Unit-6(Chapter-1)

AWT(Abstract Window Toolkit)

AWT Class Hierarchy:

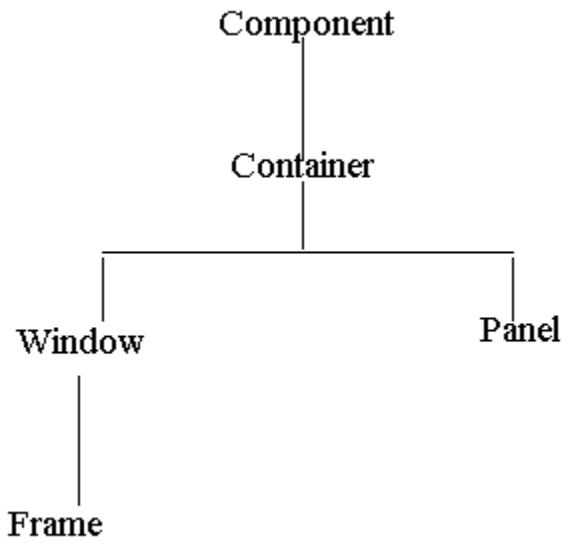


Fig: AWT Class Hierarchy

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**.
- It defines over a hundred public methods that are responsible for managing events, such as mouse and keyboard input, positioning and sizing the window, and repainting.
- A **Component** object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container

- The **Container** class is a subclass of **Component**. It has additional methods that allow other

- **Component** objects to be nested within it. Other **Container** objects can be stored inside of a
- **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system.
- A container is responsible for laying out (that is, positioning) anycomponents that it contains.

Panel

- The **Panel** class is a concrete subclass of **Container**. It doesn't add any new methods; it simply implements **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component.
- **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object.
- **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser.
- When you 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, you can position and resize them manually using the **setLocation( )**, **setSize( )**, **setPreferredSize( )**, or **setBounds( )** methods defined by **Component**.

Window

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

Frame

- **Frame** encapsulates what is commonly thought of as a "window." It is a subclass of**Window** and has a title bar, menu bar, borders, and resizing corners.

User Interface Components:

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( ) throws HeadlessException
- Label(String *str*) throws HeadlessException
- Label(String *str*, int *how*) throws HeadlessException

- The first one creates a blank label.
- The second one creates a label that contains the string specified by *str*. This string is left-justified.
- The third one 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
  Label.CENTER.

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** defines these two constructors:
- Button( ) throws HeadlessException
- Button(String *str*) throws HeadlessException
- The first one creates an empty button.
- The second one creates a button that contains *str* as a label. After a button has been created, you can set its label by calling **setLabel( )**. You can retrieve its label by calling **getLabel( )**.
- These methods are as follows:

oid setLabel(String *str*)

String getLabel( )

Here, *str* becomes the new label for the button.

Handling Buttons:

- Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered an interest in receiving action event notifications from that component.
- 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. It contains both a reference to the button that generated the event and a reference to the *action command string* associated with the button.

Check Box:

- 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. Check boxes are objects of the **Checkbox** class.
- To retrieve the current state of a check box, call **getState( )**. To set its state, call **setState( )**. You can obtain the current label associated with a check box by calling **getLabel( )**. To set the label, call **setLabel( )**. These methods are as follows:

booleangetState( )

voidsetState(boolean*on*)

String getLabel( )

voidsetLabel(String *str*)

Handling Check Boxes

- Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener**interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent**object is supplied as the argument to this method.

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. **TextField** is a subclass of **TextComponent**.

- **TextField** (and its superclass **TextComponent**) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call **getText( )**. To set the text, call **setText( )**. These methods are as follows:

String getText( )

voidsetText(String *str*)

- Also, you can select a portion of text under program control by using **select( )**.

