Unit-2
User Interface Components with Swing

Introduction

A graphical user interface (GUI) presents a user-friendly mechanism for interacting with an application. GUIs are built from GUI components (buttons, menus, labels etc.). A GUI component is an object with which the user interacts via the mouse, the keyboard or another form of input, such as voice recognition.

Java Swing is a lightweight Graphical User Interface (GUI) toolkit that includes the GUI components. Swing provides a rich set of widgets and packages to make sophisticated GUI components for Java applications. Swing is a part of Java Foundation Classes(JFC), which is an API for Java GUI programing that provide GUI. The ***javax.swing*** package provides classes for java swing API such as *JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu* etc.

Abstract Window Toolkit (AWT)

Java ***AWT (Abstract Window Toolkit)*** is an API to develop GUI or window-based applications in java. Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS. The ***java.awt*** package provides classes for AWT API such as *TextField, Label, TextArea, RadioButton, CheckBox, Choice, List* etc.

Due to its platform dependence and a kind of heavyweight nature of its components, it is rarely used in Java applications these days. Besides, there are also newer frameworks like Swing which are light-weight and platform-independent. Swing has more flexible and powerful components when compared to AWT. Swing provides components similar to Abstract Window Toolkit and also has more advanced components like trees, tabbed panels, etc.

AWT vs. Swing

AWT	***Swing***
AWT components are platform-dependent.	Java swing components are platform-independent.
AWT components are heavyweight.	Swing components are lightweight.
Java AWT has comparatively less functionality as compared to Swing.	Java Swing has more functionality as compared to AWT.
It requires more time for execution.	It requires less time for execution.
It has less powerful components compared to the Java swing.	It has more powerful components than Java AWT.
It does not support the MVC (Model-View-Controller) pattern.	It supports the MVC (Model-View-Controller) pattern.
The AWT components are provided by the package *java.awt*.	The swing components are provided by *javax.swing* package.
Using AWT , you have to implement a lot of things yourself .	Swing has them built in.

Java Applets

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

A *main()* method is not invoked on an applet, and an applet class will not define *main()*.Applets are designed to be embedded within an HTML page. When a user views an HTML page that contains an applet, the code for the applet is downloaded to the user's machine. A JVM is required to view an applet. The JVM can be either a plug-in of the Web browser or a separate runtime environment.

Advantages of Applet:

- As it operates on the client side, it requires much less response time.
- They are very secure.
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac Os etc.

Disadvantage:

- Plugin is required at client browser to execute applet.

Here is a simple Program to demonstrate Applet in Java:

```
import java.applet.Applet;
import java.awt.Graphics;
public class HelloWorld extends Applet {
   public void paint(Graphics g) {
      g.drawString("Hello World!", 50, 25);
   }
}
```

Now you have to create an HTML File that Includes the Applet.

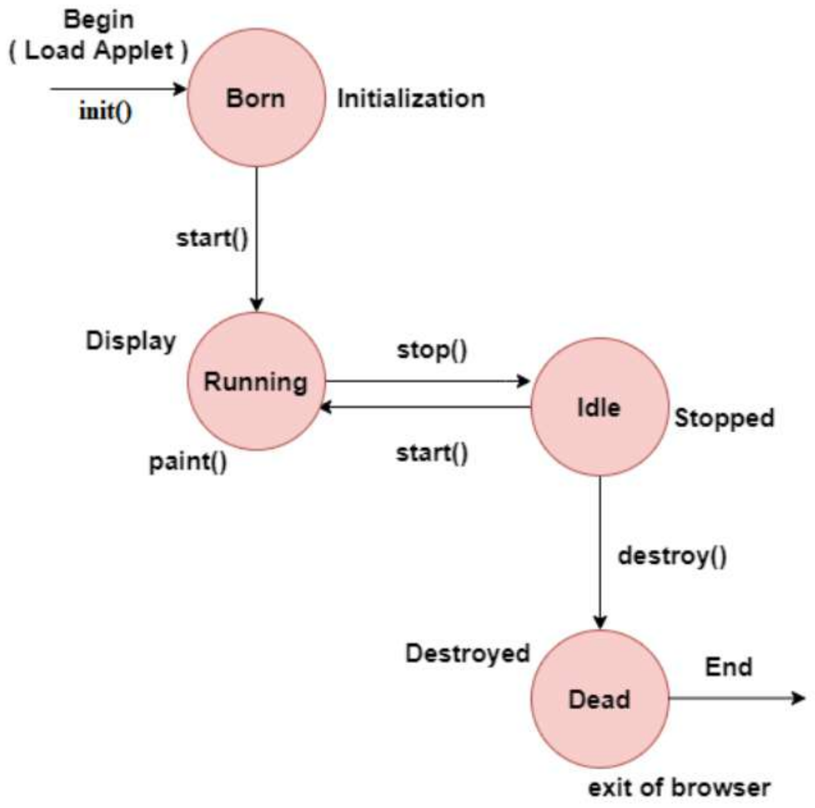
Create a file named *Hello.html* in the same directory that contains *HelloWorld.class*. This HTML file should contain the following text:

```
//Hello.html
<html>
<head>
   <title> A Simple Program </title>
</head>
<body>
   Here is the output of program:
   <applet code = "HelloWorld.class" width="150" height="25"></applet>
</body>
</html>
```

The class in the program must be declared as public, because it will be accessed by code that is outside the program. Every Applet application must declare a *paint()* method. This method is defined by AWT class and must be overridden by the applet. The *paint()* method is called each time when an applet needs to redisplay its output.

Applet Life Cycle

The life cycle of an applet is as shown in the figure below:



As shown in the above diagram, the life cycle of an applet starts with *init()* method and ends with *destroy()* method. Other life cycle methods are *start(), stop()* and *paint()*. The methods to execute only once in the applet life cycle are *init()* and *destroy()*. Other methods execute multiple times.

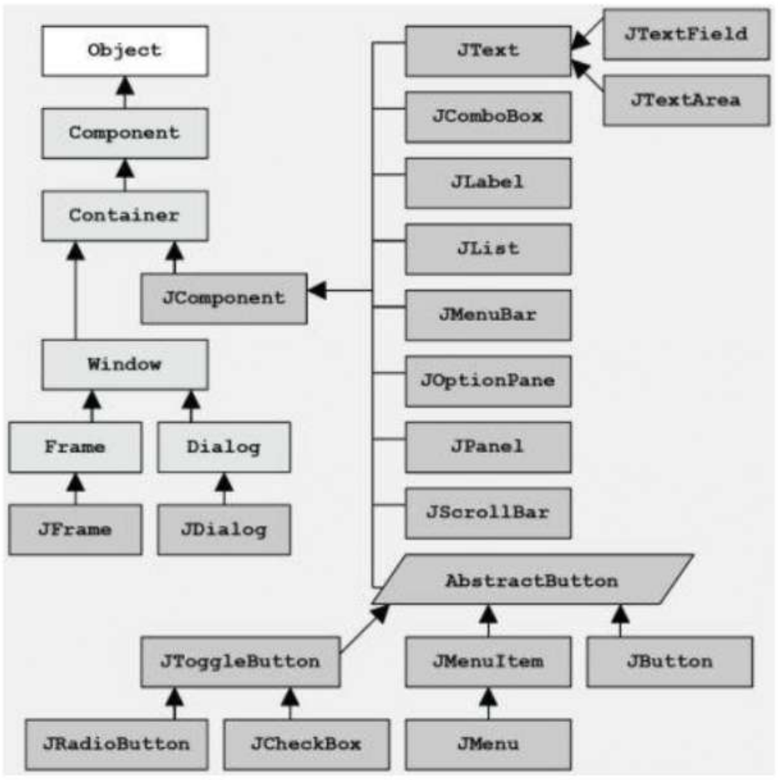
- ***Initialization or Born State:*** Applet enters the initialization state when it is first loaded. This is achieved by calling the *init()* method of the Applet class. The initialization occurs only once in the applet's life cycle.
- ***Running State:*** The *start()* method contains the actual code of the applet that should run. The *start()* method executes immediately after the *init()* method. It also executes whenever the applet is restored, maximized or moving from one tab to another tab in the browser.
- ***Idle or Stopped State:*** The *stop()* method stops the execution of the applet. The *stop()* method executes when the applet is minimized or when moving from one tab to another in the browser.
- ***Dead State:*** The *destroy()* method executes when the applet window is closed or when the tab containing the web page is closed. The *stop()* method executes just before when *destroy()* method is invoked. The *destroy()* method removes the applet object from memory.
- ***Display State:*** The *paint()* method is used to redraw the output on the applet display area. The *paint()* method executes after the execution of *start()* method and whenever the applet or browser is resized.

Example program that demonstrates the life cycle of an applet is as follows:

```
import java.awt.*;
import java.applet.*;
public class MyApplet extends Applet
{
        public void init()
        {
                System.out.println("Applet initialized");
        }
        public void start()
        {
                System.out.println("Applet execution started");
        }
        public void stop()
        {
                System.out.println("Applet execution stopped");
        }
        public void paint(Graphics g)
        {
                System.out.println("Painting...");
        }
        public void destroy()
        {
                System.out.println("Applet destroyed");
        }
}
```

Swing Class Hierarchy

The hierarchy of java swing API is given below.



Components and Containers

A swing GUI contains two main elements: *components* and *containers*.

- A ***component*** is an independent visual control like a button or label.
- A ***container*** is a special type of component that is designed to hold other components.

In order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers.

Components

In general, Swing components are derived from the *JComponent* class. All of Swing's components are represented by classes defined within the package *javax.swing*. Some of the components available in the Swings package are listed below:

JApplet	JButton	JCheckBox	JCheckBoxMenuItem
JColorChooser	JComboBox	JComponent	JDesktopPane
JDialog	JEditorPane	JFileChooser	JFormattedTextField
JFrame	JInternalFrame	JLabel	JLayer
JLayeredPane	JList	JMenu	JMenuBar
JMenuItem	JOptionPane	JPanel	JPasswordField
JPopupMenu	JProgressBar	JRadioButton	JRadioButtonMenuItem
JRootPane	JScrollBar	JScrollPane	JSeparator
JSlider	JSpinner	JSplitPane	JTabbedPane
JTable	JTextArea	JTextField	JTextPane
JTogglebutton	JToolBar	JToolTip	JTree
JViewport	JWindow		

Containers

A container holds a group of components. It provides a space where a component can be managed and displayed. Containers are of two types:

- ***Top Level Containers:*** These containers inherit the AWT classes *Component* and *Container*. The top-level containers are heavyweight. It cannot be contained within any other container. The top-level containers are ***JFrame, JApplet, JWindow,*** and ***JDialog***.
- ***Lightweight Containers:*** Lightweight containers do inherit *JComponent* class. Lightweight containers are often used to organize and manage groups of related components because a lightweight container can be contained within another container. Example of a lightweight container is ***JPanel***.

Commonly used methods of component class

Method	*Description*
`public void add(Component c)`	Add a component on another component.
`public void setSize(int width,int height)`	Sets size of the component.
`public void setLayout(LayoutManager m)`	Sets the layout manager for the component.
`public void setVisible(boolean b)`	Sets the visibility of the component. It is by default false.

Layout Management

Layout managers are used to arrange components in a particular manner within the container. LayoutManager (in package java.awt) is an interface implemented by all the classes of layout managers.

Following are the different classes of Layout manager in AWT and Swing:

1. BorderLayout
2. FlowLayout
3. GridLayout
4. GridBagLayout
5. GroupLayout

Border Layout

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.

Constants for Alignments:

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

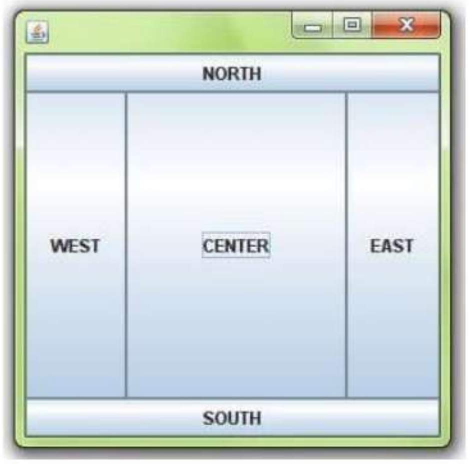
BorderLayout Constructors:

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:

```
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;

public class BorderLayoutExample {
BorderLayoutExample(){
   JFrame f=new JFrame();

   JButton b1=new JButton("NORTH");
   JButton b2=new JButton("SOUTH");
   JButton b3=new JButton("EAST");
   JButton b4=new JButton("WEST");
   JButton b5=new JButton("CENTER");

   f.add(b1,BorderLayout.NORTH);
   f.add(b2,BorderLayout.SOUTH);
   f.add(b3,BorderLayout.EAST);
   f.add(b4,BorderLayout.WEST);
   f.add(b5,BorderLayout.CENTER);
```

Output:



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

Flow Layout

The FlowLayout is used to arrange the components in a line, one after another (in a flow). It arranges components in a line, if no space left remaining components goes to next line. Align property determines alignment of the components as left, right, center etc.

FlowLayout Constructors:

FlowLayout()	creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
FlowLayout(int align)	creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
FlowLayout(int align, inthgap, intvgap)	creates a flow layout with the given alignment and the given horizontal and vertical gap.

Constants for Alignments:

- ***Static int CENTER:*** This value indicates that each row of components should be centered.
- ***Static int LEFT:*** This value indicates that each row of components should be left justified.
- ***Static int RIGHT:*** This value indicates that each row of components should be right-justified.
- ***Static int LEADING:*** This value indicates that each row of components should be justified to the leading edge of the container's orientation.
- ***Static int TRAILING**:* This value indicates that each row of components should be justified to the trailing edge of the container's orientation.

Example1:

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class FlowLayoutExample {
 FlowLayoutExample(){
    JFrame f = new JFrame("Flow
Layout");
      JButton b1 = new JButton("button 1");
      JButton b2 = new JButton("button 2");
      JButton b3 = new JButton("button 3");
      JButton b4 = new JButton("button 4");
      JButton b5 = new JButton("button 5");
      f.add(b1);
      f.add(b2);
      f.add(b3);
```

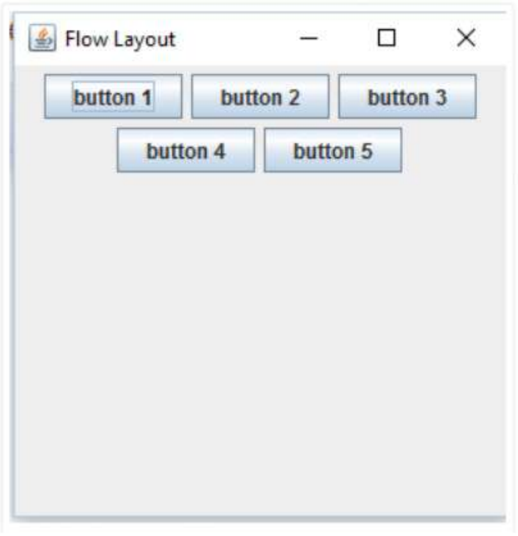
Example2:

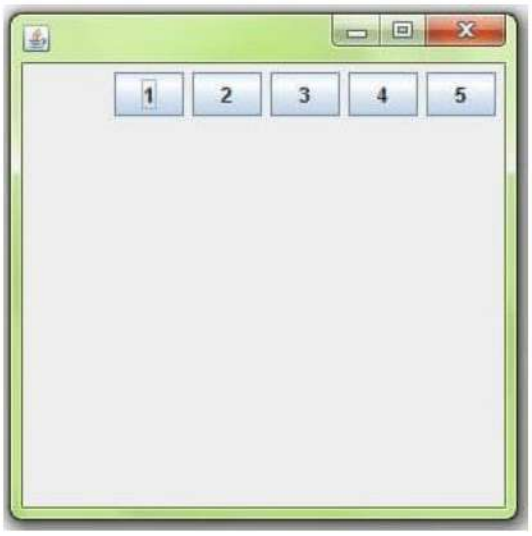
```
import java.awt.*;
import javax.swing.*;
public class MyFlowLayout{
MyFlowLayout(){
   JFrame f=new JFrame();
   JButton b1=new JButton("1");
   JButton b2=new JButton("2");
   JButton b3=new JButton("3");
   JButton b4=new JButton("4");
   JButton b5=new JButton("5");
   f.add(b1);
   f.add(b2);
   f.add(b3);
  f.add(b4);
  f.add(b5);
```

```
        f.add(b4);
        f.add(b5);
        f.setLayout(new FlowLayout());
        f.setSize(300,300);
        f.setVisible(true);
    }
    public static void main(String[] args) {
        new FlowLayoutExample();
    }
}
```

```
    f.setLayout(new
FlowLayout(FlowLayout.RIGHT));

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

Outputs:





Grid Layout

It organizes the components in a two-dimensional grid. It simply creates a collection of equal-sized components and arranges them in the desired number of rows and columns. A cell can only contain one component. All components are of equal size and cover the entire volume of the container. When the container is resized, all cells are resized as well. The order in which the components are placed in a cell is determined as they are added. Components are added from the first row to the last row and from the first column to the last column.

GridLayout Constructors:

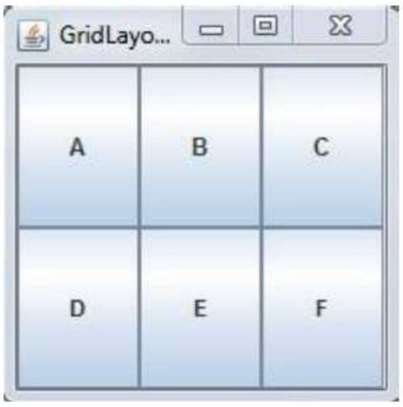
GridLayout()	creates a grid layout with one column per component in a row.
GridLayout(int rows, int cols)	creates a grid layout with the given rows and columns but no gaps between the components.
GridLayout(int rows, int cols, int int hgap, int vgap)	creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

If the value of parameter rows and cols is zero then it indicates an infinite number of rows and columns.

Example:

```
import java.awt.*;
import javax.swing.*;
public class GridDemo
{
    JButton b1, b2, b3, b4, b5, b6;
    GridDemo()
    {
        JFrame f = new JFrame("GridLayoutDemo");
        b1 = new JButton("A");
        b2 = new JButton("B");
        b3 = new JButton("C");
        b4 = new JButton("D");
        b5 = new JButton("E");
        b6 = new JButton("F");
        f.add(b1);
        f.add(b2);
        f.add(b3);
        f.add(b4);
        f.add(b5);
        f.add(b6);
        f.setLayout(new GridLayout(2, 3));
        f.setSize(200, 200);
        f.setVisible(true);
    }
    public static void main(String args[])
    {
        new GridDemo();
    }
}
```

Output:



GridBag Layout

The gridbag layout is a flexible layout manager that aligns components vertically and horizontally, without requiring that the components be of the same size. It maintains a dynamic rectangular grid of cells with each component occupying one or more cells. Each components managed by a grid bag layout is associated with an instance of *GridBagConnstraints* that specifies how the component is arranges within its display area.

Constructor:

GridBagLayout():Creates a grid bag layout manager.

We can customize *GridBagConstrainsts* object by setting one or more of its instance variables:

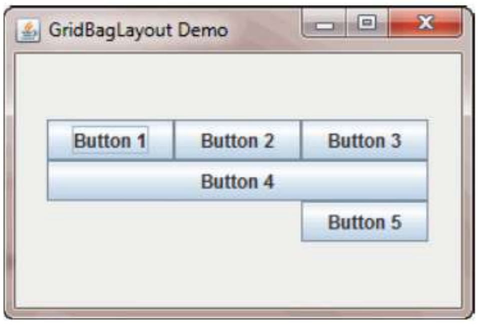
- ***gridx, gridy:*** For specifying grid location by defining x and y coordinate values.
- ***gridwidth ,gridheight:*** Used to specify the number of grids that will span a document. When the component size is greater than the area, fill is used (i.e. VERTICAL or HORIZONTAL).
- ***ipadx, ipady:*** For defining component width and height, i.e., increasing component size.
- ***insets:*** Used to define the component's surrounding space, such as top, left, right, and bottom.
- ***Anchor:*** Used when the component size is less than the area, i.e., where to place in a grid.

Example:

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

      JFrame frame = new JFrame("GridBagLayout Demo");
      JButton btn1 = new JButton("Button 1");
      JButton btn2 = new JButton("Button 2");
      JButton btn3 = new JButton("Button 3");
      JButton btn4 = new JButton("Button 4");
      JButton btn5 = new JButton("Button 5");

      JPanel panel = new JPanel(new GridBagLayout());
      GridBagConstraints cst = new GridBagConstraints();

      // add button 1 to the panel
      cst.fill = GridBagConstraints.HORIZONTAL;
      cst.gridx = 0;
      cst.gridy = 0;
      panel.add(btn1,cst);

      // add button 2 to the panel
      cst.fill = GridBagConstraints.HORIZONTAL;
      cst.gridx = 1;
      cst.gridy = 0;
      panel.add(btn2, cst);

      // add button 3 to the panel
      cst.gridx = 2;
      cst.gridy = 0;
      panel.add(btn3, cst);

       // add button 4 to the panel
      cst.fill = GridBagConstraints.HORIZONTAL;
      cst.gridwidth = 3;
      cst.gridx = 0;
      cst.gridy = 1;
      panel.add(btn4,cst);

      // add button 5 to the panel
      cst.fill = GridBagConstraints.HORIZONTAL;
      cst.gridx = 2;
      cst.gridwidth = 1;
      cst.gridy = 2;      //third row
      panel.add(btn5,cst);

      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.setSize(300,200);
      frame.getContentPane().add(panel);
      frame.setVisible(true);
   }
}
```

We can add components to any container by using following form of *add()* method.
void add (Component c, GridBagConstraints gbc)

Output:



Group Layout

GroupLayout groups its components and places them in a Container hierarchically. The grouping is done by instances of the Group class. For the components, GroupLayout is using two types of arragements:

- ***Sequential Arrangement:*** In this arrangements, the components are arranged one after the other in sequence.
- ***Parallel Arrangement:*** In this arrangement, the components are placed parallel in the same place.

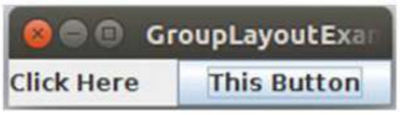
This GroupLayout class provides methods such as *createParallelGroup()* and *createSequentialGroup()* to create groups.

GroupLayout treats each axis independently. That is, there is a group representing the horizontal axis, and a group representing the vertical axis. Each component must exists in both a horizontal and vertical group. We do not need to worry about the vertical dimension when defining the horizontal layout, and vice versa, as the layout along each axis is totally independent of the layout along the other axis. When focusing on just one dimension, we only have to solve half the problem at one time.

Example:

```
import java.awt.Container;
import javax.swing.GroupLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import static javax.swing.GroupLayout.Alignment.*;

public class GroupExample {
  public static void main(String[] args) {
    JFrame frame = new JFrame("GroupLayoutExample");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Container contentPanel = frame.getContentPane();
    GroupLayout groupLayout = new GroupLayout(contentPanel);

    contentPanel.setLayout(groupLayout);

    JLabel clickMe = new JLabel("Click Here");
    JButton button = new JButton("This Button");

    groupLayout.setHorizontalGroup(
            groupLayout.createSequentialGroup()
                    .addComponent(clickMe)
                    .addGap(10, 20, 100)
                    .addComponent(button));
    groupLayout.setVerticalGroup(
             groupLayout.createParallelGroup(GroupLayout.Alignment.BASELINE)
                    .addComponent(clickMe)
                    .addComponent(button));
    frame.pack();
    frame.setVisible(true);
  }
}
```

Output:



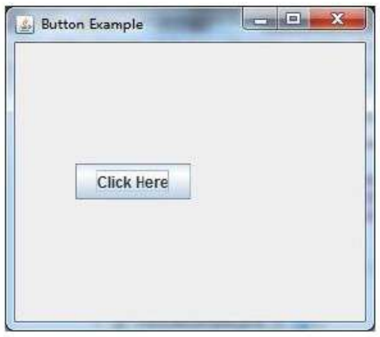
No Layout Managers

Components can be arranged in containers without the use of a layout manager. Items must be manually positioned and arranged in this case. It means that no layout manager is assigned and the components can be put at specific x, y coordinates. 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.

Example:

```
import javax.swing.*;
import java.awt.*;
public class SimpleFrame{
  public SimpleFrame() {
    JFrame f = new JFrame("Button Example");
    JButton b = new JButton("Click Here");
    b.setBounds(50, 100, 95, 30);        //(x, y, width, height)
    f.add(b);
    f.setLayout(null);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setSize(400, 400);
    f.setVisible(true);
}
public static void main(String[] args){
        new SimpleFrame();
   }
}
```

Output:



Swing GUI Controls

JLabel

The object of *JLabel* class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly.

Constructors:

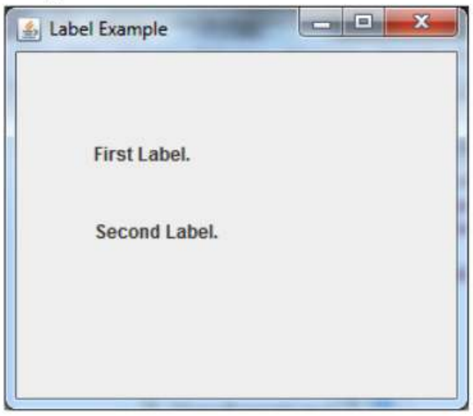
JLabel()	Creates a *JLabel* instance with no image and with an empty string for the title.
JLabel(String s)	Creates a *JLabel* instance with the specified text.
JLabel(Icon i)	Creates a *JLabel* instance with the specified image.
JLabel(String s, Icon i, int horizontalAlignment)	Creates a *JLabel* instance with the specified text, image, and horizontal alignment

Example:

```
import javax.swing.*;
class LabelExample
{
```

```
public static void main(String args[])
 {
  JFrame f= new JFrame("Label Example");
  JLabel l1, l2;
  l1=new JLabel("First Label.");
  l1.setBounds(50,50, 100,30);
  l2=new JLabel("Second Label.");
  l2.setBounds(50,100, 100,30);
  f.add(l1);
  f.add(l2);
  f.setSize(300,300);
  f.setLayout(null);
  f.setVisible(true);
 }
}
```

Output:



JTextField

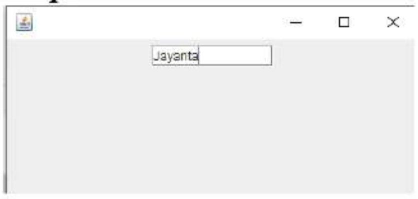
JTextField is used for taking input of single line of text. It is most widely used text component. Here are the constructors of the JTextField class:

JTextField();	Creates a new *TextField*
JTextField(int clos);	creates text field with specified number of columns
JTextField(String str);	Creates a new *TextField* initialized with the specified text.
JTextField(String str, int cols);	Creates a new text field with given text and number of columns.

Example:

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class MyTextField extends JFrame
{
 public MyTextField()
 {
  JTextField jtf = new JTextField(20);
  add(jtf);
  setLayout(new FlowLayout());
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  setSize(400, 400);
  setVisible(true);
 }
 public static void main(String[] args)
 {
  new MyTextField();
 }
}
```

Output:



JTextArea

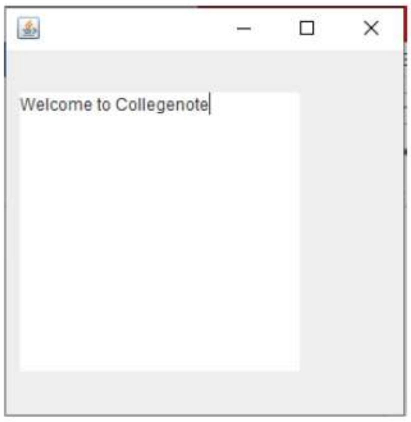
The object of a JTextArea class is a multi-line region that displays text. It allows the editing of multiple line text.

Constructors:

JTextArea ();	Creates a new text area.
JTextArea(String s);	Creates a text area that displays specified text initially.
JTextArea(int row, int column)	Creates a new text area with given rows/columns.
JTextArea(String s, int row, int column)	Creates a text area with the specified number of rows and columns that displays specified text.

Example:

```
import javax.swing.*;
public class TextAreaExample
{
    TextAreaExample(){
      JFrame f= new JFrame();
      JTextArea area=new JTextArea("Welcome to Collegenote");
      area.setBounds(10,30, 200,200);
      f.add(area);
      f.setSize(300,300);
      f.setLayout(null);
      f.setVisible(true);
    }
  public static void main(String args[])
  {
   new TextAreaExample();
  }
}
```


Welcome to Collegenote

JPasswordField

The object of a JPasswordField class is a text component specialized for password entry. It allows the editing of a single line of text. It inherits JTextField class.

Constructors:

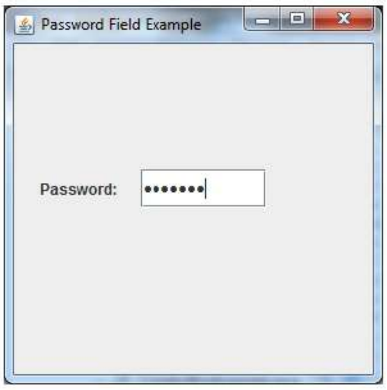
JPasswordField();	Creates a new password field.
JPasswordField(int columns);	Creates a new password field with a given columns.
JPasswordField(String text);	Creates a new password field with a given text.
JPasswordField(String text, int columns);	Creates a new password field with given text and number of columns.

Example:

```
import javax.swing.*;
public class PasswordFieldExample {
   public static void main(String[] args) {
   JFrame f = new JFrame("Password Field Example");
```

```
    JPasswordField value = new JPasswordField();
    JLabel l1 = new JLabel("Password:");
      l1.setBounds(20,100, 80,30);
       value.setBounds(100,100,100,30);
         f.add(value);
         f.add(l1);
         f.setSize(300,300);
         f.setLayout(null);
         f.setVisible(true);
   }
}
```

Output:



JCheckBox

The JCheckBox 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".

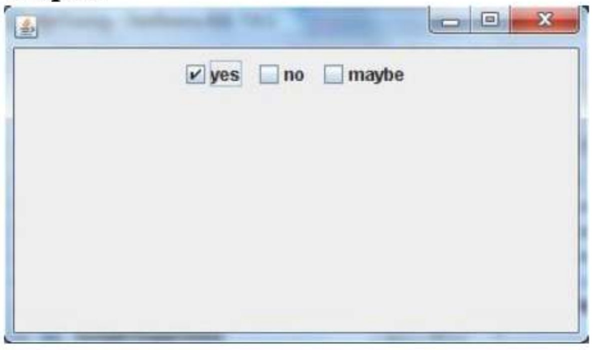
Constructors:

JCheckBox();	Creates an initially unselected check box button with no text, no icon.
JCheckBox(String s);	Creates an initially unselected check box with text.

Example:

```
import javax.swing.*;
import java.awt.*;
public class Test extends JFrame
{
  public Test()
  {
    JCheckBox jcb = new JCheckBox("yes");
    add(jcb);
    jcb = new JCheckBox("no");
    add(jcb);
    jcb = new JCheckBox("maybe");
    add(jcb);
    setLayout(new FlowLayout());
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setSize(400, 400);
    setVisible(true);
  }
  public static void main(String[] args)
  {
    new Test();
  }
}
```

Output:



JRadioButton

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz. It should be added in *ButtonGroup* to select one radio button only.

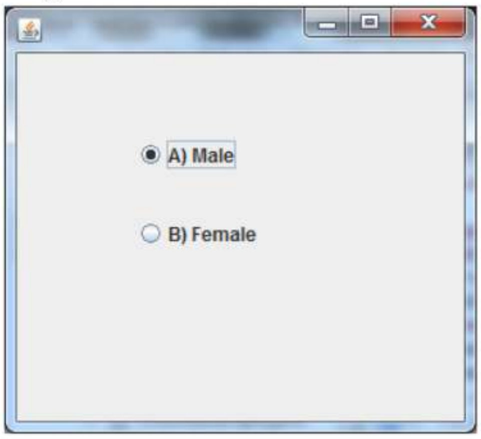
Constructors:

JRadioButton();	Creates an unselected radio button with no text.
JRadioButton(String s);	Creates an unselected radio button with specified text.
JRadioButton(String s, boolean selected);	Creates a radio button with the specified text and selected status.

Example:

```
import javax.swing.*;
public class RadioButtonExample {
JFrame f;
RadioButtonExample(){
 f=new JFrame();
 JRadioButton r1=new JRadioButton("A) Male");
 JRadioButton r2=new JRadioButton("B) Female");
 r1.setBounds(75,50,100,30);
 r2.setBounds(75,100,100,30);
 ButtonGroup bg=new ButtonGroup();
 bg.add(r1);
 bg.add(r2);
 f.add(r1);
 f.add(r2);
 f.setSize(300,300);
 f.setLayout(null);
 f.setVisible(true);
}
public static void main(String[] args) {
   new RadioButtonExample();
 }
}
```

Output:



JComboBox

Combo box is a combination of text fields and drop-down list. *JComboBox* component is used to create a combo box in Swing. Following is the constructor for JComboBox:

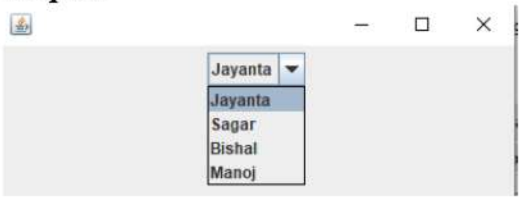
JComboBox(String arr[])

Example:

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;
public class Test extends JFrame
{
 String name[] = { "Jayanta","Sagar","Bishal","Manoj"};  //list of name.
 public Test()
 {
```

```
 JComboBox jc = new JComboBox(name);  //initialzing combo box with list of name.
 add(jc);
 setLayout(new FlowLayout());
 setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 setSize(400, 400);
 setVisible(true);
}
public static void main(String[] args)
{
 new Test();
}
}
```

Output:



JScrollPane

A JscrollPane is used to make scrollable view of a component. When screen size is limited, we use a scroll pane to display a large component or a component whose size can change dynamically.

Some of the constructors defined by JScrollPane class are as follows:

JSrollPane ();	*JScrollPane(Component component);*
JScrollPane(int ver, int hor);	*JScrollPane(Component component, int ver, int hor);*

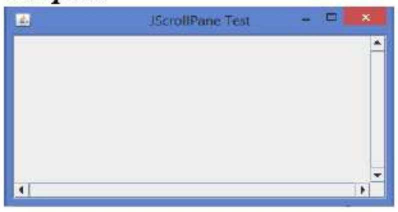
Where,

- *component* is the component to be added to the scroll pane
- *ver* and *hor* specify the policies to display the vertical and horizontal scroll bar, respectively. Some of the standard policies are: *HORIZONTAL_SCROLLBAR_ALWAYS, HORIZONTAL_SCROLLBAR_AS_NEEDED, VERTICAL_SCROLLBAR_ALWAYS, VERTICAL_SCROLLBAR_AS_NEEDED*

Example:

```
import javax.swing.*;
import java.awt.*;
public class JScrollPaneTest extends JFrame {
  JScrollPaneTest() {
    setTitle("JScrollPane Test");
    JPanel panel = new JPanel();
    panel.setLayout(new BorderLayout());
    JScrollPane jsp = new JScrollPane(panel,
       ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS,
       ScrollPaneConstants.HORIZONTAL_SCROLLBAR_ALWAYS);
    add(jsp);
    setSize(350, 275);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
    setVisible(true);
  }
  public static void main(String[] args) {
    new JScrollPaneTest();
  }
}
```

Output:



JSlider

The Java JSlider class is used to create the slider. By using JSlider, a user can select a value from a specific range. JSlider is a component that lets the users select a value by sliding a knob within a specified interval.

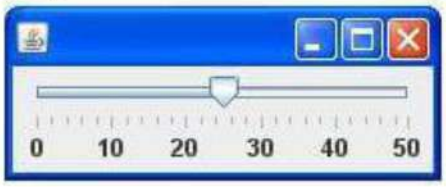
Constructors:

JSlider()	Creates a slider with the initial value of 50 and range of 0 to 100.
JSlider(int orientation)	Creates a slider with the specified orientation set by either JSlider.HORIZONTAL or JSlider.VERTICAL with the range 0 to 100 and initial value 50.
JSlider(int min, int max)	Creates a horizontal slider using the given min and max.
JSlider(int min, int max, int value)	Creates a horizontal slider using the given min, max and value.
JSlider(int orientation, int min, int max, int value)	Creates a slider using the given orientation, min, max and value.

Example:

```
import javax.swing.*;
public class SliderExample extends JFrame
{
 public SliderExample ()
 {
  JSlider slider = new JSlider (JSlider.HORIZONTAL, 0, 50, 25);

  slider.setMinorTickSpacing (2);      // set the minor tick spacing to the slider.
  slider.setMajorTickSpacing (10);     // set the major tick spacing to the slider.
  slider.setPaintTicks (true);         // determine whether tick marks are painted.
  slider.setPaintLabels (true);        // determine whether labels are painted.

  JPanel panel = new JPanel ();
  panel.add (slider);
  add (panel);
 }
 public static void main (String s[])
 {
  SliderExample frame = new SliderExample ();
  frame.pack ();
  frame.setVisible (true);
 }
}
```

Output:



An alternative to the ***pack()*** method is to establish a frame size explicitly by calling the ***setSize()*** or ***setBounds()*** methods.

Borders

Every Swing component inherits a border property from *JComponent*, so we can call *setBorder()* to specify a Border object for a Swing component. This Border object displays some kind of decoration around the outside of the component. The *javax.swing.border* package contains this Border interface and a number of useful implementations of it.

Different borders defined in *javax.swing.border* package are:

- ***Bevel Border:*** This border draws raised or lowered beveled edges, giving an illusion of depth. Following statements use Bevel border:

  For Raised Bevel Border:
  pane.setBorder(BorderFactory.createRaisedBevelBorder()); //pane is object of JPanel

  For Lowered Bevel Border:
  pane.setBorder(BorderFactory.createLoweredBevelBorder());

- ***Line Border:*** This border draws a line, with a color and thickness you specify, around the component.
  pane.setBorder(BorderFactory.createLineBorder());

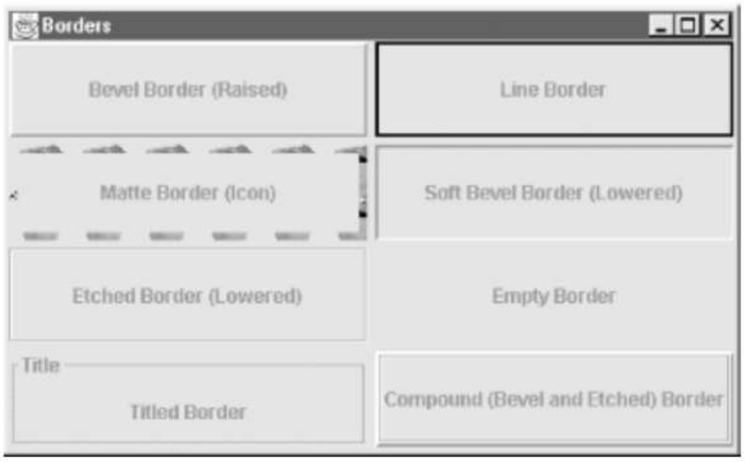
- ***Etched Border:*** This border draws a line around the component, using a 3D effect that makes the line appear etched into or raised out of the surrounding container.
  pane.setBorder(BorderFactory.createEtchedBorder());

- ***Titled Border:*** A regular border with a title. *TitledBorder* doesn't actually draw a border; it just draws a title in conjunction with another border object. We can specify the locations of the title, its justification, and its font.
  pane.setBorder(BorderFactory.createTitledBorder("Border Title"));

- ***Matte Border:*** This border draws the border using a solid color or a tiled image. We specify the border dimensions for all four sides.
  ImageIcon icon = new ImageIcon("images/icon.gif");
  pane.setBorder(BorderFactory.createMatteBorder(-1, -1, -1, -1, icon));

- ***Compound Border:*** This border combines two other Border types to create a compound border.

- ***Empty Border:*** A border with no appearance. This is a useful way to place an empty margin around a component.



Example

Program to create Raised Bevel Border:

```
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.border.Border;

public class ABevelBorder {
  public static void main(String args[]) {
    JFrame frame = new JFrame("Bevel Borders");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    Border raisedBorder = BorderFactory.createRaisedBevelBorder();

    JButton button = new JButton("Raised");
    button.setBorder(raisedBorder);
    frame.add(button);

    frame.setSize(300, 100);
    frame.setVisible(true);
  }
}
```

Swing Menu

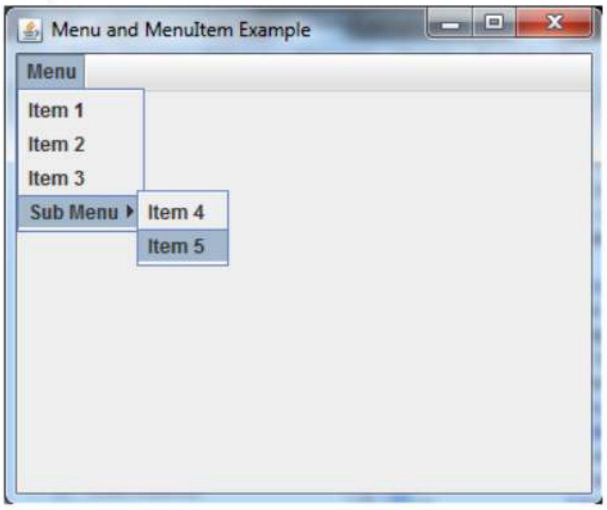
JMenuBar, JMenu and JMenuItem

- The *JMenuBar* class is used to display menubar on the window or frame. It may have several menus.
- The object of *JMenu* class is a pull down menu component which is displayed from the menu bar. It inherits the *JMenuItem* class.
- The object of *JMenuItem* class adds a simple labeled menu item. The items used in a menu must belong to the *JMenuItem* or any of its subclass.

Example:

```
import javax.swing.*;
class MenuExample
{
        JMenu menu, submenu;
        JMenuItem i1, i2, i3, i4, i5;

        MenuExample(){
        JFrame f= new JFrame("Menu and MenuItem Example");
        JMenuBar mb=new JMenuBar();
        menu=new JMenu("Menu");
        submenu=new JMenu("Sub Menu");
        i1=new JMenuItem("Item 1");
        i2=new JMenuItem("Item 2");
        i3=new JMenuItem("Item 3");
        i4=new JMenuItem("Item 4");
        i5=new JMenuItem("Item 5");
        menu.add(i1);
        menu.add(i2);
        menu.add(i3);
        submenu.add(i4);
        submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setJMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[])
{
   new MenuExample();
}
}
```

Output:



Icons in Menu Items

Icons in menu items can be added as:

//Create Image Icon

ImageIcon myIcon = new ImageIcon("E:\\temp\\myIcon.png");

//Create Menu Item

JMenuItem itm1 = new JMenuItem("Title", myIcon);

Check Box and Radio Buttons Menu Items

The ***CheckboxMenuItem*** class represents a check box that can be found in a menu. Selecting a check box in the menu toggles the status of the control from on to off or off to on. Some of the constructor to create a check box which can be included in a menu:

- *JCheckboxMenuItem()*
- *JCheckBoxMenuItem(String text)*
- *JCheckboxMenuItem(Icon icn):*
- *JCheckBoxMenuItem(String text, boolean state)*
- *JCheckBoxMenuItem(String text,Icon icn)*
- *JCheckBoxMenuItem(String text,Icon icn,boolean state)*

The ***JRadioButtonMenuItem*** class represents a Radio Button which can be included in a menu. Selecting the Radio Button in the menu changes control's state from on to off or from off to on. It can be created by using following constructors:

- *JRadioButtonMenuItem()*
- *JRadioButtonMenuItem(String text)*
- *JRadioButtonMenuItem(Icon icn)*
- *JRadioButtonMenuItem(String text,boolean state)*
- *JRadioButtonMenuItem(Icon icn,boolean state)*
- *JRadioButtonMenuItem(String text,Icon icn)*
- *JRadioButtonMenuItem(String text,Icon icn,boolean state)*

Pop-up Menus

JPopupMenu is used for creating popups dynamically on a specified position.

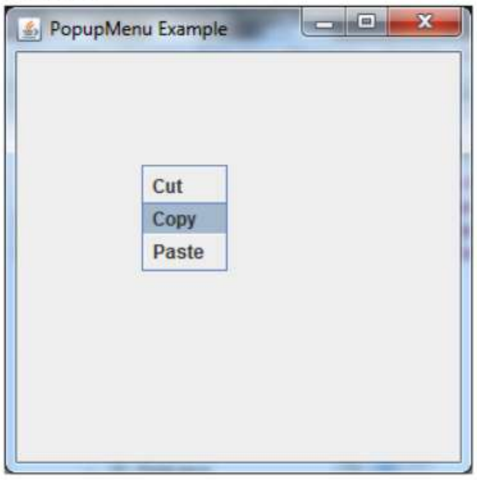
Constructors:

- *JPopupMenu():* creates a popup menu with empty name.
- *JPopupMenu(String name):* creates a popup menu with specified title.

Example:

```
import javax.swing.*;
import java.awt.event.*;
class PopupMenuExample
{
   PopupMenuExample(){
      final JFrame f= new JFrame("PopupMenu Example");
      final JPopupMenu popupmenu = new JPopupMenu("Edit");
      JMenuItem cut = new JMenuItem("Cut");
      JMenuItem copy = new JMenuItem("Copy");
      JMenuItem paste = new JMenuItem("Paste");
      popupmenu.add(cut);
      popupmenu.add(copy);
      popupmenu.add(paste);
      f.addMouseListener(new MouseAdapter() {
         public void mouseClicked(MouseEvent e) {
            popupmenu.show(f , e.getX(), e.getY());
         }
      });
      f.add(popupmenu);
      f.setSize(300,300);
      f.setLayout(null);
      f.setVisible(true);
   }
public static void main(String args[])
{
      new PopupMenuExample();
}
}
```

Output:



Keyboard Mnemonics and Accelerators

In real applications, a menu usually includes support for keyboard shortcuts because they give an experienced user the ability to select menu items rapidly. Keyboard shortcuts come in two forms: *mnemonics* and *accelerators*.

- A ***mnemonic*** is a key-press that opens a *JMenu* or selects a *MenuItem* when the menu is opened.
- An ***accelerator*** is a key-press that selects an option within the menu without it ever being open.

There are two ways to set the mnemonic for *JMenuItem*. First, it can be specified when an object is constructed using this constructor:

JMenuItem(String name, int mnem)

In this case, the name is passed in *name* and the mnemonic is passed in *mnen*. Second, you can set the mnemonic by calling *setMnemonic()*. To specify a mnemonic for *JMenu* , you must call *setMnemonic()*. This method is inherited by both classes from *AbstractButton* and is shown next:

void setMnemonic(int mnem)

Here, *mnem* specifies the mnemonic. It should be one of the constants defined in *java.awt.event.KeyEvent* , such as *KeyEvent.VK_F* or *KeyEvent.VK_Z*. Mnemonics are not case sensitive, so in the case of *VK_A*, typing either a or A will work. By default, the first matching letter in the menu item will be underscored. To underscore a letter other than the first match, specify the index of the letter as an argument to *setDisplayedMnemonicIndex()*, which is inherited by both *JMenu* and *JMenuItem* from *AbstractButton*. It is shown here:

void setDisplayedMnemonicIndex(int idx)

The index of the letter to underscore is specified by *idx*.

An accelerator can be associated with a *JMenuItem* object. It is specified by calling *setAccelerator( )*, shown next:

void setAccelerator(KeyStroke ks)

Here, *ks* is the key combination that is pressed to select the menu item. *KeyStroke* is a class that contains several factory methods that construct various types of keystroke accelerators. The following are three examples:

static KeyStroke getKeyStroke(char ch)

static KeyStroke getKeyStroke(Character ch, int modifier)

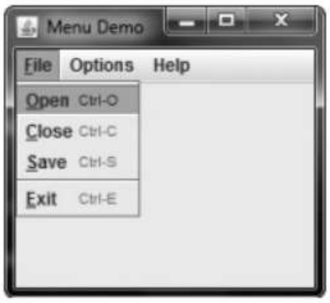
static KeyStroke getKeyStroke(int ch, int modifier)

Here, *ch* specifies the accelerator character. In the first version, the character is specified as a *char* value. In the second, it is specified as an object of type *Character*. In the third, it is a value of type *KeyEvent*, previously described. The value of *modifier* must be one or more of the following constants, defined in the *java.awt.event.InputEvent* class:

InputEvent.ALT_DOWN_MASK
InputEvent.CTRL_DOWN_MASK
InputEvent.SHIFT_DOWN_MASK
InputEvent.ALT_GRAPH_DOWN_MASK
InputEvent.META_DOWN_MASK

Therefore, if you pass ***VK_A*** for the key character and ***InputEvent.CTRL_DOWN_MASK*** for the modifier, the accelerator key combination is ctrl-a.

Example:



```
JMenu jmFile = new JMenu("File");
jmFile.setMnemonic(KeyEvent.VK_F);
JMenuItem jmiOpen = new JMenuItem("Open", KeyEvent.VK_O);
jmiOpen.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
InputEvent.CTRL_DOWN_MASK));
JMenuItem jmiClose = new JMenuItem("Close", KeyEvent.VK_C);
jmiClose.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_C,
InputEvent.CTRL_DOWN_MASK));
JMenuItem jmiSave = new JMenuItem("Save", KeyEvent.VK_S);
jmiSave.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_S,
InputEvent.CTRL_DOWN_MASK));
JMenuItem jmiExit = new JMenuItem("Exit", KeyEvent.VK_E);
jmiExit.setAccelerator( KeyStroke.getKeyStroke(KeyEvent.VK_E,
InputEvent.CTRL_DOWN_MASK));
```

Enabling and Disabling Menu Items

Since a *JMenuItem* is a *JComponent*, we can call ***menuitem.setEnabled(false)*** to disable a menu item and ***menuitem.setEnabled(true)*** to enable a menu item. A menu item that is disabled will appear "grayed out," and the user will not be able to select it, either with the mouse or with an accelerator. It's always a good idea to give the user visual feedback about the state of a program. Disabling a menu item when it doesn't make any sense to select it is one good way of doing this.

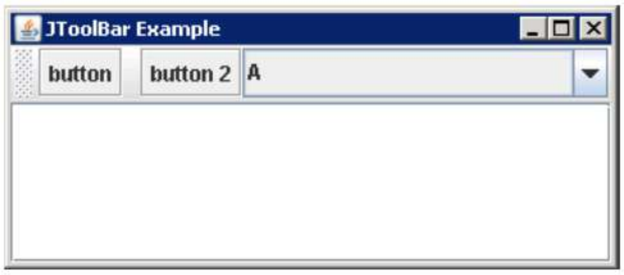
Swing ToolBar

In order to create a toolbar in Java Swing, we use ***JToolBar*** class. The *JToolBar* is a group of commonly used components such as buttons or drop down menu. *JToolBar* provides a component which is useful for displaying commonly used actions or controls. Constructors of the class are:

JToolBar()	Creates a new toolbar with default horizontal orientation.
JToolBar(int orientation)	Creates a new toolbar with a given orientation.
JToolBar(String name)	Creates a new toolbar with a given name.
JToolBar(String name, int orientation)	Creates a new toolbar with a given name and orientation.

Example:

```
import java.awt.BorderLayout;
import java.awt.Container;
import javax.swing.JButton;
import javax.swing.JComboBox;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JToolBar;

public class ToolBarSample {
  public static void main(final String args[]) {
    JFrame frame = new JFrame("JToolBar Example");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JToolBar toolbar = new JToolBar();
    toolbar.setRollover(true);
    JButton button = new JButton("button");
    toolbar.add(button);
    toolbar.addSeparator();
    toolbar.add(new JButton("button 2"));
    toolbar.add(new JComboBox(new String[]{"A","B","C"}));
    Container contentPane = frame.getContentPane();
    contentPane.add(toolbar, BorderLayout.NORTH);
    JTextArea textArea = new JTextArea();
    JScrollPane pane = new JScrollPane(textArea);
    contentPane.add(pane, BorderLayout.CENTER);
    frame.setSize(350, 150);
    frame.setVisible(true);
  }
}
```



Methods:

- ***addSeparator():*** It appends a separator of default size to the end of the tool bar.
- ***setRollover(boolean b):*** Sets the rollover state of this toolbar to *boolean b*. If the rollover state is true then the border of the toolbar buttons will be drawn only when the mouse pointer hovers over them. The default value of this property is false.

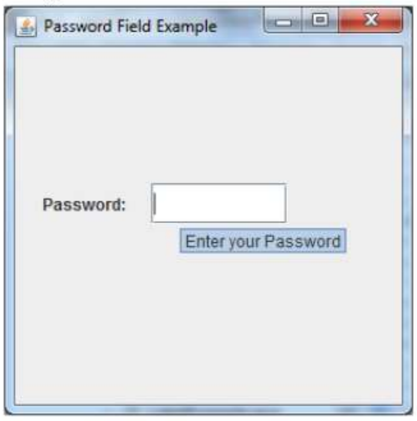
Swing ToolTips

We can add tooltip text to almost all the components of Java Swing by using the following method ***setToolTipText(String s)***. This method sets the tooltip of the component to the specified *string s*. When the cursor enters the boundary of that component a popup appears and text is displayed.

Example:

```
import javax.swing.*;
public class ToolTipExample {
   public static void main(String[] args) {
    JFrame f=new JFrame("Password Field Example");
    JPasswordField value = new JPasswordField();
    value.setBounds(100,100,100,30);
    value.setToolTipText("Enter your Password");
    JLabel l1=new JLabel("Password:");
    l1.setBounds(20,100, 80,30);
    f.add(value);
    f.add(l1);
    f.setSize(300,300);
    f.setLayout(null);
    f.setVisible(true);
}
}
```

Output:



Dialog Boxes

A Dialog window is an independent sub window meant to carry temporary notice apart from the main Swing Application Window. Most Dialogs present an error message or warning to a user, but Dialogs can present images, directory trees, or just about anything compatible with the main Swing Application that manages them.

Using JOptionPane

By using *JOptionPane* class we can create different types of dialog boxes as below:

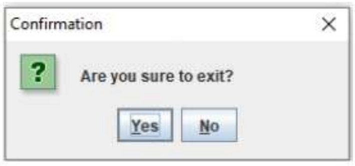
- ***Confirm Dialog:*** This dialog box asks a question that requires confirmation and receives yes/no/cancel response. We can create confirm dialog box by calling JOptionPanes *showConfirmDialog()* method.

  JOptionPane.showConfirmDialog(parent_frame, msg, title, msg_options);

  Message options can be any one of following: DEFAULT_OPTION, YES_NO_OPTION, YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION.

  Example:

  JOptionPane.showConfirmDialog (null, "Are you sure to exit?", "Confirmation", JOptionPane.YES_NO_OPTION);



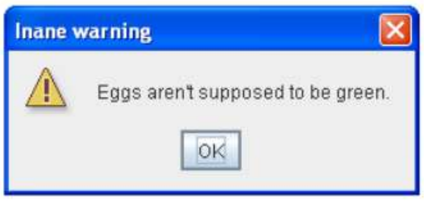
- ***Message Dialog:*** This dialog box relays a message to the user. We can create message dialog box by calling JOptionPanes *showMessageDialog()* method.

  JOptionPane.showMessageDialog(parent_frame, msg, title, msg_types, icon);

  Message types can be any one of following: ERROR_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE.

Example:

JOptionPane.showMessageDialog(null, "Eggs are not supposed to be green.", "Inane warning", JOptionPane.WARNING_MESSAGE);

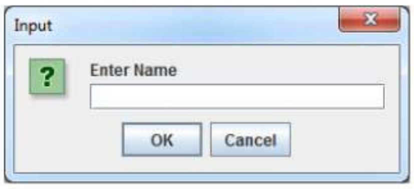


- ***Input Dialog:*** This dialog box prompts a user for input. We can create input dialog box by calling JOptionPanes *showInputDialog()* method.

  JOptionPane.showInputDialog(parent_frame, msg, title);

  Example:

  JOptionPane.showInputDialog(null, "Enter Name", "Input");



- ***Option Dialog:*** It is a combination of the three other methods. We can create option dialog box by calling JOptionPanes *showOptionDialog()* method.

  JOptionPane.showOptionDialog(parent_frame, msg, title, option_types, msg_type, icon, options, default_option);

```
// Addition program that uses JOptionPane for input and output.
import javax.swing.JOptionPane;
public class Addition{
 public static void main( String[] args )
 {
  // obtain user input from JOptionPane input dialogs
  String firstNumber = JOptionPane.showInputDialog( "Enter first integer" );
  String secondNumber = JOptionPane.showInputDialog( "Enter second integer" );

  // convert String inputs to int values for use in a calculation
  int number1 = Integer.parseInt( firstNumber );
  int number2 = Integer.parseInt( secondNumber );
  int sum = number1 + number2;

  // display result in a JOptionPane message dialog
  JOptionPane.showMessageDialog( null, "The sum is " + sum, "Sum of Two Integers",
JOptionPane.PLAIN_MESSAGE );
 }
}
```

Using JDialog

With *JOptionPane* class, it is easy to create dialog boxes, but to create custom dialog boxes we need to use *JDialog* class. *JDialog* can be customized according to user need.

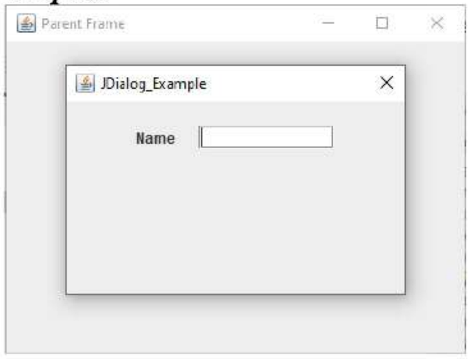
Constructor of the class are:

JDialog()
JDialog(parent_frame)
JDialog(parent_frame, modal)
JDialog(parent_frame, title)
JDialog(parent_frame, title, modal)

Example:

```
import java.awt.FlowLayout;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class JDialogExample extends JFrame {
  JDialogExample(){
    JLabel l = new JLabel("Name");
    JTextField tf = new JTextField(10);
    JDialog jd = new JDialog(this, "JDialog_Example", true);
    setSize(400, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
    setTitle("Parent Frame");
    jd.setLayout(new FlowLayout(FlowLayout.CENTER, 20, 20));
    jd.setSize(300, 200);
    jd.setLocation(50, 50);
    jd.add(l);
    jd.add(tf);
    jd.setVisible(true);
  }
  public static void main(String args[]){
    JDialogExample f = new JDialogExample();
  }
}
```

Output:



File Choosers

File choosers provide a GUI for navigating the file system, and then either choosing a file or directory from a list, or entering the name of a file or directory. To display a file chooser, we usually use the *JFileChooser* API to show a modal dialog containing the file chooser.

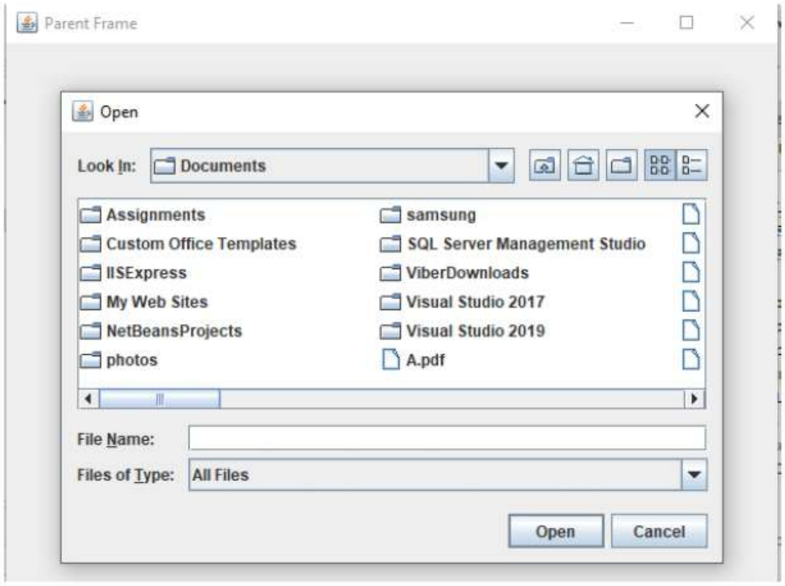
Constructors:

JFileChooser()	Constructs a *JFileChooser* pointing to the user's default directory.
JFileChooser(File currentDirectory)	Constructs a *JFileChooser* using the given File as the path.
JFileChooser(String currentDirectoryPath)	Constructs a JFileChooser using the given path.

Example:

```
import java.awt.FlowLayout;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
public class FileChooser_Example extends JFrame {
  FileChooser_Example(){
      JFileChooser jf = new  JFileChooser();
      setSize(500, 300);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setTitle("Parent Frame");
      setVisible(true);
      jf.showOpenDialog(this);
  }
   public static void main(String[] args)
  {
   new FileChooser_Example();
  }
}
```

Output:



Color Choosers

The *JColorChooser* class is used to create a color chooser dialog box so that user can select any color.

Constructors of the class:

JColorChooser()	It is used to create a color chooser panel with white color initially.
JColorChooser(color initialcolor)	It is used to create a color chooser panel with the specified color initially.

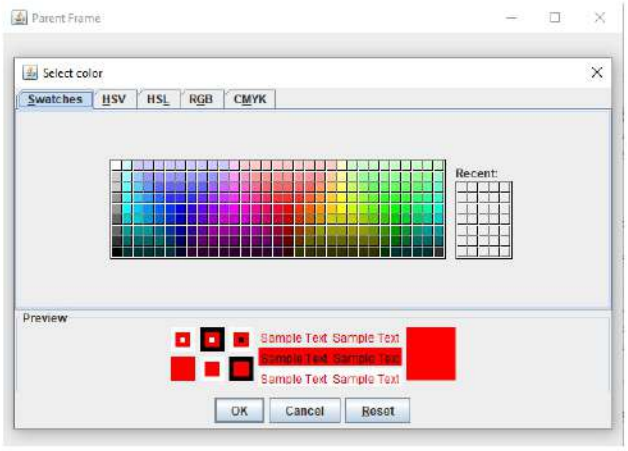
Commonly used Method:

static Color showDialog(Component c, String title, Color initialColor): It is used to show the color chooser dialog box.

Example:

```
import java.awt.Color;
import java.awt.FlowLayout;
import javax.swing.JColorChooser;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class ColorChooser_Example extends JFrame {
   ColorChooser_Example (){
      JColorChooser cc = new  JColorChooser();
      setSize(650, 500);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setTitle("Parent Frame");
      JPanel p = new JPanel();
      add(p);
      setVisible(true);
      Color c = cc.showDialog(this, "Select color", Color.red);
      p.setBackground(c);
   }
    public static void main(String[] args)
  {
   new ColorChooser_Example ();
  }
}
```

Output:



Internal Frames

An internal frame is similar to a regular frame which can be displayed within another window. It is a lightweight component which can be resized, closed, maximized or minimized depending upon the properties being set. It can also hold a title and a menu bar. With the *JInternalFrame* class we can display internal frames. Usually, we add internal frames to a desktop pane. The desktop pane, in turn, is added into *JFrame*. The desktop pane is an instance of *JDesktopPane*.

Following important points should be remember when creating internal frames:

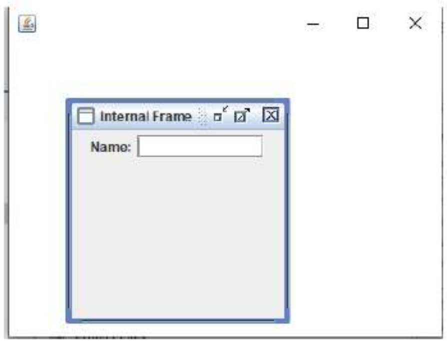
- Size of the internal frame must be set. Default size of internal frame is zero, so it will not display anything. To set the size of internal frame following methods can be used: *setSize(), pack(), setBounds()*.
- Internal frame must be added to container. If it is not added to the container it will be not appear. Generally, the internal frame is added to the *JDesktopPane*.
- Like, the JFrame to show or make visible the internal frame following method must be invoked *setVisible(true)* or *show()*.

Some of the constructors defined by *JInternalFrame* class are as follows:
JinternalFrame()
JinternalFrame(String title)
JinternalFrame(String title, boolean resizable)
JinternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)

Example:

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

public class IFExample extends JFrame {
  IFExample(){
    JDesktopPane dp = new JDesktopPane();
    JInternalFrame jif = new JInternalFrame("Internal Frame", true, true, true, true);
    JTextField tf = new JTextField(10);
    JLabel lb = new JLabel("Name:");
    setSize(400, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
    jif.setSize(200, 200);
    jif.setLocation(50,50);
    dp.add(jif);
    add(dp);
    jif.setLayout(new FlowLayout(FlowLayout.CENTER));
    jif.add(lb);
    jif.add(tf);
    jif.setVisible(true);
  }
  public static void main(String[] args){
    new IFExample();
  }
}
```

Output:



Advanced Swing Components

List

The object of JList class represents a list of text items. This list is used to show the items in a list format and get user input from the list of items. The user can choose either one item or multiple items.

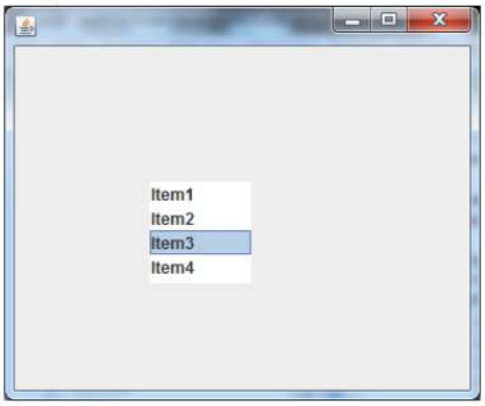
Constructors of JList are:

JList()	Creates a JList with an empty, read-only, model.
JList(ary[] listData)	Creates a JList that displays the elements in the specified array.
JList(ListModel<ary> dataModel)	Creates a JList that displays elements from the specified, non-null, model.

Example:

```
import javax.swing.*;
public class ListExample
{
    ListExample(){
      JFrame f= new JFrame();
      DefaultListModel<String> l1 = new DefaultListModel<>();
        l1.addElement("Item1");
        l1.addElement("Item2");
        l1.addElement("Item3");
        l1.addElement("Item4");
        JList<String> list = new JList<>(l1);
        list.setBounds(100,100, 75,75);
        f.add(list);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
public static void main(String args[])
   {
  new ListExample();
   }
}
```

Output:



Trees

JTree is a Swing component with which we can display hierarchical data. *JTree* is quite a complex component. A *JTree* has a 'root node' which is the top-most parent for all nodes in the tree. A node is an item in a tree. A node can have many children nodes. These children nodes themselves can have further children nodes. If a node doesn't have any children node, it is called a leaf node.

The node is represented in Swing API as *TreeNode* which is an interface. The interface *MutableTreeNode* extends this interface which represents a mutable node. Swing API provides an implementation of this interface in the form of *DefaultMutableTreeNode* class.

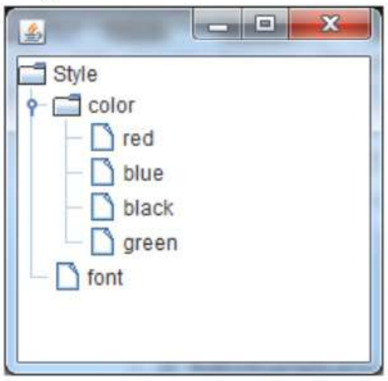
We will be using the *DefaultMutableTreeNode* class to represent our node. This class is provided in the Swing API and we can use it to represent our nodes. This class has a handy *add()* method which takes in an instance of *MutableTreeNode*.

So, we will first create the root node. And then we can recursively add nodes to that root.

Example:

```
import javax.swing.*;
import javax.swing.tree.DefaultMutableTreeNode;
public class TreeExample {
JFrame f;
TreeExample(){
  f=new JFrame();
  DefaultMutableTreeNode style=new DefaultMutableTreeNode("Style");
  DefaultMutableTreeNode color=new DefaultMutableTreeNode("color");
  DefaultMutableTreeNode font=new DefaultMutableTreeNode("font");
  style.add(color);
  style.add(font);
  DefaultMutableTreeNode red=new DefaultMutableTreeNode("red");
  DefaultMutableTreeNode blue=new DefaultMutableTreeNode("blue");
  DefaultMutableTreeNode black=new DefaultMutableTreeNode("black");
  DefaultMutableTreeNode green=new DefaultMutableTreeNode("green");
  color.add(red);
  color.add(blue);
  color.add(black);
  color.add(green);

  // create the tree by passing in the root node
  JTree jt=new JTree(style);

  f.add(jt);
  f.setSize(200,200);
  f.setVisible(true);
}
public static void main(String[] args) {
  new TreeExample();
 }
}
```

Output:



Tables

The *JTable* class is used to display data in tabular form. It is composed of rows and columns.

Constructors:

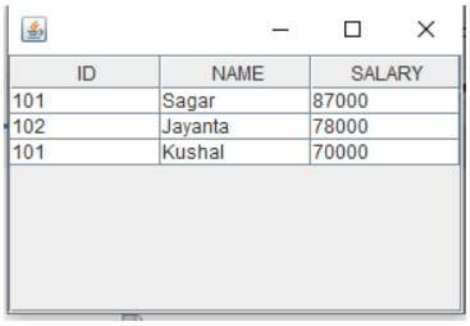
JTable()	Creates a table with empty cells
JTable(Object[][] rows, Object[] columns)	Creates a table with the specified data.

Example:

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
```

```
public class TableExample {
   TableExample(){
   JFrame f=new JFrame();
   String data[][]={ {"101","Sagar","87000"},
                   {"102","Jayanta","78000"},
                   {"101","Kushal","70000"}};
   String column[]={"ID","NAME","SALARY"};
   JTable jt=new JTable(data,column);
   jt.setBounds(30,40,200,300);
   JScrollPane sp=new JScrollPane(jt);
   f.add(sp);
   f.setSize(300,200);
   f.setVisible(true);
}
public static void main(String[] args) {
   new TableExample();
  }
}
```

Output:



ID	NAME	SALARY
101	Sagar	87000
102	Jayanta	78000
101	Kushal	70000