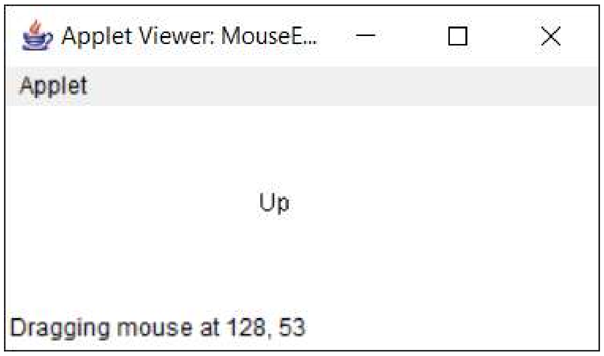
1. Implements the appropriate interface in the listener so that it will receive the type of event desired.
2. Implements code to register and unregister (if necessary) the listener as a recipient for the event notifications.

Two most commonly used event generators: the mouse and keyboard.



//Demonstrate the mouse event handlers

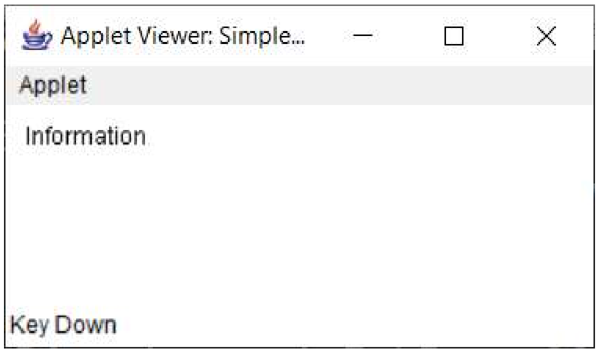
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet implements MouseListener, MouseMotionListener
{       String msg = "";
        int mouseX = 0, mouseY = 0; // coordinates of mouse

        public void init()
        {       addMouseListener(this);
                addMouseMotionListener(this);
        }
        // Handle mouse clicked.
        public void mouseClicked(MouseEvent me)
        {       mouseX = 0;
                mouseY = 10;
                msg = "Mouse clicked.";
                repaint();
        }
        // Handle mouse entered.
        public void mouseEntered(MouseEvent me)
        {       mouseX = 0;
                mouseY = 10;
                msg = "Mouse entered.";
                repaint();
        }
        // Handle mouse exited.
        public void mouseExited(MouseEvent me)
        {       mouseX = 0;
                mouseY = 10;
                msg = "Mouse exited.";
                repaint();
        }
        // Handle button pressed.
        public void mousePressed(MouseEvent me)
        {       mouseX = me.getX();
                mouseY = me.getY();
                msg = "Down";
                repaint();
        }
        // Handle button released.
        public void mouseReleased(MouseEvent me)
        {       mouseX = me.getX();
                mouseY = me.getY();
                msg = "Up";
                repaint();
        }
        // Handle mouse dragged.
        public void mouseDragged(MouseEvent me)
        {       mouseX = me.getX();
                mouseY = me.getY();
                msg = "*";
                showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
                repaint();
        }
        // Handle mouse moved.
        public void mouseMoved(MouseEvent me)
        {       showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
        }
        // Display msg in applet window at current X,Y location.
        public void paint(Graphics g)
        {       g.drawString(msg, mouseX, mouseY);
        }
}
```



```
Up
Dragging mouse at 128, 53
```

//Demonstrate the key event handlers

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet implements KeyListener
{
        String msg = "";
        int X = 10, Y = 20; // output coordinates
        public void init()
        {       addKeyListener(this);
                requestFocus(); // request input focus
        }
        public void keyPressed(KeyEvent ke)
        {       showStatus("Key Down");
        }
        public void keyReleased(KeyEvent ke)
        {       showStatus("Key Up");
        }
        public void keyTyped(KeyEvent ke)
        {       msg += ke.getKeyChar();
                repaint();
        }
        // Display keystrokes.
        public void paint(Graphics g)
        {       g.drawString(msg, X, Y);
        }
}
```



Key Down

Adaptor Classes

Java provides a special feature, called an *adaptor* class, that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface.

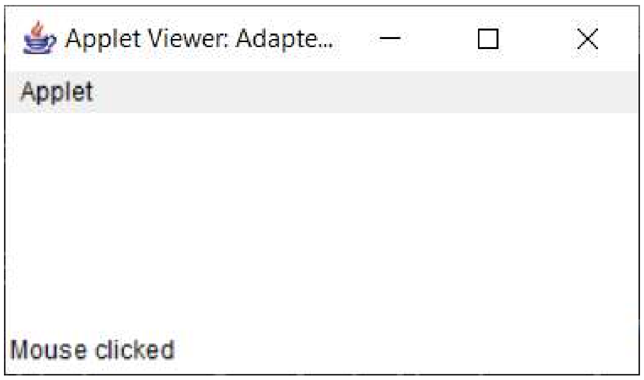
Adapter classes are useful when we want to receive and process only some of the events that are handled by a particular event listener interface. We can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which we are interested.

Adapter Class	Listener Interface
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

// Demonstrate an adapter.

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/
public class AdapterDemo extends Applet
{
        public void init()
        {       addMouseListener(new MyMouseAdapter(this));
                addMouseMotionListener(new MyMouseMotionAdapter(this));
        }
}
```

```
class MyMouseAdapter extends MouseAdapter
{       AdapterDemo adapterDemo;

        public MyMouseAdapter(AdapterDemo adapterDemo)
        {       this.adapterDemo = adapterDemo;
        }
        // Handle mouse clicked.
        public void mouseClicked(MouseEvent me)
        {
                adapterDemo.showStatus("Mouse clicked");
        }
}

class MyMouseMotionAdapter extends MouseMotionAdapter
{
        AdapterDemo adapterDemo;

        public MyMouseMotionAdapter(AdapterDemo adapterDemo)
        {
                this.adapterDemo = adapterDemo;
        }

        // Handle mouse dragged.

        public void mouseDragged(MouseEvent me)
        {
                adapterDemo.showStatus("Mouse dragged");
        }
}
```



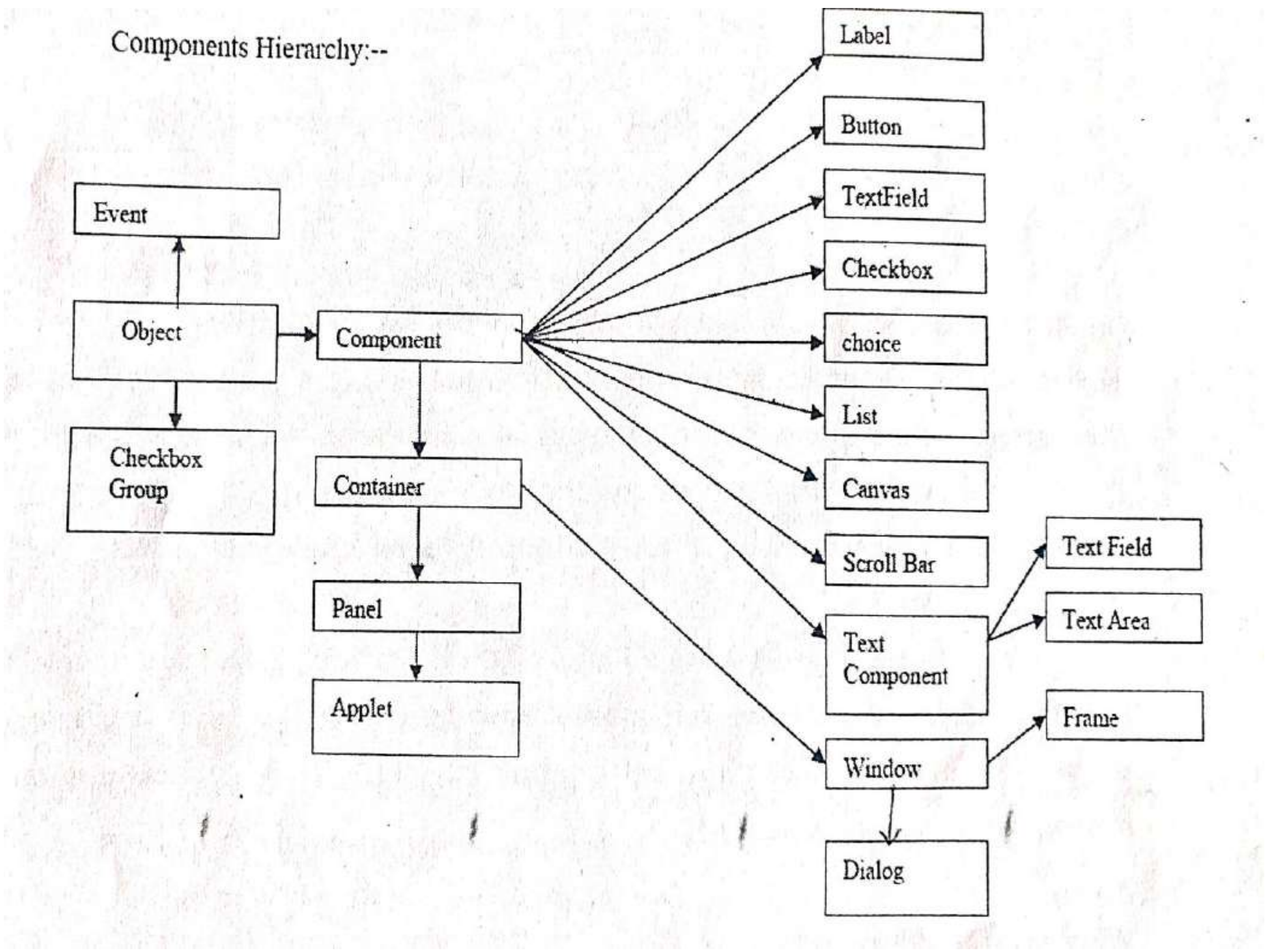
Mouse clicked

ABSTRACT WINDOW TOOLKIT (AWT)

The main purpose of the AWT is to support applet windows; it can also be used to create stand-alone windows that run in a GUI environment, such as windows.

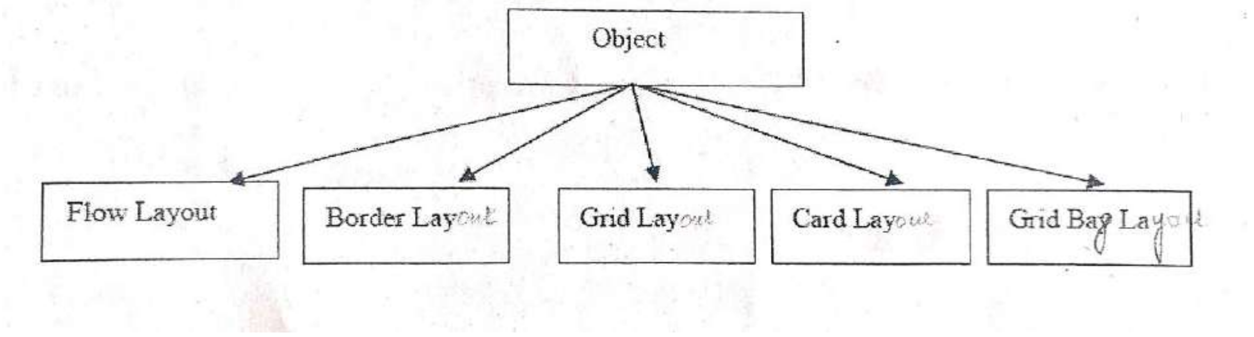
Window Fundamentals

Component: At the top of the AWT hierarchy is the **Component** class. **Component** is an abstract class that encapsulates all of the attributes of a visual component. All user interface elements that are displayed on the screen and that interact with the user are subclasses of **Component.**

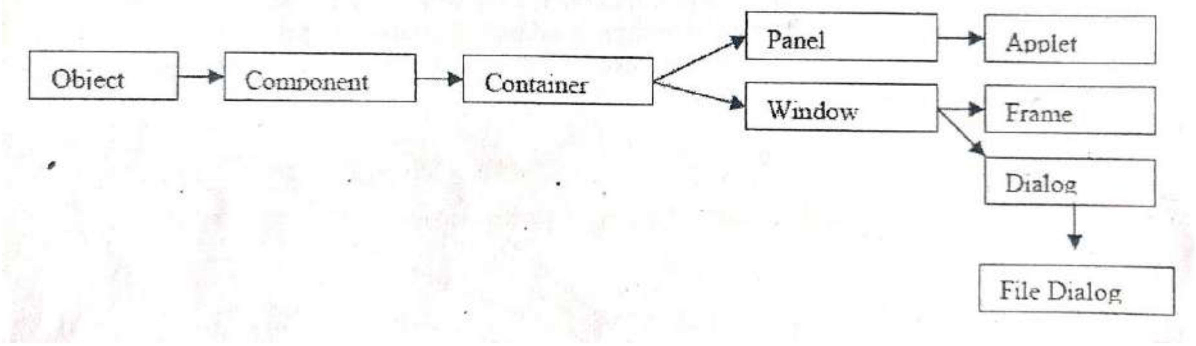


Container: The Container class is a subset of **Component**. It has additional methods that allow other **Component** objects to be nested with in it. Other **Container** object can be stored inside of a **Container.** This makes for a multileveled containment system. A container is responsible for laying out(that is positioning) any components that it contains. It does this through the use of various layout managers.

Layout Managers Hierarchy:



Container and components are abstract classes. Container holds the components.
Ex:- window, frame, applet, panel, dialog etc……..



Panel: The **Panel class** is a concrete subclass of **Container.** It doesn't add any new methods; it simply implements **Container. Panel** is the superclass for **Applet.** When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. When we run an applet using an applet viewer, the applet viewer provides the title and border.

Other components can be added to a **Panel** object by its **add()** method (inherited from **Container**)**.** Once these components have been added, we can position and resize them manually using the **setLocation()**, **setSize(), setBounds()** methods defined by **Component.**

Window: The **Windows** class creates a top-level window. A *top-level* window is not contained within any other object ; it sits directly on the desktop. Generally , we won't create window object directly. Instead , we use a subclass of **window** called **Frame.**

Frame: it is subclass of **window** and has a title bar, menu bar, borders, and resizing corners. When a **Frame** window is created by a program rather then an applet, a normal window is created.

Canvas: It is not part of the hierarchy for applet or frame windows. **Canvas** encapsulated a blank window upon which we can draw.

Working with Fonts

The AWT supports multiple type fonts. Fonts have a family name, a logical font name, and a face name. The *family name* is the general name of the font, such as Courier. The *logical name* specifies a category of font, such as Monospaced. The *face name* specifies a specific font, such as Courier Italic.
Fonts are encapsulated by the **Font** class.

The **Font** class defines these variables:

Variable	**Meaning**
String name	Name of the font
Float pointsize	Size of the font in points
int size	Size of the font in points
int style	Font style

Creating and Selecting a Font

To select a new font, we must first construct a **Font** object that describes that describes that font. One **Font** constructor has this general form:

    Font (String *fontName,* int *fontSyle,*int *pointSize*)

Here, *fontname* specifies the name of the desired font.
The style of the font is secified by *fontStyle*. It may consist of one or more of these three constants: **Font.PLAIN, Font,BOLD,** and **Font.ITALIC.**
The size, in points, of the font is specified by *pointSize.*
To use a font that we have created, we must select it using **setFont(),** which is defined by **Component.** It has this general form:

    void setFont(Font *fontObj*)

Obtained Fort Information

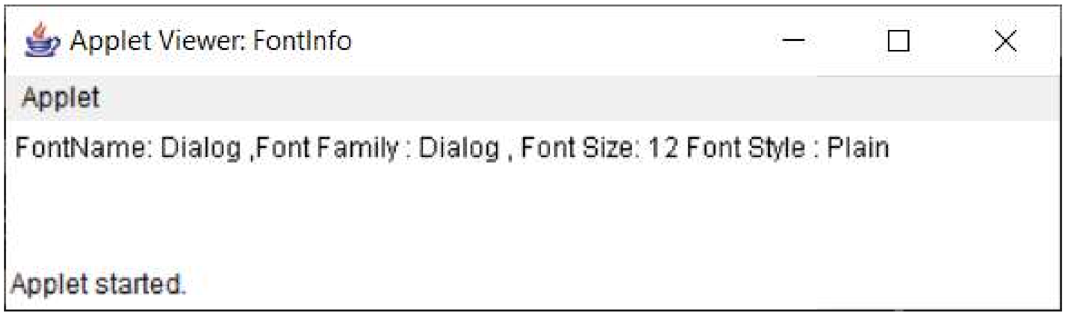
Suppose we want to obtain information about the currently selected font. To do this, we must first get the current font by calling **getFont().** This method is defined by the **Graphics** class, as shown here:

    Font getFont()

Some Methods of Font

Method	**Description**
Static Font decode(String str)	Returns a font given its name
Boolean equals(Object *FontObj*)	Returns **true** if the invoking object contains the same font as specified by FontObj. otherwise, it returns **false.**
String getFamily()	Returns the name of the font family to which the invoking font belongs.
String getFont(String property)	Returns the font associated with the system property specified by *property*. **Null** is returnedif *Property* does not exit.
static Font getFont(String property, Font defaultFont)	Returns the font associated with the system property specified by *property*. The font specified by *default* Font is returned if *property* doesn't exist.
String getFontName()	Returns the face name of the invoking font. (Added by Java 2)
String getName()	Returns the logical name of the invoking font.
int getSIze()	Returns the size, in points, of the invoking font.
int getStyle()	Returns the styles values of the invoking font.
int hashCode()	Returns the hash code associated with the invoking object.
Boolean is Bold()	Returns **true** if the font includes the **BOLD** style value. Otherwise, **false** is returned.
Boolean isItalic()	Returns **true** if the font includes the **ITALIC** style value. Otherwise, **false** is returned.
Boolean is Plain()	Returns **true** if the font includes the **PLANE** style value. Otherwise, **false** is returned.
String toString()	Returns the string equivalent of the invoking font.