Handling a TextField

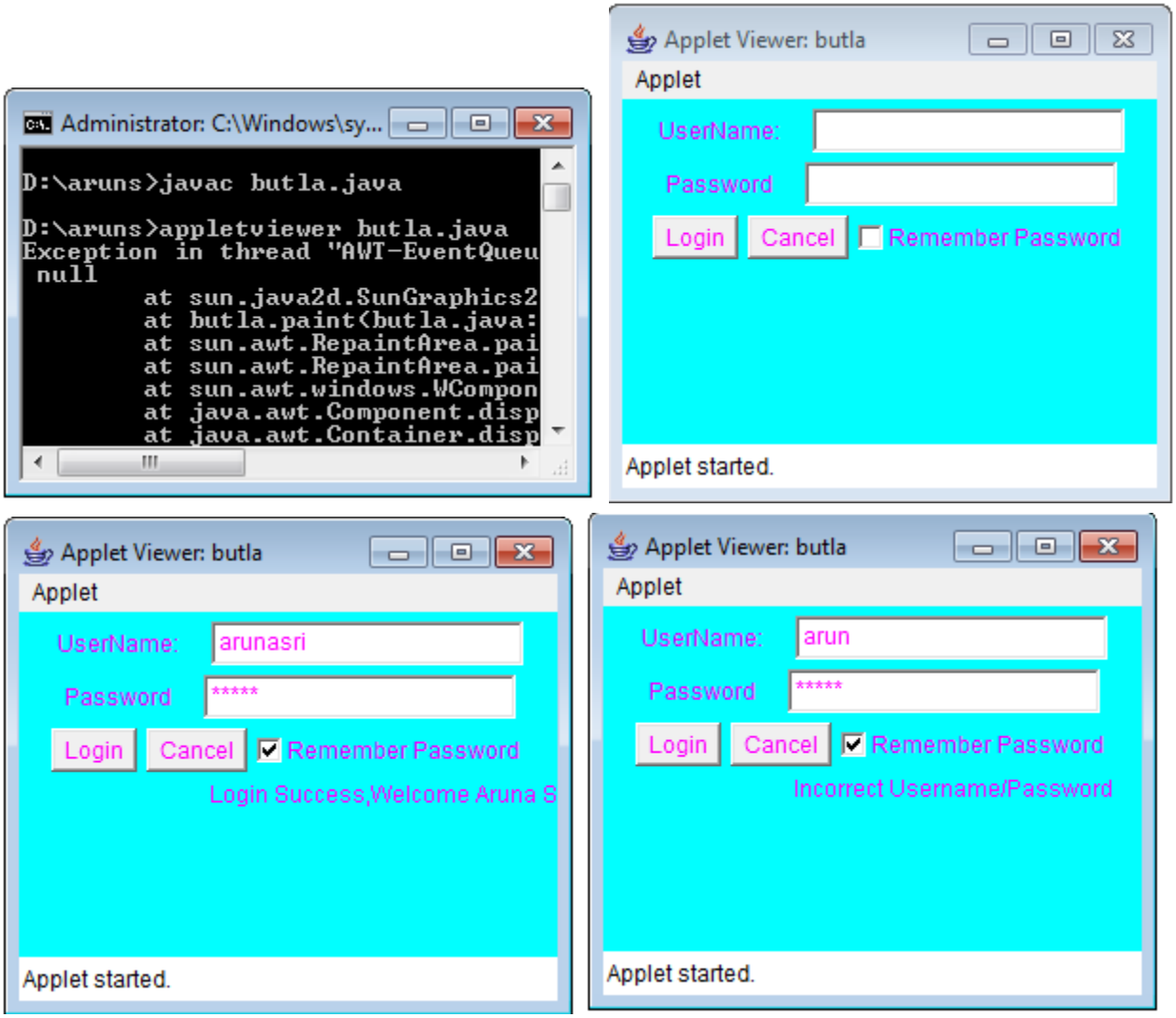
- Since text fields perform their own editing functions, your program generally will not respond to individual key events that occur within a text field. However, you may want to respond when the user presses ENTER. When this occurs, an action event is generated.

Example (Loginform (Components are Label, Button, Checkbox and TextField)):

```
importjava.awt.*;
importjava.awt.event.*;
importjava.applet.*;
/*<applet code="butla" width=500 height=500>
</applet>*/
public class butla extends Applet  implements ActionListener
{
Button b1;
Button b2;
Label l1;
Label l2;
TextField t1;
TextField t2;
Checkbox c1;
String msg;
public void init()
{
setBackground(Color.cyan);
setForeground(Color.magenta);
l1=new Label("UserName:");
t1=new TextField(20);
l2=new Label("Password");
t2=new TextField(20);
```

```
t2.setEchoChar('*');
b1=new Button("Login");
b2=new Button("Cancel");
c1=new Checkbox("Remember Password");
add(l1);
add(t1);
add(l2);
add(t2);
add(b1);
add(b2);
add(c1);
b1.addActionListener(this);
b2.addActionListener(this);

}
public void actionPerformed(ActionEvent ae)
{
String str =ae.getActionCommand();
{
String s1=t1.getText();
String s2=t2.getText();
if(s1.equals("arunasri") && s2.equals("aruns"))
{
msg="Login Success,Welcome Aruna Sri";
t1.setText("");
t2.setText("");
repaint();
}
else
{
msg="Incorrect Username/Password ";
t1.setText("");
t2.setText("");
repaint();

}
System.exit(0);
}
}
public void paint(Graphics g)
{

g.drawString(msg,100,100);

}
}
```

Output:

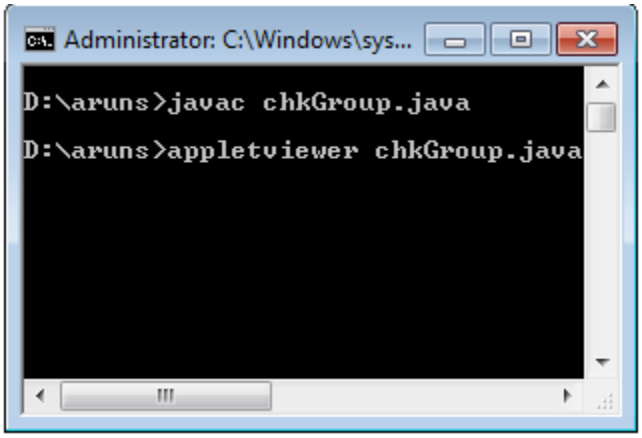


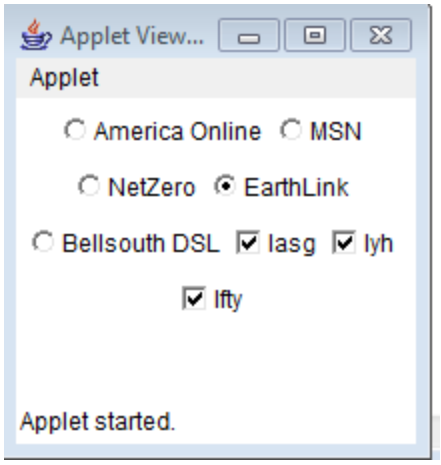
CheckboxGroup

```
importjava.awt.*;

import java.applet.*;

/*<applet code= "ChkGroup.class" width = 200 height = 150>

</applet>*/

public class ChkGroup extends Applet

{
```

```
CheckboxGroup cbgr = new CheckboxGroup();
Checkbox c1 = new Checkbox ("America Online", cbgr, false);
Checkbox c2 = new Checkbox ("MSN", cbgr, false);
Checkbox c3 = new Checkbox ("NetZero", cbgr, false);
Checkbox c4 = new Checkbox ("EarthLink", cbgr, false);
Checkbox c5 = new Checkbox ("Bellsouth DSL", cbgr, true);
public void init()
{
add(c1);    add(c2);   add(c3);  add(c4);   add(c5);
}}
```

Output:





Choice

- 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. When inactive, a **Choice** component takes uponly enough space to show the currently selected item. When the user clicks on it, the wholelist of choices pops up, and a new selection can be made. Each item in the list is a string thatappears as a left-justified label in the order it is added to the **Choice** object.
- To determine which item is currently selected, you may call either **getSelectedItem( )**or **getSelectedIndex( )**. These methods are shown here:
- String getSelectedItem( )
- intgetSelectedIndex( )
- The **getSelectedItem( )** method returns a string containing the name of the item.
- **getSelectedIndex( )** returns the index of the item. The first item is at index 0. By default,
- the first item added to the list is selected.
- To obtain the number of items in the list, call **getItemCount( )**. You can set the currentlyselected item using the **select( )** method with either a zero-based integer index or a stringthat will match a name in the list. These methods are shown here:

  intgetItemCount( )

  void select(int*index*)

  void select(String *name*)

- Given an index, you can obtain the name associated with the item at that index bycalling **getItem( )**, which has this general form:
  String getItem(int*index*)
- Here, *index* specifies the index of the desired item.

Handling Choice Lists

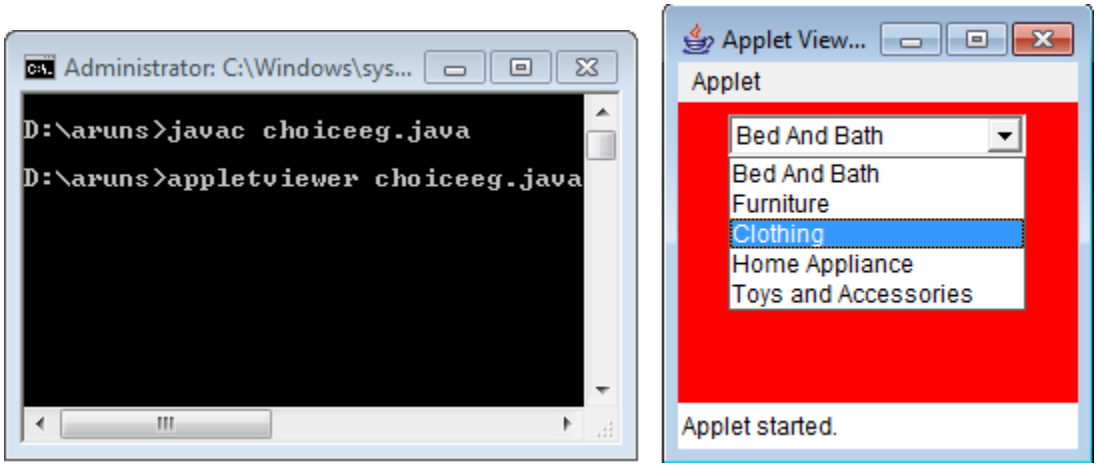
- Each time a choice is selected, an item event is generated. This is sent to any listeners thatpreviously registered an interest in receiving item event notifications from that component. Eachlistener implements the **ItemListener**interface. That interface defines the **itemStateChanged( )**method. An **ItemEvent**object is supplied as the argument to this method.

