Chapter 2

AWT

2.1 Introduction
2.2 AWT classes
2.3 Windows fundamental
2.4 Working with frame windows
2.5 Control fundamentals
2.6 Understanding layout manager

2.1 Introduction

The Abstract Window Toolkit (AWT) was introduced in previous chapter because it provides support for applets. Now we will see in detail. The Java programming language class library provides a user interface toolkit called the **Abstract Windowing Toolkit**, or the **AWT**.

The AWT is both powerful and flexible. At the lowest level, the operating system transmits information from the mouse and keyboard to the program as input, and provides pixels for program output. The AWT was designed so that programmers don't have worry about the details of tracking the mouse or reading the keyboard, nor attend to the details of writing to the screen.

The AWT provides a well-designed object-oriented interface to these low-level services and resources. Because the Java programming language is platform-independent, the AWT must also be platform-independent. The AWT was designed to provide a common set of tools for graphical user interface design that work on a variety of platforms.

2.2 AWT classes

The AWT classes are contained in the java.awt package. It is one of Java's largest packages. We will list some of the AWT classes:

Class	Description
AWTEvent	Encapsulates AWT events.
BorderLayout	The border layout manager. Border layouts use five components: North, South, East, West, and Center.
Button	Creates a push button control.
Canvas	A blank, semantics-free window.

Checkbox	Creates a check box control.
CheckboxGroup	Creates a group of check box controls.
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.
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. This context is used by the various output methods to display output in a window.
GridLayout	The grid layout manager. Grid layout displays components in a two-dimensional grid.
Label	Creates a label that displays a string.
List	Creates a list from which the user can choose. Similar to the standard Windows list box.
ScrollPane	A container that provides horizontal and/or vertical scroll bars for another component.
SystemColor	Contains the colors of GUI widgets such as windows, scroll bars, text, and others.
TextArea	Creates a multiline edit control.
TextComponent	A superclass for **TextArea** and **TextField**.
TextField	Creates a single-line edit control.

Toolkit	Abstract class implemented by the AWT.
Window	Creates a window with no frame, no menu bar, and no title.

Table: some AWT classes

2.3 Windows Fundamentals

The AWT contains numerous classes and methods that allow us to create and manage windows. Although 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.

The AWT defines windows according to a class hierarchy that adds functionality and specificity with each level. The two most common windows are those derived from Panel, which is used by applets, and those derived from Frame, which creates a standard window. Much of the functionality of these windows is derived from their parent classes. Thus, a description of the class hierarchies relating to these two classes is fundamental to their understanding.

Figure below shows the class hierarchy:

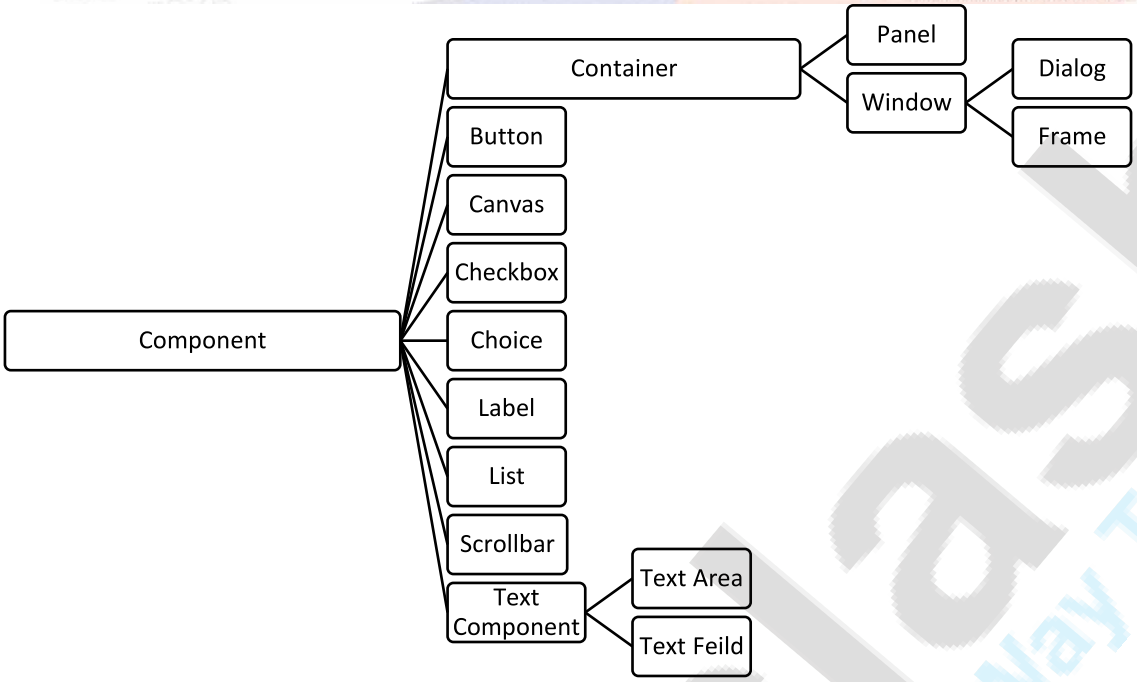


Figure: class hierarchy

2.3.1 Component

A graphical user interface is built of graphical elements called **components**. At the top of the AWT hierarchy is the Component class.

