

Chapter 01

Introduction the Abstract Window Toolkit (AWT)

Abstract Window Toolkit

Graphical User Interface (GUI) offers user interaction via some graphical components.

Java **AWT** is an **API** that contains large number of **classes** and **methods** to create and manage **graphical user interface (GUI) or windows-based applications** in Java.

Java **AWT** components are **platform-dependent** i.e. components are displayed according to the view of operating system.

That means GUI designed on one platform may look different when displayed on another platform. In simple words, an AWT application will look like a windows application in Windows OS whereas it will look like a Mac application in the MAC OS.

AWT Classes

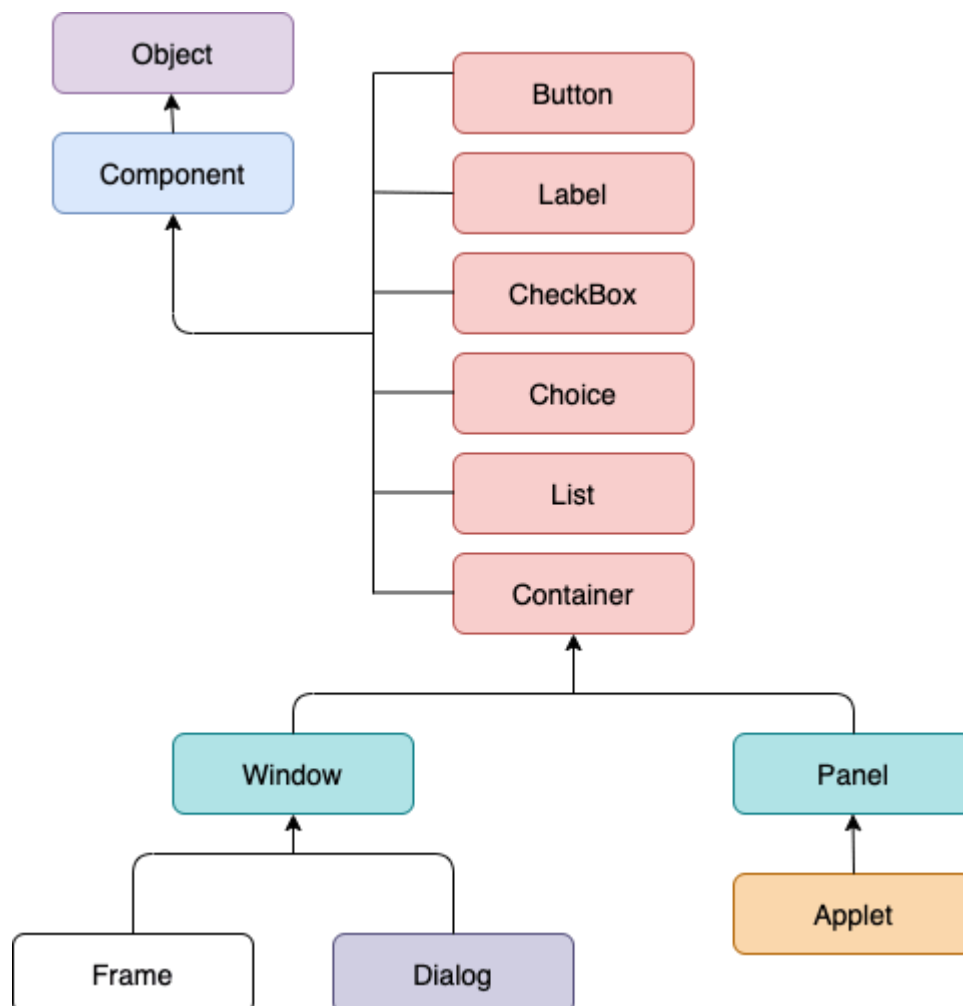
The AWT classes are contained in the java.awt package.

It is one of Java's largest packages.

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

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Components

All the **elements** like the **button**, **text fields**, **scroll bars**, etc. are called **components**.

In Java AWT, there are **classes for each component** as shown in above diagram.

In **order to place every component** in a **particular position** on a **screen**, we need to add them to a **container**.

Component class is at the **top of AWT** hierarchy.

It is an **abstract class** that encapsulates all the attributes of visual component.

A component object is responsible for remembering the current foreground and background colors and the currently selected text font.

Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc.

The **classes** that extends **Container** class are known as container such as **Window**, **Frame**, **Dialog** and **Panel**.

It is basically a screen where the where the components are placed at their specific locations.

Thus it contains and controls the layout of components.

Note: A container itself is a component, therefore we can add a container inside container.

Types of containers:

There are four types of containers in Java AWT:

1. Window
2. Panel
3. Frame
4. Dialog

Window

The window is the container that have **no borders and menu bars**.

It uses **BorderLayout** as default layout manager.

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.

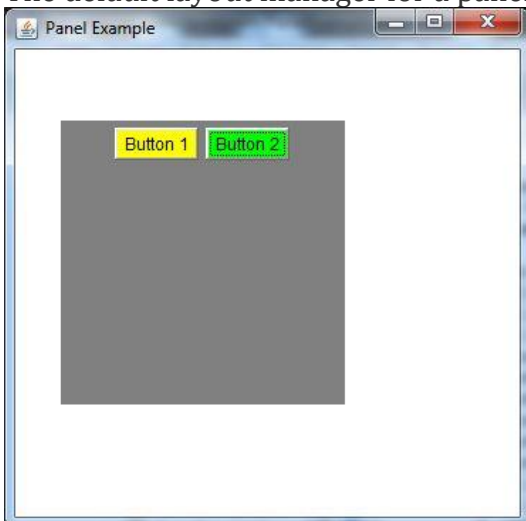
Instead, we will use a **subclass** of **Window** called **frame** or **dialog** for **creating a window**.

Panel

The class Panel is the simplest container class.