Example:

```
importjava.awt.*;
importjava.applet.*;
/*<applet code= "ChoiceTest.class" width = 200 height = 150>
</applet>*/
public class ChoiceTest extends Applet
{
Choice shoplist = new Choice();
public void init()
{
setBackground(Color.red);
shoplist.addItem("Bed And Bath");
shoplist.addItem("Furniture");
shoplist.addItem("Clothing");
shoplist.addItem("Home Appliance");
shoplist.addItem("Toys and Accessories");
add(shoplist);
```

```
}

}
```

Output:



List

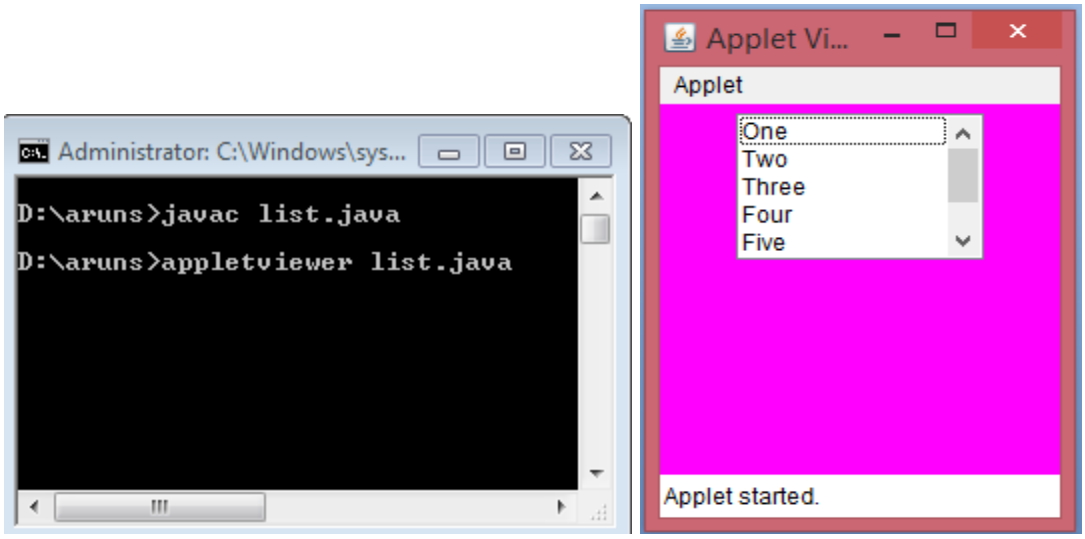
- The **List** class provides a compact, multiple-choice, scrolling selection list. Unlike the**Choice** object, which shows only the single selected item in the menu, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections.
- For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:
- String getSelectedItem( )
- intgetSelectedIndex( )
- The **getSelectedItem( )** method returns a string containing the name of the item. If more than one item is selected, or if no selection has yet been made, **null** is returned. **getSelectedIndex( )**
- returns the index of the item. The first item is at index 0. If more than one item is selected, or if
- no selection has yet been made, –1 is returned.

- For lists that allow multiple selection, you must use either **getSelectedItems( )** or **getSelectedIndexes( )**, shown here, to determine the current selections:
- String[ ] getSelectedItems( )
- int[ ] getSelectedIndexes( )
- **getSelectedItems( )** returns an array containing the names of the currently selected items. **getSelectedIndexes( )** returns an array containing the indexes of the currently selected items.

Example:

```
importjava.applet.*;
importjava.awt.*;
/*<applet code="list" width=200 height=200>
</applet>*/
public class list extends Applet
{
public void init()
{
setBackground(Color.magenta);
    List l = new List(5);
l.add("One");
l.add("Two");
l.add("Three");
l.add("Four");
l.add("Five");
l.add("Six");
l.add("Seven");
add(l);
}}
```

Output:



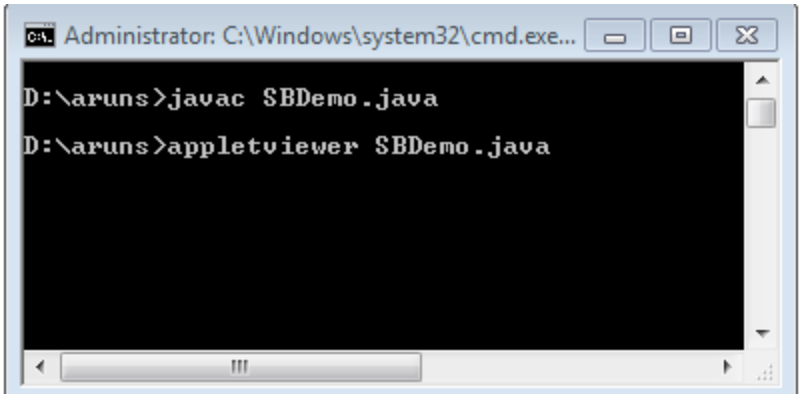
Scroll Bar

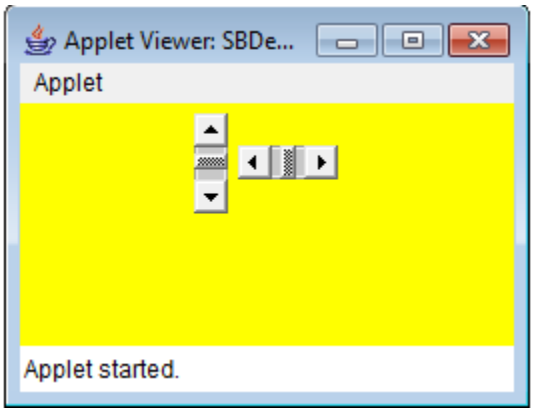
- ***Scroll bars***are used to select continuous values between a specified minimum and maximum. Scroll bars may be oriented horizontally or vertically.
- A scroll bar is actually a composite of several individual parts. Each end has an arrow that you can click to move the current value of the scroll bar one unit in the direction of the arrow. 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.
- The slider box can be dragged by the user to a new position. The scroll bar will then reflect this value. In the background space on either side of the thumb, the user can click to cause the thumb to jump in that direction by some increment larger than 1.
- **Handling Scroll Bars**
- To process scroll bar events, you need to implement the **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 scroll bar has been pressed.

Example:

```
importjava.awt.*;
importjava.awt.event.*;
importjava.applet.*;
/*<applet code="SBDemo" width=300 height=200>
</applet>*/
public class SBDemo extends Applet
{
public void init()
{
setBackground(Color.yellow);
Scrollbar  s1 = new Scrollbar(Scrollbar.VERTICAL,0, 1, 0,0);
Scrollbar  s2 = new Scrollbar(Scrollbar.HORIZONTAL,0, 1, 0,0);
add(s1);
add(s2);
}
}
```

Output:





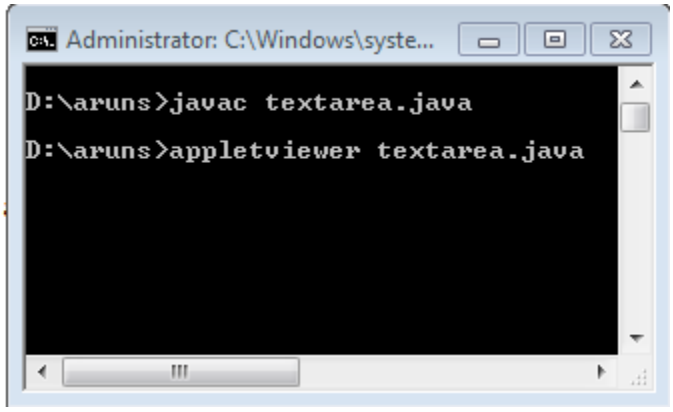
TextArea

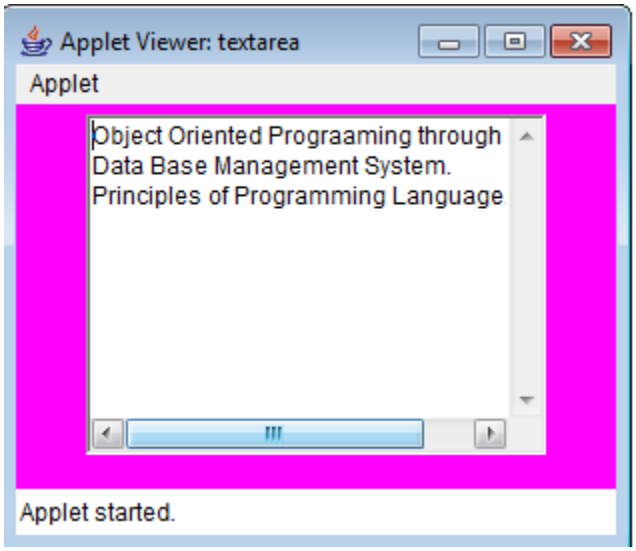
- Sometimes a single line of text input is not enough for a given task. To handle these situations,the AWT includes a simple multiline editor called **TextArea.**

```
importjava.awt.*;
importjava.applet.*;
/*
<applet code="textarea" width=300 height=250>
</applet>*/
public class textarea extends Applet
{
public void init()
{
setBackground(Color.magenta);
```

```
String val ="Object Oriented Programaming through JAVA.\n" +"Data Base Management System.\n" +"Principles of Programming Language.\n" ;
TextArea t1 = new TextArea(val, 10, 30);
add(t1);
}
}
```

Output:





Menubar

- Atop-level window can 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 the AWT by the following classes: **MenuBar**, **Menu**, and **MenuItem**.