Typical components include such items as buttons, scrollbars, and text fields. Components allow the user to interact with the program and provide the user with visual feedback about the state of the program. In the AWT, all user interface components are instances of class Component or one of its subtypes.

For the simplicity, we can split the functionality of the Component class into two categories: **appearance and behavior**. The Component class contains methods and variables that control an object's general appearance. This includes basic attributes such as whether it's visible, its current size and location, and certain common graphical defaults like font and color.

By a "component's behavior," we mean the way it responds to user-driven events. When the user performs an action (like pressing the mouse button) within a component's display area, an AWT thread delivers an event object that describes "what happened." The event is delivered to objects that have registered themselves as "listeners" for that type of event from that component.

2.3.2 Container

The **Container** class is a subclass of **Component**. Components do not stand alone, but rather are found within containers. Containers contain and control the layout of components. Containers are themselves components, and can thus be placed inside other containers. In the AWT, all containers are instances of class Container or one of its subtypes.

Because Container is a subclass of Component, a Container can go inside another Container, which can go inside another Container, and so on, like Russian nesting dolls.

Containers often take on certain responsibilities for the components that they hold. Instead of every component handling events for its own bit of the user interface, a container may register itself or another object to receive the events for its child components and "glue" those events to the correct application logic.

A component informs its container when it does something that might affect other components in the container, such as changing its size or visibility. The container can then tell the layout manager (we will see in next topic) that it is time to rearrange the child components.

2.3.3 Panel

A Panel is the basic building block of an applet. It provides a container with no special features. The default layout for a Panel is FlowLayout. The details of layout manger are discussed in Section 2.6

Panel is the super-class 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**.

This is why we don't see these items when an applet is run inside a browser. 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( ), or setBounds() methods defined by Component.

2.3.4 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, we won't create Window objects directly. Instead, we will use a subclass of Window called Frame.

2.3.5 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.** If we create a Frame object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. When a Frame window is created by a program rather than an applet, a normal window is created.

2.3.6 Canvas

Canvas encapsulates a blank window upon which we can draw. The Canvas class is just a blank area; it doesn't have any predefined appearance.

You can use Canvas for drawing images, building new kinds of components, or creating super-components that are aggregates of other components. For example, you can build a picture button by drawing a picture on a Canvas and detecting mouse click events within the area of the Canvas.

2.4 Working with frame windows

After the applet, the type of window we will most often create is derived from Frame. We will use it to create child windows within applets, and top-level or child windows for applications. It creates a standard -style window.

Following are two of Frame's constructors:

```
Frame()
Frame(String title)
```

Program for creating a frame which does not contain title:

```
import java.awt.*;

public class Frame1

{

public static void main(String [] args)

{

Frame f = new Frame();

f.show();

}

}
```

(Frame1)The first form creates a standard window that does not contain a title.

Program for creating frame which contains title:

```
import java.awt.*;

public class Frame2

{

public static void main(String [] args)

{

Frame f = new Frame("welcome");

f.show();

}

}
```

The second form creates a window with the title specified by title.

The title of the frame ("`welcome`") is set in the call to the constructor. A frame is initially invisible and must be made visible by invoking its `show()` method.

Note that we cannot specify the dimensions of the window. Instead, we must set the size of the window after it has been created.

There are several methods you will use when working with **Frame** windows.
They are discussed here:

2.4.1 Setting the Window's Dimensions

The `setSize( )` method is used to set the dimensions of the window. Its signature is shown below:

```
void setSize(int newWidth, int newHeight)
void setSize(Dimension newSize)
```

The new size of the window is specified by 'newWidth' and 'newHeight', or by the 'width' and 'height' fields of the Dimension object passed in 'newSize'. The dimensions are specified in terms of pixels.

The `getSize( )` method is used to obtain the current size of a window.

Its signature is:

```
Dimension getSize( )
```

This method returns the current size of the window contained within the 'width' and 'height' fields of a Dimension object.