```
import java.applet.*;
import java.awt.*;
/* <applet code="FontInfo" width=350 height=60>    </applet> */

public class FontInfo extends Applet
{       public void paint(Graphics g)
        {
                Font f = g.getFont();
                String fontName = f.getName();
                String fontFamily = f.getFamily();
                int fontSize=f.getSize();
                int fontStyle=f.getStyle();
                String msg="Family: " + fontName;
                msg += " ,Font: "+fontFamily;
                msg+=" , Size: "+fontSize + " , Style: ";
                if((fontStyle & Font.BOLD) == Font.BOLD)
                        msg+="Bold";
                if((fontStyle & Font.ITALIC) == Font.ITALIC)
                        msg+="Italic";
                if((fontStyle & Font.PLAIN) == Font.PLAIN)
                        msg+="Plain";
                g.drawString(msg,4,16);
        }
}
```



```
FontName: Dialog ,Font Family : Dialog , Font Size: 12 Font Style : Plain

Applet started.
```

Working with Color

Color is encapsulated by the **Color** class. **Color** defines several constants

1. Color(int *red,* int *green*, int *blue*)

This constructor takes three integers that specify the color as a mix of red, green, and blue. These values must be between 0 and 255

    new Color(255,100,100);

2. Color(int *rgbValue*)

It takes a single integer that contains the mix of red, green, and blue packed into an integer. The integer is organized with red in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7.

3. Color(float *red*, float *green*, float *blue*)

It takes three float values (between 0.0 and 1.) that specify the relative mix of red, green, and blue.

Once you have created a color, we can use it to set the foreground and/or background color by using the **setForeground()** and **setBackground()** methods.

Setting the Current Graphics Color

By default, graphics objects are drawn in the current foreground color. We can change this color by calling the **Graphics** method **setColor()**:

    void setColor(Color *newColor*)

Here, *newColor* specifies the new drawing color.

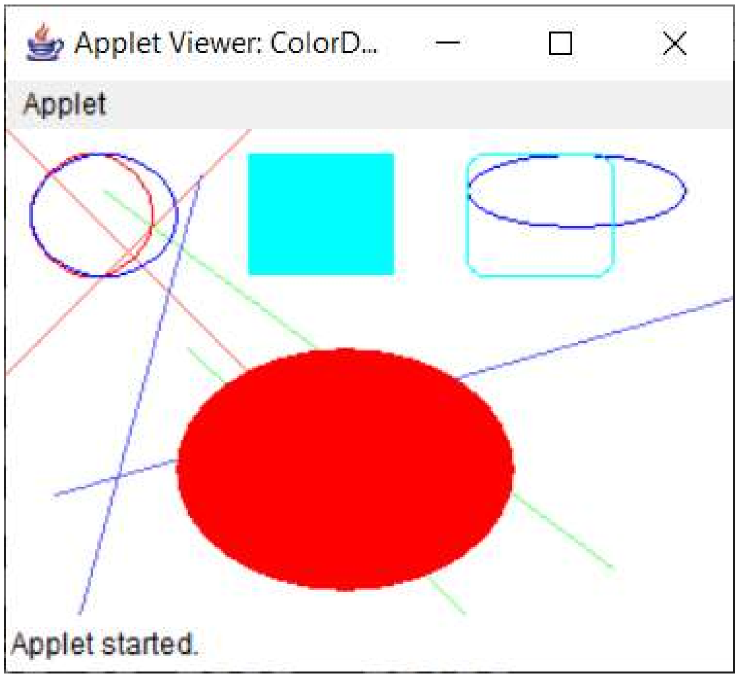
We can obtain the current color by calling **getColor()**, shown here:

    Color getColor()

// Demonstrate color

```
import java.applet.*;
import java.awt.*;
/* <applet code="ColorDemo" width=300 height=200></applet> */
public class ColorDemo extends Applet
{       public void paint(Graphics g)
        {
                Color c1 = new Color(255,100,100);
                Color c2 = new Color(100,255,100);
                Color c3 = new Color(100,100,255);
                g.setColor(c1);
                        g.drawLine(0,0,100,100);
                          g.drawLine(0,100,100,0);
```

```
                    g.setColor(c2);
                            g.drawLine(40,25,250,180);
                              g.drawLine(75,90,400,400);
                    g.setColor(c3);
                            g.drawLine(20,150,400,40);
                              g.drawLine(5,290,80,19);
                    g.setColor(Color.red);
                            g.drawOval(10,10,50,50);
                              g.fillOval(70,90,140,100);
                    g.setColor(Color.blue);
                            g.drawOval(190,10,90,30);
                              g.drawOval(10,10,60,50);
                    g.setColor(Color.cyan);
                            g.fillRect(100,10,60,50);
                              g.drawRoundRect(190,10,60,50,15,15);
    }
}
```



Working with Graphics

The origin of each window is at the top-left corner and is 0.0 . Coordinates are specifies in pixels. All output to a window takes place through a graphics context. A *graphics context* is encapsulated by the **Graphics** class and is obtained in two ways:

- It is passed to an applet when one of its various methods, such as **paint()** or **update()**, is called.
- It is returned by the **getGraphics()** method of **Component**.

The **Graphics** class defines a number of drawing functions.

Drawing Lines

Lines are drawn by means of the **drawLine()** method, shown here:

void drawLine(int *startX*, int *startY,* int *endY*, int *endX*)

drawLine() displays a line in the current drawing color that begins at *startX, startY* and ends at *endX, endY.*

Drawing Rectangles

The **drawRect()** and **fillRect()** methods display an outlined and filled rectangle, respectively. They are shown here:

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

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

The upper-left corner of the rectangle is at *top, left.* The dimensions of the rectangle are specified by *width* and *height*.

To draw a rounded rectangle, use **drawRoundRect()** or **fillRoundRect()**

void drawRoundRect(int *top*, int *left*, int *width*, int *height*, int *xDiam,* int *yDiam*)

void fillRoundRect(int *top*, int *left*, int *width*, int *height*, int *xDiam,* int *yDiam*)

Drawing Ellipses and Circles

To draw an ellipse, use **drawOval()**. To fill an ellipse, use **fillOval()**. These methods are shown here:

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

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

Drawing Arcs

Arcs can be drawn with **drawArc()** and **fillArc()**, shown here:

void drawArc(int *top*, int *left*, int *width*, int *height,* int *startAngle*, int *sweepAngle*)

void fillArc(int *top*, int *left*, int *width*, int *height,* int *startAngle*, int *sweepAngle*)

Drawing Polygons

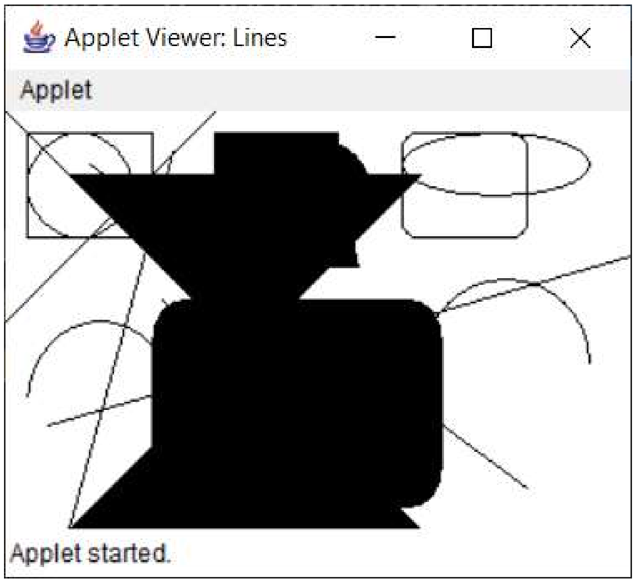
It is possible to draw arbitrarily shaped figures using **drawPolygon()** and **fillPolygon()** shown here:

void drawPolygon(int *x[]*, int *y[]*, int *numPoints*)

void fillPolygon(int *x[]*, int *y[]*, int *numPoints*)

Demonstrate Graphics

```
import java.awt.*;
import java.applet.*;
/* <applet code="Lines" width=300 height=200></applet>*/

public class Lines extends Applet
{
        public void paint(Graphics g)
        { // Draw Lines
                g.drawLine(0,0,100,100);
                g.drawLine(0,100,100,0);
                g.drawLine(40,25,250,180);
                g.drawLine(75,90,400,400);
                g.drawLine(20,150,400,40);
                g.drawLine(5,290,80,19);
         // Draw Rectangles
                g.drawRect(10,10,60,50);
                g.fillRect(100,10,60,50);
                g.drawRoundRect(190,10,60,50,15,15);
                g.fillRoundRect(70,90,140,100,30,40);
         // Draw Ellipses
                g.drawOval(10,10,50,50);
                g.fillOval(100,10,75,50);
                g.drawOval(190,10,90,30);
                g.fillOval(70,90,140,100);
         // Draw Arcs
                g.drawArc(10,40,70,70,0,75);
                g.fillArc(100,40,70,70,0,75);
                g.drawArc(10,100,70,80,0,175);
                g.fillArc(100,100,70,90,0,270);
                g.drawArc(200,80,80,80,0,180);
         // Draw Polygons
                int xpoints[]={30,200,30,200,30};
                int ypoints[]={30,200,30,200,30};
                int num=5;
                g.drawPolygon(xpoints,ypoints,num);
        }
}
```


Applet Viewer: Lines
Applet
Applet started.

AWT CONTROLS

Controls: Controls are components that allow a user to interact with our application in various ways—for example, a commonly used control is the push button.

Layout manager: A layout manager automatically positions components within a container. Thus, the appearance of a window is determined by a combination of the controls that it contains and the layout manager used to position them.

Adding and Removing Controls:

To include a control in a window, we must add it to the window.

Syn: Component add(Component compObj)

compObj is an instance of the control that we want to add.

To remove a control from a window when the control is no longer needed.

Syn: void remove(Component obj)

Buttons:

A push button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button. Button** defines these two constructors:

Button() -> creates an empty button

Button(String str) -> creates a button that contains *str* as a label

Void setLabel(String str)-> set its label

String getLabel() -> retrieve its label

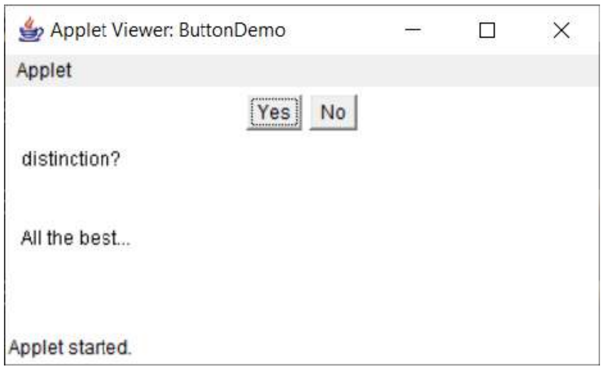
Handling Buttons

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed()** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method.

//Demonstrate Buttons

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="BottonDemo" width=250 height=150>
</applet> */
```

```
public class ButtonDemo extends Applet implements ActionListener
{       String msg="";
        Button yes,no;

        public void init()
        {
                yes=new Button("yes");
                no=new Button("no");
                add(yes); add(no);
                yes.addActionListener(this);
                no.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae)
        {
                String str=ae.getActionCommand();
                if(str.equals("yes"))
                {
                        msg="all the best";
                }
                else if(str.equals("no"))
                {
                        msg="have high confidence in yourself";
                }
                repaint();
        }
        public void paint(Graphics g)
        {       g.drawString("distinction?"10,20);
                g.drawsString(msg,6,100);
        }
}
```



```
distinction?

All the best...

Applet started.
```

Labels:

A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

Label() -> creates a blank label.

Label(String str) -> creates a label that contains the string specified by *str*. This string is left-justified.

Label(String *str*, int *how)* -> creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT, Label.RIGHT** or **Label.CENTER.**

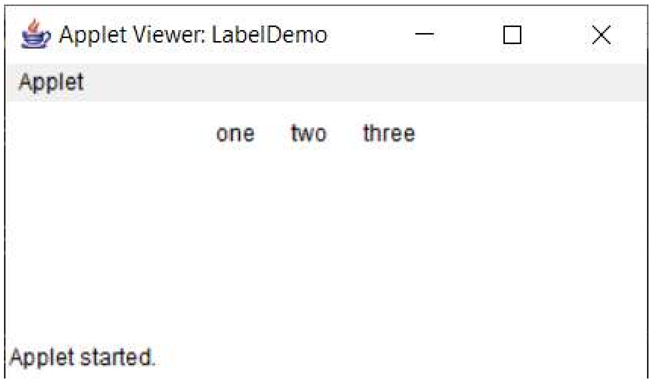
void setText(String *str*) -> set or change the text in the label

string getText() -> obtain the current label

void setAlignment(int *how*) -> set the alignment of the string within the label

int getAlignment() -> obtain the current alignment

```
//Demonstrate Labels
import java.awt.*;
import java.applet.*;
/* <applet code="LabelDemo" width=300 height=200>
</applet> */
public class LabelDemo extends Applet
{       public void init()
        {       Label one=new label("one");
                Label two=new label("two");
                Label three=new label("three");
                // add labels to applet window
                add(one);  add(two);  add(three);
        }
}
```



TextField

The **TextField** class implements a single-line text-entry area, usually called an *edit contol.* **TextField** is a subclass of **TextComponent.TextField** defines the following constructors:

TextField() -> creates a default text field
TextField(int *numChars*) -> creates a text field that is numchars characters wide
TextField(String str) -> initializes the text field with the string contained in str
TextField(String str, int numChars) -> initializes a text field and sets its width.

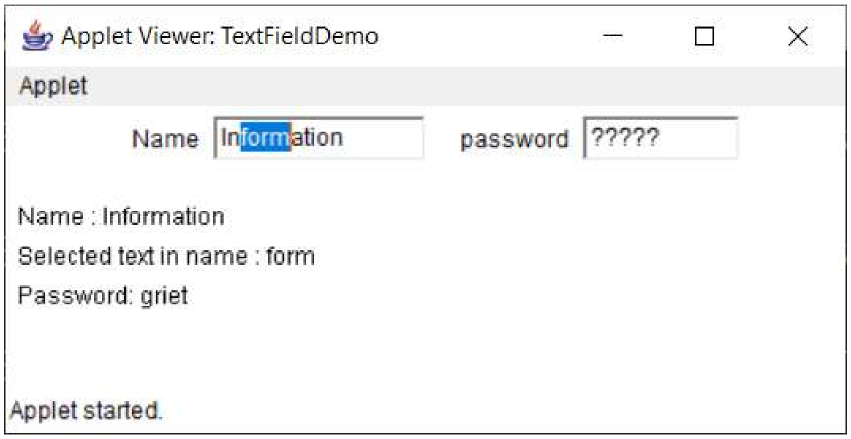
String getText() -> obtain the string currently contained in the text field
void setText(String str) -> to set the text
String getSelectedText() -> currently selected text
void select(int startIndex, int endIndex) -> selects the characters beginning at startIndex and ending at endIndex-1.

boolean isEditable() ->determines editability
void setEditable(boolean canEdit) -> if canEdit is true, the text may be changed. If it is false, the text cannot be altered

void setEchoChar(char ch) -> We will want the user to enter text that is not displayed, such as password
boolean echoCharIsSet() -> check a text field
char getEchoChar() ->retrieve the echo character

// Demonstrate Text Field

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
/*
<applet code="TextFieldDemo" width=380 height=150>
</applet>
*/
public class TextFieldDemo extends Applet implements ActionListener
{
        TextField name,pass;

        public void init()
        {
                Label namep=new Label("Name",Label.RIGHT);
                Label passp=new Label("password",Label.RIGHT);

                name=new TextField(12);
                pass=new TextField(8);
                pass.setEchoChar('?');
                add(namep);  add(name);
                add(passp);   add(pass);
// register to receive action events
                name.addActionListener(this);
                pass.addActionListener(this);
        }
        //user pressed enter.

        public void actionPerformed(ActionEvent ae)
        {
                repaint();
        }
        public void paint(Graphics g)
        {
                g.drawString("name:"+name.getText(),6,60);
                g.drawString("selected text in name"+name.getSelectedText(),6,80);
                g.drawString("password:"+pass.getText(),6,100);
        }
}
```



```
Name : Information
Selected text in name : form
Password: griet
```

Applet started.

TextArea

AWT includes a simple multiline editor called **TextArea.** Following are the constructors for **TextArea:**

```
TextArea()
TextArea(int numlines, int numchars)
TextArea(String str)
TextArea(String str, int numlines, int numchars)
TextArea(String str, int numlines, int numchars, int sBars)
```

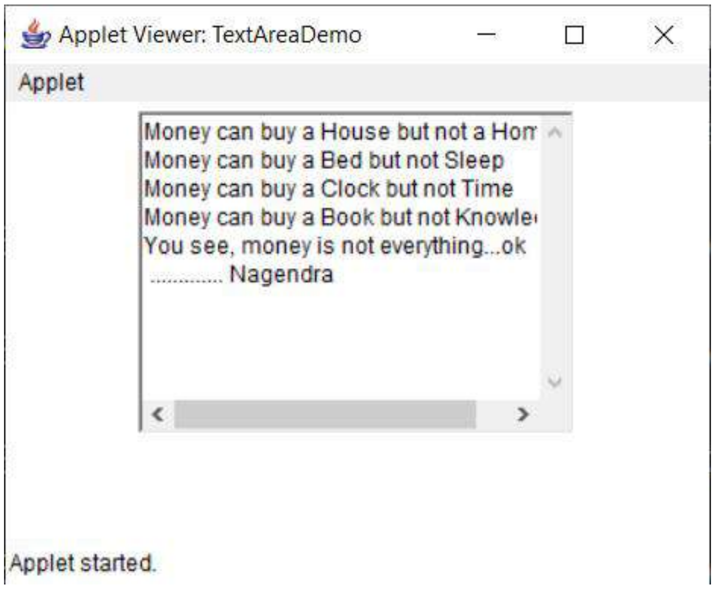
Here, numlines specifies the height, in lines, of the text area, and numChars specifies its width, in characters. Initial text can be specified by str. In the fifth form we can specify the scroll bars that you want the control to have. sBars must be one of these values:

SCROLLBARS_BOTH
SCROLLBARS_NONE
SCROLLBARS_HORIZONTAL_ONLY
SCROLLBARS_VERTICAL_ONLY

TextArea is a subclass of **TextComponent.** Therefore, it supports the **getText(), setText(), getSelectedText(), select(), isEditable(),** and **setEditable()** methods
TextArea adds the following methods:
void append(String str) -> appends the string specified by str to the end
void insert(String str, int index) ->inserts the string at the specified index
void replaceRange(String str, int startIndex, int endIndex) -> it replaces the characters from startIndex to endIndex-1, with the replacement text passed in str

//Demonstrate Text Area

```
import java.awt.*;
import java.applet.*;

/*
<applet code="TextAreaDemo" width=380 height=150>
</applet>
*/
public class TextAreaDemo extends Applet
{
        public void init()
        {
                String val ="Money can buy a horse but not a home \n"+
                                "Money can buy a bed but not sleep\n"+
                                "Money can buy a clock but not time\n"+
                                "Money can buy a book but not knowledge\n"+
                                "you see money is not everything...ok\n";

        TextArea text=new TextArea(val,10,30);
        add(text);
        }
}
```



```
Money can buy a House but not a Hom
Money can buy a Bed but not Sleep
Money can buy a Clock but not Time
Money can buy a Book but not Knowle
You see, money is not everything...ok
 ............ Nagendra
```

Applet started.

Check Boxes

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each box that describes what option the box represents.

Checkbox() supports these constructors:

Checkbox() -> creates a check box whose label is initially blank

Checkbox(String str) -> creates a check box whose label is specified by str

Checkbox(String str, boolean on) -> set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared

Checkbox(String str, boolean on, CheckboxGroup cbGroup)

Checkbox(String str, CheckboxGroup cbGroup, boolean on)->create a check box whose label is specified by str and whose group is specified by cbGroup. If this check box is not part of a group, then cbGroup must be null.

boolean getState() -> retrieve the current state of a check box

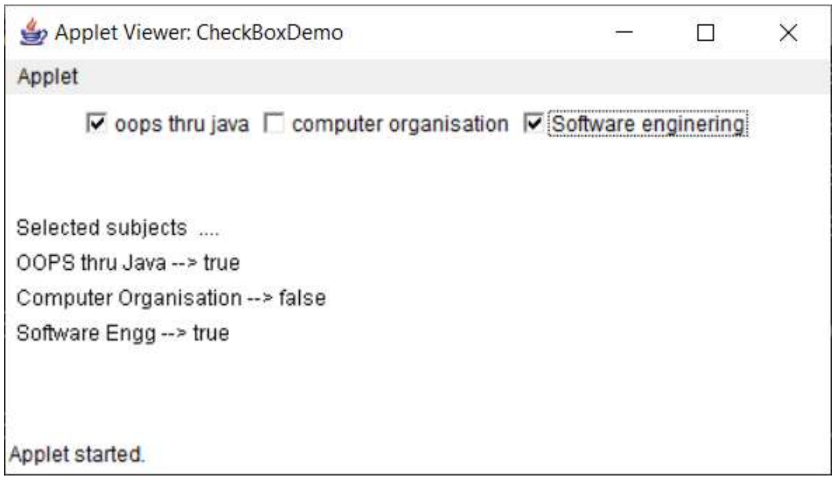
void setState(boolean on) -> to set its state
String getLabel() -> obtain the current label
void setLabel(String str) -> to set the label

Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This sent to any listeners that previously registered. Each listener implements the ItemListener interface. That interface defines the itemStateChanged() method. An ItemEvent object is supplied as the argument to this method. It contains information about the event.

//Demonstrate Check boxes

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

/*<applet code="CheckBoxDemo" width=250 height=200>
</applet>
*/
public class CheckBoxDemo extends Applet implements ItemListener
{
        String msg="";
        Checkbox javaa,cg,se;

        public void init()
        {
                javaa=new Checkbox("oops thru java",null,true);
                cg=new Checkbox("computer graphics");
                se=new Checkbox("Software enginering");
                add(javaa);add(cg);add(se);
                javaa.addItemListener(this);
                cg.addItemListener(this);
                se.addItemListener(this);
        }

        public void itemStateChanged(ItemEvent ie)
        {
                repaint();
        }
        // display current state of the check boxes.
        public void paint(Graphics g)
        {       msg="Selected subjects";
                        g.drawString(msg,6,80);
                msg="oops thru java"+javaa.getState();
                        g.drawString(msg,6,100);
                msg="computer graphics"+cg.getState();
                        g.drawString(msg,6,120);
                msg="software engg"+se.getState();
                g.drawString(msg,6,140);
        }
}
```



```
Selected subjects  ....
OOPS thru Java --> true
Computer Organisation --> false
Software Engg --> true
```

Applet started.

<u>CheckboxGroup</u>

It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called <u>Radio buttons.</u>

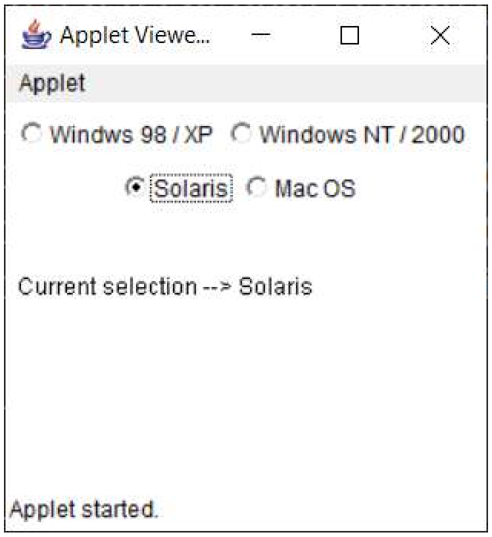
Checkbox getSelectedCheckbox() -> currently selected Checkbox
void setSelectedCheckbox(Checkbox which) -> set a check box

Program:

// Demonstrate check box group

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet code="CBGroup" width=250 height=200>
</applet> */
```

```
public class CBGroup extends Applet implements ItemListener
{       String msg="";
        Checkbox win98,winNT,solaris,mac;
        CheckboxGroup cbg;

        public void init()
        {
                cbg=new CheckboxGroup();
                win98=new Checkbox("Windows 98/xp",cbg,true);
                winNT=new Checkbox("windows nt/2000",cbg,false);
                solaris=new Checkbox("solaris",cbg,false);
                mac=new Checkbox("mac os",cbg,false);
                add(win98); add(winNT); add(solaris); add(mac);
                win98.addItemListener(this);        winNT.addItemListener(this);
                solaris.addItemListener(this);      mac.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {       repaint();
        }
        //Display current state of the check boxes.
        public void paint(Graphics g)
        {
                msg="Current selection";
                msg+=cbg.getSelectedCheckbox().getLabel();
                g.drawString(msg,6,100);
        }
}
```



```
Current selection --> Solaris
```

Choice

The **choice** class is used to create a *pop-up* list of items from which the user may chose.

Void add(String name) -> To add selection to the list

String getSelectedItem() -> To determine which item is currently selected, returns a string containg the name of the item

int getSelectedIndex() -> returns the index of the item

int getItemCount() ->To obtain the number of items in the list

void select(int index) -> currently selected item using index

void selected(String name) -> currently selected item using name

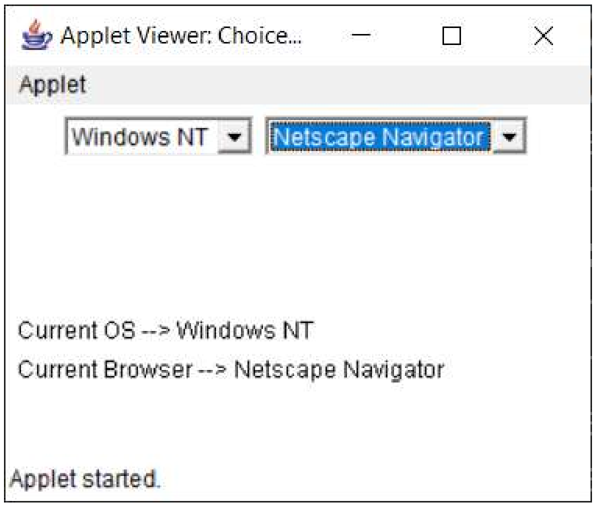
String getItem(int *index*) -> obtain the name associated with the item at that index

Handling Choice Lists

Each time a choice is selected, an item event is generated. This is sent to any Listeners that previously registered. Each listener implements the **ItemListener** Interface. That interface defines the **itemStateChange()** method. An **ItemEvent** Object is selected as the argument to this method.

Program:

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

/* <applet code="ChoiceDemo" width=300 height=180>
</applet> */

public class ChoiceDemo extends Applet implements ItemListener
{
        Choice os,browser;
        String msg="";
        public void init()
        {       os=new Choice();
                browser=new Choice();

                os.add("Windows 98");
                os.add("Windows NT");
                os.add("Solaris");

                browser.add("Mozilla");
                browser.add("Netscape Navigator");
                browser.add("IE 6");
                browser.add("Lynx 2.4");
                browser.select("IE 6");

                //add choice lists to window
                add(os);
                add(browser);
                //register to receive item events
                os.addItemListener(this);
                browser.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {       repaint();
        }
        //display current selections.

        public void paint(Graphics g)
        {
                msg="Current OS --> ";
                msg+=os.getSelectedItem();
                g.drawString(msg,6,120);

                msg="Current Browser --> ";
                msg+=browser.getSelectedItem();
                g.drawString(msg,6,140);
        }
}
```



```
Current OS --> Windows NT
Current Browser --> Netscape Navigator
```

Applet started.

Lists

The **list** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu.
List provides these constructors:
List() -> allows only one item to be selected at one time
List(int numRows) -> the value of numRows specifies the number of entries in the list that will always be visible
List(int numRows, boolean multipleSelect) -> if multipleSelect is **true**, then the user may select two or more items at a time. If it is **false,** then only one item may be selected.

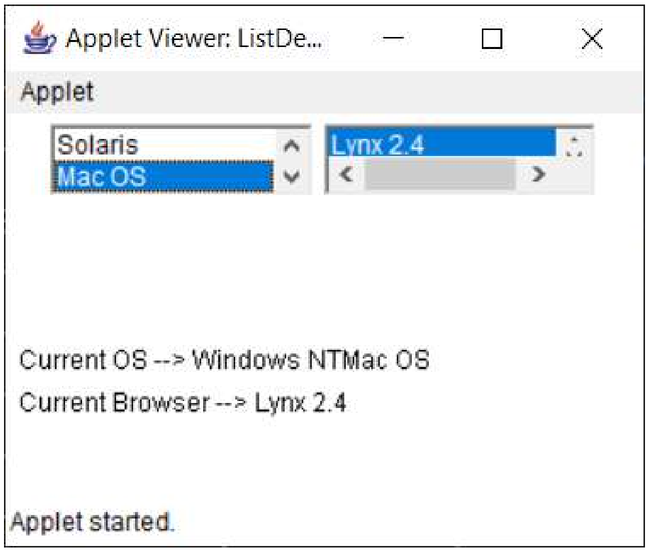
void add(String name) -> adds item to the end of the list
void add(String name, int index) -> adds the item at the index specified by index
Indexing begins at zero. You can specify -1 to add the item to the end of the list.
String getSelectedItem() -> returns a string containing the name of the item. If more than on item has been selected or if no selection has been made, -1 is returned.

String[] getSelectedItems() -> returns an array containing the names of the currently selected items
int[] getSelectedIndexes() -> returns an array containing the indexes of the currently selected items
int getItemCount() -> obtain the number of items in the list
void select(int index) -> currently selected item
String getItem(int index) -> obtain the name associated with the item at the index

Handling Lists
We will need to implement the ActionListener interface. Each time a List item is double-clicked, an ActionEvent object is generated.

Program:
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*<applet code="ListDemo" width=300 height=180>
</applet>
*/
```

```
public class ListDemo extends Applet implements ActionListener
{
  List os,browser;
  String msg="";

  public void init()
  {       os=new List(2,true);
          browser=new List(2,false);

          //add items to os list
          os.add("Windows 98");
          os.add("Windows NT");
          os.add("Solaris");
          os.add("Mac OS");

          browser.add("Mozilla");
          browser.add("Netscape Navigator");
          browser.add("IE 6");
          browser.add("Lynx 2.4");
          browser.select(3);

          //add choice lists to window

          add(os);
          add(browser);

          //register to receive action events

          os.addActionListener(this);
          browser.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae)
  {
          repaint();
  }

  //display current selections.
  public void paint(Graphics g)
  {
          int idx[];

          msg="Current OS --> ";

          idx=os.getSelectedIndexes();

          for(int i=0;i<idx.length;i++)
                  msg+=os.getItem(idx[i])+"";

          g.drawString(msg,6,120);

          msg="Current Browser --> ";

          msg+=browser.getSelectedItem();
          g.drawString(msg,6,140);
  }
}
```



```
Current OS --> Windows NTMac OS
Current Browser --> Lynx 2.4
```

Scroll Bars

Scroll bars are used select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. The current value of the scroll bar relative to its minimum and maximum values is indicated by the *slider box(or thumb)* for the scroll bar.

Scrollbar defines the following constructors:

Scrollbar() -> creates a vertical scroll bar

Scrollbar(int style) -> specify the orientation of the scroll bar. If style is **Scrollbar.VERTICAL,** a vertical scroll bar is created. It style is **Scrollbar.HORIZONTAL,** the scrollbar is horizontal.

Scrollbar(int style, int initialValue, int thumSize, int min, int max) -> the initial value of the scroll bar is passed in the initialValue. The number of units represented by the height of the thumb is passed in thumbSize. The minimum and maximum values for the scroll bar is specified by min and max.

void setValues(int initialValues, int thumdSize,int min, int max) -> set its parameters

int getValue() -> obtain the current value of the scroll bar

void setValue(int newValue) -> set the current value

int getMinimum() -> retrieve the minimum value

int setMaximum() -> retrieve the maximum value

Handling Scroll Bars

Implement using **AdjustmentListener** interface. Each time a user interacts with a scroll bar, an **AdjustmentEvent** object is generated. Its **getAdjustmentType()** method can be used to determine the type of the adjustment.

The types of adjustment events are as follows:

BLOCK_DECREMENT A page-down event has been generated.

BLOCK_INCREMENT A page-up event has been generated.

TRACK An absolute tracking event has been generated.

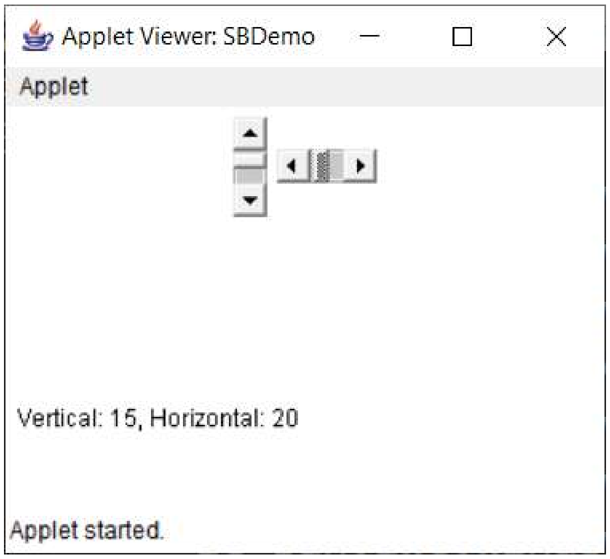
UNIT_DECREMENT The line-down button in a scroll bar has been pressed.

UNIT_INCREMENT The line-up button in a sroll bar has been pressed.

Program:

```
// Demonstrate scroll bars
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/* <applet code="SBDemo" width=300 height=200></applet>*/
```

```
public class SBDemo extends Applet implements AdjustmentListener, MouseMotionListener
{
        String msg="";
        Scrollbar vertSB,horzSB;

        public void init()
        {
                int width=Integer.parseInt(getParameter("width"));
                int height=Integer.parseInt(getParameter("height"));

                vertSB=new Scrollbar(Scrollbar.VERTICAL,0,1,0,height);
                horzSB=new Scrollbar(Scrollbar.HORIZONTAL,0,1,0,width);

                add(vertSB);
                add(horzSB);

                vertSB.addAdjustmentListener(this);
                horzSB.addAdjustmentListener(this);
                addMouseMotionListener(this);
        }

        public void adjustmentValueChanged(AdjustmentEvent ae)
        {
                repaint();
        }
        public void mouseDragged(MouseEvent me){}
        public void mouseMoved(MouseEvent me){}

        public void paint(Graphics g)
        {       msg="Vertical: "+ vertSB.getValue();
                msg+=", Horizontal: " + horzSB.getValue();
                g.drawString(msg,6,160);
        }
}
```



```
Vertical: 15, Horizontal: 20
```

Menu Bars and Menus

A top level windows will have a menu bar associated with it. A menu bar displays a list of top-level menu choices. Each choice is associated with a drop-down menu. This concept is implemented in java by the following classes : **MenuBar,Menu** and **MenuItem.** In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of **MenuItems** objects. Each **MenuItem** object represents something that can be selected by user.

To create a menubar, first create an instance of **MenuBar**

Menu()

Menu(String *optionName*)

Menu(String *optionName*, boolean *removable*)

optionName specifies the name of the menu selection. If *removable* is **true,** the pop-up menu can be removed and allowed to float free. Otherwise, it will remain attached to the menu bar.

MenuItem defines these constructors:

MenuItem()

MenuItem(String *itemName*)

MenuItem(String *itemName,* MenuShortcut *keyAccel*) void setEnabled(boolean *enabledFlag)* → disabled or enabled a menu item boolean isEnabled() → returns true if the menu item on which it is called is enabled. Otherwise, it returns false.

void setLabel(String *newName*) → changes the name of a menu item

We can create a checklabel menu item by using a subclass of **MenuItem** called **CheckMenuItem**. It has these constructors :

CheckboxMenuItem()

CheckboxMenuItem(String *itemName*)

CheckboxMenuItem(String *itemName,* boolean *on*)

boolean getState() → If the item is checked it return true. Otherwise, false void setState(boolean checked) → To check an item, pass true to setState(). To clear an item, pass false.

MenuItem add(MenuItem item) → add the item to a Menu object

Each time a menu item is selected, an ActionEvent object is generated. Each time a check box menu item is checked or unchecked, an ItemEvent object is generated. Thus, we must implement the ActionListener and ItemListener interface in order to handle these events.

Object getItem( ) → The **getItem( )** method of **ItemEvent** returns a reference to the item that generated this event.

Program:

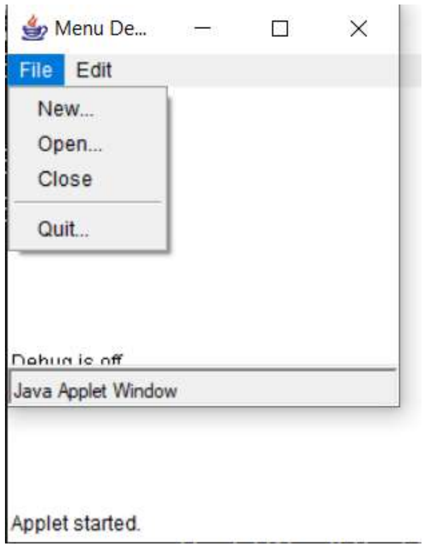
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*
<applet code="MenuDemo" width=250 height=250>
</applet>
*/
```

```
// Create a subclass of Frame
class MenuFrame extends Frame
{
                                        String msg = "";
                                        CheckboxMenuItem debug, test;

                                        MenuFrame(String title)
                                        {
                                            super(title);
                                            // create menu bar and add it to frame
                                            MenuBar mbar = new MenuBar();
                                            setMenuBar(mbar);

                                            // create the menu items
                                            Menu file = new Menu("File");
                                            MenuItem item1, item2, item3, item4, item5;
                                            file.add(item1 = new MenuItem("New..."));
                                            file.add(item2 = new MenuItem("Open..."));
                                            file.add(item3 = new MenuItem("Close"));
                                            file.add(item4 = new MenuItem("-"));
                                            file.add(item5 = new MenuItem("Quit..."));
                                            mbar.add(file);

                                            Menu edit = new Menu("Edit");
                                            MenuItem item6, item7, item8, item9;
                                            edit.add(item6 = new MenuItem("Cut"));
                                            edit.add(item7 = new MenuItem("Copy"));
                                            edit.add(item8 = new MenuItem("Paste"));
                                            edit.add(item9 = new MenuItem("-"));

                                            Menu sub = new Menu("Special");
                                            MenuItem item10, item11, item12;
                                            sub.add(item10 = new MenuItem("First"));
                                            sub.add(item11 = new MenuItem("Second"));
                                            sub.add(item12 = new MenuItem("Third"));
                                            edit.add(sub);

                                            // these are checkable menu items
```

```
        debug = new CheckboxMenuItem("Debug");
        edit.add(debug);
        test = new CheckboxMenuItem("Testing");
        edit.add(test);
        mbar.add(edit);

        // create an object to handle action and item events
        MyMenuHandler handler = new MyMenuHandler(this);

        // register it to receive those events
        item1.addActionListener(handler);
        item2.addActionListener(handler);
        item3.addActionListener(handler);
        item4.addActionListener(handler);
        item5.addActionListener(handler);
        item6.addActionListener(handler);
        item7.addActionListener(handler);
        item8.addActionListener(handler);
        item9.addActionListener(handler);
        item10.addActionListener(handler);
        item11.addActionListener(handler);
        item12.addActionListener(handler);

        debug.addItemListener(handler);
        test.addItemListener(handler);

        // create an object to handle window events
        MyWindowAdapter adapter = new MyWindowAdapter(this);

        // register it to receive those events
        addWindowListener(adapter);
    }

    public void paint(Graphics g)
    {
        g.drawString(msg, 10, 200);

        if(debug.getState())
```

```
                                        g.drawString("Debug is on.", 10, 220);
                                    else
                                        g.drawString("Debug is off.", 10, 220);

                                    if(test.getState())
                                        g.drawString("Testing is on.", 10, 240);
                                    else
                                        g.drawString("Testing is off.", 10, 240);
                                }
}
class MyWindowAdapter extends WindowAdapter
{
                                MenuFrame menuFrame;

                                public MyWindowAdapter(MenuFrame menuFrame)
                                {
                                    this.menuFrame = menuFrame;
                                }
                                public void windowClosing(WindowEvent we)
                                {
                                    menuFrame.setVisible(false);
                                }
}
class MyMenuHandler implements ActionListener, ItemListener
{
                                MenuFrame menuFrame;
                                public MyMenuHandler(MenuFrame menuFrame)
                                {
                                    this.menuFrame = menuFrame;
                                }
                                // Handle action events
                                public void actionPerformed(ActionEvent ae)
                                {
                                    String msg = "You selected ";
                                    String arg = (String)ae.getActionCommand();

                                    if(arg.equals("New..."))
                                        msg += "New.";
                                        else if(arg.equals("Open..."))
                                                msg += "Open.";
                                        else if(arg.equals("Close"))
                                                msg += "Close.";
                                        else if(arg.equals("Quit..."))
                                                msg += "Quit.";
                                        else if(arg.equals("Edit"))
                                                msg += "Edit.";
                                        else if(arg.equals("Cut"))
                                                msg += "Cut.";
                                        else if(arg.equals("Copy"))
                                                msg += "Copy.";
                                        else if(arg.equals("Paste"))
                                                msg += "Paste.";
                                        else if(arg.equals("First"))
                                                msg += "First.";
                                        else if(arg.equals("Second"))
                                                msg += "Second.";
                                        else if(arg.equals("Third"))
                                                msg += "Third.";
                                        else if(arg.equals("Debug"))
                                                msg += "Debug.";
                                        else if(arg.equals("Testing"))
                                                msg += "Testing.";
                                        menuFrame.msg = msg;
                                        menuFrame.repaint();
                                }
                                // Handle item events
                                public void itemStateChanged(ItemEvent ie)
                                {
                                        menuFrame.repaint();
                                }
}
```

```
// Create frame window.
public class MenuDemo extends Applet
{
                                Frame f;
                                public void init()
                                {
                                   f = new MenuFrame("Menu Demo");
                                   int width = Integer.parseInt(getParameter("width"));
                                   int height = Integer.parseInt(getParameter("height"));
                                   setSize(new Dimension(width, height));
                                   f.setSize(width, height);
                                   f.setVisible(true);
                                }
                                public void start()
                                {
                                   f.setVisible(true);
                                }
                                public void stop()
                                {
                                   f.setVisible(false);
                                }
}
```



LAYOUT MANAGERS

A layout manager automatically arranges our controls within a window. A layout manager is a instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. If no calls to **setLayout( )** is made, then default layout manager is used.

FlowLayout :

FlowLayout is the default layout manager. Components are laid out from the upper-left corner, left to right and top to bottom. When on more components fit on line, the next one appears on the next line.

FlowLayout( )

FlowLayout(int *how*)

FlowLayout(int *how,*int *horz,*int *vert*)

Valid values for the *how* are as follows:

FlowLayout.LEFT

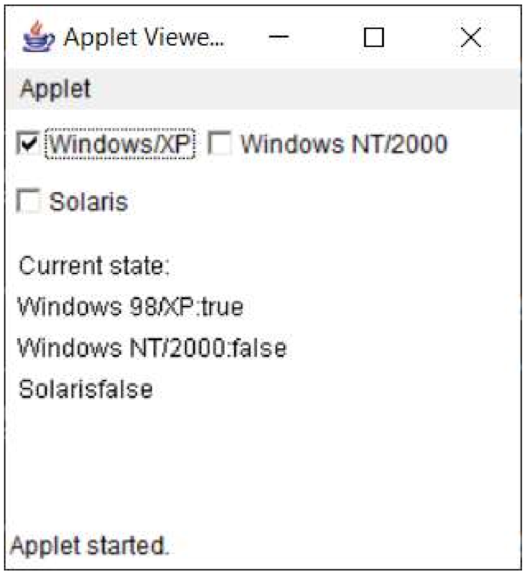
FlowLayout.CENTER

FlowLayout.RIGHT

The third form allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

PROGRAM:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class FlowLayoutDemo extends Applet implements ItemListener
{
String msg="";
Checkbox Win98,winNT,solaris,mac;
public void init()
{
                                setLayout(new FlowLayout(FlowLayout.LEFT));
                                Win98=new Checkbox("Windows/XP",null,true);
                                winNT=new Checkbox("Windows NT/2000");
                                solaris=new Checkbox("Solaris");
                                add(Win98);
                                add(winNT);
                                add(solaris);
                                Win98.addItemListener(this);
                                winNT.addItemListener(this);
                                solaris.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie)
{
                                repaint();
}
public void paint(Graphics g)
{
                                msg="Current state:";
                                g.drawString(msg,6,80);

                                msg="Windows 98/XP:"+Win98.getState();
                                g.drawString(msg,6,100);

msg="Windows NT/2000:"+winNT.getState();
g.drawString(msg,6,120);

msg="Solaris"+solaris.getState();
g.drawString(msg,6,140);
}
}
/*
<applet code="FlowLayoutDemo.class" width=250 height=200>
</applet>
*/
```



```
Current state:
Windows 98/XP:true
Windows NT/2000:false
Solarisfalse
```

Applet started.

BorderLayout

It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north,south,east and west. The middle area is called the center. Here are the constructors defined by **BorderLayout.**

BorderLayout( )

BorderLayout(int *horz*, int *vert*)

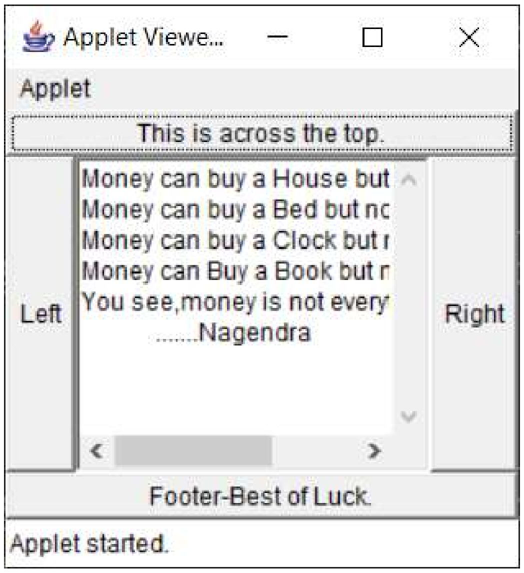
This first form creates a default border layout. The second allows us to specify the horizontal and vertical spaces left between components in *horz* and *vert*, respectively. **BorderLayout** defines the following constant that specify the regions :

BorderLayout.CENTER
BorderLayout.SOUTH
BorderLayout.EAST
BorderLayout.WEST
BorderLayout.NORTH

void add(Component *compObj,* Object *region*); → *compObj* is the component to be added, and *region* specifies where the component will be added.

PROGRAM

```
//Demostrator BorderLayout

 import java.awt.*;
 import java.applet.*;
 import java.util.*;

 /*<applet code="BorderLayoutDemo" width=400 height=200></applet>*/

 public class BorderLayoutDemo extends Applet
 {   public void init()
    {     setLayout(new BorderLayout());

add(new Button("This is across the top."),BorderLayout.NORTH);
add(new Button("Footer-Best of Luck."),BorderLayout.SOUTH);
add(new Button("Right"),BorderLayout.EAST);
add(new Button("Left"),BorderLayout.WEST);

String msg="Money can buy a House but not a Home\n"+
                                    "Money can buy a Bed but not Sleep\n"+
                                    "Money can buy a Clock but not Time\n"+
                                    "Money can Buy a Book but not Knowledge\n"+
                                            "You see,money is not everything .. Ok\n"+
                                            "           .......Nagendra";

                                      add(new TextArea(msg), BorderLayout.CENTER);
    }
 }
```



```
Money can buy a House but
Money can buy a Bed but nc
Money can buy a Clock but r
Money can Buy a Book but r
You see,money is not every
        .......Nagendra
```

Applet started.

GridLayout

GridLayout lays out components in a tow-dimensional grid. We specify the number of rows and columns, the number of rows only and let the layout manager determine the number of columns, or the number of columns only and let the layout manager determine the number of rows.

The cells in the grid are equal size based on the largest component in the grid.

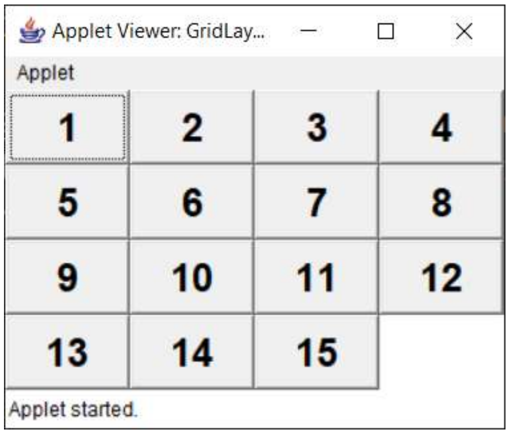
GridLayout() → creates a single-column grid layout

GridLayout(int *numRows,* int *numcolumn*) → creates a grid layout with the specified number of rows and columns.

GridLayout(int *numRow,* int *numColumn,* int *horz,* int *vert*) → specify the horizontal and vertical spaces left between components.

PROGRAM

```
//demonstrate GridLayout

import java.awt.*;
import java.applet.*;

/*
 <applet code="GridLayoutDemo" width=300 height=200>
 </applet>
*/

public class GridLayoutDemo extends Applet
{
   static final int n=4;

    public void init()
    {
       setLayout(new GridLayout(n,n));
       setFont(new Font("SansSerif",Font.BOLD,24));

                                        for(int i=0;i<n;i++)
                                         {
                                           for(int j=0;j<n;j++)
                                             {
                                                     int k=i*n+j;
                                                     if(k>0)
                                                       add(new Button(""+ k));
                                             }
                                         }
    }
}
```



GridBagLayout

The Grid bag layout (like grid layout) arranges components into a grid of rows and columns, but lets us specify a number of settings. Unlike the grid layout, the rows and columns are not constrained to be a uniform size. For example, a component can be set to span multiple rows or columns, or we can change its position on the grid. Each GridBagLayout object maintains a dynamic, rectangular grid of cells, with each component occupying one or more cells, called its *display* area. Each component managed by a GridBagLayout is associated with an instance of GridBagConstraints.

The constraints objects specifies where a component's display area should be located on the grid and how the component should be positioned within its display area.

GridBagConstraints :

1. To adjust the anchor positioned of the component, click one of the compass buttons:

➔ **Anchor northwest** – Position the components in the upper-left corner of the grid cell.

➔ **Anchor north** -Positions the component in the top center of the grid cell.

➔ **Anchor west** – Positions the component in the left middle of the grid cell.

➔ **Anchor center** – Positions the component in the center of the grid.

➔ **Anchor east** – Positions the component in the middle of the grid cell.

➔ **Anchor southwest** – Positions the component in the lower-left corner of the grid cell.

➔ **Anchor south** – Positions the component in the bottom center of the grid cell.

➔ **Anchor southeast** – Positions the component in the lower-right corner of the grid cell.

2. To adjust the padding between the grid cell border and the component, enter a value (in pixels) for any of the following fields:

- **Top** – sets the inset, or padding, above the component.
- **Left** – sets the inset, or padding, to the left of the component.
- **Bottom** – sets the inset, or padding, below the component.
- **Right** – sets the inset, or padding, to the right of the component.

3. To adjust how much the component fills the grid cells, click either or both of fill buttons :

- **Fill horizontal** – specifies that the component should occupy the full height of the grid cell.
- **Fill vertical** – specifies that the component should occupy the full height of the grid cell.

4. To specify how many cells a component should span, enter values for **Width** and **Height :**

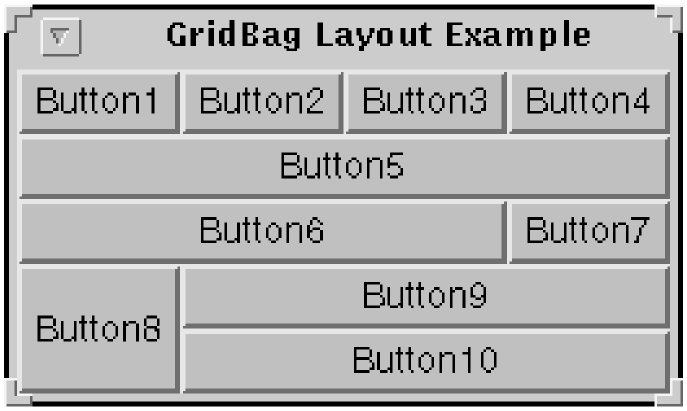
- **Width -** specifies the number of columns the components occupies (the x- axis).
- **Height -** specifies the number of rows the components occupies (the y- axis).

5. To specify internal padding for a component, enter values(in pixels) for the following fields :

- X –
- Y-

6. To specify how to distribute extra space across rows or columns, enter values(in relative numerical values, in relation to the weights specified for other components) for the **X** and **Y** fields.

- **X –** specifies the weight for distribution of extra space between columns.
- **Y –** specifies the weight for distribution of extra space between rows.



Card Layout:

The CardLayout class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.

CardLayout ()

CardLayout (int *horz*, int *vert*)

Specify the horizontal and vertical space left between components in horz and vert.

When card panels are added to a panel, they are usually given a name. Thus, most of the time, we will use this form of add () when adding cards to a panel:

Void add (Component *panelObj*, Object *name*);

Following methods defined by CardLayout:

Void first (Container *deck*)

void last(Container *deck*)

void next(Container *deck*)

void previous(Container *deck*)

void show(Container *deck*, String *cardName*)

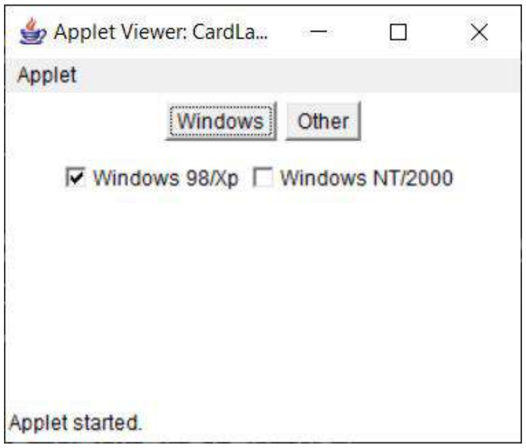
deck is a reference to the container.

PROGRAM:

```
//Demonstrate CardLayout
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<apllet code="CardLayoutDemo"width=300 height=100></applet>*/


public class CardLayoutDemo extends Applet implements ActionListener,MouseListener
{
        Checkbox Win98,WinNT,solaris,mac;
        Panel osCards;
        CardLayout cardLO;
```

```
    Button Win,Other;

  public void init()
  {

    Win = new Button("Windows");

    Other = new Button("Other");

    add(Win);add(Other);

    cardLO = new CardLayout();

    osCards = new Panel();

   osCards.setLayout(cardLO);//set panel layout to card layout

    Win98 = new Checkbox("Windows 98/Xp",null,true);

    WinNT = new Checkbox("Windows NT/2000");

    solaris = new Checkbox("solaris");

   mac = new Checkbox("MacOS");


  //add Windws check boxes to a panel

    Panel winPan = new Panel();
        winPan.add(Win98);
        winPan.add(WinNT);


 //add other OS check boxes to a panel
    Panel otherPan = new Panel();
         otherPan.add(solaris);
         otherPan.add(mac);


//add panels to card deck panel
    osCards.add(winPan,"Windows");
   osCards.add(otherPan,"Other");


//add cards to main applet panel
    add(osCards);


//register to receive action events
   Win.addActionListener(this);
```

```
        Other.addActionListener(this);


    //register mouse events
        addMouseListener(this);
     }


   //Cycle througth panels

   public void mousePressed(MouseEvent me)
    {
        cardLO.next(osCards);
    }

  //provide empty implementations for the other MouseListener methods.

   public void mouseClicked(MouseEvent me){}

   public void mouseEntered(MouseEvent me){}

   public void mouseExited(MouseEvent me){}

   public void mouseReleased(MouseEvent me){}

   public void actionPerformed(ActionEvent ae)
     {
        if(ae.getSource()==Win)
           {
               cardLO.show(osCards,"Windows");
           }
        else
           {
               cardLO.show(osCards,"Other");
           }
     }
}
```



SWINGS

Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT. Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables.

Swing components are LightWeight Components and where as AWT Components are Heavyweight Components.

JApplet :

Fundamental to Swing is the JApplet class, which extends Applet. JApplet supports various "panes," such as the content pane, the glass pane, and the root pane. 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.

Content Pane:

The Content pane is associated with JFrame. The content pane is the container that contains the components which resides in JFrame. Components should be added to a JFrame through content pane and not directly to the JFrame itself. **getContentPane()** method of JFrame returns an object of Container class as follows..

```
Container c=getContentPane();
```

Icons and Labels:

In Swing, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image. Two of its constructors are shown here..

ImageIcon(String *filename*)

ImageIcon(URL *url*)

The **ImageIcon** class implements the **Icon** interface that declares the following methods.

int getIconHeight()

int getIconWidth()

void paintIcon(Component *comp,* Graphics *g,* int *x*, int *y*)

Swing labels are instances of the **JLabel** class, which extends **JComponent**. It can display text and /or an icon. Some of its constructors….

JLabel(Icon *i*)

JLabel(String *s*)

JLabel(String *s*, Icon *i*, int *align*)→ Here s and i are the text and icon used for the label. The *align* argument is either **LEFT, RIGHT, CENTER, LEADING**, or **TRAILING**.

The icon and text associated with the label can be read and written by the following methods…

Icon getIcon()

String getText()

Void setIcon(Icon *i*)

Void setText(String *s*)

<u>Text Fields</u>

The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent.**

JTextField()

JTextField(int *cols*)

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

JTextField(String *s*)

JButtons:

Swing buttons are subclasses of the **AbstractButton** class, which extends **JComponent.** We can define different icons that are displayed for the component when it is disabled , pressed, or selected. Another icon can be used as a *over* icon, which is displayed when the mouse is positioned over that component.

Void setDisabledIcon(Icon *di*)

Void setPressedIcon(Icon *pi*)

Void setSelectedIcon(Icon *si*)

Void setRolloverIcon(Icon *ri*)

String getText()

Void setText(String *s*)

Void addActionListener(ActionListener *al*)

Void removeActionListener(ActionListener *al*)

JButton allows an icon, a string , or both to be associated with the push button.

JButton(Icon i)

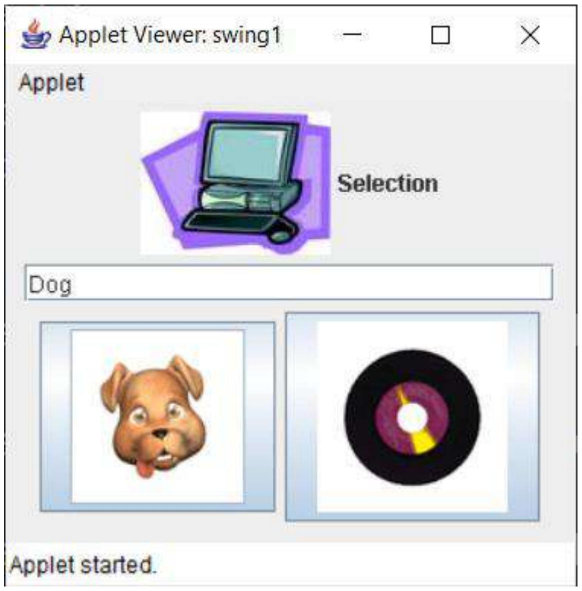
JButton(String s)

JButton(String s, Icon i)

Program 1:

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

/* <applet code="swing1" width=250 height=150>
</applet>

*/
```

```
public class swing1 extends JApplet implements ActionListener
{
      JTextField jtext=new JTextField("Hi",25);
       public void init()
       {
              Container contentPane=getContentPane();
              contentPane.setLayout(new FlowLayout());

                 ImageIcon img=new ImageIcon("1.jpg");
         JLabel jlab=new JLabel("Selection",img,JLabel.CENTER);

         contentPane.add(jlab);
                contentPane.add(jtext);

         ImageIcon img1=new ImageIcon("2.gif");
              JButton jb=new JButton(img1);
              jb.setActionCommand("Dog");
              jb.addActionListener(this);
              contentPane.add(jb);

         ImageIcon img2=new ImageIcon("3.gif");
              jb=new JButton(img2);
              jb.setActionCommand("CompactDisc");
              jb.addActionListener(this);
              contentPane.add(jb);
        }

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

}
```



<u>**Check Boxes:**</u>

The **JCheckBox** class , which provides the functionality of a check box, is a concrete implementation of **AbstractButton.**

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)

Void setSelected(Boolean state)→state of the check box can be changed

Event is handled by **itemStatechanged().**

getItem()→gets the **JCheckBox** object

getText()→gets the text for that check box

<u>**Radio Buttons**</u>

Radio buttons are supported bt the **JRadioButton** class,which is a concrete implementation of **AbstractButton.**

JRadioButton(Icon i)

JRadioButton(Icon I,Boolean state)

JRadioButton(String s)

JRadioButton(String s,Boolean state)

JRadioButton(String s,Icon i)

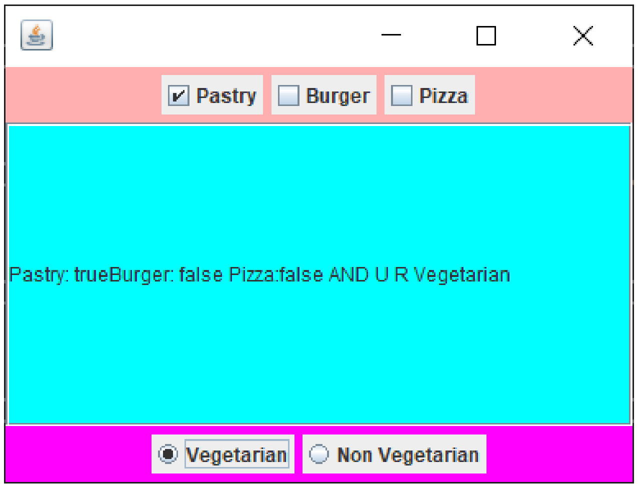
JRadioButton(String s, Icon I, Boolean state)

Radio buttons must be configured into a group. Only one of the buttons in that group can be selected at any time. The **ButtonGroup** class is instatiated to create a button group.

Program 2:

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

public class swing2 extends JFrame implements ItemListener
{
      JPanel north,south;
      JTextField jtext;
      JCheckBox pastry,burger,pizza;
      JRadioButton veg,nonveg;

   public swing2()
   {
      north=new JPanel();
      south=new JPanel();

      jtext=new JTextField("selection is yours......");
      north.setBackground(Color.pink);
      south.setBackground(Color.magenta);
      jtext.setBackground(Color.cyan);

      pastry=new JCheckBox("Pastry");
      burger=new JCheckBox("Burger");
      pizza=new JCheckBox("Pizza");
```

```
        veg=new JRadioButton("Vegetarian");
        nonveg=new JRadioButton("Non Vegetarian");

        ButtonGroup bg=new ButtonGroup();
        bg.add(veg);
        bg.add(nonveg);

        north.add(pastry);
        north.add(burger);
        north.add(pizza);

        south.add(veg);
        south.add(nonveg);

        pastry.addItemListener(this);
        burger.addItemListener(this);
        pizza.addItemListener(this);

        veg.addItemListener(this);
        nonveg.addItemListener(this);

        Container c=getContentPane();
        c.add(north,BorderLayout.NORTH);
        c.add(south,BorderLayout.SOUTH);
        c.add(jtext,BorderLayout.CENTER);

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

public void itemStateChanged(ItemEvent ie)

  {
        String str=" ";
        if(veg.isSelected()) str=" U R Vegetarian";
        if(nonveg.isSelected()) str=" U R Non-Vegetarian";
        jtext.setText("Pastry:    "+pastry.isSelected()+    "Burger:    "+burger.isSelected()+"
Pizza:"+pizza.isSelected()+" AND"+str);

    }

public static void main(String args[])
    {
        JFrame f=new swing2();
        f.setSize(400,300);
        f.setVisible(true);
    }
}
```



```
Pastry: trueBurger: false Pizza:false AND U R Vegetarian
```

Combo Boxes

Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent.** A combo box normally displays one entry. However , it can also display a drop-down list that allows a user to select a different entry. This is similar to AWT Choice Component.

JComboBox()

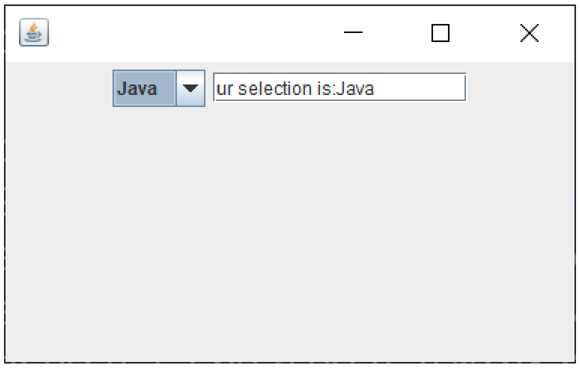
JComboBox(Vector v)→ *v* is a vector that initializes the combo box.

Void addItem(Object *obj*)→*obj* is the object to be added to the combo box.

Example 3:

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

public class swing3 extends JFrame
{
       JComboBox cbox;
       JTextField jtext;

       public swing3()
       {
         Container c=getContentPane();
         c.setLayout(new FlowLayout());
         String sub[]={"Java","SE","DBMS"};
         jtext=new JTextField("Select ur subject....",15);
         cbox=new JComboBox(sub);

         cbox.addItem("CO");
         cbox.addItem("OS");
         cbox.addItem("ES");

         help h=new help(jtext);
         cbox.addItemListener(h);
         c.add(cbox);
         c.add(jtext);

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

      public static void main(String ar[])
      {
         JFrame f=new swing3();
         f.setSize(300,200);
         f.setVisible(true);
      }
}
```

```
class help implements ItemListener
{
    JTextField jf;
    help(JTextField j)
    {
      jf=j;
    }
    public void itemStateChanged(ItemEvent ie)
    {
        String s=(String)ie.getItem();
        jf.setText("ur selection is:"+s);
    }
}
```



```
ur selection is:Java
```

<u>Tabbed Panes</u>

A *tabbed pane* is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent.**

Void addTab(String *str*, Component *comp*)→*str* is the title for the tab, and *comp* is the component that should be added to the tab.

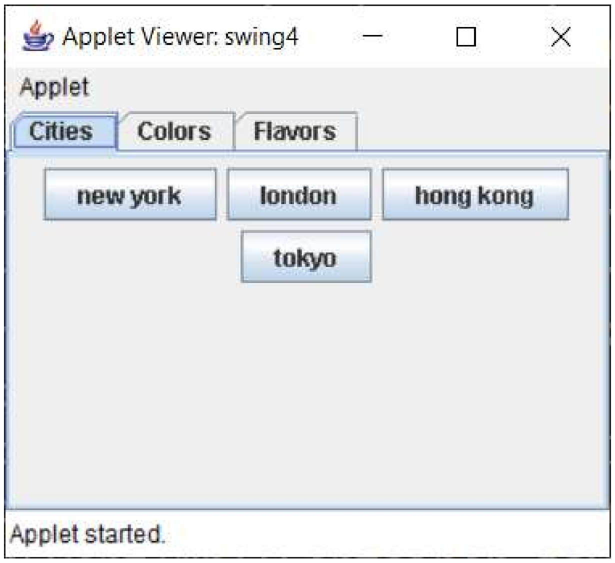
Procedure:

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

Program 4:

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

/* <applet code="swing4" width=400 height=100> </applet> */

public class swing4 extends JApplet
{
      public void init()
      {
           Container c=getContentPane();
                                JTabbedPane jtp=new JTabbedPane();
           jtp.addTab("Cities",new CitiesPanel());
           jtp.addTab("Colors",new ColorsPanel());
           jtp.addTab("Flavors",new FlavorsPanel());
           c.add(jtp);
      }
}

class CitiesPanel extends JPanel
{
      public CitiesPanel()
                          {
                              JButton b1=new JButton("new york");
                              add(b1);
                              JButton b2=new JButton("london");
                              add(b2);
                              JButton b3=new JButton("hong kong");
                              add(b3);
                              JButton b4=new JButton("tokyo");
                              add(b4);
                          }
}

c
```

```
        lass ColorsPanel extends JPanel
        {
                        public ColorsPanel()
                        {
                          JCheckBox cb1=new JCheckBox("Red");
                          add(cb1);
                          JCheckBox cb2=new JCheckBox("green");
                          add(cb2);
                          JCheckBox cb3=new JCheckBox("blue");
                          add(cb3);
                        }
        }
        class FlavorsPanel extends JPanel
        {
                        public FlavorsPanel()
                        {
                          JComboBox jcb=new JComboBox();
                          jcb.addItem("vanilla");
                          jcb.addItem("chocolate");
                          jcb.addItem("strawberry");
                          add(jcb);
                        }
    }
```



Applet started.

Scroll Panes

A *scroll pane* is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary. Scroll panes are implemented in Swing by the **JScrollPane** class, which extends **JComponent.**

JComponent(Component *comp*)

Jcomponent(int vsb, int hsb)

Jcomponent(Component comp, int vsb, int hsb)àcomp is the component to be added to the scroll pane, vsb and hsb are int constants.

HORIZONTAL_SCROLLBAR_ALWAYS

HORIZONTAL_SCROLLBAR_AS_NEEDED

VERTICAL_SCROLLBAR_ALWAYS

VERTICAL_SCROLLBAR_AS_NEEDED

Procedure:

1. Create a **JCompnent** object.

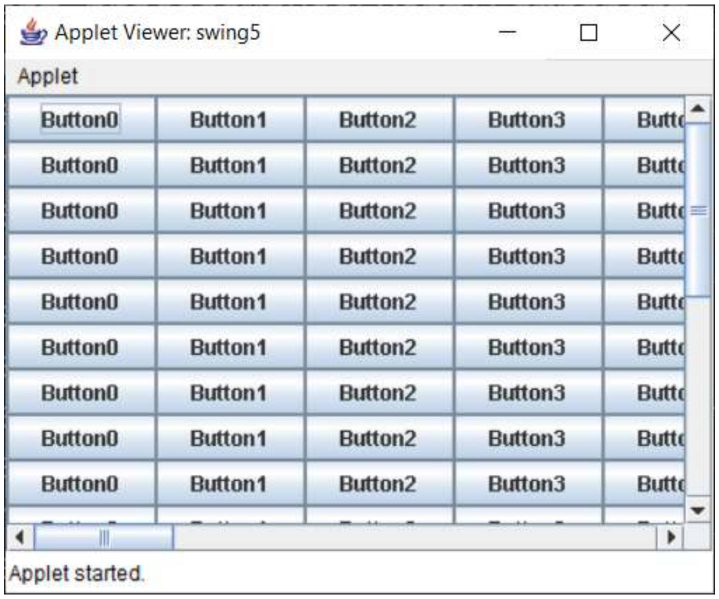
2. Create a **JScrollPane** object.

3. Add the scroll pane to the content of the applet.

Program 5:

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

/* <applet code="swing5" width=300 height=200> </applet> */

public class swing5 extends JApplet
{
  public void init()
  {
      Container c=getContentPane();
      c.setLayout(new BorderLayout());
```

```
     JPanel jp=new JPanel();
     jp.setLayout(new GridLayout(20,20));

     for(int i=0;i<20;i++)
     for(int j=0;j<20;j++)
        jp.add(new JButton("Button"+j));

     int v=ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
     int h=ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;

     JScrollPane js=new JScrollPane(jp,v,h);
     c.add(js,BorderLayout.CENTER);
  }
}
```



Trees

A tree is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **Jtree** class, which extends **Jcomponent.**

Jtree(Hashable ht)àCreates a tree in which each element of the hash table ht is a child node

Jtree(Object obj[])àEach element of the array obj is a child node

Jtree(TreeNode tn)àThe tree node tn is the root of the tree

Jtree(Vector v)àThe elements if the vector v as child nodes

The **getPathForLocation()** method is used to translate a mouse click on a specific point of the tree to a tree path.

TreePath getPathForLocation(int x, int y)

A **JTree** object generates events when a node is expanded or collapsed.

Void addTreeExpansionListner(TreeExpansionListener tel)

Void removeTreeExpansionListner(TreeExpansionListener tel)

The **TreePath** class encapsulates information about a path to a particular node in a tree. The **TreeNode** interface declares methods that obtain information about a tree node. The **MutableTreeNode** 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.

DefaultMutableTreeNode(Object *obj*)

Void add(MutableTreeNode *child*)à*child* is a mutable tree node that is to be added as a child to the current node.

1.Create a **Jtree** object.

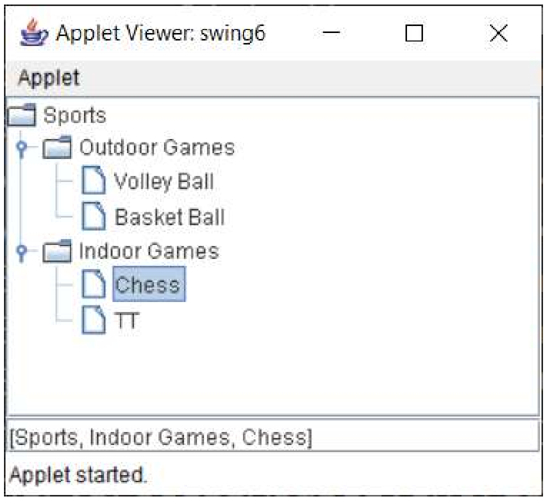
2.Create a **JScrollPane** object.

3.Add the tree to the scroll pane.

4.Add the scroll pane to the content pane of the applet.

Program 6:

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

public class swing6 extends JApplet
{
        JTree jt;
        JTextField jtext;
        public void init()
        {
                Container c = getContentPane();
                c.setLayout(new BorderLayout());

DefaultMutableTreeNode root = new DefaultMutableTreeNode("Sports");
DefaultMutableTreeNode out = new DefaultMutableTreeNode("Outdoor Games");
DefaultMutableTreeNode volley = new DefaultMutableTreeNode("Volley Ball");
        DefaultMutableTreeNode basket = new DefaultMutableTreeNode("Basket Ball");

        out.add(volley);        out.add(basket);        root.add(out);

        DefaultMutableTreeNode in = new DefaultMutableTreeNode("Indoor Games");
        DefaultMutableTreeNode chess = new DefaultMutableTreeNode("Chess");
        DefaultMutableTreeNode table = new DefaultMutableTreeNode("TT");

        in.add(chess);          in.add(table); root.add(in);

        jt = new JTree(root);
        int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;

        JScrollPane js = new JScrollPane(jt,v,h);
        c.add(js,BorderLayout.CENTER);
        jtext= new JTextField(20);
        c.add(jtext,BorderLayout.SOUTH);

        jt.addMouseListener(new MouseAdapter()
                        {
                                public void mouseClicked(MouseEvent me)
                                { display(me); }
                        });
        }
        public void display(MouseEvent e)
        {       TreePath tp =jt.getPathForLocation(e.getX(),e.getY());
                jtext.setText(tp.toString());
        }
}
```



[Sports, Indoor Games, Chess]

Applet started.

Tables

A *table* is a component that displays rows and columns if data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Tables are implemented by the **JTable** class, which extends **JComponent.**

JTable(Object *data*[][], Object *colHeads*[])à*data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

Procedure:

1.Create a **JTable** object.

2.Create a **JScrollPane** object.

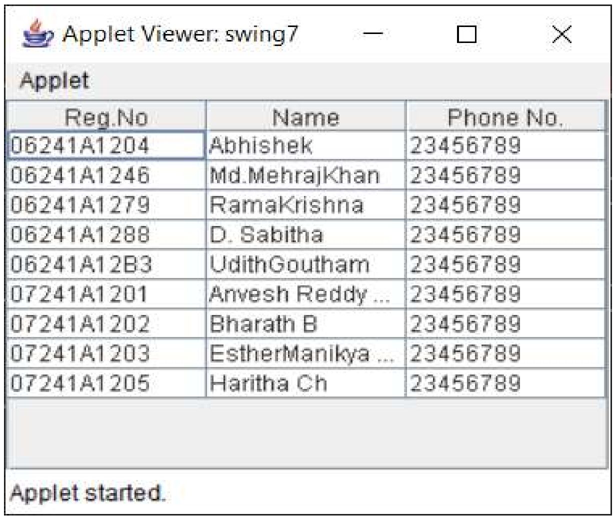
3.Add the table to the scroll pane.

4.Add the scroll pane to the content pane of the applet.

Program 7:

```
import java.awt.*;
import javax.swing.*;
/* <applet code="swing7" width=400 height=100></applet>*/
public class swing7 extends JApplet
{
  public void init()
  {
    Container contentPane=getContentPane();
    contentPane.setLayout(new BorderLayout());
    final String[] colHeads = {"Reg.No","Name","Phone No."};
```

```
        final Object[][] data={
                        {"06241A1204","Abhishek","23456789"},
                          {"06241A1246","Md.MehrajKhan","23456789"},
                          {"06241A1279","RamaKrishna","23456789"},
                          {"06241A1288","D. Sabitha","23456789"},
                          {"06241A12B3","UdithGoutham","23456789"},
                          {"07241A1201","Anvesh Reddy Ch","23456789"},
                          {"07241A1202","Bharath B","23456789"},
                          {"07241A1203","EstherManikya Latha","23456789"},
                  {"07241A1205","Haritha Ch","23456789"},
                         };
        JTable table=new JTable(data,colHeads);
        int v=ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h=ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane js=new JScrollPane(table,v,h);
        contentPane.add(js,BorderLayout.CENTER);
      }
    }
```



Reg.No	Name	Phone No.
06241A1204	Abhishek	23456789
06241A1246	Md.MehrajKhan	23456789
06241A1279	RamaKrishna	23456789
06241A1288	D. Sabitha	23456789
06241A12B3	UdithGoutham	23456789
07241A1201	Anvesh Reddy ...	23456789
07241A1202	Bharath B	23456789
07241A1203	EstherManikya ...	23456789
07241A1205	Haritha Ch	23456789

Applet started.