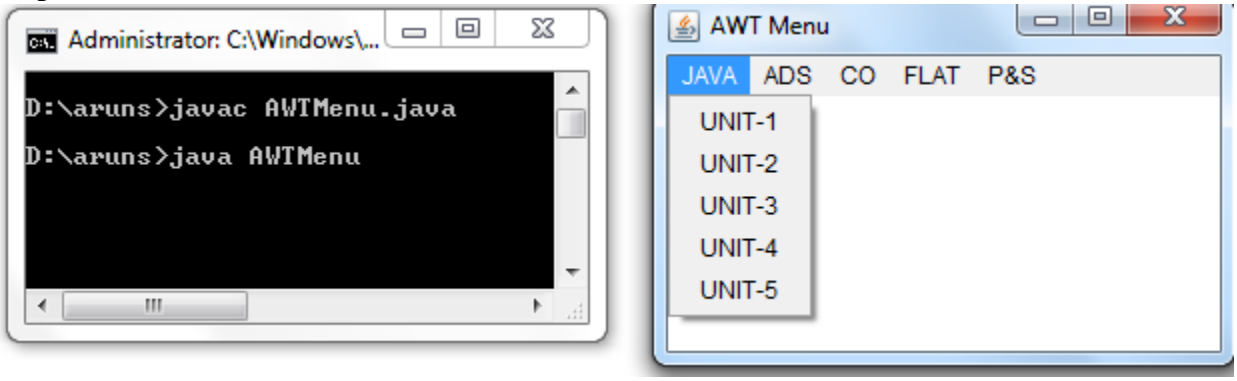
- In general, a menu bar contains one or more **Menu** objects. Each **Menu** object contains a list of**MenuItem**objects. Each **MenuItem**object represents something that can be selected by theuser. Since **Menu** is a subclass of **MenuItem**, a hierarchy of nested submenus can be created. It is also possible to include checkable menu items. These are menu options of type **CheckboxMenuItem**and will have a check mark next to them when they are selected.
- To create a menu bar, first create an instance of **MenuBar**. This class only defines the default constructor. Next, create instances of **Menu** that will define the selections displayed on the bar.

Example:

```
importjava.awt.*;
classAWTMenu extends Frame
{
MenuBar mb1;
Menu m1,m2,m3,m4,m5;
MenuItem mi1,mi2,mi3,mi4,mi5;

publicAWTMenu()
    {
setTitle("AWT Menu");
setSize(400,400);
setVisible(true);
    //setLocationRelativeTo(null);
    mb1=new MenuBar();

    m1=new Menu("JAVA");
    m2=new Menu("ADS");
    m3=new Menu("CO");
    m4=new Menu("FLAT");
    m5=new Menu("P&S");
   // Create MenuItems
    mi1=new MenuItem("UNIT-1");
    mi2=new MenuItem("UNIT-2");
    mi3=new MenuItem("UNIT-3");
    mi4=new MenuItem("UNIT-4");
    mi5=new MenuItem("UNIT-5");

    // Attach menu items to menu
m1.add(mi1);
m1.add(mi2);
```

```
m1.add(mi3);
m1.add(mi4);
m1.add(mi5);


    // Attach menu to menu bar
mb1.add(m1);
mb1.add(m2);
mb1.add(m3);
mb1.add(m4);
mb1.add(m5);

    // Set menu bar to the frame
setMenuBar(mb1);

    }
public static void main(String args[])
    {
newAWTMenu();

    }

}
```

Output:



Example:

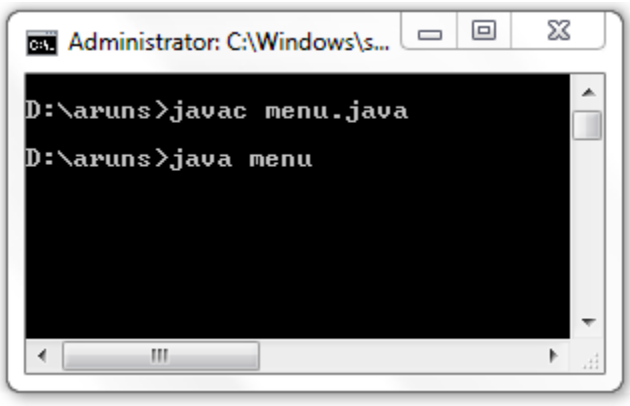
```
importjava.awt.*;
importjava.awt.event.*;

public class menu {
    Frame mainFrame;
    Label headerLabel;
    Label statusLabel;
    Panel controlPanel;

public menu(){
```

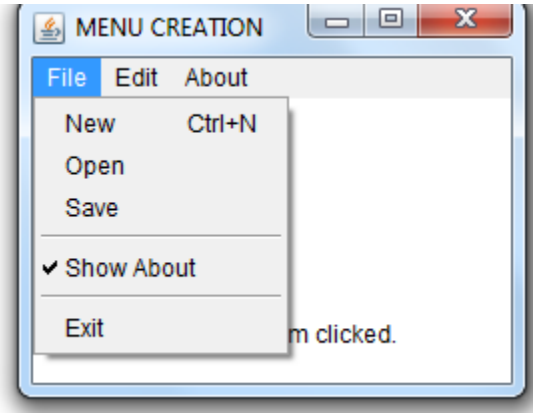
```
menudisplay();
   }

public static void main(String[] args)
  {
menu  obj = new menu();
obj.showMenu();
   }

public  voidmenudisplay()
  {
mainFrame = new Frame("MENU CREATION");
mainFrame.setSize(200,200);
mainFrame.setLayout(new GridLayout(3, 1));
mainFrame.addWindowListener(new WindowAdapter()
      {
public void windowClosing(WindowEventwindowEvent)
      {
System.exit(0);

      }
      });

headerLabel = new Label();
headerLabel.setAlignment(Label.CENTER);
statusLabel = new Label();
statusLabel.setAlignment(Label.CENTER);
statusLabel.setSize(350,100);

controlPanel = new Panel();
controlPanel.setLayout(new FlowLayout());

mainFrame.add(headerLabel);
mainFrame.add(controlPanel);
mainFrame.add(statusLabel);
mainFrame.setVisible(true);
   }

public void showMenu()
     {
      //create a menu bar
final  MenuBarmenuBar = new MenuBar();

      //create menus
      Menu fMenu = new Menu("File");
      Menu eMenu = new Menu("Edit");
final Menu aMenu = new Menu("About");
```