Program:

```
import java.awt.*;
public class ResizeFrame
{
public static void main(String [] args)
{
Frame f = new Frame("Resize");
f.setSize(500,500);
f.show();
}
```

```
}
```

2.4.2 Hiding and Showing a Window

After a frame window has been created, it will not be visible until we call `setVisible( )`.

Its signature is:

```
void set Visible(boolean visibleFlag)
```

The component is visible if the argument to this method is true. Otherwise, it is hidden.

2.4.3 Setting a Window's Title

We can change the title in a frame window using `setTitle( )`, which has this general form:

```
void setTitle(String newTitle)
```

Here, 'newTitle' is the new title for the window.

2.4.4 Closing a Frame Window

When using a frame window, our program must remove that window from the screen when it is closed, by calling set `Visible(false)`. To intercept a window close event, we must implement the `windowClosing( )` method of the Window Listener interface. Inside window `Closing( )`, we must remove the window from the screen.

2.4.5 Creating a Frame Window in an Applet

The following applet creates a subclass of Frame called SampleFrame. A window of this subclass is instantiated within the `init( )` method of AppletFrame. Note that 'SampleFrame' calls Frame's constructor. This causes a standard frame window to be created with the title passed in title. This example overrides the applet window's `start( )` and `stop( )` method s so that they show and hide the child window, respectively. It also causes the child window to be shown when the browser returns to the applet.

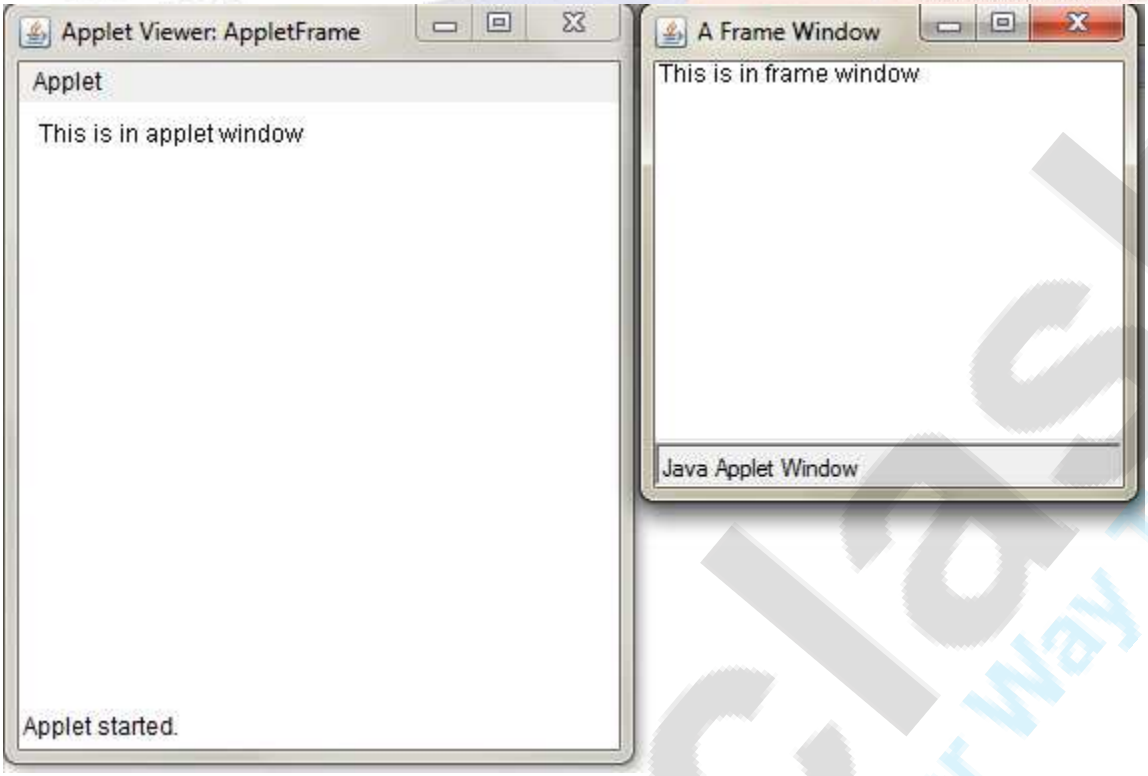
Program:

```
import java.awt.*;

import java.awt.event.*;

import java.applet.*;

/*

<applet code="AppletFrame" width=300 height=300>

</applet>

*/

// Create a subclass of Frame.

class SampleFrame extends Frame

{

SampleFrame(String title)

{

super(title);

// 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("This is in frame window", 10, 40);

}
```

```
}

class MyWindowAdapter extends WindowAdapter

{

SampleFrame sampleFrame;

public MyWindowAdapter(SampleFrame sampleFrame)

{

this.sampleFrame = sampleFrame;

}

public void windowClosing(WindowEvent we)

{

sampleFrame.setVisible(false);

}

}

// Create frame window.

public class AppletFrame extends Applet

{

Frame f;

public void init()

{

f = new SampleFrame("A Frame Window");

f.setSize(250, 250);

f.setVisible(true);
```

```
}

public void start()

{

f.setVisible(true);

}

public void stop()

{

f.setVisible(false);

}

public void paint(Graphics g)

{

g.drawString("This is in applet window", 10, 20);

}

}
```