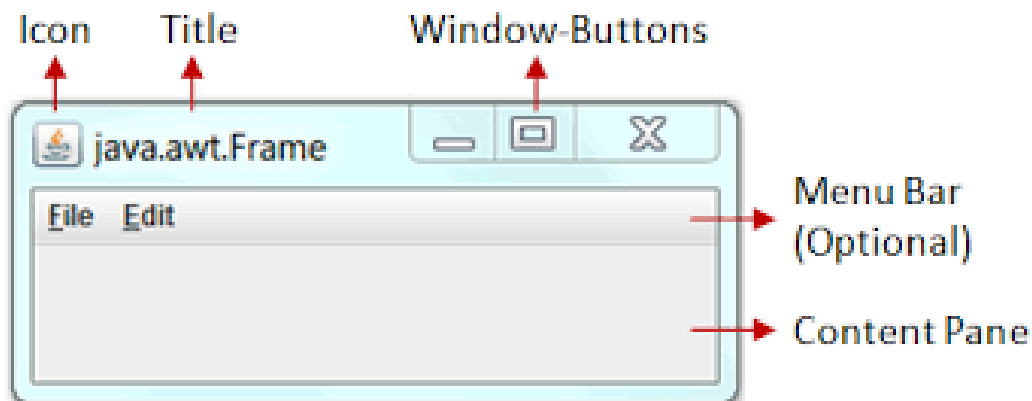
It provides space in which an application can attach any other component, including other panels.

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



Frame

The Frame is the container that contain **title bar** and **border** and can **have menu bars**. It can have other components like **button**, **text field**, **scrollbar** etc. Frame is most widely used container while developing an AWT application. It uses **BorderLayout** as default layout manager.

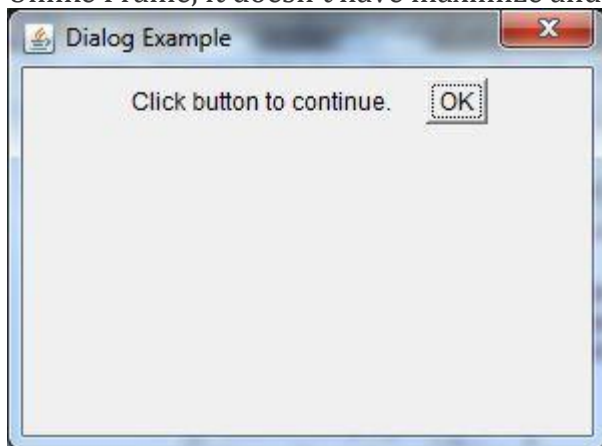


Dialog

The Dialog control represents a top level window with a border and a title used to take some form of input from the user.

It inherits the Window class.

Unlike Frame, it doesn't have maximize and minimize buttons



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 setVisible(boolean status)	Changes the visibility of the component, by default false.

Java AWT Example

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

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

1. By extending Frame class (inheritance)

Example 1:

```
import java.awt.*;
/* We have extended the Frame class here, thus our class "Sample" would behave like a Frame */
class Sample extends Frame
{
    public static void main(String args[])
    {
        // Creating the instance of Frame
        Sample fr=new Sample();
        fr.setVisible(true);
        fr.setSize(300,300);
        fr.setTitle("Welcome to Arrow");
    }
}
```

Frame()

Constructs a new instance of Frame that is initially invisible.

Example 2: Using Frame() constructor

```
import java.awt.*;
class Sample extends Frame
{
    Sample() //default
    {
        setVisible(true);
        setSize(300,300);
        setTitle("Welcome to Arrow");
    }
    public static void main(String args[])
    {
        // Creating the instance of Frame
        Sample fr=new Sample();
    }
}
```

Frame(String title)

Constructs a new, initially invisible Frame object with the specified title.

```
import java.awt.*;
class Sample extends Frame
{
    Sample(String title)
    {
        super(title);
    }
    public static void main(String args[])
    {
        Sample fr=new Sample("Welcome to Arrow World");
        fr.setVisible(true);
        fr.setSize(300,300);
    }
}
```

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

Let's see a simple example of AWT where we are creating instance of Frame class.

```
import java.awt.*;
class Sample
{
    public static void main(String args[])
    {
        // Creating the instance of Frame
        Frame fr=new Frame();
        fr.setVisible(true);
        fr.setSize(300,300);
        fr.setTitle("Welcome to Arrow");
    }
}
```

Label

It is used to display a single line of **read only text**. The text can be changed by a programmer but a user cannot edit it directly.

It is called a **passive control** as it does not create any event when it is accessed.

To create a label, we need to create the object of **Label** class.

Label class Constructors

Sr. no.	Constructor	Description
1.	Label()	It constructs an empty label.
2.	Label(String text)	It constructs a label with the given string (left justified by default).
3.	Label(String text, int alignment)	It constructs a label with the specified string and the specified alignment.

Label Class Methods

Sr. no.	Method name	Description
1.	void <u>setText(String text)</u>	It sets the texts for label with the specified text.
2.	void setAlignment(int alignment)	It sets the alignment for label with the specified alignment.
3.	String getText()	It gets the text of the label
4.	int getAlignment()	It gets the current alignment of the label.

In the absence of a layout manager, the position and size of the components have to be set manually.

The `setBounds()` method is used in such a situation to set the position and size.

To specify the position and size of the components manually, the layout manager of the frame can be null.

The `setBounds(int x-axis, int y-axis, int width, int height)` method is used in the above example that sets the position of the awt label.

Program 1 :

```
import java.awt.*;
class Sample
{
    public static void main(String args[])
    {

        // creating the object of Frame class
        Frame f = new Frame ("Label example");

        // initializing the labels
        Label l1 = new Label("First Label : ");
        Label l2 = new Label("Second Label : ");

        // adding labels to the frame
        f.add(l1);
        f.add(l2);

        // set the location of label
        l1.setBounds(50, 100, 100, 30);
        l2.setBounds(50, 150, 100, 30);

        // setting size, layout and visibility of frame
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

Program 2: Using setText() and setAlignment ()

```
import java.awt.*;

class Sample
{
    public static void main(String args[])
    {

        // creating the object of Frame class
        Frame f = new Frame ("Label example");

        // initializing the labels
        Label l1 = new Label("First Label : ", Label.RIGHT);    // using Label(String text, int alignment)
        Label l2 = new Label("Second Label : ", Label.LEFT);    // using Label(String text, int alignment)
        Label l3 = new Label();
        l3.setText("Next Label : ");                                // Using setText()
        l3.setAlignment(Label.CENTER);                            // Using SetAlignment()

        // adding labels to the frame
        f.add(l1);
        f.add(l2);
        f.add(l3);

        // set the location of label
        l1.setBounds(100, 100, 100, 30);
        l2.setBounds(100, 150, 100, 30);
        l3.setBounds(100, 200, 100, 30);

        // setting size, layout and visibility of frame
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```


Button Class Constructors

Following table shows the types of Button class

A button is basically a control component with a label that generates an event when pushed. The **Button** class is used to create a labeled button.