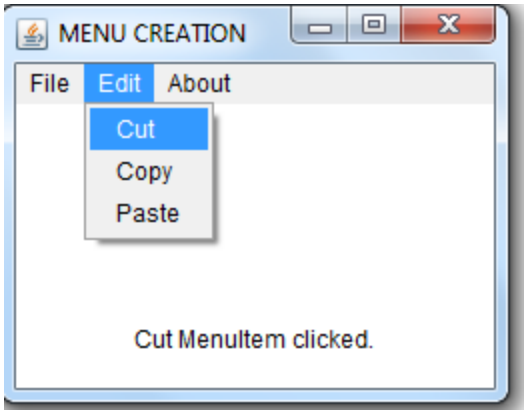
```
      //create menu items
MenuItemnewi=new MenuItem("New",newMenuShortcut(KeyEvent.VK_N));
newi.setActionCommand("New");

MenuItem open= new MenuItem("Open");
open.setActionCommand("Open");

MenuItem save= new MenuItem("Save");
save.setActionCommand("Save");

MenuItem exit= new MenuItem("Exit");
exit.setActionCommand("Exit");

MenuItem cut = new MenuItem("Cut");
cut.setActionCommand("Cut");

MenuItem copy = new MenuItem("Copy");
copy.setActionCommand("Copy");

MenuItem paste = new MenuItem("Paste");
paste.setActionCommand("Paste");

MenuItemListener mil = new MenuItemListener();

newi.addActionListener(mil);
open.addActionListener(mil);
save.addActionListener(mil);
exit.addActionListener(mil);
cut.addActionListener(mil);
copy.addActionListener(mil);
paste.addActionListener(mil);

final                  CheckboxMenuItemshowWindowMenu         =new
CheckboxMenuItem("Show About", true);
showWindowMenu.addItemListener(new ItemListener()
      {
public void itemStateChanged(ItemEvent e)
         {
if(showWindowMenu.getState())
            {
menuBar.add(aMenu);
            }
else
           {
menuBar.remove(aMenu);
            }
```

```
            }
        });

        //add menu items to menus
fMenu.add(newi);
fMenu.add(open);
fMenu.add(save);
fMenu.addSeparator();
fMenu.add(showWindowMenu);
fMenu.addSeparator();
fMenu.add(exit);

eMenu.add(cut);
eMenu.add(copy);
eMenu.add(paste);

        //add menu to menubar
menuBar.add(fMenu);
menuBar.add(eMenu);
menuBar.add(aMenu);

        //add menubar to the frame
mainFrame.setMenuBar(menuBar);
mainFrame.setVisible(true);
    }

classMenuItemListener implements ActionListener
    {
public void actionPerformed(ActionEvent e)
        {
statusLabel.setText(e.getActionCommand()+ " MenuItem clicked.");
        }
    }
}
```

Output:







Layout Managers

FlowLayout

- **FlowLayout**is the default layout manager. **FlowLayout**implements a simple layout style, which is similar to how words flow in a text editor. The direction of the layout is governed by the container's component orientation property, which, by default, is left to right, top to bottom. Therefore, by default, components are laid out line-by-line beginning at the upper-left corner. In all cases, when a line is filled, layout advances to the next line. A small space is left between
- each component, above and below, as well as left and right.