Output:



2.5 Control fundamental

The AWT supports the following types of controls:

- Labels
- Buttons
- Check boxes and Radio Buttons
- TextField

These controls are subclasses of **Component**.

2.5.1 Label

A label is an object of type Label, and it contains a string, which it displays. The Label class can be used to add text labels to a container.

Label defines the following constructors:

`Label ()`

Creates a blank Label instance.

`Label (String Text)`

Creates a Label instance with the specified text.

`Label (String Text, int Alignment)`

Creates a Label instance with the specified text and horizontal alignment.

Common Methods:

`void setBackground (Color BgdColor)`
Sets the color of the background for the label.

`void setAlignment (int Alignment)`
Sets the alignment of the label's contents along the X axis.

`void setFont (Font TextFont)`
Sets the font for this component.Arguments

`void setForeground (Color TextColor)`
Sets the color of the text for the label.

`void setText (String Text)`
Sets the text for the label.

Arguments:

`Alignment`

The horiziontal position of the text within the label. One of:

- `Label.CENTER`

  The central position in an area.

- `Label.LEFT`

  Box-orientation constant used to specify the left side of a box.

- `Label.RIGHT`

  Box-orientation constant used to specify the right side of a box.

`BgdColor`

The color to be used for the background of the label.

`Text`

The text to appear in the label.

`TextColor`

The color to be used for the text of the label.

`TextFont`

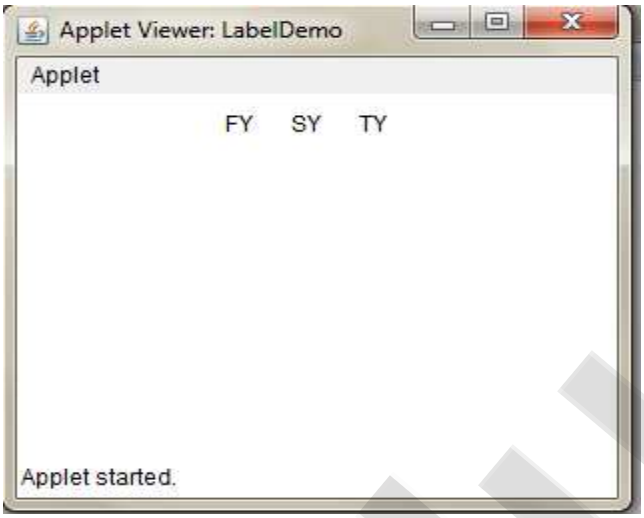
The font to be used for the text of the label.

Program:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="LabelDemo" width=300 height=200>
</applet>
*/
public class LabelDemo extends Applet
{
public void init() {
Label a = new Label("FY");
Label b = new Label("SY");
Label c = new Label("TY");
// add labels to applet window
```

```
add(a);

add(b);

add(c);

}

}
```

Output:



2.5.2 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 a blank Button instance.

`Button (String Text)`
Creates a Button instance with the specified text.

Common Methods:

`void addActionListener`
(ActionListener Handler)

`void setActionCommand`
(String ActionText)

`void setBackground (Color BackgroundColor)`
Sets the background color of the button.

`void setEnabled (boolean State)`
Enables or disables the button.

`void setFont (Font TextFont)`
Sets the font for this component.

`void setForeground (Color TextColor)`
Sets the color of the text for the button.

`void setLabel (String Text)`
Sets the text for the button.

Arguments:

`ActionText`
The ActionCommand text for the event associated with the button.

`BackgroundColor`

The color to be used for the background of the button.

`Handler`

The object which handles action events from this combo box.

`State`

The enable state of the button (true or false.)

`Text`

The text to appear next to the button.

`TextColor`

The color to be used for the text of the button.

`TextFont`

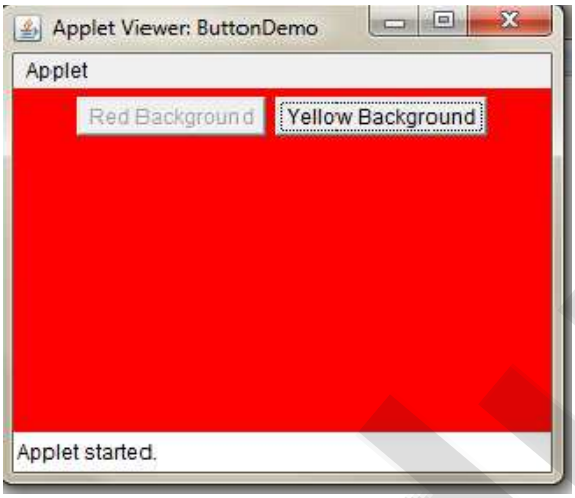
The font to be used for the text of the button.

Program:

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="ButtonDemo" width=300 height=200>
</applet>
*/
public class ButtonDemo extends Applet implements ActionListener
{
  /* Declaration */
  private LayoutManager Layout;
  private Button Red;
  private Button Yellow;
  public ButtonDemo ()
{
    /* Instantiation */
    Red = new Button ();
    Yellow = new Button ();
    Layout = new FlowLayout ();
    /* Location */
    setLayout (Layout);
    add (Red);
    add (Yellow);
    /* Decoration */
    Red.setLabel ("Red Background");
    Yellow.setLabel ("Yellow Background");
    /* Configuration */
    Red.addActionListener (this);
    Red.setActionCommand ("red");
    Yellow.addActionListener (this);
    /* Initialization */
    setBackground (Color.yellow);
    Yellow.setEnabled (false);
  }
  public void actionPerformed(ActionEvent e)
{
    String Action;
    Action = e.getActionCommand ();
```

```
    if (Action.equals ("red")) {
      setBackground (Color.red);
      Red.setEnabled (false);
      Yellow.setEnabled (true);
 }
else if (Action.equals ("Yellow Background"))
{
      setBackground (Color.yellow);
      Red.setEnabled (true);
      Yellow.setEnabled (false);
}
}
}
```

Output:



2.5.3 Check boxes and Radio Buttons

A check box is a graphical component that can be in either an "on" (true) or "off" (false) state. It consists of a small box that can either contain a check mark or not. 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.*

Checkboxes are used for multi option user input that the user may select or de-select by clicking on them. A *RadioButton*is similar to a *Checkbox*; they are basically used as option buttons to specify choices. The difference between radiobuttons and checkboxes is that only a single radiobutton can be selected in a *RadioButton* group, as opposed to checkboxes where more than one *CheckBox* can be selected.

Check box defines the following constructors:

`CheckBox ()`
Creates a blank CheckBox instance.

`CheckBox (String Text)`
Creates a CheckBox instance with the specified text.

`CheckBox (String Text, boolean Selected)`
Creates a CheckBox instance with the specified text and the designated selection state.

`CheckBox (String Text, CheckboxGroup Group, boolean Selected)`
Creates a CheckBox instance with the specified text and the designated selection state as part of a CheckbosGroup.
This causes it to interact with other members of the group, so that only one is checked at a time, and the checkbox shape is changed to round.

Common Methods:

`void addItemListener (ItemListener Handler)`
Configures an event handler for the checkbox.

`boolean isSelected ()`
Returns the state (checked or not checked) of the checkbox.

`void setBackground (Color BackgroundColor)`
Sets the background color of the checkbox.

`void setSelected (boolean State)`
Sets the state (checked or not checked) of the checkbox.

`void setFont (Font TextFont)`
Sets the font for this component.

`void setForeground (Color TextColor)`
Sets the color of the text for the checkbox.

`void setLabel (String Text)`
Sets the text for the checkbox.

Arguments:

`BackgroundColor`
The color to be used for the background of the checkbox.

`Group`
: A CheckboxGroup for interactive boxes to act as radiobuttons.

`Handler`
: The object which handles action events from this combo box.

`State`
: The checked state of the checkbox (true or false.)

`Text`
: The text to appear in the checkbox.

`TextColor`
: The color to be used for the text of the checkbox.

`TextFont`
: The font to be used for the text of the checkbox.

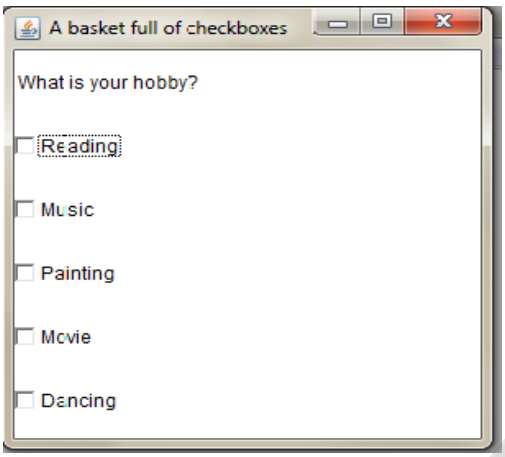
Program:

The following code demonstrates the use of checkboxes:

```
import java.awt.*;
import java.awt.event.*;
class Hobbies extends Frame
{
    Checkbox cboxRead = new Checkbox("Reading", false);
    Checkbox cboxMus = new Checkbox("Music", false);
    Checkbox cboxPaint = new Checkbox("Painting", false);
    Checkbox cboxMovie = new Checkbox("Movie", false);
    Checkbox cboxDance = new Checkbox("Dancing", false);
    Label lblQts = new Label("What is your hobby?");
    public Hobbies(String str)
    {
        super(str);
        setLayout(new GridLayout(6,1));
        add(lblQts);
        add(cboxRead);
        add(cboxMus);
        add(cboxPaint);
        add(cboxMovie);
        add(cboxDance);
        addWindowListener (new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                setVisible(false);
```

```
                System.exit(0);
            }
        });
    }
    public static void main(String args[])
    {
        Hobbies ObjHobby = new Hobbies("A basket full of checkboxes");
        ObjHobby .setSize(300, 300);
        ObjHobby.show();
    }
}
```

Output:



The following code demonstrates the use of Radio Button:

```
import java.awt.*;
import java.awt.event.*;
class Qualification extends Frame
{
    CheckboxGroup cg = new CheckboxGroup();
    Checkbox radUnder = new Checkbox("Undergraduate", cg, false);
    Checkbox radGra = new Checkbox("Graduate", cg, false);
    Checkbox radPost = new Checkbox("Post Graduate", cg, false);
    Checkbox radDoc = new Checkbox("Doctorate", cg, false);
    Label lblQts = new Label("What is your Qualification?");
    public Qualification(String str)
    {
        super(str);
        setLayout(new GridLayout(6,1));
        add(lblQts);
        add(radUnder);
```

```
        add(radGra);
        add(radPost);
        add(radDoc);
        addWindowListener (new WindowAdapter()
        {
            public void windowClosing(WindowEvent we)
            {
                setVisible(false);
                System.exit(0);
            }
        });
    }
    public static void main(String args[])
    {
        Qualification ObjQualification = new
Qualification("Literacy!!");
        ObjQualification.pack();
        ObjQualification.show();
    }
}
```

The *CheckboxGroup* class can be used to create radiobuttons. In this example, three parameters are passed to the constructor; the String label, the *CheckboxGroup* instance and the initial state of the radiobutton.

Output:

2.5.4 TextField

The TextField class can be used to add single line text input to a container. Text fields allow the user to enter strings and to edit the text.

TextField defines the following constructors:

`TextField ()`
Creates a blank TextField instance.

`TextField (String Text)`
Creates a TextField instance with the specified text.

`TextField (int Columns)`
Creates a blank TextField instance with the specified number of columns.

`TextField (String Text, int Columns)`
Creates a TextField instance with the specified text and the specified number of columns.

Common Methods:

`void addActionListener (ActionListener Handler)`
Configures an event handler for the TextField.

`String getText ()`
Returns the text in the field.

`void setBackground (Color BackgroundColor)`
Sets the background color of the TextField.

`void setEditable (boolean Editable)`
Sets the field as being editable or fixed.

`void setFont (Font TextFont)`
Sets the font for this component.

`void setText (String Text)`
Sets the text for the field.

Arguments:

`BackgroundColor`
The color to be used for the background of the field.

`Editable`
The state of the field as editable or fixed.

Handler

The object which handles action events from this field.

Text

The text to appear in the field.

TextFont

The font to be used for the text of the button.

Program:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
/*
<applet code="TextField2" width=380 height=150>
</applet>
*/
public class TextField2 extends Applet implements ActionListener
{
  /* Declaration */
  TextField Input;
  TextField Echo;
  LayoutManager Layout;
  public TextField2 ()
{
    /* Instantiation */
    Input = new TextField ("Input", 35);
    Echo = new TextField ("Text entered above will appear here.", 35);
    Layout = new FlowLayout ();
    /* Decoration */
    setBackground (Color.white);
    Input.setBackground (Color.green);
    Echo.setForeground (Color.blue);
    /* Location */
    setLayout (Layout);
    add (Input);
    add (Echo);
    /* Configuration */
    Echo.setEditable (false);
    Input.addActionListener (this);
  }
  public void actionPerformed (ActionEvent e)
{
    Echo.setText (Input.getText());
}
}
```

Output:

2.6 Understanding layout manager

Layouts allow you to format components on the screen in a platform-independent way. Without layouts, you would be forced to place components at explicit locations on the screen, creating obvious problems for programs that need to run on multiple platforms.
There is no guarantee that a TextArea or a Scrollbar or any other component will be the same size on each platform; in fact, you can bet they won't be.

In an effort to make your Java creations portable across multiple platforms, Sun created a LayoutManager interface that defines methods to reformat the screen based on the current layout and component sizes. Layout managers try to give programs a consistent and reasonable appearance, regardless of the platform, the screen size, or actions the user might take.

A container has a so-called *layout manager* to arrange its components. The layout managers provide a level of abstraction to map your user interface on all windowing systems, so that the layout can be *platform-independent*.

The standard JDK provides five classes that implement the LayoutManager interface.

They are:

- FlowLayout,
- GridLayout,
- BorderLayout,
- CardLayout
- Grid- BagLayout

Some of these layouts are used in previous topics.

This section also discusses how to create complex layouts by combining layout managers and how to write your own LayoutManager. This interface extends the LayoutManager inter face for managers that provide constraint-based layouts.

A container has a setLayout() method to set its layout manager:

```
// java.awt.Container
public void setLayout(LayoutManager mgr)
```

To set up the layout of a Container (such as Frame, JFrame, Panel, or JPanel), you have to:

- Construct an instance of the chosen layout object, via new and constructor, e.g., new FlowLayout())
- Invoke the setLayout() method of the Container, with the layout object created as the argument;
- Place the GUI components into the Container using the add() method in the correct order; or into the correct zones.

6.1 FlowLayout

The FlowLayout is the **default layout** for the Panel class, which includes its most famous subclass, Applet. When you add components to the screen, they flow left to right (centered within the applet) based upon the order added and the width of the applet. When there are too many components to fit, they "wrap" to a new row, similar to a word processor with word wrap enabled. If you resize an applet, the components' flow will change based upon the new width and height.
Constructors:

```
public FlowLayout();

public FlowLayout(int align);
```
align: FlowLayout.LEFT (or LEADING), FlowLayout.RIGHT (or TRAILING), or FlowLayout.CENTER

```
public FlowLayout(int align, int hgap, int vgap);
```
hgap, vgap: horizontal/vertical gap between the components
By default: hgap=5, vgap=5, align=CENTER

Program:

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

// An AWT GUI program inherits the top-level container java.awt.Frame
```

```
public class AWTFlowLayoutDemo extends Frame
{
   private Button btn1, btn2, btn3, btn4, btn5, btn6;
   public AWTFlowLayoutDemo ()
{
      setLayout(new FlowLayout());
         // "this" Frame sets layout to FlowLayout, which arranges the
components
         // from left-to-right, and flow from top-to-bottom.

      btn1 = new Button("Button 1");
      add(btn1);
      btn2 = new Button("This is Button 2");
      add(btn2);
      btn3 = new Button("3");
      add(btn3);
      btn4 = new Button("Another Button 4");
      add(btn4);
      btn5 = new Button("Button 5");
      add(btn5);
      btn6 = new Button("One More Button 6");
      add(btn6);

      setTitle("FlowLayout Demo");
      setSize(280, 150);
      setVisible(true);
}
   public static void main(String[] args)
{
      new AWTFlowLayoutDemo();
}
}
```

Output:

6.2 GridLayout

The GridLayout is widely used for arranging components in rows and columns. As with FlowLayout, the order in which you add components is relevant. You start at row one, column one, move across the row until it's full, then continue on to the next row.
GridLayout can reposition or resize objects after adding or removing components. Whenever the area is resized, the components within it are resized.
Constructors:

```
public GridLayout(int rows, int columns);

public GridLayout(int rows, int columns, int hgap, int vgap);
```

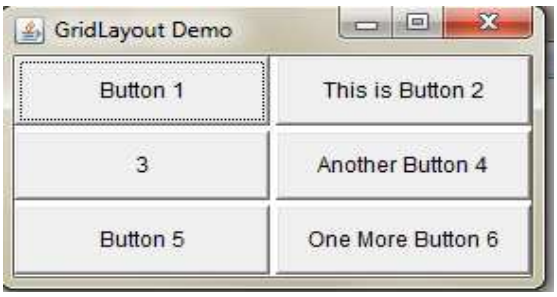
By default: rows=1, cols=0, hgap=0, vgap=0

Program:

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

// An AWT GUI program inherits the top-level container java.awt.Frame
public class AWTGridLayoutDemo extends Frame
{
   private Button btn1, btn2, btn3, btn4, btn5, btn6;
   public AWTGridLayoutDemo ()
{
      setLayout(new GridLayout(3, 2, 3, 3));
         // "this" Frame sets layout to 3x2 GridLayout, horizontal and
verical gaps of 3 pixels

      // The components are added from left-to-right, top-to-bottom
      btn1 = new Button("Button 1");
      add(btn1);
      btn2 = new Button("This is Button 2");
      add(btn2);
      btn3 = new Button("3");
      add(btn3);
      btn4 = new Button("Another Button 4");
      add(btn4);
      btn5 = new Button("Button 5");
      add(btn5);
      btn6 = new Button("One More Button 6");
      add(btn6);
      setTitle("GridLayout Demo");
      setSize(280, 150);
      setVisible(true);
   }
   public static void main(String[] args)
{
      new AWTGridLayoutDemo();
}
```

```
}
```

Output:



6.3 BorderLayout

BorderLayout is one of the more unusual layouts provided. It is the default layout for Window, along with its children, Frame and Dialog. BorderLayout provides five areas to hold components. These areas are named after the four different borders of the screen, North, South, East, and West, with any remaining space going into the Center area. When you add a component to the layout, you must specify which area to place it in. The order in which components are added to the screen is not important, although you can have only one component in each area.

The NORTH and SOUTH components may be stretched horizontally; the EAST and WEST components may be stretched vertically; the CENTER component may stretch both horizontally and vertically to fill any space left over.

Constructors:

```
public BorderLayout();

public BorderLayout(int hgap, int vgap);
```

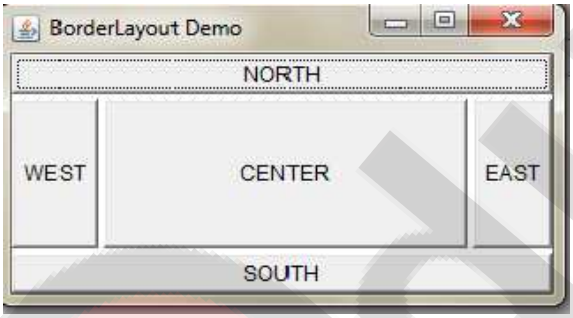
By default hgap=0, vgap=0

Program:

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

// An AWT GUI program inherits the top-level container java.awt.Frame
public class AWTBorderLayoutDemo extends Frame
{
   private Button btnNorth, btnSouth, btnCenter, btnEast, btnWest;
   public AWTBorderLayoutDemo ()
{
      setLayout(new BorderLayout(3, 3));
```

```
        // horizontal and vertical gaps of 3 pixels
      // The components are added to the specified zone
      btnNorth = new Button("NORTH");
      add(btnNorth, BorderLayout.NORTH);
      btnSouth = new Button("SOUTH");
      add(btnSouth, BorderLayout.SOUTH);
      btnCenter = new Button("CENTER");
      add(btnCenter, BorderLayout.CENTER);
      btnEast = new Button("EAST");
      add(btnEast, BorderLayout.EAST);
      btnWest = new Button("WEST");
      add(btnWest, BorderLayout.WEST);
      setTitle("BorderLayout Demo");
      setSize(280, 150);
      setVisible(true);
}
   public static void main(String[] args)
{
      new AWTBorderLayoutDemo();
}
}
```

Output:



6.4 CardLayout

The CardLayout is a bit on the strange side. A CardLayout usually manages several components, displaying one of them at a time and hiding the rest. All the components are given the same size. Usually, the CardLayout manages a group of Panels (or some other container), and each Panel contains several components of its own. With a little work, you can use the CardLayout to create tabbed dialog

boxes or property sheets, which are not currently part of AWT. CardLayout lets you assign names to the components it is managing and lets you jump to a component by
name. You can also cycle through components in order.

Note:
The CardLayout class manages two or more components that share the same display space. Conceptually, each component is like a playing card in a stack, where only the top card is visible at any time.

Methods:

```
public void first(Container parent)
public void next(Container parent)
public void previous(Container parent)
public void last(Container parent)
public void show(Container parent, String name)
```

6.5 GridBagLayout

GridBagLayout is the most sophisticated, flexible and complex of the layouts provided in the development kit. With the GridBagLayout, you can organize components in multiple rows and columns, stretch specific rows or columns when space is available, and anchor objects in different corners. You provide all the details of each component through instances of the GridBagConstraints class. The rows in the grid can have different heights, and grid columns can have different widths.

Constructors:

```
public GridBagLayout();
```