Sr. No.	Constructor	Description
1.	Button()	It constructs a new button with an empty string i.e. it has no label.
2.	Button (String text)	It constructs a new button with given string as its label.

Sr. no.	Method	Description
1.	void setLabel (String label)	It sets the label of button with the specified string.
2.	String getLabel()	It fetches the label of the button.

Example:

```
import java.awt.*;

class ButtonExample {
    public static void main (String[] args) {

        // create instance of frame with the label
        Frame f = new Frame("Button Example");

        // create instance of button with label
        Button b1 = new Button("Ok");
        Button b2 = new Button("Cancel");
        // set the position for the button in frame
        b1.setBounds(50,100,80,30);
        b2.setBounds(200,100,80,30);
        // add button to the frame
        f.add(b1);
        f.add(b2);
        // set size, layout and visibility of frame
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

Java AWT TextField

The object of a **TextField** class is a text component that **allows a user to enter a single line text and edit it.**

Sr. no.	Constructor	Description
1.	TextField()	It constructs a new text field component.
2.	TextField(String text)	It constructs a new text field initialized with the given string text to be displayed.

```
// importing AWT class
import java.awt.*;

class Sample {
    // main method
    public static void main(String args[]) {
        // creating a frame
        Frame f = new Frame("TextField Example");

        // creating objects of textfield
        Label l1=new Label("Enter First Name");
        TextField t1 = new TextField();
        l1.setBounds(50,100,150,30);
        t1.setBounds(200,100,200,30);
        Label l2=new Label("Enter Last Name");
        TextField t2 = new TextField("Welcome to Arrow.");
        l2.setBounds(50,200,150,30);
        t2.setBounds(200,200,200, 30);
        // adding the components to frame
        f.add(l1);
        f.add(t1);
        f.add(l2);
        f.add(t2);
        // setting size, layout and visibility of frame
        f.setSize(500,500);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

Java AWT TextArea

The object of a TextArea class is a multiline region that displays text. It allows the editing of multiple line text.

Constructors of TextArea

Constructor	Description
public TextArea()	Creates a new TextArea.
public TextArea(String text)	Creates a new TextArea with specified text.
public TextArea(int rows, int columns)	Creates a new TextArea with specified number of rows and columns.
public TextArea(String text, int rows, int columns)	Creates a new TextArea with a text and a number of rows and columns.

Methods of TextArea class

Methods	Description
public void setText(String text)	Sets a String message on the TextArea.
public String getText()	Gets a String message of TextArea.
public int getRows()	Gets the total number of rows in TextArea.
public int getColumns()	Gets the total number of columns in TextArea.

```
import java.awt.*;

public class TextAreaExample
{
    public static void main(String args[])
    {
        Frame f = new Frame();
        // creating a text area
        TextArea area = new TextArea();
        // setting location of text area in frame
        Label l1 = new Label("Enter Address");
        area.setBounds(100, 30, 200, 100);
        l1.setBounds(20, 50, 80, 30);
        f.add(l1);
        f.add(area);
        f.setSize(500,500);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

Java AWT Checkbox

The Checkbox class is used to create a checkbox. It is used to turn an option on (true) or off (false). Clicking on a Checkbox changes its state from "on" to "off" or from "off" to "on".

Constructors of Checkbox

Constructor	Description
Checkbox()	Creates a checkbox with no text, this checkbox is unchecked by default..
Checkbox(String text)	Creates a checkbox with a text, this checkbox is unchecked by default..
Checkbox(String text, boolean b)	Creates a checkbox with a text, this checkbox is checked or unchecked depending on the boolean value.

```
import java.awt.*;

public class CheckboxExample1
{
    public static void main (String args[])
    {
        Frame f = new Frame("Checkbox Example");
        // creating the checkboxes
        Checkbox checkbox1 = new Checkbox("C++",true);
        checkbox1.setBounds(100, 100, 50, 50);
        Checkbox checkbox2 = new Checkbox("Java");
        checkbox2.setBounds(150, 100, 50, 50);
        Checkbox checkbox3 = new Checkbox("Python");
        checkbox3.setBounds(200, 100, 50, 50);

        // adding checkboxes to frame
        f.add(checkbox1);
        f.add(checkbox2);
        f.add(checkbox3);

        // setting size, layout and visibility of frame
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

CheckboxGroup

The **CheckboxGroup** class is used to group together a set of Checkbox buttons.

Exactly one check box button in a CheckboxGroup can be in the "on" state at any given time.

Pushing any button sets its state to "on" and forces any other button that is in the "on" state into the "off" state.

CheckboxGroup enables you to create radio buttons in AWT.

There is no special control for creating radio buttons in AWT.

Checkbox(**String** label, **CheckboxGroup** group, boolean state)

Creates a check box with the specified label, in the specified check box group, and set to the specified state.

```
import java.awt.*;

class CheckboxGroupExample
{
    public static void main(String args[])
    {
        Frame f= new Frame("CheckboxGroup Example");
        CheckboxGroup cbg = new CheckboxGroup();
        Checkbox checkBox1 = new Checkbox("C++", cbg, true);
        checkBox1.setBounds(100,100, 50,50);
        Checkbox checkBox2 = new Checkbox("Java",cbg, false);
        checkBox2.setBounds(100,150, 50,50);

        f.add(checkBox1);
        f.add(checkBox2);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

Choice

Choice control is used to show pop up menu of choices. Selected choice is shown on the top of the menu.

Choice Class constructor

Sr. no.	Constructor	Description
1.	Choice()	It constructs a new choice menu.

Java AWT Choice Example

In the following example, we are creating a choice menu using Choice() constructor. Then we add 5 items to the menu using add() method and Then add the choice menu into the Frame.

```
import java.awt.*;

class ChoiceExample1
{
    public static void main(String args[])
    {
        Frame f = new Frame();
        // creating a choice component
        Choice c = new Choice();
        // setting the bounds of choice menu
        c.setBounds(100, 100, 75, 75);
        // adding items to the choice menu
        c.add("Item 1");
        c.add("Item 2");
        c.add("Item 3");
        c.add("Item 4");
        c.add("Item 5");
        // adding choice menu to frame
        f.add(c);
        // setting size, layout and visibility of frame
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
    }
}
```

Java AWT List

The object of List class represents a list of text items. With the help of the List class, user can choose either one item or multiple items

AWT List Class Constructors

Sr. no.	Constructor	Description
1.	List()	It constructs a new scrolling list.
2.	List(int row_num)	It constructs a new scrolling list initialized with the given number of rows visible.
3.	List(int row_num, Boolean multipleMode)	It constructs a new scrolling list initialized which displays the given number of rows. If multipleMode is true then multiple selection allowed.

```
import java.awt.*;

public class ListExample1 {
    public static void main(String args[]) {
        Frame f = new Frame();
        List l1 = new List(5);
        List l2 = new List(5,true);
        l1.setBounds(100, 100, 75, 75);
        l2.setBounds(200, 100, 75, 75);

        l1.add("C");
        l1.add("C++");
        l1.add("Java");
        l1.add("DCC");
        l1.add("SEN");

        l2.add("CSS");
        l2.add("AJP");
        l2.add("OSY");
        l2.add("STE");
        l2.add("EST");

        f.add(l1);
        f.add(l2);
    }
}
```

```

        // setting size, layout and visibility of frame
        f.setSize(400, 400);
        f.setLayout(null);
        f.setVisible(true);
    }
}

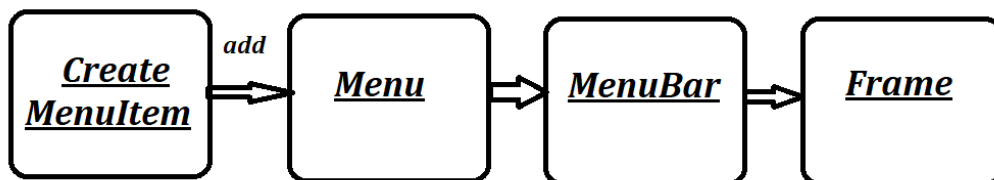
```

Java AWT MenuItem and Menu

- A menu bar can be created using **MenuBar** class.
- In order to associate the menu bar with a Frame object, call the frame's **setMenuBar** method.
- A menu bar may contain one or multiple menus, and these menus are created using **Menu** class.
- The object of **Menu** class is a pull down menu component which is displayed on the menu bar.
- A menu may contain one of multiple menu items and these menu items are created using **MenuItem** class. The object of MenuItem class adds a simple labeled menu item on menu

Simple constructors of MenuBar, Menu and MenuItem

Constructor	Description
public MenuBar()	Creates a menu bar to which one or many menus are added.
public Menu(String title)	Creates a menu with a title.
public MenuItem(String title)	Creates a menu item with a title.



//Program to create Menubar with Menu and MenuItems

```

import java.awt.*;
class MenuExample
{
    public static void main(String args[])
    {
        Frame f= new Frame("Menu and MenuItem Example");

        //Creating Menu Bar
        MenuBar mb=new MenuBar();

        //Creating Menu
        Menu m1=new Menu("File");

        //Creating Menu Items
        MenuItem i1=new MenuItem("New");
    }
}

```



```
MenuItem i2=new MenuItem("Open");
MenuItem i3=new MenuItem("Save");

m1.add(i1);
m1.add(i2);
m1.add(i3);

mb.add(m1);
f.setMenuBar(mb);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}
}
```

//Adding Submenu to Menu

```
import java.awt.*;
class MenuExample
{
    public static void main(String args[])
    {
        Frame f = new Frame("MenuBar, Menu and MenuItems");

        //Creating a menu bar
        MenuBar mb = new MenuBar();

        //Creating first menu
        Menu m1 = new Menu("File");
        Menu m2 = new Menu("Edit");

        MenuItem mItem1 = new MenuItem("New");
        MenuItem mItem2 = new MenuItem("Open");
        MenuItem mItem3 = new MenuItem("Save");

        MenuItem mItem4 = new MenuItem("Cut");
        MenuItem mItem5 = new MenuItem("Copy");
        MenuItem mItem6 = new MenuItem("Paste");

        //Adding menu items to the File menu
        m1.add(mItem1);
        m1.add(mItem2);
        m1.add(mItem3);

        //Adding menu items to the Edit menu
```

```

        m2.add(mItem4);
        m2.add(mItem5);
        m2.add(mItem6);

//Creating a second sub-menu
Menu m3 = new Menu("Save-as");
MenuItem mItem7 = new MenuItem(".jpeg");
MenuItem mItem8 = new MenuItem(".png");
MenuItem mItem9 = new MenuItem(".pdf");

//Adding menu items to the sub-menu
        m3.add(mItem7);
        m3.add(mItem8);
        m3.add(mItem9);

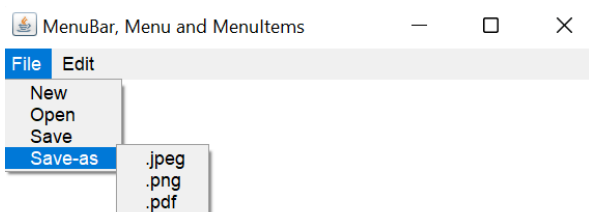
//Adding the sub-menu Save-as to the first menu File
        m1.add(m3);

//Adding our menu to the menu bar
        mb.add(m1);
        mb.add(m2);

//Adding my menu bar to the frame by calling setMenuBar() method
        f.setMenuBar(mb);

        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
}

```



Java AWT Scrollbar

The **object** of Scrollbar class is used to add horizontal and vertical scrollbar.

Scrollbar is a GUI component allows us to **see invisible number of rows and columns**.

Constructor and Description

Scrollbar()

Constructs a new vertical scroll bar.

Scrollbar(int orientation)

Constructs a new scroll bar with the specified orientation.

Scrollbar(int orientation, int value, int visible, int minimum, ~~int maximum~~)

Constructs a new scroll bar with the specified orientation, initial value, visible amount, and minimum and maximum values.

- **Value:** specify the starting position of the knob of Scrollbar on its track.
- **visible amount :** The value range represented by the bubble.
- **Minimum:** specify the minimum width of track on which scrollbar is moving.
- **Maximum:** specify the maximum width of track on which scrollbar is moving.

A scroll bar can represent a range of values. For example, if a scroll bar is used for scrolling through text, the width of the "bubble" (also called the "thumb" or "scroll box") can be used to represent the amount of text that is visible. Here is an example of a scroll bar that represents a range:



The horizontal scroll bar in this example could be created with code like the following:

```
Scrollbar sc = new Scrollbar(Scrollbar.HORIZONTAL, 0, 60, 0, 300);
```

```
import java.awt.*;

public class ScrollbarExample1
{
    public static void main(String args[])
    {
        // creating a frame
        Frame f = new Frame("Scrollbar Example");
```

```

// creating a scroll bar
Scrollbar sb1=new Scrollbar(Scrollbar.VERTICAL,0,20,0,100);
Scrollbar sb2=new Scrollbar(Scrollbar.HORIZONTAL,0,20,0,100);

// setting the position of scroll bar
sb1.setBounds(450,40,50,420);
sb2.setBounds(10,450,420,50);

// adding scroll bar to the frame
f.add(sb1);
f.add(sb2);

// setting size, layout and visibility of frame
f.setSize(500, 500);
f.setLayout(null);
f.setVisible(true);
}
}

```

Layout

Layout means the **arrangement of components** within the **container**.

In other way we can say that **placing the components at a particular position within the container**. The task of layouting the controls is done automatically by the **Layout Manager**.

Layout Manager

The layout manager automatically positions all the components within the container.

If we do not use layout manager then also the components are positioned by the default layout manager.

It is possible to layout the controls by hand but it becomes very difficult because of the following two reasons.

- It is very tedious to handle a large number of controls within the container.
- Oftenly the width and height information of a component is not given when we need to arrange them.

Java provide us with various layout manager to position the controls.

The properties like size,shape and arrangement varies from one layout manager to other layout manager.

When the size of the applet or the application window changes the size, shape and arrangement of the components also changes in response i.e. the layout managers adapt to the dimensions of appletviewer or the application window.

The layout manager is associated with every Container object.

- To apply Layout Manager, we use `setLayout()`.

void setLayout(LayoutManager object);

- If you want to use setBounds() then please make sure that you have set null to setLayout() method.

Java BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the **default layout of a frame or window**. The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **BorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class: Using BorderLayout() constructor

```
import java.awt.*;

class BorderLayoutDemo
{
    public static void main(String args[])
    {
        Frame f1 = new Frame("BorderLayout Manager");
        BorderLayout bl=new BorderLayout();
        f1.setLayout(bl);

        Button b1=new Button("B1");
        Button b2=new Button("B2");
        Button b3=new Button("B3");
        Button b4=new Button("B4");
        Button b5=new Button("B5");

        f1.add(b1, BorderLayout.NORTH);
        f1.add(b2, BorderLayout.SOUTH);
        f1.add(b3, BorderLayout.EAST);
        f1.add(b4, BorderLayout.WEST);
```

```
f1.add(b5, BorderLayout.CENTER);
```

```
f1.setVisible(true);
```

```
f1.setSize(500,500);
```

```
}
```

```
}
```

Example of BorderLayout class: Using BorderLayout(int hgap, int vgap) constructor

The following example inserts horizontal and vertical gaps between buttons using the parameterized constructor BorderLayout(int hgap, int vgap)

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

```
class BorderLayoutDemo
```

```
{
```

```
public static void main(String args[])
```

```
{
```

```
Frame f1 = new Frame("BorderLayout Manager");
```

```
BorderLayout bl=new BorderLayout(20,20);
```

```
f1.setLayout(bl);
```

```
Button b1=new Button("B1");
```

```
Button b2=new Button("B2");
```

```
Button b3=new Button("B3");
```

```
Button b4=new Button("B4");
```

```
Button b5=new Button("B5");
```

```
f1.add(b1, BorderLayout.NORTH);
```

```
f1.add(b2, BorderLayout.SOUTH);
```

```
f1.add(b3, BorderLayout.EAST);
```

```
f1.add(b4, BorderLayout.WEST);
```

```
f1.add(b5, BorderLayout.CENTER);
```

```
f1.setVisible(true);
```

```
f1.setSize(500,500);
```

```
}
```

```
}
```

Java BorderLayout: Without Specifying Region

The add() method of the Frame class can work even when we do not specify the region. In such a case, only the latest component added is shown in the frame, and all the components added previously get discarded. The latest component covers the whole area.

```
import java.awt.*;

class BorderLayoutDemo
{
    public static void main(String args[])
    {
        Frame f1 = new Frame("BorderLayout Manager");
        BorderLayout bl=new BorderLayout(20,20);
        f1.setLayout(bl);

        Button b1=new Button("B1");
        Button b2=new Button("B2");
        Button b3=new Button("B3");
        Button b4=new Button("B4");
        Button b5=new Button("B5");

        // each button covers the whole area
        // however, the B5 is the latest button
        // that is added to the frame; therefore, B5
        // is shown

        f1.add(b1);
        f1.add(b2);
        f1.add(b3);
        f1.add(b4);
        f1.add(b5);

        f1.setVisible(true);
        f1.setSize(500,500);
    }
}
```

Flow

Panel

↓

Window

FlowLayout

The **Java FlowLayout** class is used to arrange the components **in a line**, one after another (in a flow). It is the default layout of the applet or panel.

Class constructors

S.N.	Constructor & Description
1	FlowLayout() Constructs a new FlowLayout with a centered alignment and a default 5-unit horizontal and vertical gap.
2	FlowLayout(int align) Constructs a new FlowLayout with the specified alignment and a default 5-unit horizontal and vertical gap.
3	FlowLayout(int align, int hgap, int vgap) Creates a new flow layout manager with the indicated alignment and the indicated horizontal and vertical gaps.

Example 1: Default Constructor

```
import java.awt.*;

class FlowLayoutDemo
{
    public static void main(String args[])
    {
        Frame f1 = new Frame("FlowLayout Manager");

        // parameter less constructor is used
        // therefore, alignment is center
        // horizontal as well as the vertical gap is 5 units.

        FlowLayout fl=new FlowLayout();
        f1.setLayout(fl);
        Button b1=new Button("Ok");
        Button b2=new Button("Cancel");
        Button b3=new Button("RETRY");
        Button b4=new Button("B4");
        Button b5=new Button("B5");
        Button b6=new Button("B6");
```



```
Button b7=new Button("B7");
Button b8=new Button("B8");
Button b9=new Button("B9");
Button b10=new Button("B10");
```

```
f1.add(b1);
f1.add(b2);
f1.add(b3);
f1.add(b4);
f1.add(b5);
f1.add(b6);
f1.add(b7);
f1.add(b8);
f1.add(b9);
f1.add(b10);
```

```
f1.setVisible(true);
f1.setSize(500,500);
}
}
```

Example 2

```
import java.awt.*;
class FlowLayoutDemo
{
public static void main(String args[])
{
Frame f1 = new Frame("FlowLayout Manager");
// parameterized constructor is used
// where alignment is left
// horizontal gap is 20 units and vertical gap is 20 units.

FlowLayout fl=new FlowLayout(FlowLayout.LEFT,20,20);
f1.setLayout(fl);
Button b1=new Button("Ok");
Button b2=new Button("Cancel");
Button b3=new Button("RETRY");
Button b4=new Button("B4");
Button b5=new Button("B5");
Button b6=new Button("B6");
Button b7=new Button("B7");
```

```
Button b8=new Button("B8");
Button b9=new Button("B9");
Button b10=new Button("B10");
```

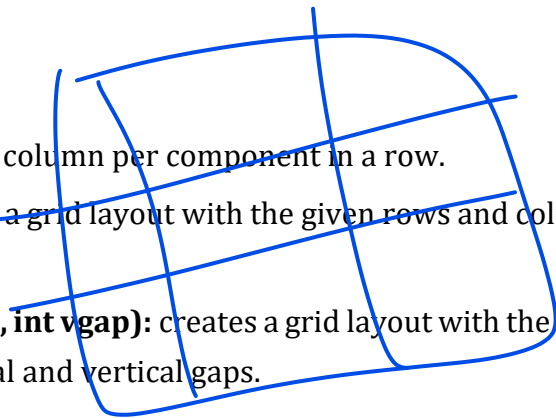
```
f1.add(b1);
f1.add(b2);
f1.add(b3);
f1.add(b4);
f1.add(b5);
f1.add(b6);
f1.add(b7);
f1.add(b8);
f1.add(b9);
f1.add(b10);
f1.setVisible(true);
f1.setSize(500,500);
}
}
```

Java GridLayout

The Java GridLayout class is used to arrange the components in a **rectangular grid**. One component is displayed in each rectangle.

Constructors of GridLayout class

1. **GridLayout()**: creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns)**: creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap)**: creates a grid layout with the given rows and columns along with given horizontal and vertical gaps.



Example of GridLayout class: Using GridLayout() Constructor

The GridLayout() constructor creates only one row. The following example shows the usage of the default constructor.

```
import java.awt.*;
class GridLayoutDemo
{
    public static void main(String args[])
    {
        Frame f1 = new Frame("GridLayout Manager");
```

```
GridLayout gl=new GridLayout();
f1.setLayout(gl);
Button b1=new Button("B1");
Button b2=new Button("B2");
Button b3=new Button("B3");
Button b4=new Button("B4");
Button b5=new Button("B5");
Button b6=new Button("B6");
Button b7=new Button("B7");
Button b8=new Button("B8");
Button b9=new Button("B9");
Button b10=new Button("B10");

f1.add(b1);
f1.add(b2);
f1.add(b3);
f1.add(b4);
f1.add(b5);
f1.add(b6);
f1.add(b7);
f1.add(b8);
f1.add(b9);
f1.add(b10);
f1.setVisible(true);
f1.setSize(500,500);
}
}
```

Example of GridLayout class: Using GridLayout(int rows, int columns) Constructor

```
import java.awt.*;
class GridLayoutDemo
{
public static void main(String args[])
{
Frame f1 = new Frame("GridLayout Manager");
GridLayout gl=new GridLayout(3,3);
f1.setLayout(gl);
Button b1=new Button("B1");
Button b2=new Button("B2");
```

```
Button b3=new Button("B3");
Button b4=new Button("B4");
Button b5=new Button("B5");
Button b6=new Button("B6");
Button b7=new Button("B7");
Button b8=new Button("B8");
Button b9=new Button("B9");
```

```
f1.add(b1);
f1.add(b2);
f1.add(b3);
f1.add(b4);
f1.add(b5);
f1.add(b6);
f1.add(b7);
f1.add(b8);
f1.add(b9);
```

```
f1.setVisible(true);
f1.setSize(500,500);
}
}
```

Example of GridLayout class: Using GridLayout(int rows, int columns, int hgap, int vgap) Constructor

```
import java.awt.*;
class GridLayoutDemo
{
    public static void main(String args[])
    {
        Frame f1 = new Frame("GridLayout Manager");
        GridLayout gl=new GridLayout(3,3,20,30);
        f1.setLayout(gl);
        Button b1=new Button("B1");
        Button b2=new Button("B2");
        Button b3=new Button("B3");
        Button b4=new Button("B4");
        Button b5=new Button("B5");
        Button b6=new Button("B6");
```

```
Button b7=new Button("B7");
Button b8=new Button("B8");
Button b9=new Button("B9");
```

```
f1.add(b1);
f1.add(b2);
f1.add(b3);
f1.add(b4);
f1.add(b5);
f1.add(b6);
f1.add(b7);
f1.add(b8);
f1.add(b9);
```

```
f1.setVisible(true);
f1.setSize(500,500);
}
}
```

Java ActionListener Interface

The Java **ActionListener** is notified whenever you click on the button or menu item. It is notified against **ActionEvent**. The **ActionListener** interface is found in **java.awt.event** package. It has only one method: **actionPerformed()**.

actionPerformed() method

The **actionPerformed()** method is **invoked automatically** whenever you click on the registered component.

```
public abstract void actionPerformed(ActionEvent e);
```

How to write ActionListener

The common approach is to implement the ActionListener. If you implement the ActionListener class, you need to follow 3 steps:

1) Implement the ActionListener interface in the class:

```
public class ActionListenerExample Implements ActionListener
```

2) Register the component with the Listener:

```
    component.addActionListener(instanceOfListenerclass);
```

3) Override the actionPerformed() method:

```
public void actionPerformed(ActionEvent e)
{
```

//Write the code here

}

Java ActionListener Example : (implements ActionListener)

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

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

```
class ActionListenerExample extends Frame implements ActionListener
```

```
{
```

```
    TextField tf;
```

```
    public ActionListenerExample()
```

```
    {
```

```
        tf = new TextField();
```

```
        tf.setBounds(50,50,150,20);
```

```
        Button b=new Button("Click Here");
```

```
        b.setBounds(50,100,60,30);
```

```
        add(b);
```

```
        add(tf);
```

```
        b.addActionListener(this);
```

```
    }
```

```
    public void actionPerformed(ActionEvent e)
```

```
    {
```

```
        tf.setText("Welcome to Arrow");
```

```
    }
```

```
    public static void main(String args[])
```

```
    {
```

```
        ActionListenerExample f1 = new ActionListenerExample();
```

```
        f1.setTitle("ActionListenerExample");
```

```
        f1.setSize(400,400);
```

```
        f1.setLayout(null);
```

```
        f1.setVisible(true);
```

```
    }
```

```
}
```

Java ActionListener Example: On Button click

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

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

```
public class ActionListenerExample
```

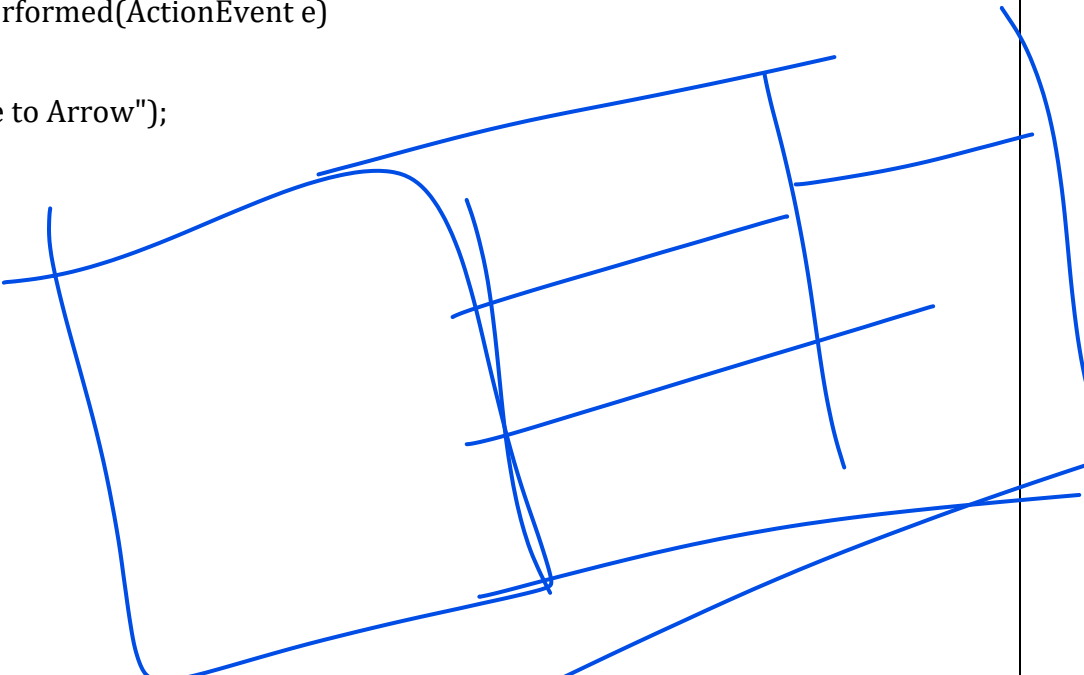
```
{
```

```
    public static void main(String[] args)
```

```

{
    Frame f=new Frame("ActionListener Example");
    TextField tf=new TextField();
    tf.setBounds(50,50, 150,20);
    Button b=new Button("Click Here");
    b.setBounds(50,100,60,30);
    b.addActionListener( new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            tf.setText("Welcome to Arrow");
        }
    } );
    f.add(b);
    f.add(tf);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
}

```



CardLayout

The `java.awt.CardLayout` layout manager is significantly different from the other layout managers. Unlike other layout managers, that display all the components within the container at once, a `CardLayout` layout manager displays only one component at a time (The component could be a component or another container).

Each component in a container with this layout fills the entire container. The name *cardlayout* evolves from a stack of cards where one card is piled upon another and only one of them is shown. In this layout, the first component that you add to the container will be at the top of stack and therefore visible and the last one will be at the bottom.

Commonly Used Methods of CardLayout Class

- **public void next(Container parent):** is used to flip to the next card of the given container.
- **public void previous(Container parent):** is used to flip to the previous card of the given container.
- **public void first(Container parent):** is used to flip to the first card of the given container.
- **public void last(Container parent):** is used to flip to the last card of the given container.
- **public void show(Container parent, String name):** is used to flip to the specified card with the given name.

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

```

import java.awt.event.*;
class CardLayoutExample extends Frame implements ActionListener
{
    CardLayout card = new CardLayout(20,20);
    CardLayoutExample()
    {
        setLayout(card);
        Button Btnfirst = new Button("first ");
        Button BtnSecond = new Button ("Second");
        Button BtnThird = new Button("Third");
        add(Btnfirst);
        add(BtnSecond);
        add(BtnThird);
        Btnfirst.addActionListener(this);
        BtnSecond.addActionListener (this);
        BtnThird.addActionListener(this);
    }
    public void actionPerformed(ActionEvent e)
    {
        card.next(this);
    }
    public static void main(String args[])
    {
        CardLayoutExample frame = new CardLayoutExample();
        frame.setTitle("CardLayout in Java Example");
        frame.setSize(220,150);
        frame.setVisible(true);
    }
}

```


GridBagLayout

The java.awt.GridBagLayout layout manager is the most powerful and flexible of all the predefined layout managers but more complicated to use. Unlike **GridLayout** where the component are arranged in a rectangular grid and **each component** in the **container** is **forced** to be the **same size**, in **GridBagLayout**, components are also arranged in rectangular grid but can have **different sizes** and can occupy **multiple rows** or **columns**.

In order to create **GridBagLayout**, we first **instantiate** the **GridBagLayout** class by using its only no-arg constructor and **defining** it as the **current layout manager**. The following statements accomplish this task;

```
GridBagLayout layout = new GridBagLayout();  
setLayout(layout);
```

A GridBagLayout layout manager **requires a lot of information to know where to put a component in a container**. A **helper class** called `GridBagConstraints` provides all this information. It specifies constraints on **how to position a component**, **how to distribute the component** and **how to resize and align them**. **Each component** in a **GridBagLayout** has its own **set of constraints**, so you have to **associate an object** of type **GridBagConstraints** with each **component** before adding component to the container.

gridx, gridy

Specify the row and column at the upper left of the component. The leftmost column has address gridx=0 and the top row has address gridy=0.

gridwidth, gridheight

Specify the number of columns (for gridwidth) or rows (for gridheight) in the component's display area. These constraints specify the number of cells the component uses, not the number of pixels it uses. The default value is 1.

ipadx, ipady

Specifies the internal padding: how much to add to the size of the component.

fill

Used when the component's display area is larger than the component's requested size to determine whether and how to resize the component.

Valid values include **NONE** (the default), **HORIZONTAL** (make the component wide enough to fill its display area horizontally, but do not change its height), **VERTICAL** (make the component tall enough to fill its display area vertically, but do not change its width), and **BOTH** (make the component fill its display area entirely).

```
import java.awt.*;  
  
public class GridBagLayoutExample extends Frame  
{  
    public GridBagLayoutExample()  
    {  
        GridBagLayout grid = new GridBagLayout();
```

```

GridBagConstraints gbc = new GridBagConstraints();
setLayout(grid);
setTitle("GridBag Layout Example");

gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridx = 0;
gbc.gridy = 0;
add(new Button("Button One"), gbc);

gbc.gridx = 1;
gbc.gridy = 0;
add(new Button("Button two"), gbc);

gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.ipady = 20;
gbc.gridx = 0;
gbc.gridy = 1;
add(new Button("Button Three"), gbc);

gbc.gridx = 1;
gbc.gridy = 1;
add(new Button("Button Four"), gbc);

gbc.gridx = 0;
gbc.gridy = 2;
gbc.fill = GridBagConstraints.HORIZONTAL;
gbc.gridwidth = 2;
add(new Button("Button Five"), gbc);
setSize(300, 300);
setVisible(true);
}

public static void main(String[] args)
{
    GridBagLayoutExample a = new GridBagLayoutExample();
}

}

```

Panel

The `java.awt.Panel` or `Panel` is a container that is designed to group a set of components, including other panels. It is visually represented as window that does not contain a [title bar](#), menu bar or border. Panels are represented by objects created from `Panel` class.

```
import java.awt.*;

class PanelJavaExample
{
    public static void main(String[] args)
    {
        Frame f1 = new Frame();
        f1.setLayout(new FlowLayout());
        Panel panel1 = new Panel();
        Label lb1 = new Label("Panel with Green Background");
        lb1.setForeground(Color.red);
        panel1.add(lb1);
        Button bt1 = new Button("1");
        panel1.add(bt1);
        panel1.setBackground(Color.green);
        f1.add(panel1);

        Panel panel2=new Panel();
        Label lb2 = new Label("Panel with Pink Background");
        panel2.add(lb2);
        Button bt2 = new Button("2");
        panel2.add(bt2);
        panel2.setBackground(Color.pink);
        f1.add(panel2);
        f1.setTitle("Panels in Java Example");
        f1.setSize(500,500);
        f1.setBackground(Color.yellow);
        f1.setVisible(true);
    }
}
```

Dialog

- The Dialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Window class.
- Unlike Frame, it doesn't have maximize and minimize buttons.
- **Modal dialog boxes**, which require the user to respond before continuing the program.
- **Modeless dialog boxes**, which stay on the screen and are available for use at any time but permit other user activities. User does not have to close it in order to continue using application.

Class constructors

S.N.	Constructor & Description
1	Dialog(Dialog owner) Constructs an initially invisible, modeless Dialog with the specified owner Dialog and an empty title.
2	Dialog(Dialog owner, String title) Constructs an initially invisible, modeless Dialog with the specified owner Dialog and title.
3	Dialog(Dialog owner, String title, boolean modal) Constructs an initially invisible Dialog with the specified owner Dialog, title, and modality.

Example

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample1
{
    public static void main(String args[])
    {

        Frame f = new Frame();
        Dialog d=new Dialog(f,"Dialog Box",true);
        d.setLayout(new FlowLayout());
        Button b = new Button ("OK");
        Label l = new Label("Click button to continue.");
        d.add(b);
        d.add(l);
        d.setSize(400,400);
        d.setVisible(true);
    }
}
```

Example with ActionListener

```
import java.awt.*;
import java.awt.event.*;
public class DialogExample extends Frame implements ActionListener
{
    Dialog d;
    DialogExample f;
    DialogExample()
    {
        d = new Dialog(f,"Dialog Example", true);
        d.setLayout( new FlowLayout());
        Button b = new Button ("OK");
        b.addActionListener (this);
        d.add( new Label ("Click button to continue."));
        d.add(b);
        d.setSize(300,300);
        d.setVisible(true);
    }
    public void actionPerformed((ActionEvent e)
    {
        d.setVisible(false);
    }
    public static void main(String args[])
    {
        DialogExample f = new DialogExample();
    }
}
```

FileDialog

The FileDialog is a subclass of **Dialog** class that displays a dialog window from which the user can select a file.

Since it is a **modal** dialog, when the application calls its show method to display the dialog, it **blocks the rest of the application** until the user has chosen a file.

There is **no Layout Manager** for FileDialog.

1	FileDialog(Frame parent) Creates a file dialog for loading a file.
2	FileDialog(Frame parent, String title) Creates a file dialog window with the specified title for loading a file.
3	FileDialog(Frame parent, String title, int mode) Creates a file dialog window with the specified title for loading or saving a file.

Field

Following are the fields for **java.awt.Image** class:

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

```
import java.awt.*;
class FileDialogDemo
{
public static void main(String args[])
{
Frame f1=new Frame();
f1.setVisible(true);
f1.setTitle("Parent Window");
f1.setSize(800,800);
//FileDialog fd1=new FileDialog(f1,"Save",FileDialog.SAVE);
FileDialog fd1=new FileDialog(f1,"Open",FileDialog.LOAD);
//fd1.setDirectory("C:\\");
fd1.setSize(500,500);
fd1.setVisible(true);
}
}
```