FlowLayout.LEFT

FlowLayout.CENTER

FlowLayout.RIGHT

FlowLayout.LEADING

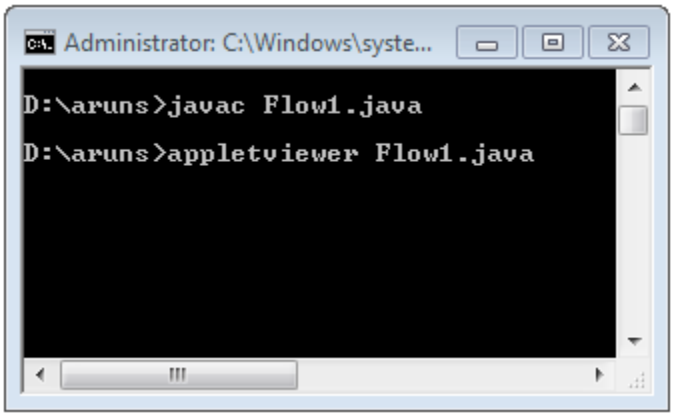
FlowLayout.TRAILING

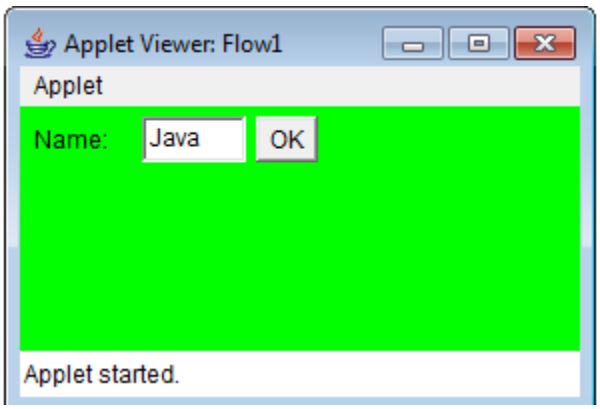
Example:

```
importjava.applet.Applet;
importjava.awt.*;
/*<applet code="Flow1" width=500 height=500>
</applet>*/
public class Flow1 extends Applet
{
public void init()
  {
setBackground(Color.green);
setLayout ( new FlowLayout(FlowLayout.LEFT) );
    Label label = new Label("Name:");
```

```
add(label);
TextFieldtf = new TextField("Java");
add(tf);
    Button b = new Button("OK");
add(b);
  }
}
```

Output:





BorderLayout

- The **BorderLayout**class implements a common layoutstyle for top-level windows. 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.

BorderLayout.CENTER

BorderLayout.SOUTH

BorderLayout.EAST

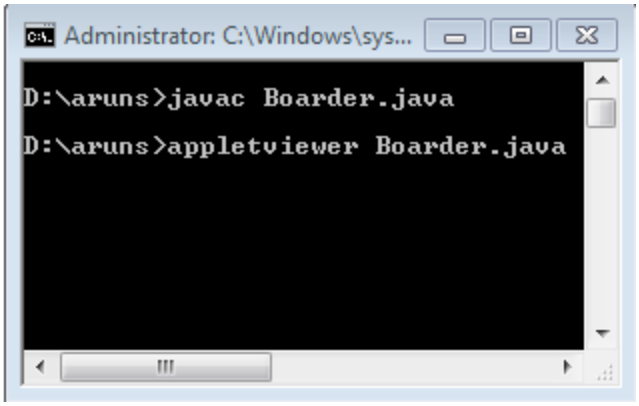
BorderLayout.WEST

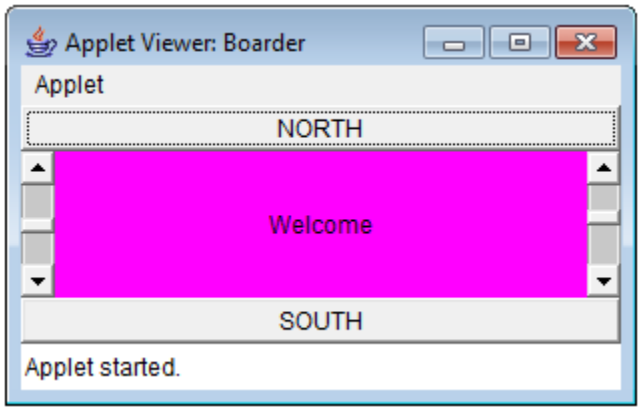
BorderLayout.NORTH

Example:

```
importjava.applet.*;
importjava.awt.*;
/*<applet code="Boarder" width=500 height=500>
</applet>*/
public class Boarder extends Applet
{
public void init()
  {
setBackground(Color.magenta);
setLayout (new BorderLayout());
    Label l = new Label("Welcome", Label.CENTER);
    Scrollbar sb1 = new Scrollbar();
    Scrollbar sb2 = new Scrollbar();
    Button b1=new Button("NORTH");
      Button b2=new Button("SOUTH");
add(b1,BorderLayout.NORTH);
add(b2,BorderLayout.SOUTH);
add(l, BorderLayout.CENTER);
add(sb1, BorderLayout.WEST);
add(sb2, BorderLayout.EAST);
  }
}
```

Output:





GridLayout

GridLayoutlays out components in a two-dimensional grid. When you instantiate a **GridLayout**, you define the number of rows and columns.

// Demonstrate GridLayout

```
importjava.awt.*;
importjava.applet.*;
/*
<applet code="Grid" width=300 height=200>
</applet>
*/
public class Grid extends Applet
 {
static final int n = 4;
public void init()
```

```
{
setBackground(Color.cyan);
setLayout(new GridLayout(n, n));
setFont(new Font("SansSerif", Font.BOLD, 24));
for(inti = 0; i< n; i++)
{
for(int j = 0; j < n; j++)
{
int k = i * n + j;
if(k > 0)
add(new Button("" + k));
}
}
}
}
```

Output: