UNIT V: GUI Programming with Java – AWT class hierarchy, component, container, panel, window, frame, graphics.

AWT controls: Labels, button, text field, check box, check box groups, choices, lists, scrollbars, and graphics.

Layout Manager – Layout manager types: border, grid and flow.

Swing – Introduction, limitations of AWT, Swing vs AWT.

GUI PROGRAMMING WITH JAVA

ABSTRACT WINDOW TOOLKIT (AWT)

Java AWT (Abstract Window Toolkit) is ***an API to develop GUI*** *or window-based application in java*. Java AWT components are **platform-dependent** i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components uses the resources of system. The Abstract Window Toolkit(AWT) support for applets. The AWT contains numerous classes and methods that allow you to create and manage windows.

The **java.awt package** provides classes for AWT api such as **TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.**

AWT Classes

The AWT classes are contained in the **java.awt package**. It is one of Java's largest packages.

Class	**Description**
AWTEvent	Encapsulates AWT events.
AWTEventMulticaster	Dispatches events to multiple listeners.
BorderLayout	Border layouts use five components: North, South, East, West, and Center.
CardLayout	Card layouts emulate index cards. Only the one on top is showing.
Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
CheckboxMenuItem	Creates an on/off menu item.
Choice	Creates a pop-up list.
Color	Manages colors in a portable, platform-independent fashion.
Component	An abstract superclass for various AWT components.
Container	A subclass of Component that can hold other components.
Cursor	Encapsulates a bitmapped cursor.
Dialog	Creates a dialog window.
Dimension	Specifies the dimensions of an object. The width is stored in width , and the height is stored in height .
Event	Encapsulates events.
FlowLayout	The flow layout manager. Flow layout positions components left to right, top to bottom.
Frame	Creates a standard window that has a title bar, resize corners, and a menu bar.
Graphics	Encapsulates the graphics context.

Control Fundamentals

The AWT supports the following types of controls:

3. Labels
4. Push buttons
5. Check boxes
6. Choice lists
7. Lists
8. Scroll bars
9. Text editing

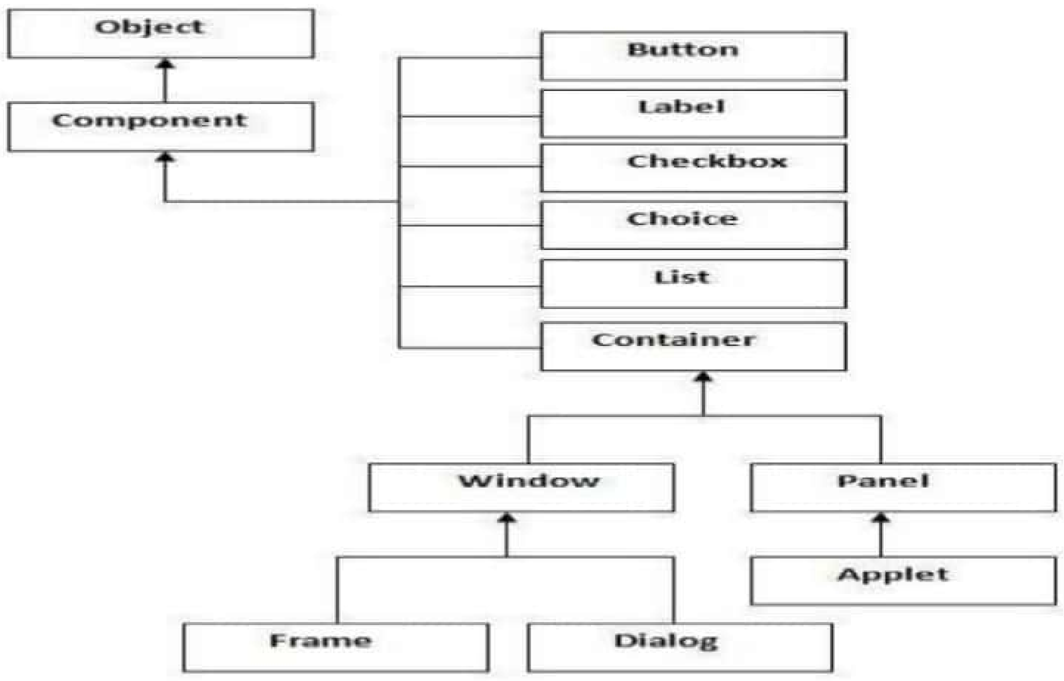
User interaction with the program is of two types:

CUI (Character User Interface): In CUI user interacts with the application by typing characters or commands. In CUI user should remember the commands. It is not user friendly.

2. **GUI (Graphical User Interface):** In GUI user interacts with the application through graphics. GUI is user friendly. GUI makes application attractive. It is possible to simulate real object in GUI programs. In java to write GUI programs we can use awt (Abstract Window Toolkit) package.

Java AWT Class Hierarchy

The hierarchy of Java AWT classes is given below.



Container

The Container is a component in AWT that can contain other components like buttons, textfields, labels etc. The classes that extend Container class are known as container such as **Frame, Dialog and Panel**.

Window

The window is the container that has no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size(width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Listeners and Listener Methods:

Listeners are available for components. A Listener is an interface that listens to an event from a component. Listeners are available in **java.awt.event package**. The methods in the listener interface are to be implemented, when using that listener.

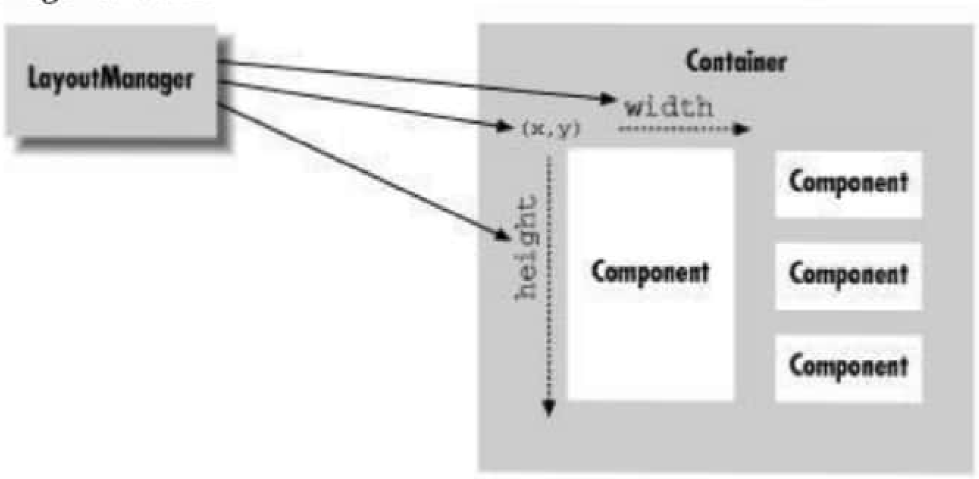
Component	Listener	Listener methods
Button	ActionListener	public void actionPerformed (ActionEvent e)
Checkbox	ItemListener	public void itemStateChanged (ItemEvent e)
CheckboxGroup	ItemListener	public void itemStateChanged (ItemEvent e)
TextField	ActionListener FocusListener	public void actionPerformed (ActionEvent e) public void focusGained (FocusEvent e) public void focusLost (FocusEvent e)
TextArea	ActionListener FocusListener	public void actionPerformed (ActionEvent e) public void focusGained (FocusEvent e) public void focusLost (FocusEvent e)
Choice	ActionListener ItemListener	public void actionPerformed (ActionEvent e) public void itemStateChanged (ItemEvent e)
List	ActionListener ItemListener	public void actionPerformed (ActionEvent e) public void itemStateChanged (ItemEvent e)
Scrollbar	AdjustmentListener MouseMotionListener	public void adjustmentValueChanged (AdjustmentEvent e) public void mouseDragged (MouseEvent e) public void mouseMoved (MouseEvent e)
Label	No listener is needed	

Layout Managers

A layout manager arranges the child components of a container. It positions and sets the size of components within the container's display area according to a particular layout scheme.

The layout manager's job is to fit the components into the available area, while maintaining the proper spatial relationships between the components. AWT comes with a few standard layout managers that will collectively handle most situations; you can make your own layout managers if you have special requirements.

LayoutManager at work



Every container has a **default layout manager**; therefore, when you make a new container, it comes with a LayoutManager object of the appropriate type. You can install a new layout manager at any time with the setLayout() method. Below, we set the layout manager of a container to a BorderLayout:

setLayout (new BorderLayout( ));

Every component determines three important pieces of information used by the layout manager in placing and sizing it: a minimum size, a maximum size, and a preferred size.

These are reported by the getMinimumSize(), getMaximumSize(), and getPreferredSize(), methods of Component, respectively.

When a layout manager is called to arrange its components, it is working within a fixed area. It usually begins by looking at its container's dimensions, and the preferred or minimum sizes of the child components.

Layout manager types

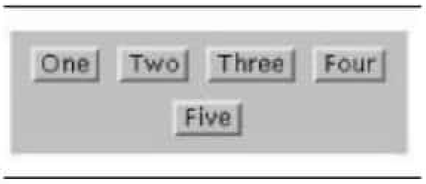
Flow Layout

`FlowLayout` is a simple layout manager that tries to arrange components with their preferred sizes, from left to right and top to bottom in the display. A `FlowLayout` can have a specified justification of `LEFT`, `CENTER`, or `RIGHT`, and a fixed horizontal and vertical padding.

By default, a flow layout uses **`CENTER`** justification, meaning that all components are centered within the area allotted to them. `FlowLayout` is the default for `Panel` components like `Applet`.

The following applet adds five buttons to the default `FlowLayout`.

```
import java.awt.*;
/*
<applet code="Flow" width="500" height="500">
</applet>
*/
public class Flow extends java.applet.Applet
{
   public void init()
  {
       //Default for Applet is FlowLayout
            add( new Button("One") );
      add( new Button("Two") );
      add( new Button("Three") );
      add( new Button("Four") );
      add( new Button("Five") );
   }
}
```



If the applet is small enough, some of the buttons spill over to a second or third row.

Grid Layout

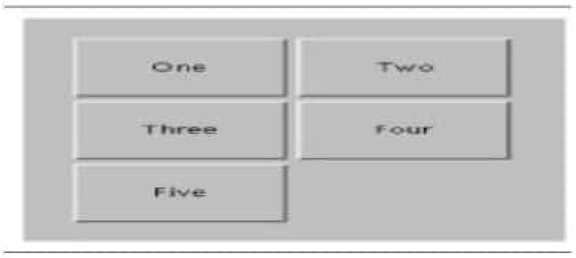
`GridLayout` arranges components into regularly spaced rows and columns. The components are arbitrarily resized to fit in the resulting areas; their minimum and preferred sizes are consequently ignored.

`GridLayout` is most useful for arranging very regular, identically sized objects and for allocating space for Panels to hold other layouts in each region of the container.

`GridLayout` takes the number of rows and columns in its constructor. If you subsequently give it too many objects to manage, it adds extra columns to make the objects fit. You can also set the number of rows or columns to zero, which means that you don't care how many elements the layout manager packs in that dimension.

For example, `GridLayout(2,0)` requests a layout with two rows and an unlimited number of columns; if you put ten components into this layout, you'll get two rows of five columns each. The following applet sets a `GridLayout` with three rows and two columns as its layout manager;

```
import java.awt.*;
/*
   <applet code="Grid" width="500"
    height="500"> </applet>
*/
public class Grid extends java.applet.Applet
  {
   public void init()
    {
      setLayout( new GridLayout( 3, 2 ));
```



```
        add( new Button("One") );
        add( new Button("Two") );
        add( new Button("Three") );
        add( new Button("Four") );
        add( new Button("Five") );
    }
 }
```

The five buttons are laid out, in order, from left to right, top to bottom, with one empty spot.

Border Layout

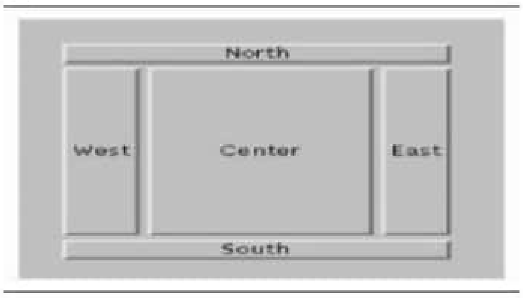
`BorderLayout` is a little more interesting. It tries to arrange objects in one of five geographical locations: "North," "South," "East," "West," and "Center," possibly with some padding between.

`BorderLayout` is the default layout for `Window` and `Frame` objects. Because each component is associated with a direction, `BorderLayout` can manage at most five components; it squashes or stretches those components to fit its constraints.

When we add a component to a border layout, we need to specify both the component and the position at which to add it. To do so, we use an overloaded version of the **`add()`** method that takes an additional argument as a constraint.

The following applet sets a `BorderLayout` layout and adds our five buttons again, named for their locations;

```
import java.awt.*;
/*
<applet code="Border" width="500" height="500">
</applet>
*/
public class Border extends java.applet.Applet
  {
   public void init()
   {
        setLayout( new java.awt.BorderLayout() );
        add( new Button("North"), "North" ); add(
        new Button("East"), "East" );
        add( new Button("South"), "South" );
        add( new Button("West"), "West" );
        add( new Button("Center"), "Center" );
    }
}
```



Compile: javac Border.java
Run   : appletviewer Border.java

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

// By extending Frame class (inheritance)

2. By creating the object of Frame class (association)

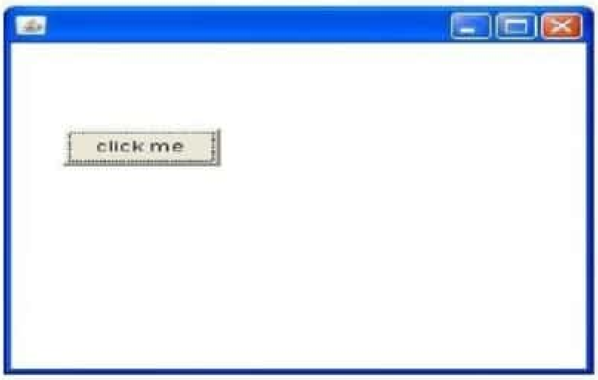
Simple example of AWT by inheritance

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300
height setLayout(null);//no layout manager
setVisible(true);//now frame will be visible }

public static void main(String args[]){
First f=new First();
}
}
```

Simple example of AWT by association

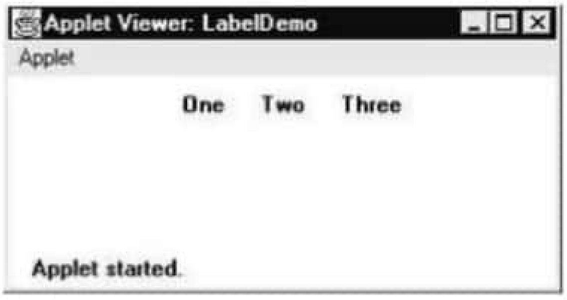
```
import java.awt.*;
class First2{
First2(){
Frame f=new Frame();
 Button b=new Button("click me");
b.setBounds(30,50,80,30);
f.add(b);
f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String args[]){
First2 f=new First2();
}
}
```



AWT controls

Labels:
The easiest control to use is a label. 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.

```
 // 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);
}
}
```



Buttons:
The most widely used control is the push button. 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 class is useful to create push buttons. A push button triggers a series of events.

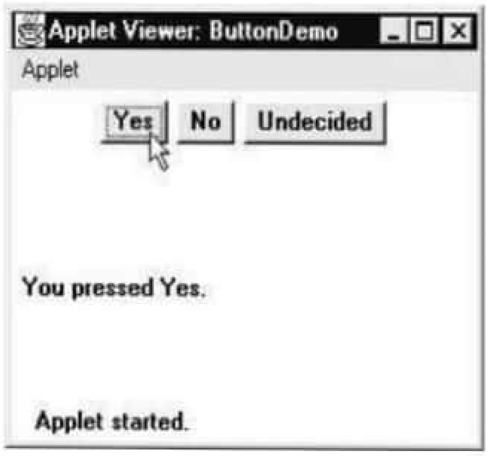
To create push button: Button b1 =new Button("label");
To get the label of the button: String l = b1.getLabel();
To set the label of the button: b1.setLabel("label");
To get the label of the button clicked: String str = ae.getActionCommand();

-

```
\{ Demonstrate Buttons
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/* <applet code="ButtonDemo" width=250 height=150>
</applet> */

public class ButtonDemo extends Applet implements ActionListener
 {
String msg = "";
Button yes, no, maybe;
public void init()
{
yes = new Button("Yes");
no = new Button("No");
maybe = new Button("Undecided");
add(yes);
add(no);
add(maybe);
yes.addActionListener(this);
no.addActionListener(this);
maybe.addActionListener(this);
}

public void actionPerformed(ActionEvent ae)
{
String str = ae.getActionCommand();
  if(str.equals("Yes"))
   {
    msg = "You pressed Yes.";
   }
else if(str.equals("No"))
  {
   msg = "You pressed No.";
  }
else
 {
   msg = "You pressed Undecided.";
 }
 repaint();
}
public void paint(Graphics g)
{
  g.drawString(msg, 6, 100);
}
  }
```



You pressed Yes.

Applet started.

Check Boxes:

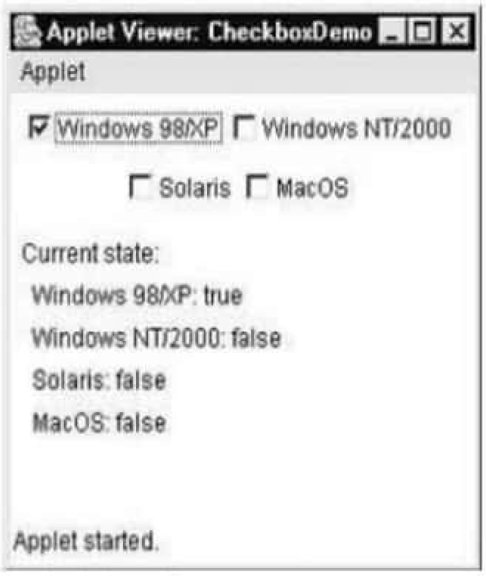
A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group.

```
\{    Demonstrate check boxes.
 import java.awt.*;
 import java.awt.event.*;
 import java.applet.*;
```

```
/*
<applet code="CheckboxDemo" width=250 height=200>
</applet>
*/
 public class CheckboxDemo extends Applet implements ItemListener
{
String msg = "";
checkbox Win98, winNT, solaris, mac;

public void init()
 {
  win98 = new Checkbox("Windows 98/XP", null, true);
  winNT = new Checkbox("Windows NT/2000");
  solaris = new Checkbox("Solaris"); mac = new
  Checkbox("MacOS");
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 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);
  msg = " MacOS: " + mac.getState();
   g.drawString(msg, 6, 160);
 }
}
```



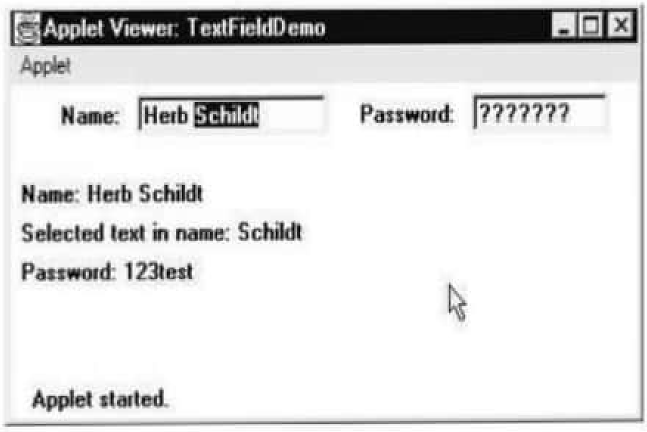
```
Current state:
 Windows 98/XP: true
 Windows NT/2000: false
 Solaris: false
 MacOS: false
```

Applet started.

TextField:
The TextField class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections.

```
// Demonstrate text field.
   import java.awt.*;
   import
   java.awt.event.*;
   import java.applet.*;
/*
<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: Herb Schildt
Selected text in name: Schildt
Password: 123test
```

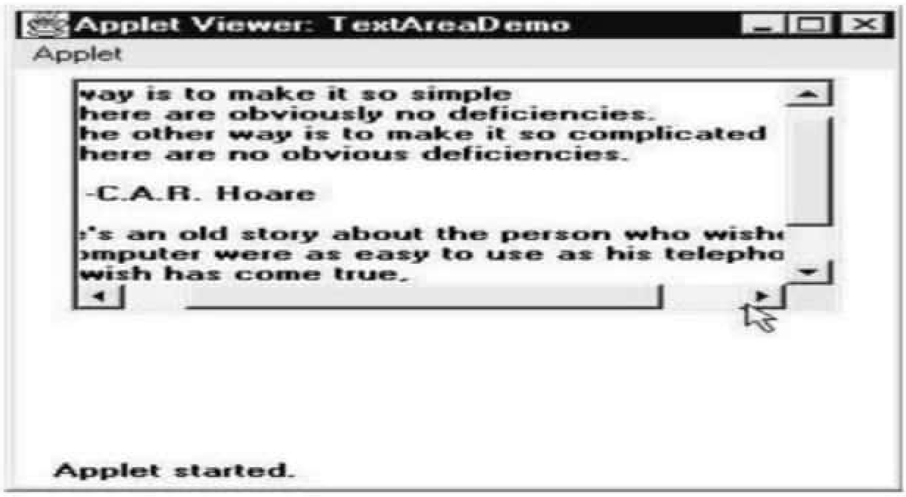
Applet started.

TextArea:

Sometimes a single line of text input is not enough for a given task. To handle these situations, the AWT includes a simple multiline editor called TextArea .

```
  \{ Demonstrate
     TextArea. import
     java.awt.*; import
     java.applet.*;
/*
<applet code="TextAreaDemo" width=300
height=250> </applet>
*/
public class TextAreaDemo extends Applet
{
 public void init()
 {
String val = "There are two ways of constructing " + "a software design.\n" + "One way is to make it so simple\n" + "that there are obviously no deficiencies.\n" + "And the other way is to make it so complicated\n" + "that there are no obvious deficiencies.\n\n" + " -C.A.R. Hoare\n\n" + "There's an old story about the person who wished\n" + "his computer were as easy to use as his telephone.\n" + "That wish has come true,\n" + "since I no longer know how to use my telephone.\n\n" + " -Bjarne Stroustrup, AT&T, (inventor of C++)";

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



```
way is to make it so simple
here are obviously no deficiencies.
he other way is to make it so complicated
here are no obvious deficiencies.

-C.A.R. Hoare

:'s an old story about the person who wish
omputer were as easy to use as his telepho
wish has come true.
```

Applet started.

CheckboxGroup

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 radio buttons. A Radio button represents a round shaped button such that only one can be selected from a panel. Radio button can be created using CheckboxGroup class and Checkbox classes.

· To create a radio button: CheckboxGroup cbg = new CheckboxGroup ();
Checkbox cb = new Checkbox ("label", cbg, true);

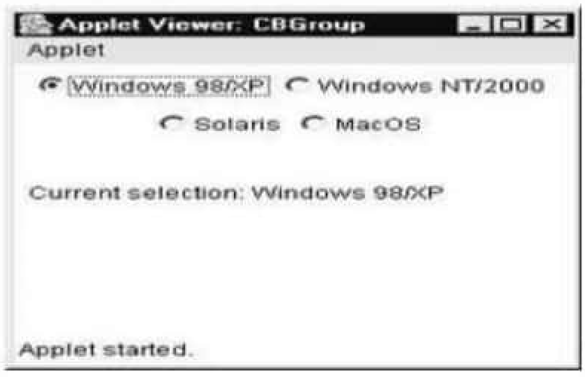
· To know the selected checkbox: Checkbox cb = cbg.getSelectedCheckbox ();

·To know the selected checkbox label: String label = cbg.getSelectedCheckbox().getLabel ();

```
\{ 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("MacOS", 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: Windows 98/XP

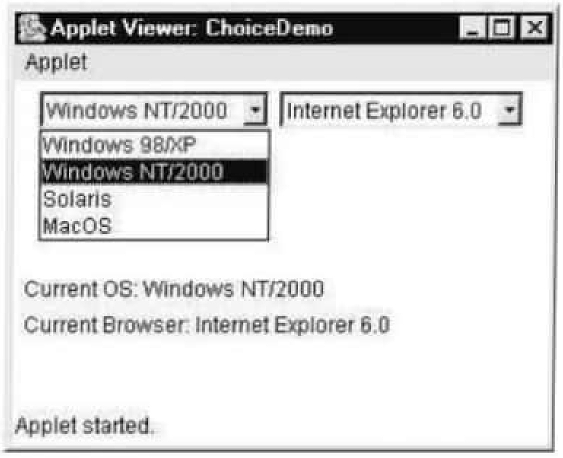
Choice Controls

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. Choice menu is a popdown list of items. Only one item can be selected.

- To create a choice menu: Choice ch = new Choice();
- To add items to the choice menu: ch.add ("text");
- To know the name of the item selected from the choice menu: String s = ch.getSelectedItem ();
- To know the index of the currently selected item: int i = ch.getSelectedIndex();

This method returns -1, if nothing is selected.

```
  //Demonstrate Choice
      lists. 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();
  //add items to os list
os.add("Windows 98/XP");
os.add("Windows NT/2000");
os.add("Solaris");
os.add("MacOS");
  \{ add items to browser list
      browser.add("Netscape 3.x");
      browser.add("Netscape 4.x");
      browser.add("Netscape 5.x");
      browser.add("Netscape 6.x");
      browser.add("Internet Explorer
      4.0"); browser.add("Internet
      Explorer 5.0");
      browser.add("Internet Explorer
      6.0"); browser.add("Lynx 2.4");
      browser.select("Netscape 4.x");
  \{ add choice lists to
      window add(os);
add(browser);
  \{ register to receive item
      events
      os.addItemListener(this);
      browser.addItemListener(
      this);
```



```
Current OS: Windows NT/2000
Current Browser: Internet Explorer 6.0
```

Applet started.

```
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);
}
}
```

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, a List object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. List provides these constructors: List( )

List(int numRows)

List(int numRows , boolean multipleSelect)

A List box is similar to a choice box, it allows the user to select multiple items.

· To create a list box: List lst = new List();

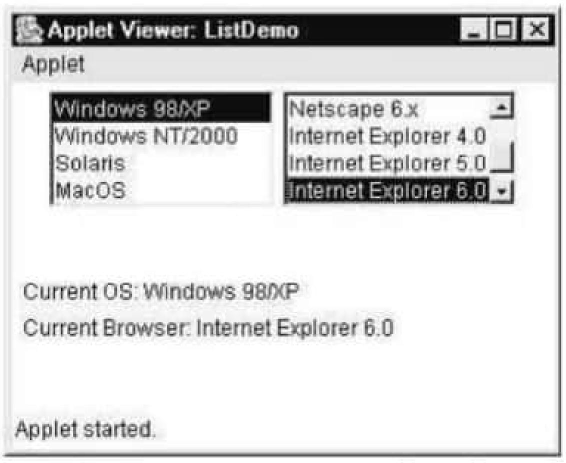
(or)

List lst = new List (3, true);

This list box initially displays 3 items. The next parameter true represents that the user can select more than one item from the available items. If it is false, then the user can select only one item.

= To add items to the list box: lst.add("text");
= To get the selected items: String x[] = lst.getSelectedItems();
= To get the selected indexes: int x[] = lst.getSelectedIndexes ();

```
//      Demonstrate Lists.
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(4, true);
  browser = new List(4, false);
\} add items to os list
    os.add("Windows 98/XP");
    os.add("Windows
    NT/2000");
    os.add("Solaris");
    os.add("MacOS");
\} add items to browser list
    browser.add("Netscape 3.x");
    browser.add("Netscape 4.x");
    browser.add("Netscape 5.x");
    browser.add("Netscape 6.x");
    browser.add("Internet Explorer
    4.0"); browser.add("Internet
    Explorer 5.0");
    browser.add("Internet Explorer
    6.0"); browser.add("Lynx 2.4");
    browser.select(1);
\} add 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 98/XP
Current Browser: Internet Explorer 6.0
Applet started.
```

Scroll Bars

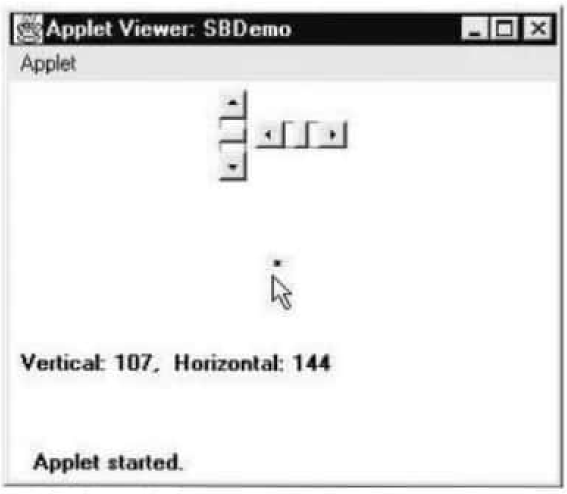
Scroll bars are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically. Scrollbar class is useful to create scrollbars that can be attached to a frame or text area. Scrollbars can be arranged vertically or horizontally.

\{ To create a scrollbar : Scrollbar sb = new Scrollbar (alignment, start, step, min, max);
alignment: Scrollbar.VERTICAL, Scrollbar.HORIZONTAL
start: starting value (e.g. 0)
step: step value (e.g. 30) // represents scrollbar length
min: minimum value (e.g. 0)
max: maximum value (e.g. 300)

· To know the location of a scrollbar:	int n = sb.getValue ();
· To update scrollbar position to a new position:	sb.setValue (int position);
· To get the maximum value of the scrollbar:	int x = sb.getMaximum ();
· To get the minimum value of the scrollbar:	int x = sb.getMinimum ();
· To get the alignment of the scrollbar:	int x = getOrientation ();

This method return 0 if the scrollbar is aligned HORIZONTAL, 1 if aligned VERTICAL.

```
// 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);
// register to receive adjustment events
 vertSB.addAdjustmentListener(this);
 horzSB.addAdjustmentListener(this);
 addMouseMotionListener(this);
}
public void adjustmentValueChanged(AdjustmentEvent ae) {
repaint();
}
// Update scroll bars to reflect mouse dragging.
public void mouseDragged(MouseEvent me) {
int x = me.getX();
int y = me.getY();
vertSB.setValue(y);
horzSB.setValue(x);
repaint();
}
//    Necessary for MouseMotionListener
public void mouseMoved(MouseEvent me) {
}
//    Display current value of scroll bars.
public void paint(Graphics g) {
msg = "Vertical: " + vertSB.getValue();
msg += ", Horizontal: " + horzSB.getValue();
g.drawString(msg, 6, 160);
\} show current mouse drag position
g.drawString("*", horzSB.getValue(),
vertSB.getValue());
}
}
```



```
Vertical: 107, Horizontal: 144
```

Applet started.

Graphics

The AWT supports a rich assortment of graphics methods. All graphics are drawn relative to a window.

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.

Drawing Lines

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

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

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

The following applet draws several lines:

```
//    Draw lines import
java.awt.*; import
java.applet.*; /*

<applet code="Lines" width=300
height=200> </applet>
*/
public class Lines extends Applet
{ public void paint(Graphics g) {
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);
} }
```

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( ), both shown here: 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)

```
// Draw rectangles
import java.awt.*;
import java.applet.*;
/*
<applet code="Rectangles" width=300
height=200> </applet>
*/
public class Rectangles extends Applet {
public void paint(Graphics g) {
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);
}
}
```

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)
```

```
// Draw
    Ellipses
    import
    java.awt.*
    ; import
    java.apple
    t.*; /*
<applet code="Ellipses" width=300
height=200> </applet>
*/
public class Ellipses extends Applet
{ public void paint(Graphics g) {
g.drawOval(10, 10, 50, 50);
g.fillOval(100, 10, 75, 50);
g.drawOval(190, 10, 90, 30);
g.fillOval(70, 90, 140, 100);
} }
```

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)
```

The arc is bounded by the rectangle whose upper-left corner is specified by top,left and whose width and height are specified by width and height. The arc is drawn from startAngle through the angular distance specified by sweepAngle. Angles are specified in degrees.

Zero degrees is on the horizontal, at the three o' clock position. The arc is drawn counterclockwise if sweepAngle is positive, and clockwise if sweepAngle is negative. Therefore, to draw an arc from twelve o' clock to six o' clock, the start angle would be 90 and the sweep angle 180.

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)
```

The polygon' s endpoints are specified by the coordinate pairs contained within the x and y arrays. The number of points defined by x and y is specified by numPoints. There are alternative forms of these methods in which the polygon is specified by a Polygon object.

The following applet draws several arcs:

```
   //Draw Arcs
      import
      java.awt.*;
      import
      java.applet.*;
      /*

<applet code="Arcs"
width=300 height=200>
</applet>
*/
public class Arcs extends Applet {
public void paint(Graphics g) {
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);
}
}
```

The following applet draws an hourglass shape:

```
  // Draw Polygon
      import
      java.awt.*;
      import
      java.applet.*;
      /*
<applet code="HourGlass"
width=230 height=210>
</applet>
*/
public class HourGlass extends Applet {
public void paint(Graphics g) {
int xpoints[] = {30, 200, 30, 200, 30};
int ypoints[] = {30, 30, 200, 200, 30};
int num = 5; g.drawPolygon(xpoints,
ypoints, num);
}
```

SWINGS

- Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT. **Swing** is a GUI widget toolkit for **Java**. It is part of Oracle's **Java** Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

- In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables. Even familiar components such as buttons have more capabilities in Swing. For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.

- Unlike AWT components, Swing components are not implemented by platform specific code. Instead, they are written entirely in Java and, therefore, are platform-independent. The term *lightweight* is used to describe such elements.

✓ The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Differences between AWT and Swing

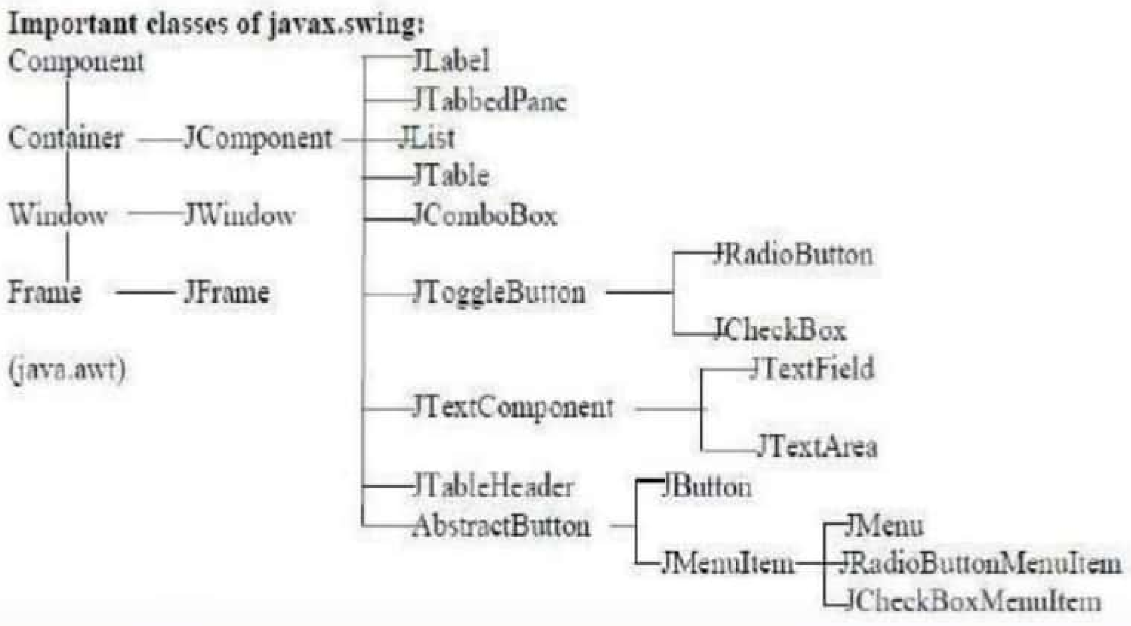
AWT	Swing
AWT components are called Heavyweight component.	Swings are called light weight component because swing components sits on the top of AWT components and do the work.
AWT components are platform dependent.	Swing components are made in purely java and they are platform independent.
AWT components require java.awt package.	Swing components require javax.swing package.
AWT is a thin layer of code on top of the OS.	Swing is much larger. Swing also has very much richer functionality.
AWT stands for Abstract windows toolkit.	Swing is also called as JFC's (Java Foundation classes).
This feature is not supported in AWT.	We can have different look and feel in Swing.
Using AWT, you have to implement a lot of things yourself.	Swing has them built in.

This feature is not available in AWT.	Swing has many advanced features like JTabel, Jtabbed pane which is not available in AWT. Also.Swing components are called "lightweight" because they do not require a native OS object to implement their functionality. JDialog and JFrame are heavyweight, because they do have a peer. So components like JButton, JTextArea, etc., are lightweight because they do not have an OS peer.

The Swing component classes are:

Class	**Description**
AbstractButton	Abstract superclass for Swing buttons.
ButtonGroup	Encapsulates a mutually exclusive set of buttons.
ImageIcon	Encapsulates an icon.
JApplet	The Swing version of Applet.
JButton	The Swing push button class.
JCheckBox	The Swing check box class.
JComboBox	Encapsulates a combo box (combination of a drop-down list & text field).
JLabel	The Swing version of a label.
JRadioButton	The Swing version of a radio button.
JScrollPane	Encapsulates a scrollable window.
JTabbedPane	Encapsulates a tabbed window.
JTable	Encapsulates a table-based control.
JTextField	The Swing version of a text field.
JTree	Encapsulates a tree-based control.

Hierarchy for Swing components:



JApplet
Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that use Swing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is not foundin **Applet**.

The content pane can be obtained via the method shown here:
Container getContentPane( )

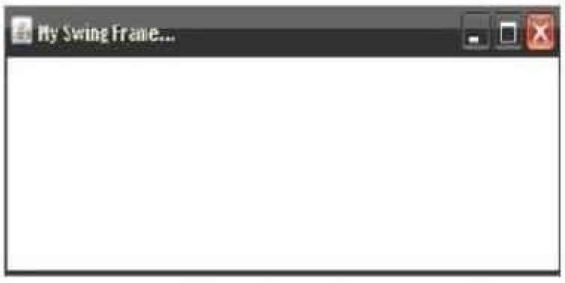
The **add( )** method of **Container** can be used to add a component to a content pane. Its form is shown here:
void add(*comp*)

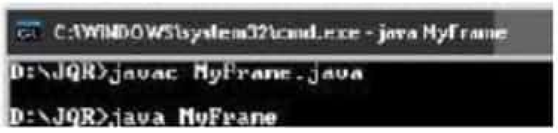
Here, *comp* is the component to be added to the content pane.

JFrame
Create an object to JFrame: JFrame ob = new JFrame ("title"); (or)
Create a class as subclass to JFrame class: MyFrame extends JFrame
Create an object to that class : MyFrame ob = new MyFrame ();

Example: Write a program to create a frame by creating an object to JFrame class. //A swing Frame

```
import javax.swing.*;
class MyFrame
{
public static void main (String agrs[])
{ JFrame jf = new JFrame ("My Swing Frame...");
jf.setSize (400,200);
jf.setVisible (true);
jf.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
}
}
```





```
C:\WINDOWS\system32\cmd.exe - java MyFrame
D:\JQR>javac MyFrame.java
D:\JQR>java MyFrame
```

Note: To close the frame, we can take the help of getDefaultCloseOperation () method of JFrame class, as shown here: **getDefaultCloseOperation (constant);**

where the constant can be any one of the following:

- JFrame.EXIT_ON_CLOSE: This closes the application upon clicking on close button.
- JFrame.DISPOSE_ON_CLOSE: This disposes the present frame which is visible on the screen. The JVM may also terminate.
- JFrame.DO_NOTHING_ON_CLOSE: This will not perform any operation upon clicking on close button.
- JFrame.HIDE_ON_CLOSE: This hides the frame upon clicking on close button.

Window Panes: In swings the components are attached to the window panes only. A window pane represents a free area of a window where some text or components can be displayed. For example, we can create a frame using JFrame class in javax.swing which contains a free area inside it, this free area is called 'window pane'. Four types of window panes are available in javax.swing package.

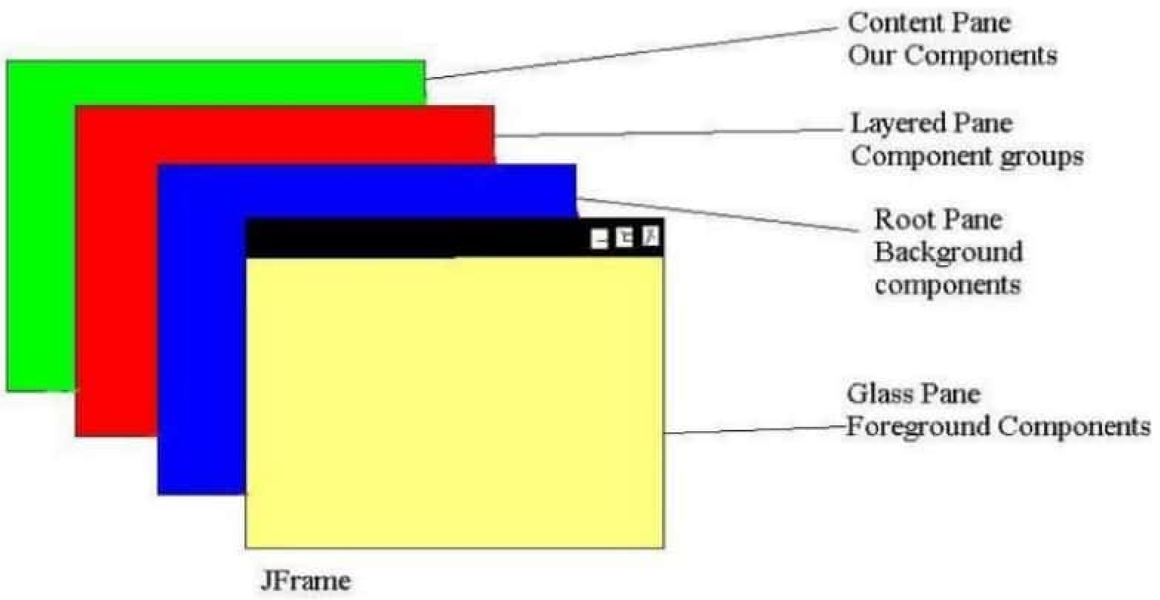
Glass Pane: This is the first pane and is very close to the monitors screen. Any components to be displayed in the foreground are attached to this glass pane. To reach this glass pane, we use getGlassPane () method of JFrame class.

Root Pane: This pane is below the glass pane. Any components to be displayed in the background are displayed in this pane. Root pane and glass pane are used in animations also.
For example, suppose we want to display a flying aeroplane in the sky. The aeroplane can be displayed as a .gif or .jpg file in the glass pane where as the blue sky can be displayed in the root pane in the background. To reach this root pane, we use getRootPane () method of JFrame class.

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

Content Pane: This is the bottom most pane of all. Individual components are attached to this pane. To reach this pane, we can call getContentPane () method of JFrame class.

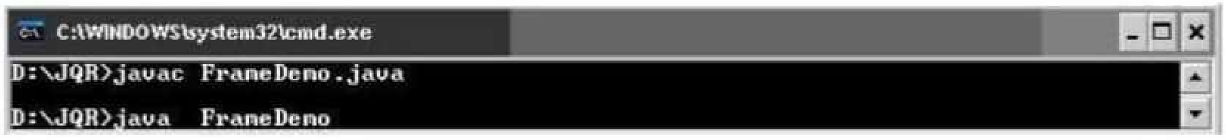
Displaying Text in the Frame:
paintComponent (Graphics g) method of JPanel class is used to paint the portion of a component in swing. We should override this method in our class. In the following example, we are writing our class MyPanel as a subclass to JPanel and override the painComponent () method.



Write a program to display text in the frame

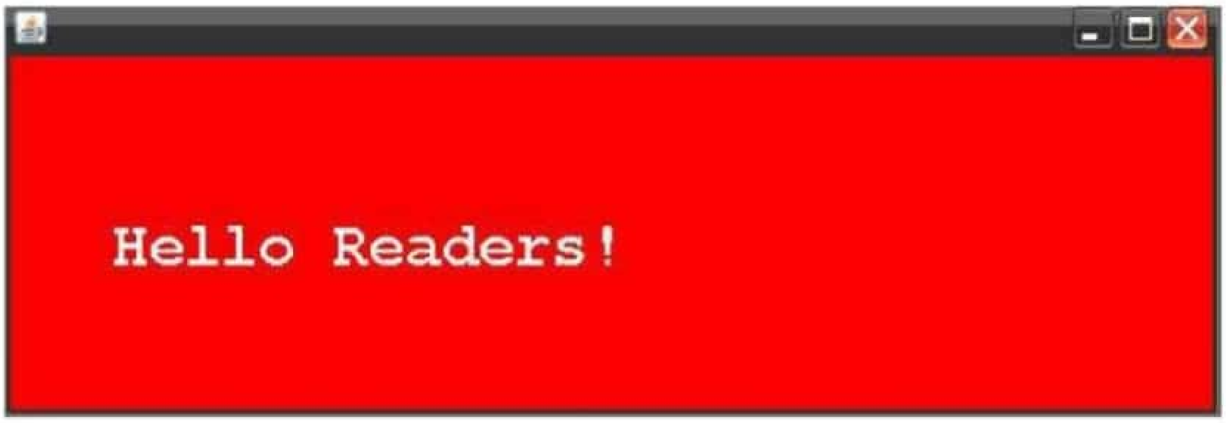
```
import javax.swing.*;
import java.awt.*;
class MyPanel extends JPanel
{ public void paintComponent (Graphics g)
{ super.paintComponent (g); //call JPanel' s method
setBackground (Color.red);
g.setColor (Color.white);
g.setFont (new Font("Courier New",Font.BOLD,30));
```

```
                    rawString ("Hello Readers!", 50, 100);
}
}
class FrameDemo extends JFrame
{ FrameDemo ()
{
Container c = getContentPane ();
MyPanel mp = new MyPanel ();
c.add (mp);
setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
}
public static void main(String args[])
{ FrameDemo ob = new FrameDemo ();
ob.setSize (600, 200);
ob.setVisible (true);
}
}
```



```
D:\JQR>javac FrameDemo.java

D:\JQR>java  FrameDemo
```



```
Hello Readers!
```

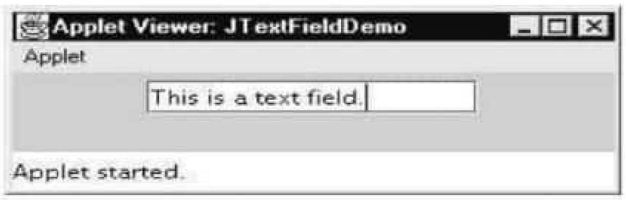
TEXT FIELDS

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

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

```
import java.awt.*;
import javax.swing.*;
/*
<applet code="JTextFieldDemo" width=300 height=50>
```

```
</applet>
*/
public class JTextFieldDemo extends JApplet {
 JTextField jtf;
public void init()
{
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
   // Add text field to content
      pane jtf = new
      JTextField(15);
      contentPane.add(jtf);
}
}
```



BUTTONS

Swing buttons provide features that are not found in the Button class defined by the AWT. For example, you can associate an icon with a Swing button. Swing buttons are subclasses of the AbstractButton class, which extends JComponent. AbstractButton contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons.

The JButton Class

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

- To create a JButton with text: JButton b = new JButton (“OK”);
- To create a JButton with image: JButton b = new JButton (ImageIcon ii);
- To create a JButton with text & image: JButton b = new JButton (“OK” , ImageIcon ii);

It is possible to create components in swing with images on it. The image is specified by ImageIcon class object.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JButtonDemo" width=250 height=300>
</applet>
*/
public class JButtonDemo extends JApplet implements ActionListener {
JTextField jtf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Add buttons to content pane
ImageIcon france = new ImageIcon("france.gif");
JButton jb = new JButton(france);
jb.setActionCommand("France");
jb.addActionListener(this); contentPane.add(jb);

ImageIcon germany = new ImageIcon("germany.gif");
jb = new JButton(germany);
jb.setActionCommand("Germany");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon italy = new ImageIcon("italy.gif");
jb = new JButton(italy);
jb.setActionCommand("Italy");
jb.addActionListener(this);
contentPane.add(jb);
ImageIcon japan = new ImageIcon("japan.gif");
jb = new JButton(japan);
jb.setActionCommand("Japan");
jb.addActionListener(this);
contentPane.add(jb);
    // Add text field to
        content pane jtf = new
        JTextField(15);
        contentPane.add(jtf);
}
public void actionPerformed(ActionEvent ae)
{ jtf.setText(ae.getActionCommand());
}
}
```

CHECK BOXES

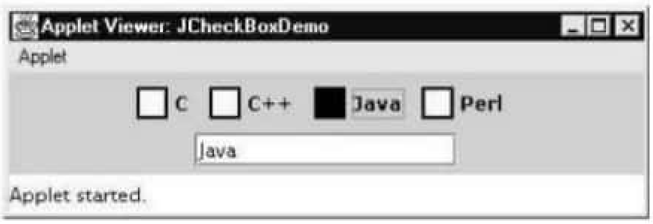
The JCheckBox class, which provides the functionality of a check box, is a concrete implementation of AbstractButton. Its immediate superclass is JToggleButton, which provides support for two-state buttons. Some of its constructors are shown here:

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

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JCheckBoxDemo" width=400 height=50>
</applet>
*/
```

```
public class JCheckBoxDemo extends JApplet implements ItemListener {
JTextField jtf;
public void init()
 {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
// Create icons
ImageIcon normal = new ImageIcon("normal.gif");
ImageIcon rollover = new ImageIcon("rollover.gif");
ImageIcon selected = new ImageIcon("selected.gif");
// Add check boxes to the content pane
JCheckBox cb = new JCheckBox("C", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this); contentPane.add(cb);

cb = new JCheckBox("C++",
normal); cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Java", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
cb = new JCheckBox("Perl", normal);
cb.setRolloverIcon(rollover);
cb.setSelectedIcon(selected);
cb.addItemListener(this);
contentPane.add(cb);
// Add text field to the content pane
jtf = new JTextField(15);
contentPane.add(jtf);
}
public void itemStateChanged(ItemEvent ie) {
JCheckBox cb = (JCheckBox)ie.getItem();
jtf.setText(cb.getText());
}
}
```

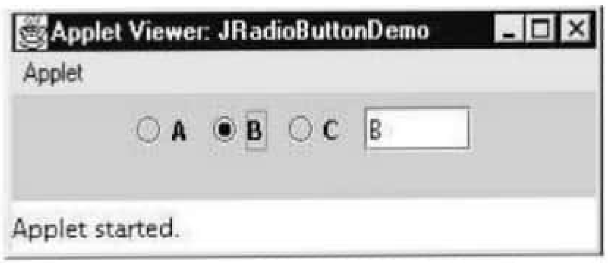


RADIO BUTTONS

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

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

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JRadioButtonDemo" width=300 height=50>
</applet>
*/
public class JRadioButtonDemo extends JApplet implements ActionListener {
JTextField tf;
public void init() {
// Get content pane
Container contentPane = getContentPane();
contentPane.setLayout(new FlowLayout());
1. Add radio buttons to content pane
JRadioButton b1 = new JRadioButton("A");
b1.addActionListener(this);
contentPane.add(b1);
JRadioButton b2 = new JRadioButton("B");
b2.addActionListener(this);
contentPane.add(b2);
JRadioButton b3 = new JRadioButton("C");
b3.addActionListener(this);
contentPane.add(b3);
2. Define a button group
ButtonGroup bg = new ButtonGroup();
bg.add(b1);
bg.add(b2);
bg.add(b3);
// Create a text field and add it
// to the content pane
tf = new JTextField(5);
contentPane.add(tf);
}
public void actionPerformed(ActionEvent ae) {
tf.setText(ae.getActionCommand());
}
}
```



Limitations of AWT:

The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface. One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents or peers. This means that the look and feel of a component is defined by the platform, not by java. Because the AWT components use native code resources, they are referred to as heavy weight.

The use of native peers led to several problems.

First, because of variations between operating systems, a component might look, or even act, differently on different platforms. This variability threatened java's philosophy: write once, run anywhere.

Second, the look and feel of each component was fixed and could not be changed. Third, the use of heavyweight components caused some frustrating restrictions. Due to these limitations Swing came and was integrated to java. Swing is built on the AWT. Two key Swing features are: Swing components are light weight, Swing supports a pluggable look and feel.