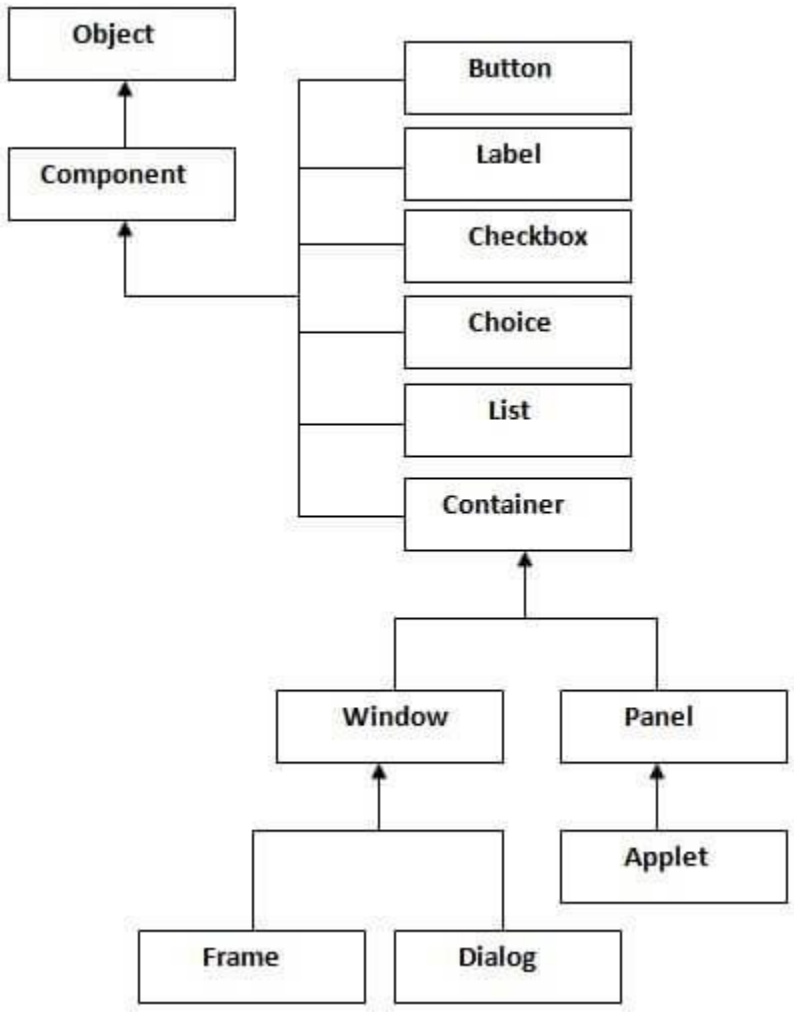
Chapter – 1
Java Application

AWT and Swing both are used to create GUI interface in Java.

Java AWT

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

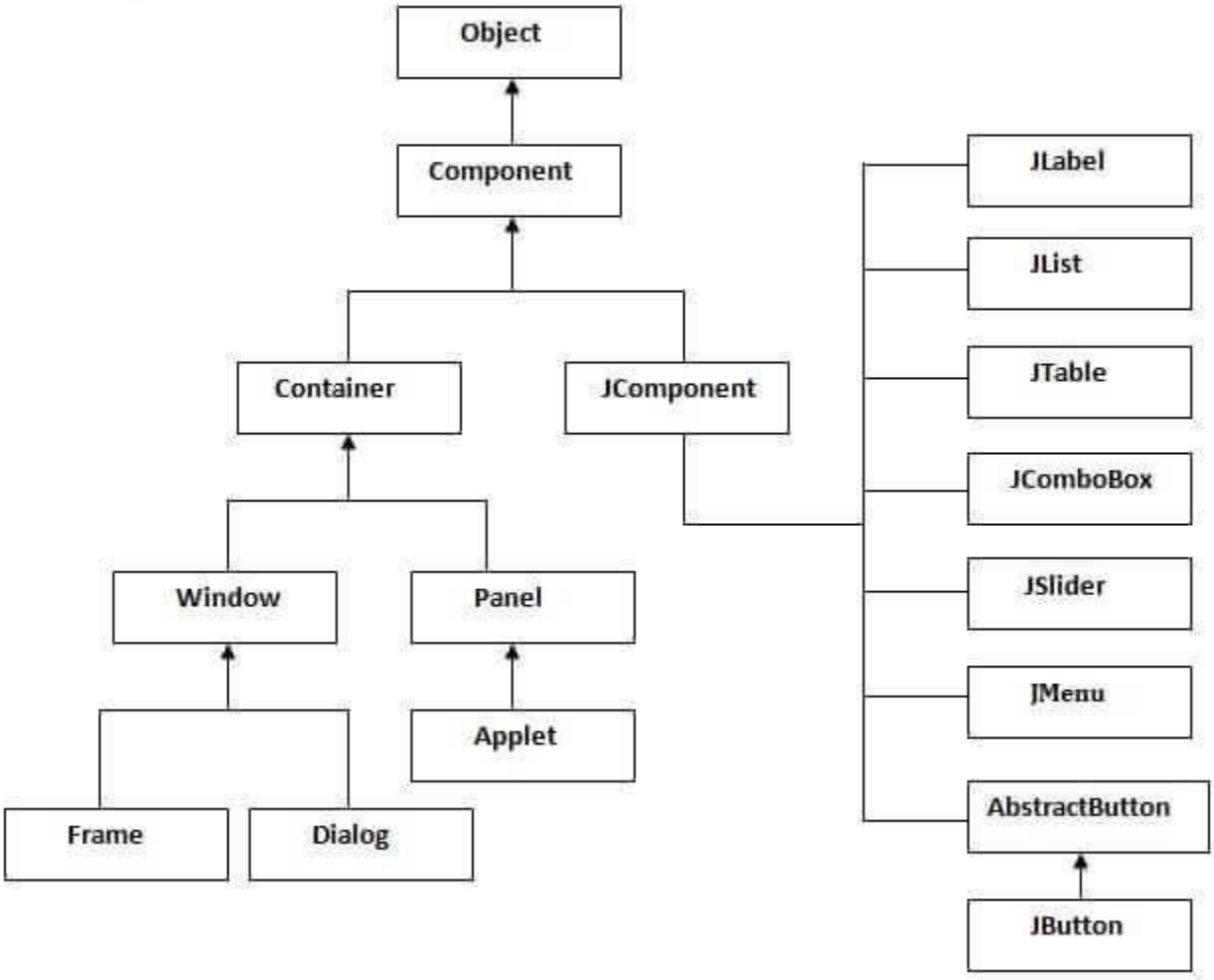
Java AWT hierarchy



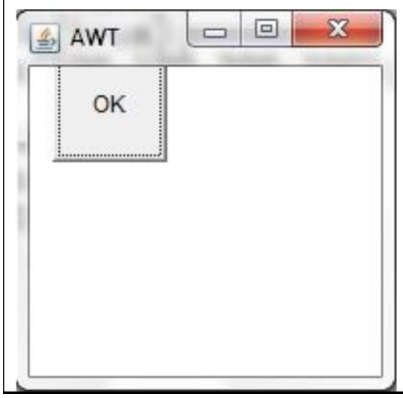
Java Swing

Java Swing is a part of java Foundation Classes (JFC) that is used to create window – based applications. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java. Unlike AWT, java Swing provides platform – independent and lightweight components. The java,swing package provides classes foe java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Java Swing Hierarchy



Difference between java AWT and Swing

AWT	Swing
AWT stands for Abstract Window Toolkit.	Swing is a part of Java Foundation Class (JFC).
AWT components are heavy weight.	Swing components are light weight.
AWT components are platform dependent so there look and feel changes according to OS.	Swing components are platform independent so there look and feel remains constant.
AWT components are not very good in look and feel as compared to Swing components. See the button in below image, its look is not good as button created using Swing. 	Swing components are better in look and feel as compared to AWT. See the button in below image, its look is better than button created using AWT. 

JFrame

- Whenever you create a graphical user interface with java swing functionality, you will need a container for your application. In this case of swing, this container is called a JFrame. All GUI applications requires a JFrame. In fact, some Applets even use a JFrame. Why?
- You can't build a house without a foundation. The same is true in java: without a container in which to put all other elements, you won't have a GUI application. In other words, the JFrame is required as the foundation or base container for all other graphical components.
- Java Swing applications can be run on any system that supports Java. These applications are lightweight. This means that don't take up much space or use many system resources.
- JFrame is a class in java and has its own methods and constructors. Methods are functions that impact the JFrame, such as setting the size or visibility. Constructors are run when the instance is created: One constructor can create a blank JFrame, while another can create it with a default title.
- When a new JFrame is created, you actually create an instance of the JFrame class. You can create an empty one, or one with a title. If you pass a string into the constructor, a title is created as follows

  JFrame f = new JFrame();

  Or overload the constructor and give it a title.

JFrame f1 = new JFrame("its me Nishanta");

Jpanel in Swing

The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class. It doesn't have title bar

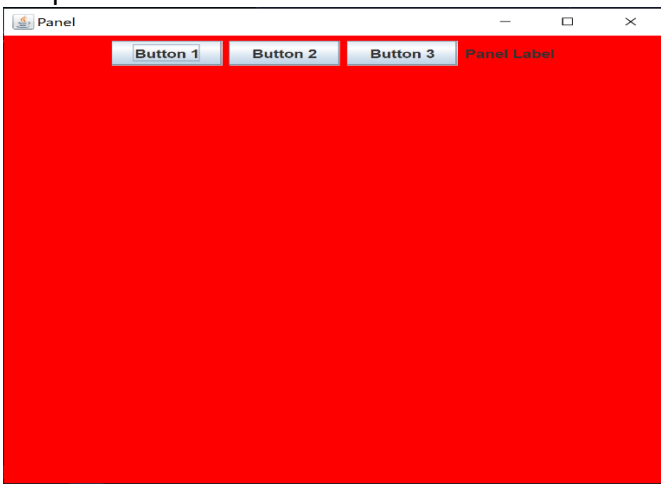
JFrame f= new JFrame("Panel Example");
JPanel panel=new JPanel();
JLabel lbl=new JLabel("Testing label");
panel.add(lbl);
f.add(panel);

Example:

```
import java.awt.*;
import javax.swing.*;
class PanelDemo extends JFrame
{
   static JFrame f;
   static JButton b1,b2,b3;
   static JLabel l;
   public static void main(String args[])
```

```
  {
    f = new JFrame("Panel");
    l = new JLabel("Panel Label");
    b1 = new JButton("Button 1");
    b2 = new JButton("Button 2");
    b3 = new JButton("Button 3");
    JPanel p = new JPanel();
    p.add(b1);
    p.add(b2);
    p.add(b3);

    p.add(l);
    p.setBackground(Color.red);
    f.add(p);
    f.setSize(500, 500);
    f.show();
  }
}
```

Output:



Swing Components and Containers

A component is an independent visual control. Swing Framework contains a large set of components which provide rich functionalities and allow high level of customization. They all are derived from JComponent class. All these components are lightweight components. This class provides some common functionlity like pluggable look and feel, support for accessibility, drag and drop, layout etc.

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

1. **Top level Containers**
   - It inherits Component and Container of AWT.
   - It cannot be contained within other container.
   - Heavyweight

- Example: JFrame, JDialog, JApplet

2. **Lightweight Container**
   - It inherits JComponent class.
   - It is a general purpose container.
   - It can be used to organize related components together.
   - Example: JPanel

JPanel

JPanel, a part of java Swing package is a container that can store a group of components. The main task of JPanel is to organize components; various layouts can be set in JPanel which provides better organization of components and however it does not have a title bar.
Constructor of JPanel are:

- JPanel(): Creates a new panel with flow layout.
- JPanel(LayoutManager l): Creates a new JPanel with specified layoutManager.
- JPanel(boolean isDoubleBuffered): Creates a new JPanel with a specified buffering strategy.
- JPanel(LayoutManager l, boolean isDoubleBuffered): Creates a new JPanel with specified layoutManager and a specified buffereng strategy.

Commonly used function:

- Add(Component c): adds component to a specified container.
- SetLayout(LayoutManager l): Sets the layout of the container to specified layout manager.
- UpdateUI(): resets the UI property with a value from the current look and feel.
- SetUI(PanelUI ui): sets the look and feel object that renders this component.
- GetUI(): returns the look and feel object that renders this component.
- ParamString(): returns a string representation of this JPanel.
- GetUIClassID(): returns the name of the look and feel class that renders this component.
- GetAccessibleContext(): gets the AccessibleContext associated with this JPanel.

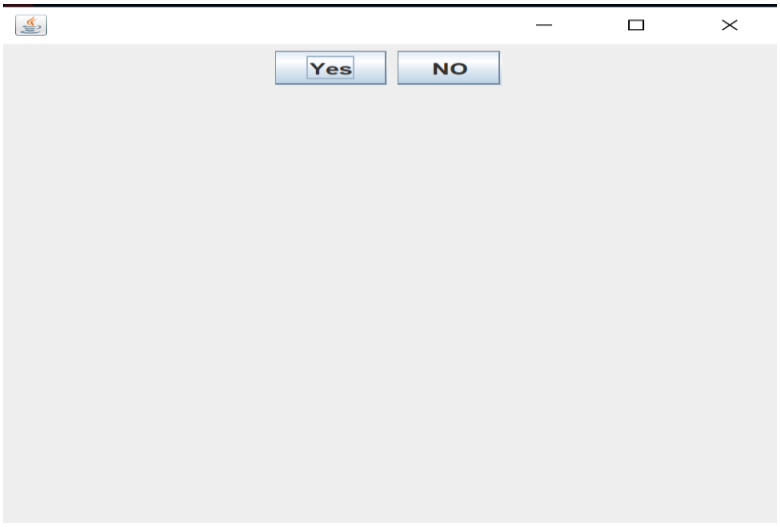
JButton:
JButton is in implementation of a push button. It is used to trigger an action if the user clicks on it. JButton can display a text, an icon or both.
JButton bt = new jButton("Click Me");

Example

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

```
class MyButtonFrame extends JFrame
{
   MyButtonFrame() // Use the class name for the constructor
   {
      JButton bt1 = new JButton("Yes"); // Creating yes button
      JButton bt2 = new JButton("NO");  // Creating No button
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setLayout(new FlowLayout());
      setSize(400, 400);  // Setting size of JFrame
      add(bt1);      // adding yes button to frame
      add(bt2);      // adding No button to frame
      setVisible(true);
   }

   public static void main(String args[])
   {
      {
         new MyButtonFrame(); // Create an instance of your class
      }
   }
}
```

Output



JLabel

Label is a simple component for displaying text, images or both. It doesnot react to input events.

JLabel lb = new Label("This is a label");

JTextField

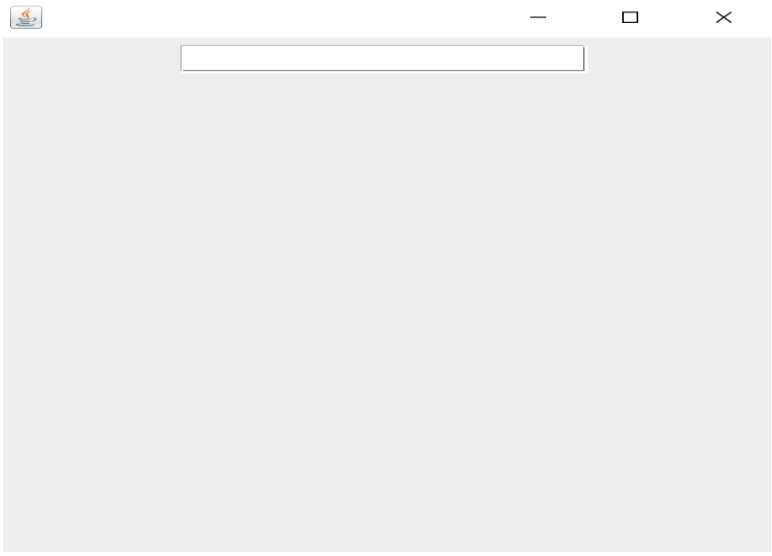
JTextField is a text component that allows editing of a single line of non – formatted text.

JTextField txt = new JTextField();

Example:

```
import java.awt.*;
import javax.swing.*;
class MyTextField extends JFrame
{
   public MyTextField() // Use the class name for the constructor
   {
      JTextField tf = new JTextField(20); // creating JTextField
      add(tf);
      setLayout(new FlowLayout());
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(400, 400);
      setVisible(true);
   }

   public static void main(String args[])
   {
      {
         new MyTextField(); // Create an instance of your class
      };
   }
}
```

Output:



JRadio Button

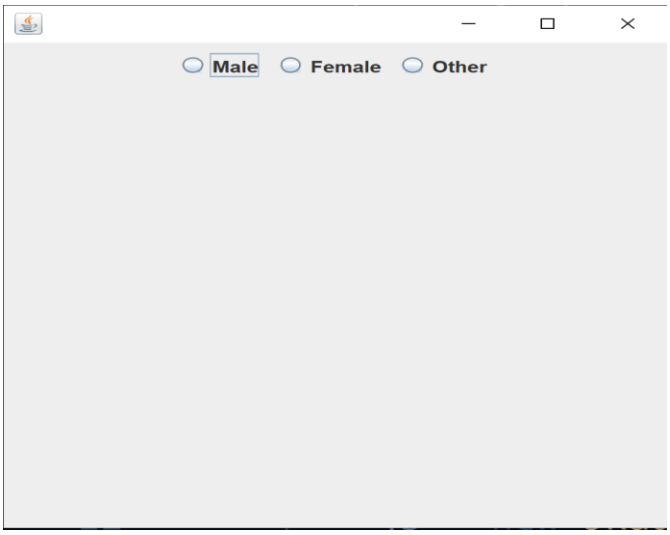
JRadioButton allows the user to select a single exclusive choice from a group of options. It is used with the ButtonGroup Component.

```
JRadioButton rb1 = new JRadioButton("Male");
JRadioButton rb2 = new JRadioButton("Female");
ButtonGroup bg = new ButtonGroup();
bg.add(rb1);
bg.add(rb2);
```

Example:

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

class MyRadioButton extends JFrame
{
    MyRadioButton() // Use the class name for the constructor
    {
        JRadioButton jb = new JRadioButton("Male"); // Creating JRadioButton
        add(jb); //adding JRadioButton to frame
        jb = new JRadioButton("Female");    //creating JRadioButton
        add(jb); //adding JRadioButton to frame
        jb = new JRadioButton("Other"); //creating JRadioButton
        add(jb);  //adding JRadioButton to frame
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400,400);
        setVisible(true);
    }

    public static void main(String args[])
    {
            new MyRadioButton(); // Create an instance of your class
    }
}
```

Output:



JComboBox

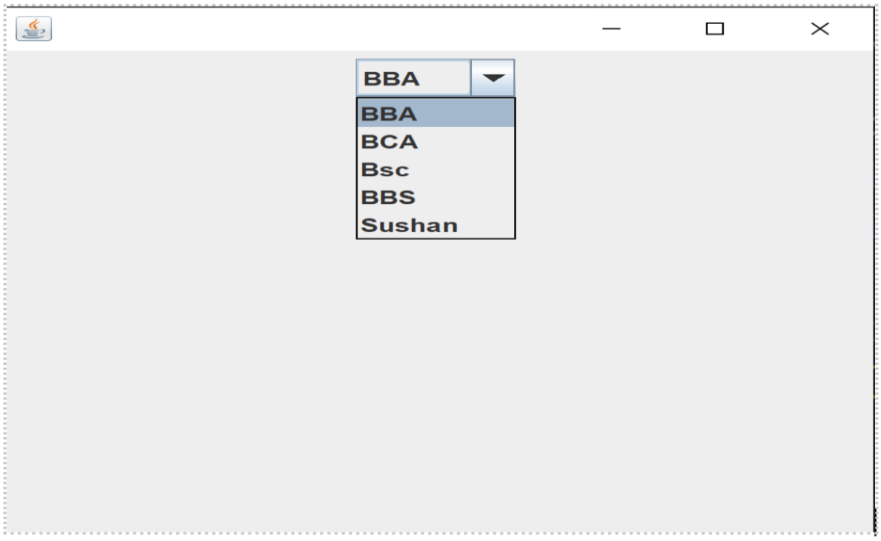
JComboBox is a component that combiles a button or editable ield and drip – down list. The user can select a value from the drop – down list, which appears at the user's request. If you make the combo box editable, then the combo box includes an editable filed into which the user can type a value.

String arr[] = {"BCA", "BBA", "BBS", "BSC"};
JComboBox cb = new JComboBox(arr);
Or
JComboBox<String> cb = new JComboBox<String>();
cb.addItem("BCA");
cb.addItem("BBA);
cb.addItem("BBS");
cb.addItem("BSC");

Example

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

class MyComboBox extends JFrame
{
    String faculty[] = {"BBA", "BCA", "Bsc", "BBS","Sushan"};
    MyComboBox()
    {
        JComboBox cb = new JComboBox<>(faculty);
        add(cb); // adding JComboBox to frame
        setLayout(new FlowLayout());
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(width:400,height:400);
        setVisible(b:true);
    }
    Run | Debug
    public static void main(String args[])
    {
        {
            new MyComboBox(); // Create an instance of your class
        }
    }
}
```

Example



JList:

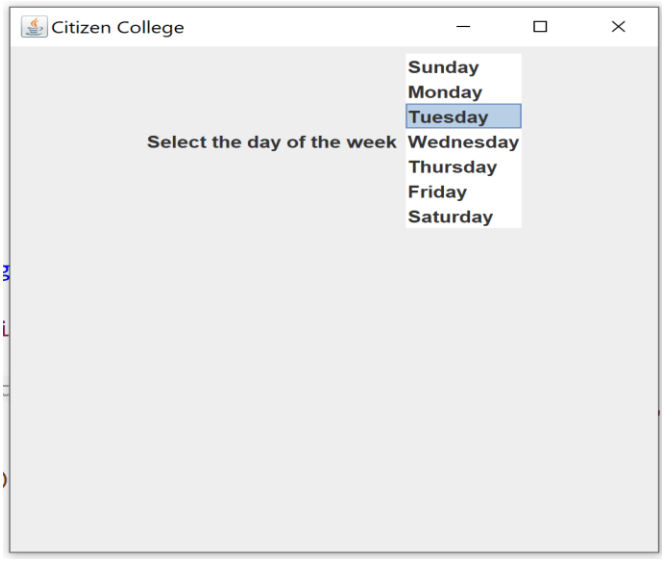
JList is a component that displays a list of objects. It allows the user to select one or more items.

String arr[] = {"BCA", "BBA", "BBS", "BSC"};

JList cb = new JList(arr);

```
1   import java.awt.*;
2   import javax.swing.*;
3
4   class MyJlist extends JFrame
5   {
6       static JFrame f;
7       static JList l;
8
        Run | Debug
9       public static void main(String args[])
10      {
11          f = new JFrame(title:"Citizen College");
12          JPanel p = new JPanel();
13          JLabel lb = new JLabel(text:"Select the day of the week");
14          String week[] = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"};
15          l = new JList<>(week);
16          l.setSelectedIndex(index:2);
17          p.add(lb);
18          p.add(l);
19          f.add(p);
20          f.setSize(width:400,height:400);
21          f.setVisible(b:true);
22      }
23  }
```

Output



JTextArea

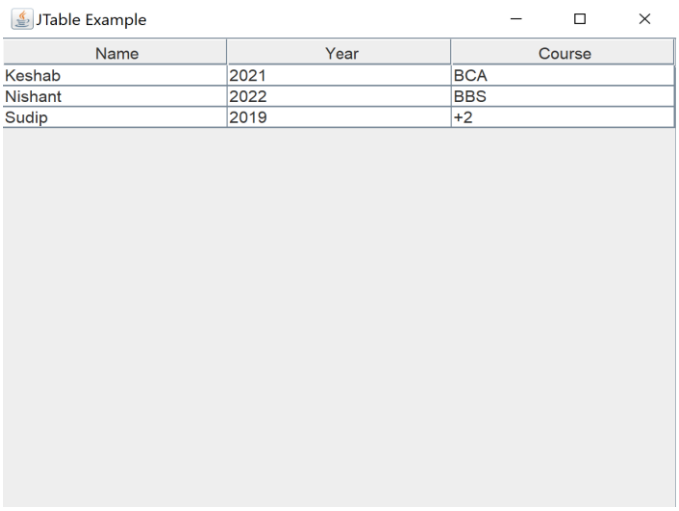
A JTextArea is a multiline text area that displays plain text. It is lightweight component for working with text. The component does not handle scrolling. For this task, we use JScrollPane component.

JTextArea txt = new JTextArea();

JTable

The JTable class is a part of java Swing Package and is generally used to display or edit two dimensional data that is having both rows and columns. It is similar to a spreadsheet. This arranges data in a tabular form.

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

class Jtable1
{
    JFrame f;
    JTable j;
    String[] columnNames = {"Name", "Year", "Course"};

    Jtable1()
    {
        f = new JFrame();
        f.setTitle(title:"JTable Example");
        String data[][] = {{"Keshab", "2021", "BCA"}, {"Nishant", "2022", "BBS"}, {"Sudip", "2019", "+2"}};
        j = new JTable(data, columnNames);
        JScrollPane sp = new JScrollPane(j);
        f.add(sp);
        f.setSize(width:500, height:400);
        f.setVisible(b:true);
    }

    Run | Debug
    public static void main(String args[])
    {
        new Jtable1();
    }
}
```

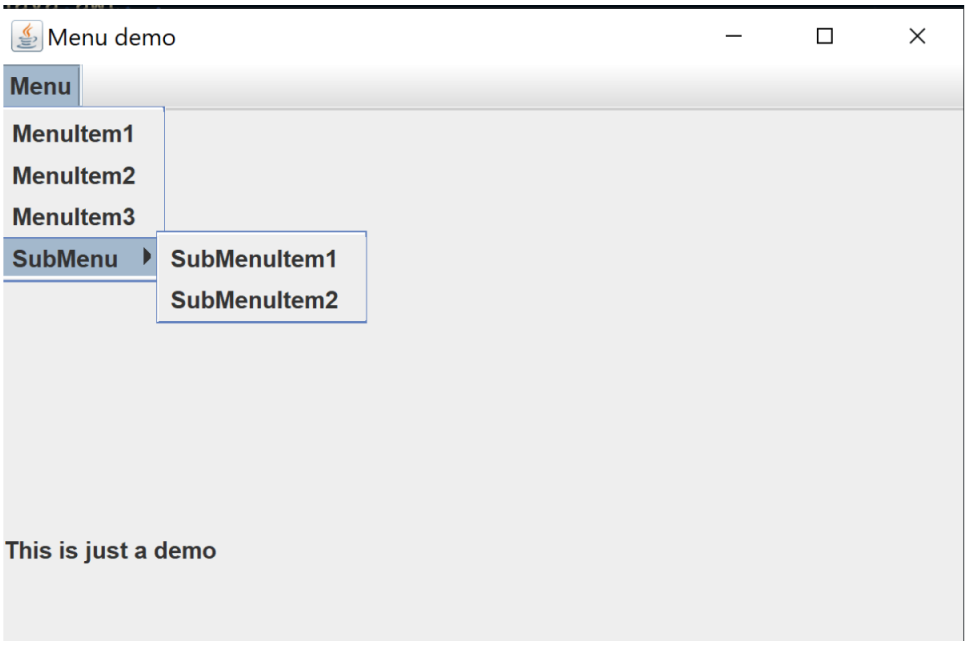


JMenu

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 item used in a menu must belong to the JMenuItem or any of its subclass.

```
JMenuBar mb = new JMenuBar();
JMenu menu1 = new JMenu("Courses");
JMenuItem item1 = new JMenuItem("BCA");
JMenuItem item2 = new JMenuItem("BBA");
menu1.add(item1);
menu1.add(item2);
mb.add(menu1);
```

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

class MenuDemo extends JFrame
{
    static JFrame f;
    static JLabel l;
    static JMenuBar mb;
    static JMenu x1, x2;
    static JMenuItem m1, m2, m3, s1, s2;

    Run | Debug
    public static void main(String args[]) {
        f = new JFrame(title:"Menu demo");
        l = new JLabel(text:" This is just a demo");
        mb = new JMenuBar();
        x1 = new JMenu(s:"Menu");
        x2 = new JMenu(s:"SubMenu");
        m1 = new JMenuItem(text:"MenuItem1");
        m2 = new JMenuItem(text:"MenuItem2");
        m3 = new JMenuItem(text:"MenuItem3");
        s1 = new JMenuItem(text:"SubMenuItem1");
        s2 = new JMenuItem(text:"SubMenuItem2");

        x1.add(m1);
        x1.add(m2);
        x1.add(m3);

        x2.add(s1);
        x2.add(s2);

        x1.add(x2);

        mb.add(x1);

        f.setJMenuBar(mb);
        f.add(l);
        f.setSize(width:500, height:500);
        f.setVisible(b:true);
    }
}
```



Dialog Boxes in swing

Dialog windows or dialogs are an indispensable part of most modern GUI applications. A dialog is defined as a conversation between two or more persons. In a computer application a dialog is a window which is used to "talk" to the application. A dialog is used to input data, modify data, change the application settings etc. Dialogs are important means of communication between a user and computer program.
In java Swing, we can create two kinds of dialogs: standard dialogs and custom dialogs. Custom dialogs are created by programmers. They are based on the JDialog class. Standard dialogs are predefined dialogs available in the Swing toolkit, for example the JColorChooser or the JFileChooser. These are dialogs for common programming tasks like showing text, receiving input, loading and saving files. They save programmer's time and enhance using some standard behaviour.

Modal and Modeless Dialog
There are two basic types of dialogs: Modal and Modeless.
Modal dialogs block input to other top – level windows. Modeless dialogs allow input to other windows. An open file dialog is a good example of a modal dialog. While choosing a file to open, no other operation should be permitted.
A typical modeless dialog is a find text dialog. It is handy to have the ability to move the cursor in the text control and define, where to start the finding of the particular text.

Standard Dialog Example – Message Box
Message dialogs are simple dialogs that provide information to the user. Message dialogs are created with the JOptionPane.showMessageDialog() method.

JOptionPane.showMessageDialog(null,"This is a test message");

Custom Dialog Creation

JDialog jd = new JDialog();
Jd.setTitle("This is a test message");
JLabel lb = new JLabel("Do you want to exit ? ");
Jd.add(lb);
JButton yes = new JButton("Yes");
Jd.add(yes);

```
import javax.swing.*;
class SimpleDialog
{
    Run | Debug
    public static void main(String args[])
    {
        JFrame frame = new JFrame(title:"Main Window");
        JOptionPane.showMessageDialog(frame, message:"Message for the dialog box goes here.",title:"Error", JOptionPane.ERROR_MESSAGE);
        frame.setSize(width:350,height:350);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(b:true);
    }
}
```

Layout Management

FlowLayout

The FlowLayout arranges the components in a directional flow, either from left to right or from right to left. Normally all components are set to one row, according to the order of different components, if all components cannot be fit into one row, it will start a new row and fit the rest in.

To construct a FlowLayout, three options could be chosen:

- **FlowLayout():** Construct a new FlowLayout object with center alignment and horizontal and vertical gap to be default size of 5 pizel.

- **FlowLayout(int align):** Construct similar object with different settings on alignment.
- **FLowLayout(int align, int hgap, int vgap):** Construct similar object with different settings on alignment and gaps between components.

```java
import javax.swing.*;
import java.awt.*;
public class FlowLayoutDemo
{
    JFrame f1;
    JButton b1,b2,b3,b4,b5,b6,b7,b8;
    FlowLayoutDemo()
    {
        f1 = new JFrame();
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setSize(width:400,height:400);
        f1.setLayout(new FlowLayout(align:0,hgap:15,vgap:15));

        b1 = new JButton(text:"Button 1");
        b2 = new JButton(text:"Button 2");
        b3 = new JButton(text:"Button 3");
        b4 = new JButton(text:"Button 4");
        b5 = new JButton(text:"Button 5");
        b6 = new JButton(text:"Button 6");
        b7 = new JButton(text:"Button 7");
       // b8 = new JButton("Button 8");

        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.setVisible(b:true);
    }
    Run | Debug
    public static void main(String args[])
    {
        new FlowLayoutDemo();
    }
}
```

Output

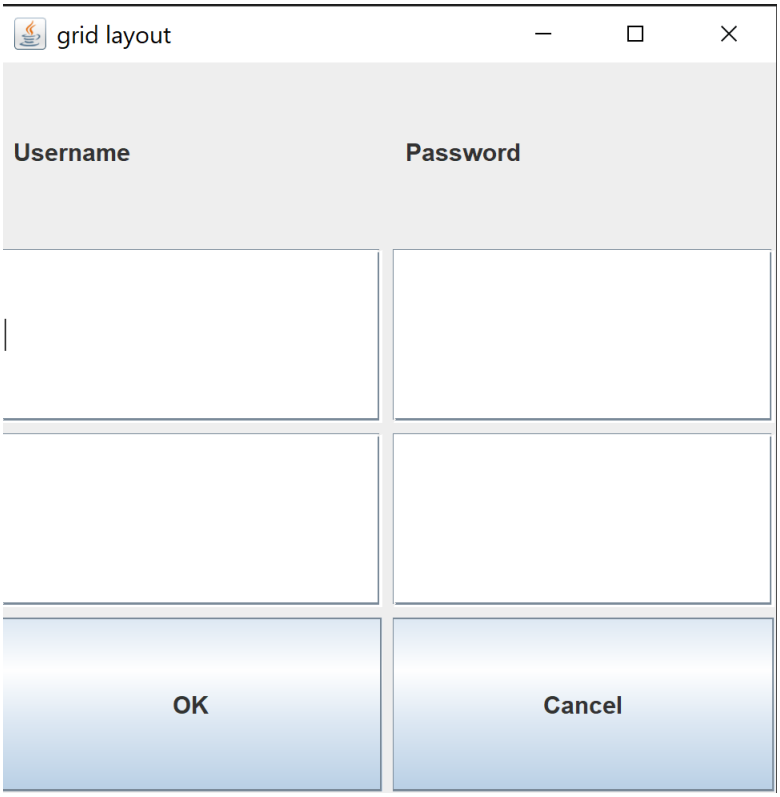
GridLayout

The GridLayout manager is used to lay out the components in a rectangle grid, which has been divided into equal – sized rectangles and one component is placed in each rectangle.

It can constructed with following methods

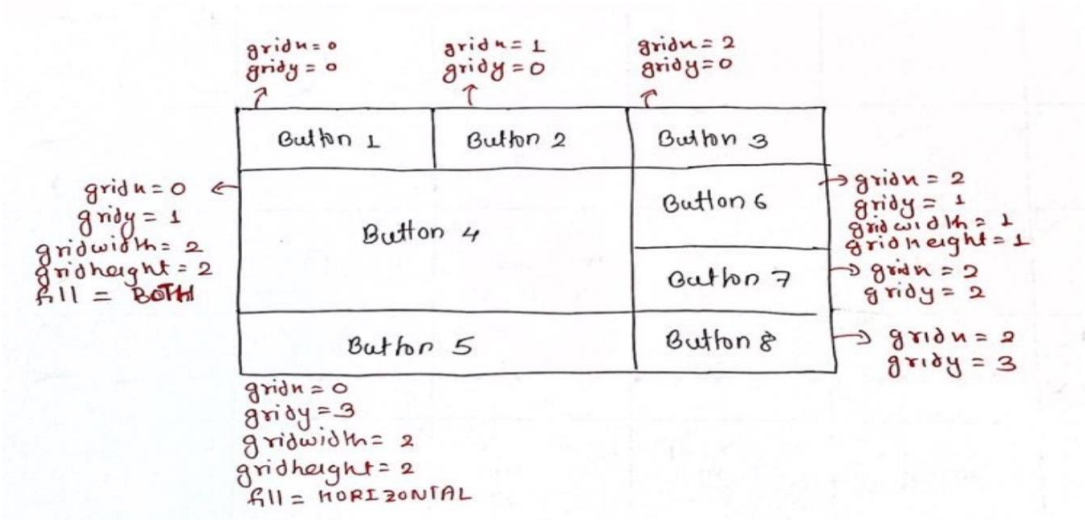
- **GridLayout():** Construct a grid layout with one column per component in a single row.
- **GridLayout(int row, int col):** Construct a grid layout with specified numbers of rows and columns.
- **GridLayout(int row, int col, int hgap, int vgap):** Construct a grid layout with specified rows, columns and gaps between components.

```java
import javax.swing.*;
import java.awt.*;
public class GridLayoutDemo
{
    JFrame f1;
    JLabel l1,l2;
    JButton b1,b2;
    JTextField t1,t2;
    JPasswordField p1,p2;
    GridLayoutDemo()
    {
        f1 = new JFrame(title:" grid layout");
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setSize(width:400,height:400);
        f1.setLayout(new GridLayout(rows:4,cols:2,hgap:5,vgap:5));

        l1 = new JLabel(text:"  Username");
        l2 = new JLabel(text:"  Password");
        b1 = new JButton(text:"OK");
        b2 = new JButton(text:"Cancel");
        t1 = new JTextField();
        t2 = new JTextField();
        p1 = new JPasswordField();
        p2 = new JPasswordField();

        f1.add(l1);
        f1.add(l2);
        f1.add(t1);
        f1.add(p1);
        f1.add(t2);
        f1.add(p2);
        f1.add(b1);
        f1.add(b2);


        f1.setVisible(b:true);
    }
    Run | Debug
    public static void main(String args[])
    {
        new GridLayoutDemo();
    }
}
```



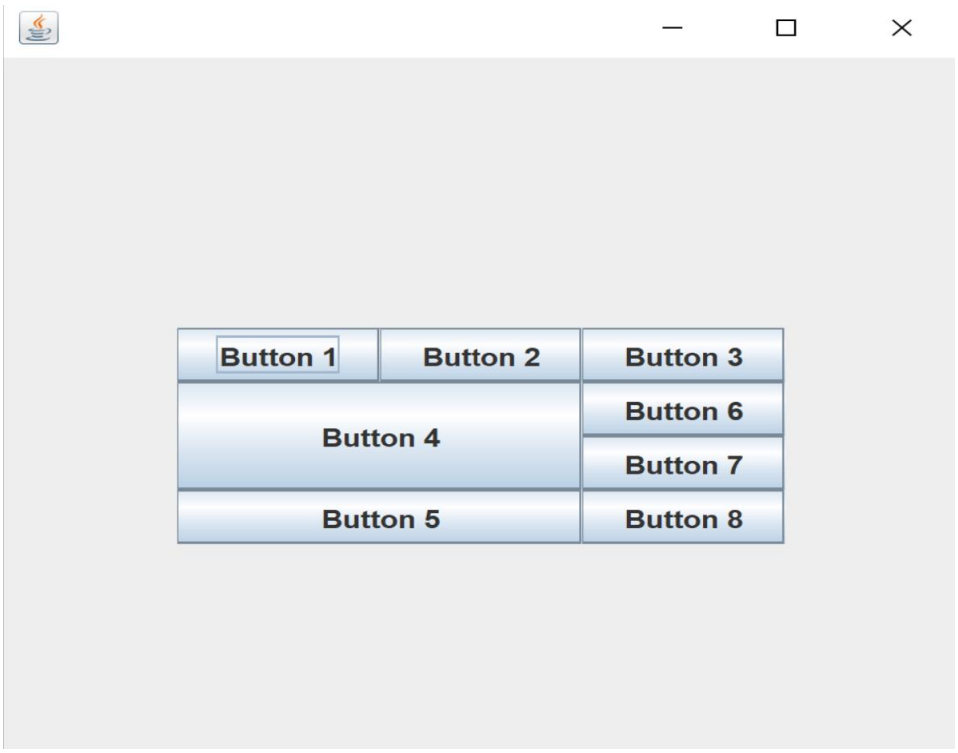
GridBagLayout

- More powerful than GridLayout.
- Size of each component can be controlled.
- It has a single constructor
- It uses a helper class “GridBagConstraints” and its properties gridx, gridy, gridwidth, gridheight, fill etc.


gridx = 0
gridy = 0
gridx = 1
gridy = 0
gridx = 2
gridy = 0
Button 1
Button 2
Button 3
Button 4
Button 6
Button 7
Button 5
Button 8
gridx = 0
gridy = 1
gridwidth = 2
gridheight = 2
fill = BOTH
gridx = 2
gridy = 1
gridwidth = 1
gridheight = 1
gridx = 2
gridy = 2
gridx = 2
gridy = 3
gridx = 0
gridy = 3
gridwidth = 2
gridheight = 2
fill = HORIZONTAL

```java
import javax.swing.*;
import java.awt.*;
public class GridBagDemo
{
    JFrame f1;
    JButton b1,b2,b3,b4,b5,b6,b7,b8;
    GridBagDemo()
    {
        f1 = new JFrame();
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setSize(width:400,height:400);

        GridBagLayout g1 = new GridBagLayout();
        f1.setLayout(g1);
        GridBagConstraints gbc = new GridBagConstraints();

        b1 = new JButton(text:"Button 1");
        gbc.gridx = 0;
        gbc.gridy= 0;
        g1.setConstraints(b1, gbc);
        f1.add(b1);

        b2 = new JButton(text:"Button 2");
        gbc.gridx = 1;
        gbc.gridy = 0;
        g1.setConstraints(b2, gbc);
        f1.add(b2);

        b3 = new JButton(text:"Button 3");
        gbc.gridx = 2;
        gbc.gridy = 0;
        g1.setConstraints(b3, gbc);
        f1.add(b3);

        b4 = new JButton(text:"Button 4");
        gbc.gridx = 0;
        gbc.gridy = 1;
        gbc.gridwidth = 2;
        gbc.gridheight = 2;
        gbc.fill = GridBagConstraints.BOTH;
        g1.setConstraints(b4, gbc);
        f1.add(b4);
```

```
44          b5 = new JButton(text:"Button 5");
45          gbc.gridx = 0;
46          gbc.gridy = 3;
47          gbc.gridwidth = 2;
48          gbc.gridheight = 1;
49          gbc.fill = GridBagConstraints.BOTH;
50          g1.setConstraints(b5, gbc);
51          f1.add(b5);
52
53          b6 = new JButton(text:"Button 6");
54          gbc.gridx = 2;
55          gbc.gridy = 1;
56          gbc.gridwidth = 1;
57          gbc.gridheight = 1;
58          g1.setConstraints(b6, gbc);
59          f1.add(b6);
60
61          b7 = new JButton(text:"Button 7");
62          gbc.gridx = 2;
63          gbc.gridy = 2;
64          //gbc.gridwidth = 1;
65          //gbc.gridheight = 1;
66          g1.setConstraints(b7, gbc);
67          f1.add(b7);
68
69          b8 = new JButton(text:"Button 8");
70          gbc.gridx = 2;
71          gbc.gridy = 3;
72          //gbc.gridwidth = 1;
73          //gbc.gridheight = 1;
74          g1.setConstraints(b8, gbc);
75          f1.add(b8);
76
77          f1.setVisible(b:true);
78      }
            Run | Debug
79      public static void main(String args[])
80      {
81          new GridBagDemo();
82      }
83  }
```



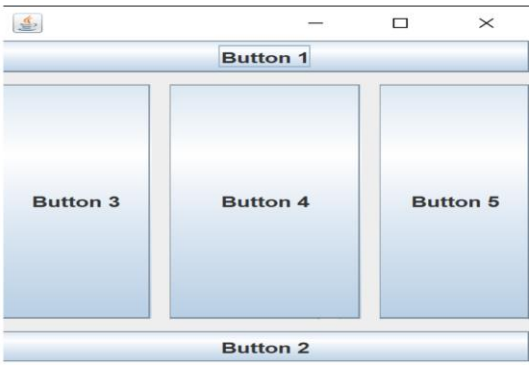
Border Layout:

A BorderLayout lays out a container, arranging its components to fit into five regions: NORTH, SOUTH, EAST, WEST and CENTER. For each region, it may contain no more than one component. When adding different components, we need to specify the orientation of it to be the one of the five region.

For BorderLayout, it can be constructed like below

- **BorderLayout():** Construct a border lauout with no gaps between components.
- **BorderLayout(int hgap, int vgap):** Construct a border layout with specified gaps between components.

```java
import javax.swing.*;
import java.awt.*;
public class BorderLayoutDemo
{
    JFrame f1;
    JButton b1,b2,b3,b4,b5;
    BorderLayoutDemo()
    {
        f1 = new JFrame();
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setSize(width:400,height:400);
        f1.setLayout(new BorderLayout(hgap:10,vgap:10));

        b1 = new JButton(text:"Button 1");
        b2 = new JButton(text:"Button 2");
        b3 = new JButton(text:"Button 3");
        b4 = new JButton(text:"Button 4");
        b5 = new JButton(text:"Button 5");

        f1.add(b1, BorderLayout.NORTH);
        f1.add(b2, BorderLayout.SOUTH);
        f1.add(b3, BorderLayout.WEST);
        f1.add(b4, BorderLayout.CENTER);
        f1.add(b5, BorderLayout.EAST);

        f1.setVisible(b:true);
    }
    Run | Debug
    public static void main(String args[])
    {
        new BorderLayoutDemo();
    }
}
```



Card Layout

The card layout manager displays the container we pass to it as cards, where cash container can have different layout manager. Only one card is visible at a time and the container acts as stack of cards. We can give each card a name and can move from card to card by using show() method.

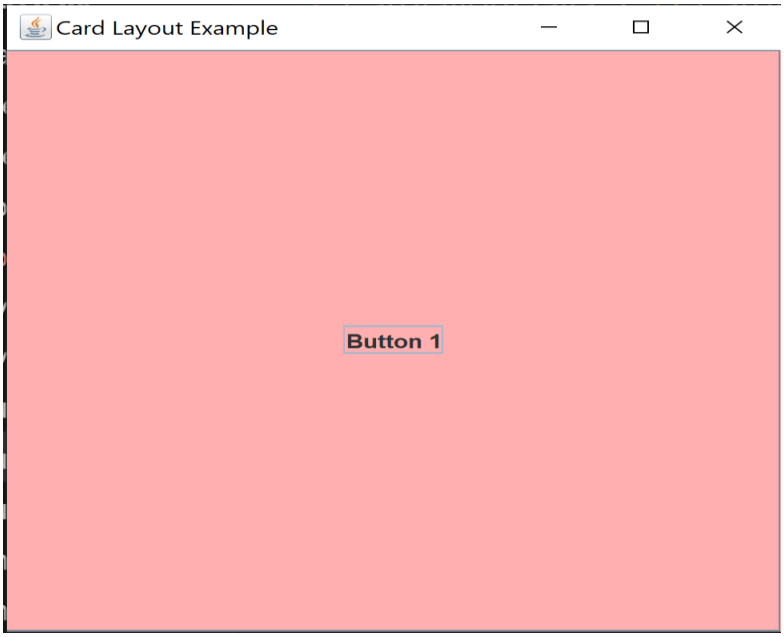
Commonly used constructor

CardLayout(): Creates a new card layout with gaps of size zero.

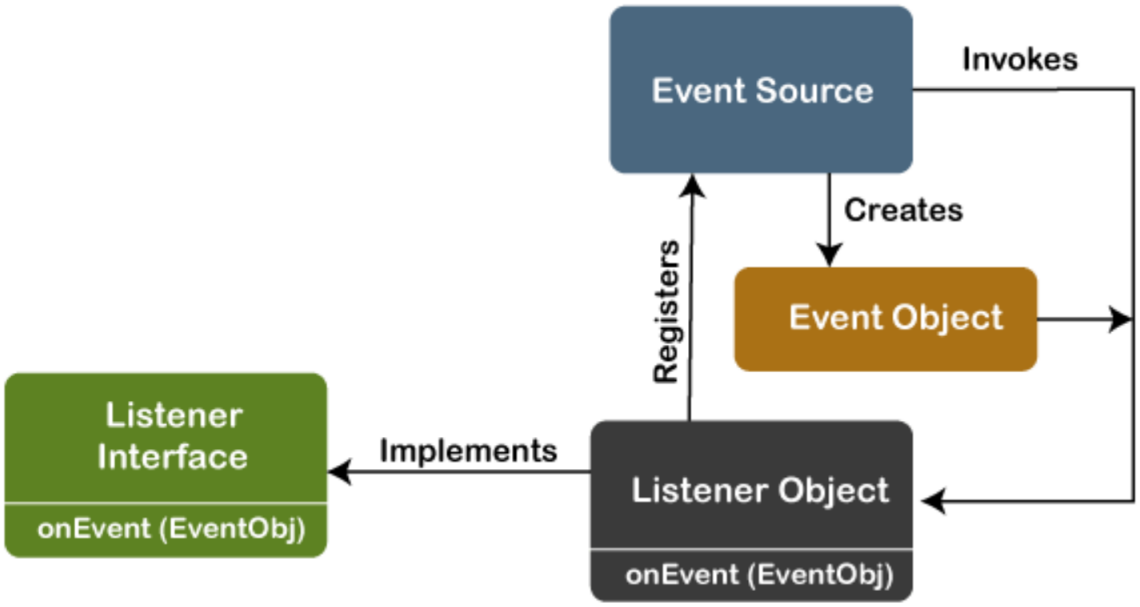
CardLayout(int hgap, int vgap): Creates a new card layout with the specified horizontal and vertical gaps.

```java
import java.awt.CardLayout;
import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

class CardlayoutDemo implements ActionListener {
    JFrame f1;
    JButton b1, b2, b3, b4;
    CardLayout cl;

    CardlayoutDemo() {
        f1 = new JFrame(title:"Card Layout Example");
        f1.setSize(width:400, height:400);
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        cl = new CardLayout();
        f1.setLayout(cl);

        b1 = new JButton(text:"Button 1");
        b2 = new JButton(text:"Button 2");
        b3 = new JButton(text:"Button 3");
        b4 = new JButton(text:"Button 4");

        b1.setBackground(Color.pink);
        b2.setBackground(Color.GREEN);
        b3.setBackground(Color.blue);
        b4.setBackground(Color.red);
```

```
30          // Add components to the frame with unique identifiers
31          f1.add(b1, constraints:"1");
32          f1.add(b2, constraints:"2");
33          f1.add(b3, constraints:"3");
34          f1.add(b4, constraints:"4");
35
36          // Make the frame visible
37          f1.setVisible(b:true);
38
39          // Add action listeners to the buttons
40          b1.addActionListener(this);
41          b2.addActionListener(this);
42          b3.addActionListener(this);
43          b4.addActionListener(this);
44      }
45
            Run | Debug
46      public static void main(String[] args) {
47          new CardlayoutDemo();
48      }
49
50      public void actionPerformed(ActionEvent e) {
51          cl.next(f1.getContentPane());
52      }
53  }
```



Event Delegation model



- The GUI programming is inherently event-driven; whenever a user initiates an activity such as a mouse activity, clicks, scrolling, etc., each is known as an event that is mapped to a code to respond to functionality to the user. This is known as event handling.
- The modern approach for event processing is based on the Delegation Model. It defines a standard and compatible mechanism to generate and process events. In this model, a source generates an event and forwards it to one or more listeners. The listener waits until it receives an event. Once it receives the event, it is processed by the listener and returns it. The UI elements are able to delegate the processing of an event to a separate function.
- The key advantage of the Delegation Event Model is that the application logic is completely separated from the interface logic.
- In this model, the listener must be connected with a source to receive the event notifications. Thus, the events will only be received by the listeners who wish to receive them. So, this approach is more convenient than the inheritance-based event model

Design goals of the event delegation model are as following

- It is easy to learn and implement.
- It supports a clean separation between application and GUI code.

- It provides robust event handling program code which is less error-prone (strong compile-time checking)
- It is Flexible, can enable different types of application models for event flow and propagation.
- It enables run-time discovery of both the component-generated events as well as observable events
- It provides support for the backward binary compatibility with the previous model

Event handling:
An event is a signal to the program that something has happened. It can be triggered by typing in a text field, selecting an item from the menu etc. The action is initiated outside the scope of the program and it is handled by a piece of code inside the program. Events may also be triggered when timer expires, hardware and software failure occurs, operation completes, counter is increased or decreased by a value etc.

- **Event handler**

The code that performs a task in response to an event is called event handler.

- **Event Handling**

It is process of responding to events that can occur at any time during execution of a program.

- **Event Source**

It is an object that generates the events. Usually the event source is a button or the other component that the user can click but any swing component can be an event source. The job of the event source is to accept registrations, get events from the user and call the listener's event handling methods.

Event Source	Description
Button	Generate action events when button is pressed.
CheckBox	Generates item events when the check box is selected or deselected
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double clicked; generates item events when an item is selected or deselected
Menu Item	Generates action events when a menu item is selected, generates item events when a checkable menu item is selected or deselected.
Srollbar	Generates adjustment events when the scroll bar is manipulated
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed deactivated, disconified, iconified, opened or quit.

- **Event listener**

It is an object that watch for (i.e listen for) events and handles them when they occur. It is basically a consumer that receives events from the source. To sum up, the job of an event listener is to implement the interface, register with the source and provide the event handling.

- **Listener Interface**

It is an interface which contains methods that the listener must implement and the source of the event invokes when the event occurs.

Event classes	Listener Interface
ActionEvent • When button is clicked. • MenuItem is clicked. • ListItem is doubled clicked. • Enter key is pressed in a text field.	ActionListener Public void actionPerform(Action Event e)
ItemEvent • When list item, radio button, checkbox, item in a combo box is clicked.	ItemListener Public void itemStateChange(ItemEvent e)
FocusEvent • When a textfield looses or gain keyboard focus.	FocusListener Public void focusGained(FocusEvent e) Public void focusLost(FocusEvent e)
KeyEvent • When a key is pressed in a text field or text area.	KeyListener Public void keyPressed(keyEvent e). Public void keyReleased(keyEvent e). Public void keyTyped(keyEvent e).
Container Event • When a component is added to or removed from container.	Container Listener Public void componentAdded(Container Event e) Public void componentRemoved(ContainerEvent e)
Mouse Event • When the mouse is clicked, enters a component, exist a component, is pressed or is released	Mouse Listener Public void mouseClicked(MouseEvent e). Public void mouseEntered(MouseEvent e). Public void mouseExited(MouseEvent e). Public void mousePressed(MouseEvent e). Public void mouseReleased(MouseEvent e).

Window event When a window is activated, closed, deactivated, deiconified, iconiied, opened or quit.	Window Listener Void windowActivated(WindowEvent e) Void windowClosed(WindowEvent e) Void windowClosing(WindowEvent e) Void windowDeactivated(WindowEvent e) Void windowDeiconified(WindowEvent e) Void windowIconified(WindowEvent e) Void windowOpened(WindowEvent e)

Handling Action Event

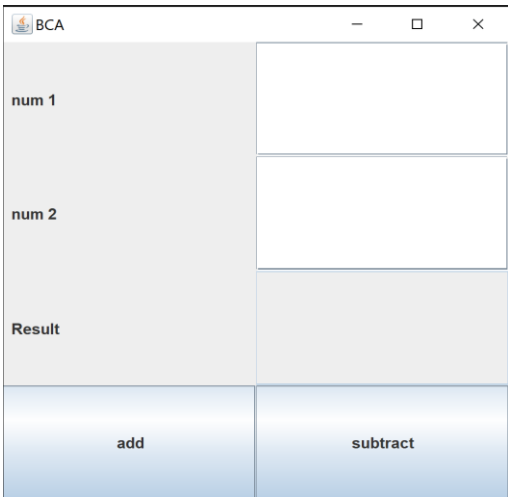
The class *ActionEvent* is defined in *java.awt.event* package. The *ActionEvent* is generated when button is clicked or the item of a list is double clicked. The object that implements the *ActionListener* interface gets this *ActionEvent* when the event occurs and hence must handle the event by overriding *actionPerformed()* method of *ActionListener* interface.

```java
import java.awt.FlowLayout;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;

class EventHandling2 implements ActionListener
{
    JFrame f1;
    JButton b1, b2;
    JTextField t1, t2, t3;
    JLabel l1, l2, l3;

    EventHandling2()
    {
        f1 = new JFrame(title:"BCA");
        f1.setSize(width:400, height:400);
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setLayout(new GridLayout(rows:4,cols:2));

        t1 = new JTextField(columns:10);
        t2 = new JTextField(columns:10);
        t3 = new JTextField(columns:10);
        t3.setEditable(b:false); // Make the result field read-only

        b1 = new JButton(text:"add");
        b2 = new JButton(text:"subtract");

        l1 = new JLabel(text:"  num 1");
        l2 = new JLabel(text:"  num 2");
        l3 = new JLabel(text:"  Result");
```

```
33          // Add components to the frame
34          f1.add(l1);
35          f1.add(t1);
36          f1.add(l2);
37          f1.add(t2);
38
39          f1.add(l3);
40          f1.add(t3);
41          f1.add(b1);
42          f1.add(b2);
43
44          // Make the frame visible
45          f1.setVisible(b:true);
46
47          // Add action listeners to the buttons
48          b1.addActionListener(this);
49          b2.addActionListener(this);
50      }
51
        Run | Debug
52      public static void main(String args[])
53      {
54          new EventHandling2();
55      }
```

```
57      public void actionPerformed(ActionEvent e)
58      {
59          int a, b, c;
60
61          a = Integer.parseInt(t1.getText());
62          b = Integer.parseInt(t2.getText());
63
64          if (e.getSource() == b1) //Code to be executed when the event is triggered by the button 'b1'
65          {
66              c = a + b;
67          }
68          else if (e.getSource() == b2)
69          {
70              c = a - b;
71          }
72          else
73          {
74              c = 0;
75          }
76
77          t3.setText(String.valueOf(c));
78      }
79  }
```

Output



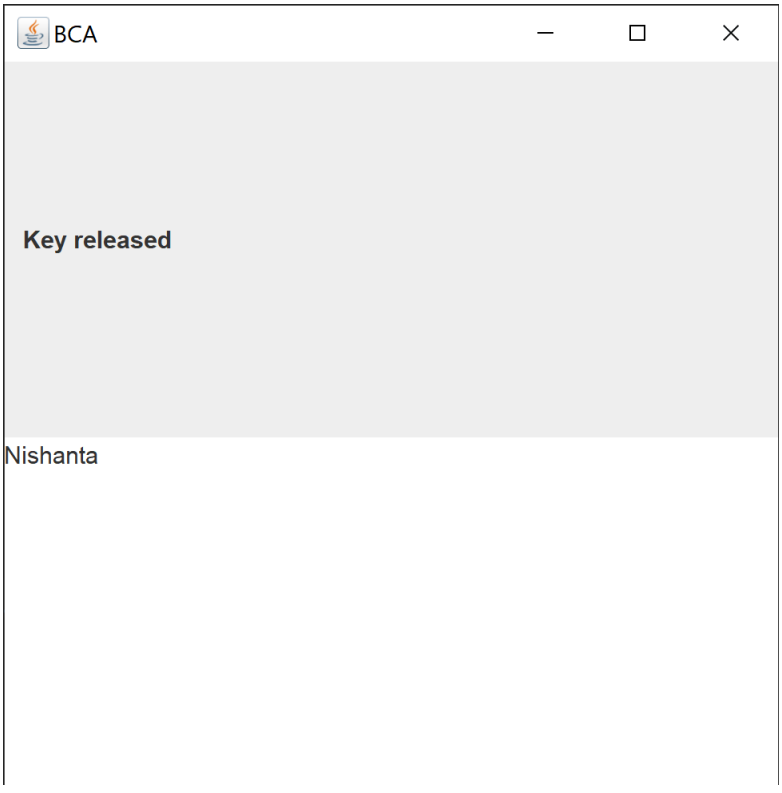
Key Event:

An event which indicates that a keystoke occurred in a component. Thie low level event is generated by a component object(such as a text field) when a key is pressed, released or typed. The event is passed to every KeyListener or KeyAdapter object which registered to receive such events using the component's addKeyListener method.

```
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import javax.swing.*;

class keyevent1 extends JFrame implements KeyListener
{
    JFrame f1;
    JLabel l1;
    JTextArea t1;
    keyevent1()
    {
        f1 = new JFrame(title:"BCA");

        f1.setSize(width:400, height:400);
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setLayout(new GridLayout(rows:2,cols:1,hgap:10,vgap:10));

        l1 = new JLabel(  text:"Email");
        t1 = new JTextArea();

        f1.add(l1);
        f1.add(t1);

        f1.setVisible(b:true);
        t1.addKeyListener(this);
    }
```

```
28
29          public void keyPressed(KeyEvent e)
30          {
31              l1.setText(text:"   Key pressed");
32          }
33      public void keyReleased(KeyEvent e)
34      {
35          l1.setText(text:"   Key released");
36      }
37          public void keyTyped(KeyEvent e)
38          {
39              l1.setText(text:"    Key typed");
40      }
        Run | Debug
41      public static void main(String args[])
42      {
43          new keyevent1();
44      }
45  }
```

output



Mouse Event

The MouseEvent Interface represents events that occur due to the user interacting with a pointing device(such as a mouse). Common events using this interface include click, double click, mouseup, mousedown. MouseListener and MouseMotionListener is an interface in a java.awt.event package. Mouse event sare of two types. MouseListener handles the events when the mouse is not in motion.

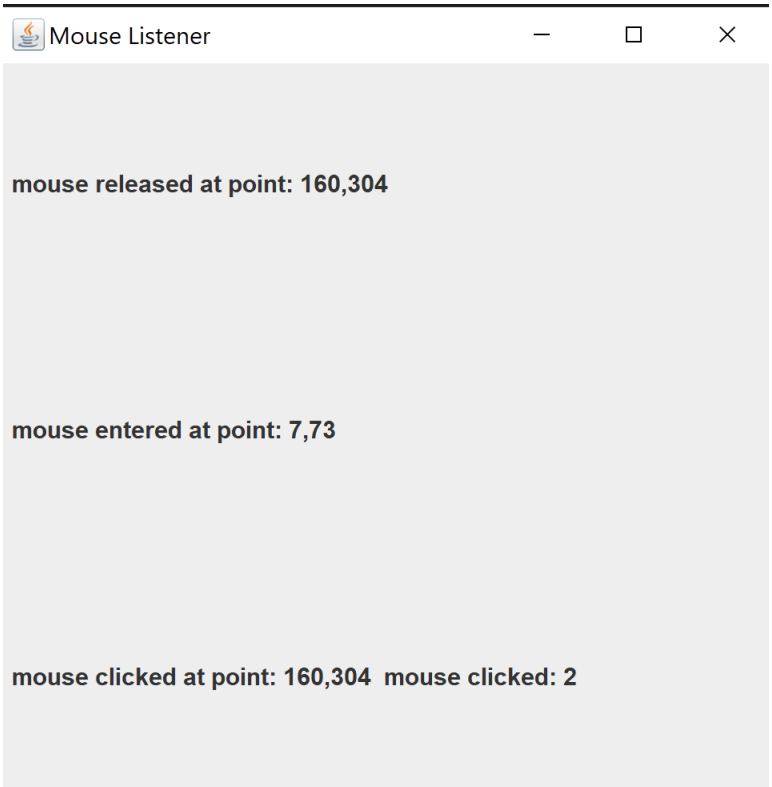
The abstract functions are:

- Void mouseReleased(MotionEvent e): Mouse key is released.
- Void mouseClicked(MouseEvent e): Mouse key is pressed / released.
- Void mouseExited(MouseEvent e): Mouse exited the component.
- Void mouseEntered(MouseEvent e): Mouse entered the component.
- Void mouse pressed(MouseEvent e): Mouse key is pressed

```java
import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.*;

class MouseEvent1 extends JFrame implements MouseListener
{
    static JLabel l1, l2, l3;
    Run | Debug
    public static void main(String args[])
    {
        JFrame f1 = new JFrame(title:"Mouse Listener");
        f1.setSize(width:400,height:400);
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel p = new JPanel();
        p.setLayout(new GridLayout(rows:3,cols:1,hgap:5,vgap:5));

        l1 = new JLabel(text:" no event 1: ");
        l2 = new JLabel(text:"  no event 2: ");
        l3 = new JLabel(text:"  no event 3: ");
        MouseEvent1 m = new MouseEvent1();

        p.add(l1);
        p.add(l2);
        p.add(l3);
        f1.add(p);

        f1.addMouseListener(m);

        f1.setVisible(b:true);
    }
```

```
31      public void mousePressed(MouseEvent e)
32      {
33          l1.setText("  mouse pressed at point: " + e.getX()+","+e.getY());
34      }
35      public void mouseReleased(MouseEvent e)
36      {
37          l1.setText("  mouse released at point: " + e.getX()+","+e.getY());
38      }
39      public void mouseExited(MouseEvent e)
40      {
41          l2.setText("  mouse exited at point: " + e.getX()+","+e.getY()+"\n");
42      }
43      public void mouseEntered(MouseEvent e)
44      {
45          l2.setText("  mouse entered at point: " + e.getX()+","+e.getY()+"\n");
46      }
47      public void mouseClicked(MouseEvent e)
48       {
49          l3.setText("  mouse clicked at point: " + e.getX()+","+e.getY()+"  mouse clicked: "+e.getClickCount());
50       }
51  }
```

Output



Focus Events

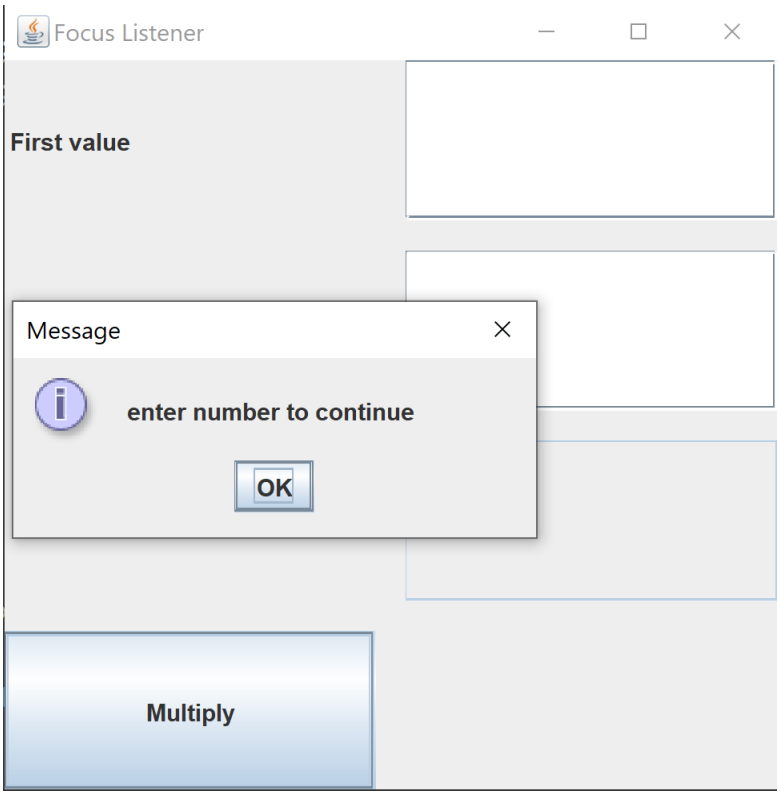
The class **FocusEvent** us defined **in java.awt.event** package. Focus events occur when any component gains or losses input focus on a GUI. Focus applies to all components that can receive input. To handle a focus event, a class must implement the **FocusListener** interface or extend **FocusAdapter** class. The object that implements the **FocusListener** interface gets this **FocusEvent** when the event occurs and hence must handle the event by overriding **focusGained()** and **focusLost()** methods of **FocusListener** interface. These object are invoke automatically for a component, which is registered with **FocusListener**, when focus is gained or lost by it respectively.

```
1  import java.awt.*;
2  import java.awt.event.FocusEvent;
3  import java.awt.event.FocusListener;
4  import javax.swing.*;
5
6  class Focusevent1 extends JFrame implements FocusListener
7  {
8      static JTextField t1, t2, t3;
9      static JLabel l1, l2, l3;
10     static JButton b1;
11
12     public Focusevent1()
13     {
14         JFrame f1 = new JFrame(title:"Focus Listener");
15         f1.setSize(width:400,height:400);
16         f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17         //JPanel p = new JPanel();
18         f1.setLayout(new GridLayout(rows:4,cols:2,hgap:15,vgap:15));
19
20         l1 = new JLabel(text:" First value ");
21         l2 = new JLabel(text:"  Second value ");
22         l3 = new JLabel(text:"  Third value ");
23
24         t1 = new JTextField(columns:10);
25         t2 = new JTextField(columns:10);
26         t3 = new JTextField(columns:10);
27         t3.setEditable(b:false);
```

```
29          b1 =  new JButton(text:"Multiply");
30
31          f1.add(l1);
32          f1.add(t1);
33          f1.add(l2);
34          f1.add(t2);
35          f1.add(l3);
36          f1.add(t3);
37          f1.add(b1);
38          //f1.add(p);
39
40
41          t1.addFocusListener(this);
42          t2.addFocusListener(this);
43          b1.addFocusListener(this);
44
45          f1.setVisible(b:true);
46      }
47
        Run | Debug
48      public static void main(String args[])
49      {
50          new Focusevent1();
51      }
```

```
53      public void focusGained(FocusEvent e)
54      {
55          int a, b, c;
56
57          a = Integer.parseInt(t1.getText());
58          b = Integer.parseInt(t2.getText());
59
60          if (e.getSource() == b1) //Code to be executed when the event is triggered by the button 'b1'
61          {
62              c = a * b;
63              t3.setText(String.valueOf(c));
64          }
65      }
66
67
68      public void focusLost(FocusEvent e)
69      {
70          if(e.getSource() == t1 && t1.getText().equals(anObject:""))
71          {
72              JOptionPane.showMessageDialog(this, message:"enter number to continue");
73              t1.requestFocus();
74          }
75
76          if(e.getSource() == t2 && t2.getText().equals(anObject:""))
77          {
78              JOptionPane.showMessageDialog(this, message:"enter number to continue");
79              t2.requestFocus();
80          }
81      }
82  }
```

Output



ItemEvent

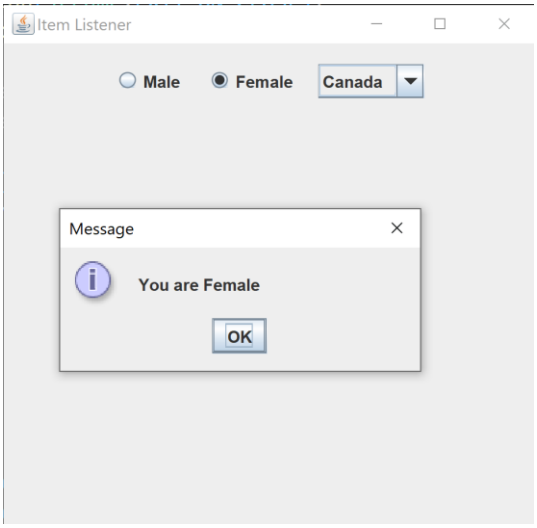
Item event is generated whenever user selects or deselects a selectable object such as radio button, checkbox or list. To handle an item event, a class must implement the **ItemListener** interface. There is no corresponding adapter class for this listener because there is only one method declared within it. The object that implements the **Itemlistener** interface gets item event when the event occurs and hence must handle the event by overriding, **itemStateChanged()** method of **itemListener** interface. The **stateChange** of any **ItemEvent** instance takes one of the following values:

ItemEvent SELECTED

ItemEvent DESELECTED

```java
import java.awt.*;
import java.awt.event.ItemEvent;
import java.awt.event.ItemListener;
import javax.swing.*;

class Itemevent1 extends JFrame implements ItemListener
{
    JFrame f1;
    static JRadioButton rb1, rb2;
    static JComboBox<String> cb;
    static ButtonGroup bg;

    Itemevent1()
    {
        f1 = new JFrame(title:"Item Listener");
        f1.setSize(width:400, height:400);
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setLayout(new FlowLayout(FlowLayout.CENTER, hgap:15, vgap:15));

        rb1 = new JRadioButton(text:"Male");
        rb2 = new JRadioButton(text:"Female");
        bg = new ButtonGroup();
        String country_name[] = {"Nepal", "India", "Canada", "Bhutan", "Thailand"};
        cb = new JComboBox<>(country_name);

        bg.add(rb1);
        bg.add(rb2);

        f1.add(rb1);
        f1.add(rb2);
        f1.add(cb);
```

```java
        rb1.addItemListener(this);
        rb2.addItemListener(this);
        cb.addItemListener(this);

        f1.setVisible(b:true);
    }

    Run | Debug
    public static void main(String[] args)
    {
        new Itemevent1();
    }

    public void itemStateChanged(ItemEvent e)
    {
        if (e.getSource() == rb1 && e.getStateChange() == ItemEvent.SELECTED)
        {
            JOptionPane.showMessageDialog(this, message:"You are Male");
        }
        else if (e.getSource() == rb2 && e.getStateChange() == ItemEvent.SELECTED)
        {
            JOptionPane.showMessageDialog(this, message:"You are Female");
        }
        else if (e.getSource() == cb && e.getStateChange() == ItemEvent.SELECTED)
        {
            JOptionPane.showMessageDialog(this, "Your country is " + cb.getSelectedItem());
        }
    }
}
```



Adapter class

- Java adapter classes provide the default implementation of listener interfaces.
- If we inherit adapter class, we will not be forced to provide the implementation of all the methods of listener interfaces. So it saves code.
- Event adapter classes are abstract class that provides some methods used for avoiding the heavy coding.
- Adapter class is defined for the listener that has more than one abstract methods.
- A source generates an event and sends it to one or more listeners registered with the source.
- Once event is received by the listener, they process the event and then return.
- Events are supported by a number of java packages, like java.util, java.awt and java.awt.event.

Java.awt.event Adapter classes

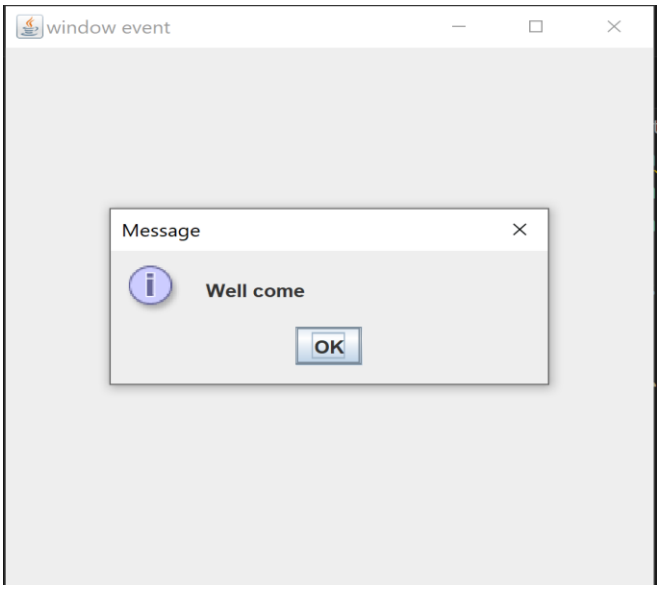
Adapter Class	**Listener Interface**
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener

Window Event by using adapter class

The class **WindowEvent** is defined in java.awt.event package. This is generated whenever there us a change in the window state such as minimized, maximized, activate, deactivated, opened, closing, closed. Window event is generated by components such as JFrame. JInternalFrame, JDialog etc. To handle a window event, a class must implement the **Windowlistene**r interface or extend WindowAdapter class. The object that implements the **WindowListener** interface gets the window when the event occurs and hence must handle the event by overriding **windowClosing()**, **WindowsClosed()**, **WindowIconified()**, **WindowDeconified()**, **windowActivated()**, **WindowDeactivated()** and **WindowOpened()** methods of **WindowListener** interface.

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

public class Adapter extends WindowAdapter
{
    JFrame f1;
    Adapter()
    {
        f1 = new JFrame(title:"window event");
        f1.setSize(width:400, height:400);
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        f1.addWindowListener(this);
        f1.setVisible(b:true);
    }

    public void windowClosing(WindowEvent e)
    {
        JOptionPane.showMessageDialog(f1, message:"Good Bye");
    }

    public void windowOpened(WindowEvent e)
    {
        JOptionPane.showMessageDialog(f1, message:"Well come");
    }

    public void windowIconified(WindowEvent e)
    {
        JOptionPane.showMessageDialog(f1, message:"See you later");
    }
```

```java
    public void windowDeiconified(WindowEvent e)
    {
        JOptionPane.showMessageDialog(f1, message:"Well come back");
    }

    Run | Debug
    public static void main(String[] args)
    {
        new Adapter();
    }
}
```



Introducing Graphics

The awt includes several methods that support graphics. All graphics are drawn relative to a window. This can be the main window of an application or a child window. The origin of each window is at the top – left corner and is 0.0. Coordinates are specified in pixels. All output to a widow takes place through a graphics context.

A graphics context is encapsulated by the graphics class. Here are two ways in which a graphics context can be obtained:

- It is passed to a method, such as paint() or update() as an argument.
- It is returned by the getGraphics() method of component.

Drawing Lines

Lines are drawn by means of the drawLine() method.

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

Drawing Rectangle

The **drawRect()** and **fillRect()** methods display an outlined and filled rectangle, respectively they are shown as:

Void drawRect(int left, int top, int width, int height)

Void fillRect(int left, int top, int width, int height)

Here the upper – left corner of the rectangle is at left, top. To draw a rounded rectangle, use **drawRoundRect()** and **fillRoundRect()** are shown below:

Void drawRoundRect(int left, int top, int width, int height, int xDiam, int yDiam)

Void fillRoundRect(int left, int top, int width, int height, int xDiam, int yDiam)

Drawing Ellipse and Circle

To draw an ellipse, use **drawOval().** To fill an elipse, use **fillOval().** These methods are shown here:

Void drawOval(int left, int top, int width, int height)

Void fillOval(int left, int top, int width, int height)

Drawing Arcs

Arcs can be drawn with **drawArc()** and **fillArc(),** shown here

Void drawArc(int left, int top, int width, int height, int startAngle, int sweepAngle)

Void fillArc(int left, int top, int width, int height, int startAngle, int sweepAngle)

Drawing polygon

It is possible to draw arbitrarily shaped figures using **drawPolygon()** and **fillPolygon()** shown here:

Void drawPolygon(int x[], int y[], int numPoints)

Void fillPolygon(int x[], int y[], int numPoints)

```
1  import java.awt.*;
2  import javax.swing.*;
3
4  public class Drawing extends JFrame
5  {
6      public Drawing()
7      {
8          setSize(width:400, height:700);
9          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         setVisible(b:true);
11     }
12
13     public void paint(Graphics g)
14     {
15         // Draw lines
16         g.drawLine(x1:80, y1:40, x2:100, y2:90);
17         g.drawLine(x1:20, y1:90, x2:100, y2:40);
18         g.drawLine(x1:40, y1:45, x2:250, y2:80);
19
20         // Draw rectangles
21         g.drawRect(x:20, y:150, width:60, height:50);
22         g.fillRect(x:110, y:150, width:60, height:50);
23         g.drawRoundRect(x:200, y:150, width:60, height:50, arcWidth:15, arcHeight:15);
24         g.fillRoundRect(x:290, y:150, width:60, height:50, arcWidth:30, arcHeight:40);
25
26         //Draw elipse and circle
27         g.drawOval(x:20, y:250, width:50, height:50);
28         g.fillOval(x:100, y:250, width:75, height:50);
29         g.drawOval(x:200, y:260, width:100, height:40);
```

```
31          //Draw arcs
32          g.drawArc(x:20, y:350, width:70, height:70, startAngle:0, arcAngle:180);
33          g.fillArc(x:70, y:350, width:70, height:70, startAngle:0, arcAngle:75);
34
35          //Draw polygon
36          int xpoints[] = {20,200,20,200,20};
37          int ypoints[] = {450,450,650,650,450};
38          int num = 5;
39
40          g.drawPolygon(xpoints, ypoints,num);
41
42      }
43
        Run | Debug
44      public static void main(String[] args) {
45          new Drawing();
46      }
47  }
```

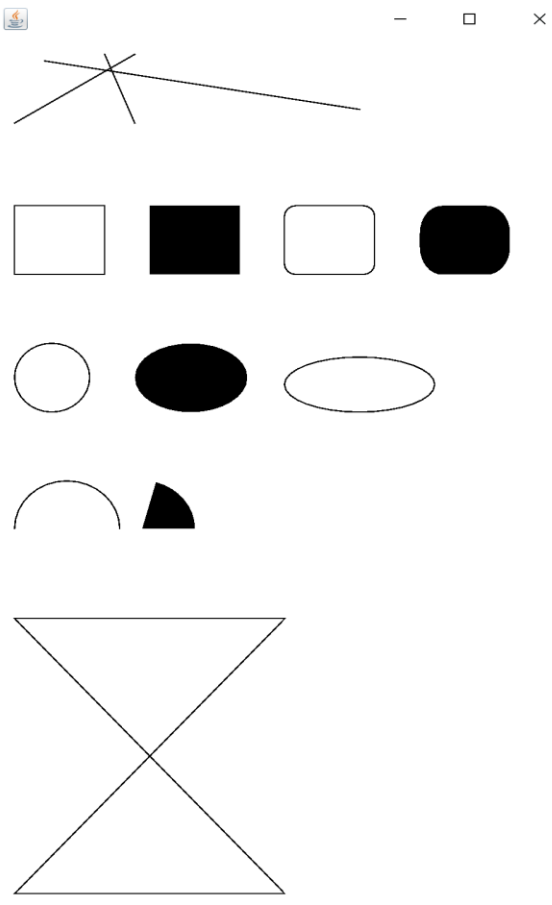


Image Fundamentals

There are three common operations that occurs when you are working with image

- Creating an image
- Loading an image
- Displaying an image

Creating an Image object

The **createImage()** method has the following two forms

Image createImage(imageProducer imgProd)

Image createImage(int width, int height)

Here the first form return an image produce by imgProd, which is an object of a class that implements the ImageProducer interface. The second form returns a blank(i.e empty) image that has the specified width and height.

For example

Canvas c = new Canvas();

Image test = c.createImage(200,100)

This creates an instance of Canvas and then calls the createImage() method to actually make an Image object.

Loading an image

The method that loads an image is called **read()**.

Static BufferedImage read(File imageFile) throws IOException

Here imageFile specifies the file that contains the image. It returns a reference to the image in the form of a BufferedImage, which is a subclass of Image that includes a buffer. Null is returned if the file does not contain a valid image

Displaying an Image

We can display image by using **drawImage(),** which is a member of the Graphics class.

boolean drawImage(Image *imgObj*, int *left*, int *top*, ImageObserver *imgOb*)

This display the image passed in imgObj with its upper – left corner specified by left and top imgObj is a reference to a class that implements the ImageObserver interface. An image observer is an object that can monitor an image while it loads. When no image observer is needed, imgOb can be null.

```
import java.awt.*;
import javax.swing.*;
import java.io.*;
import javax.imageio.ImageIO;

class Image1 extends JFrame
{
    Image img;

    Image1()
    {
        setTitle(title:"Displaying image");
        setSize(width:400, height:400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        try
        {
            // Specify the correct path to your image file
            File imageFile = new File(pathname:"D:\\images.jpeg");

            img = ImageIO.read(imageFile);

        }
```

```
24          catch (IOException e)
25          {
26              System.out.println("Cannot load image file: " + e.getMessage());
27              System.exit(status:0);
28          }
29
30          setVisible(b:true);
31      }
32
33      @Override
34      public void paint(Graphics g)
35      {
36          super.paint(g);
37          if (img != null)
38          {
39              g.drawImage(img, getInsets().left, getInsets().top, observer:null);
40          }
41      }
42
        Run | Debug
43      public static void main(String[] args)
44      {
45          new Image1();
46      }
47  }
```

MDI using JDesktop Pane and JInternal Frame

MDI stands for Multiple Document Interface. In an MDI application , one main window is opened and multiple child window are open within the main window. In an MDI application, we can open multiple frames that will be instances of the JInternalFrame class. We can organize multiple internal frames in many ways. For example, we can maximize and minimize them; we can view them side by side in a tiled fashion, or we can view them in a cascaded from. The following are four classes we needed for working with in an MDI application:

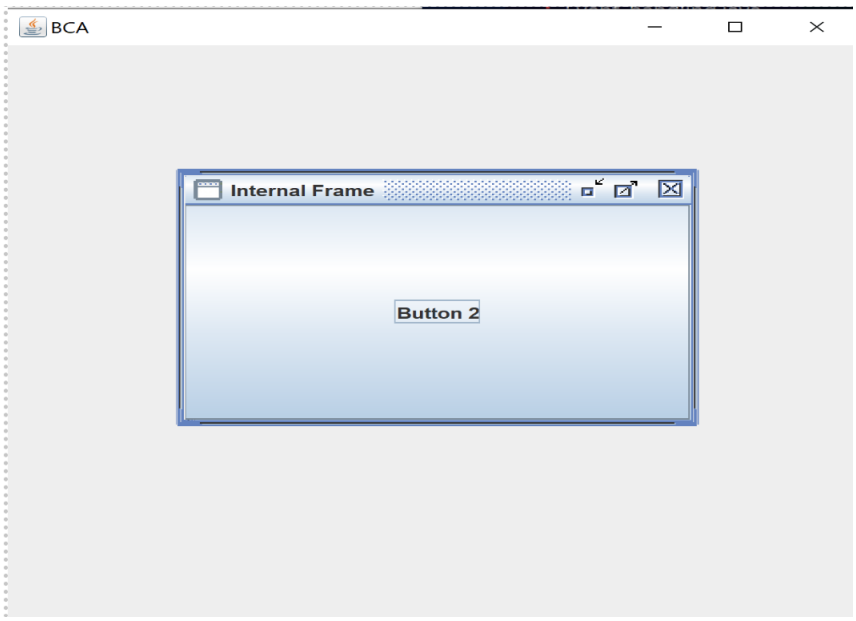
- JInternalFrame
- JDesktopPane
- DesktopManager

JInternalFrame

- A JInternalFrame is a light – weight component that has all the features of a JFrame.
- As we know JFrame is a heavy – weight component that can not be added to a Container that's why JInternalFrame comes into the picture that includes all the features of a JFrame.

```java
import javax.swing.*;
import java.awt.*;
class JinternalFrame extends JFrame
{
    JFrame f1;
    JButton b1;
    JinternalFrame()
    {
        f1 = new JFrame(title:"BCA");
        f1.setSize(width:500, height:500);
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setVisible(b:true);
        f1.setLocationRelativeTo(c:null);
        f1.setLayout(manager:null);

        //creating an internal frame
        JInternalFrame iframe = new JInternalFrame(title:"Internal Frame", resizable:true, closable:true, maximizable:true, iconif…true);
        internalframe(iframe);
    }
    private void internalframe(JInternalFrame iframe)
    {
        iframe.setSize(width:300,height:200);
        iframe.setDefaultCloseOperation(JInternalFrame.DISPOSE_ON_CLOSE);
        iframe.setLocation(x:50, y:50);
        iframe.setVisible(aFlag:true);

        b1 = new JButton(text:"Button 1");
        iframe.add(b1);
        f1.add(iframe);
    }
    Run | Debug
    public static void main(String args[])
    {
        new JinternalFrame();
    }
}
```

Output

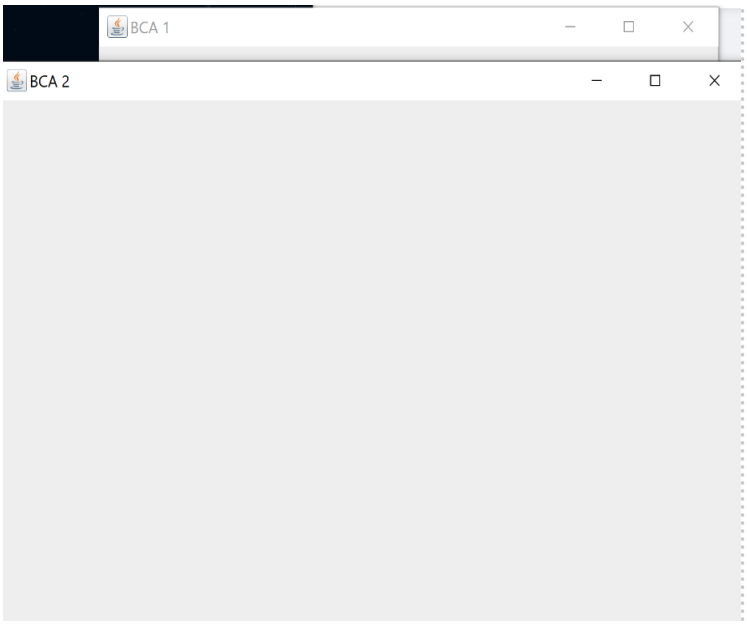


JDesktop Pane:

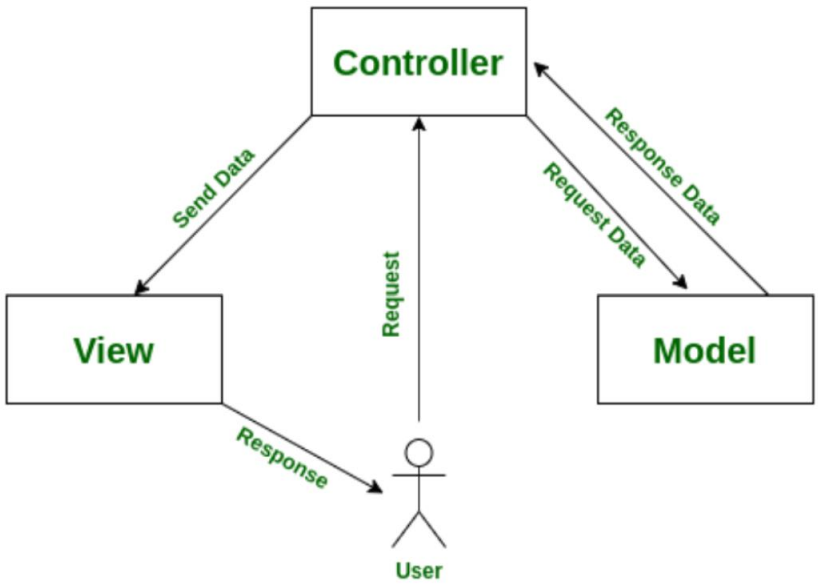
- JDesktopPane can be used to create “multi - document” application.
- A multi – document application can have many windows included in it. Example Photoshop.
- A JDesktopPane is a component by which you can add multiple small windows instances to the main window.

```java
import javax.swing.*;
import java.awt.*;
class JDesktoppane1 extends JFrame
{
    JFrame f1;
    JDesktoppane1()
    {
        f1 = new JFrame(title:"BCA 1");
        f1.setSize(width:500, height:500);
        f1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f1.setLocationRelativeTo(c:null);
        f1.setVisible(b:true);

        //creating JDesktopPane
        JDesktopPane dpane = new JDesktopPane();
        desktopPane(dpane);
        f1.add(dpane,BorderLayout.CENTER);
    }
    private void desktopPane(JDesktopPane dpane)
    {
        dpane.setLayout(mgr:null);
        JFrame f2 = new JFrame(title:"BCA 2");
        f2.setSize(width:600,height:500);
        f2.setDefaultCloseOperation(JInternalFrame.DISPOSE_ON_CLOSE);
        f2.setLocation(x:500, y:500);
        f2.setVisible(b:true);
        dpane.add(f2);
    }
    Run | Debug
    public static void main(String args[])
    {
        new JDesktoppane1();
    }
}
```

Output



Model View Controller (MVC) design pattern



Swing uses the model-view-controller architecture (MVC) as the fundamental design behind each of its components. MVC breaks GUI components into three elements. Each of these elements plays an important role in how the component behaves. It is a design pattern for the architecture of web applications whose purpose is to achieve a clean separation between three components of most any web application.

Model

The model represents the state data for each component. Model represents knowledge. A model stores data that is retrieved to the controller and displayed in the view. Whenever there is change in the data it is updated by the controller. There are different models for different types of components. For example, the model of a menu may contain a list of the menu items the user can select from. This information remains the same no matter how the component is painted on the screen; model data is always independent of the component's visual representation.

View

The view refers to how we see the component on the screen. It is a visual representation of its model. A view is attached to its model and gets the data necessary for the presentation from the model by asking questions. It may also update the model by sending appropriate messages. A view requests information from the model that it uses to generate an output representation to the user. As an example we can look at an application window on two different GUI platforms. Almost all window frames have a title bar spanning the top of the window. However, the title bar may have a close box on the left side (like the Mac OS platform), or it may have the close box on the right side (as in the Windows platform). These are examples of different types of views for the same window object.

Controller

The controller is the portion of the user interface that dictates how the component interacts with events. The controller decides how each component reacts to the event. It is a link between the user and the system. It provides the user with input by arranging relevant views to present themselves in appropriate places on the screen. It provides means for user output by presenting the user with menus or other means of giving commands and data. It receives such user output, translates it into the appropriate messages and passes these messages on to one or more of the views.

A controller can send commands to the model to update the model's state (e.g editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g by scrolling through a document)

Chapter – 2

JDBC

JDBC

JDBC stands for Java Database connectivity. Its advancement for ODBC (Open Database Connectivity). JDBC is a standard API specification developed in order to move data from frontend to backend. This API consists of classes and interface written in java. It is basically acts as an interface(not the one we use in java) or channel between our java program and databases i.e it establishes a link between the two so that a programmer could send data from java code and store it in the database for future use.

Steps for connectivity between java program and database

1. **Loading the Driver**

To begin with, we first need load the driver or register it before using it in the program Registration is to be done once in our program. We can register a driver in a way mentioned before.

2. **Class.forName()**

Here we load the driver's class file into memory at the runtime. No need of using new or creation of object. The following examples uses Class.forName() to load the Mysql driver

```
Class.forName("com.mysql.jdbc.Driver");
```

3. **Create the connection**

```
Connection con = DriverManager.getConnection(url, user, password)
```

User – username from which our sql command prompt can be accessed.

Password – password from which our sql command prompt can be accessed.

Con – it is a reference to connection interface.

Url – Uniform Resource Locator. It can be created as follows:

String url = "jdbc:mysql://localhost:3306/database_name"

4. **Create a statement**

Once a connection is established we can interact with the database. The JDBCStatement, CallableStatement and PreparedStatement interfaces define the methods that enable us to send SQL commands and receive data from our database.

5. **Use of JDBC statement is as follows**

```
Statement st = con.createStatement();
```

Here, con is a reference to connection interface used in previous step.

6. **Execute the query**

The executeQuery() method of statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table. The executeUpdate(sql query) method ofStatement interface is used to execute queries of updating / inserting.

Example:

```
int m = st.executeUpdate(sql);
If (m ==1)
        System.out.println("inserted successfully: " +sql)
Else
System.out.println("insertion failed");
Here sql is sql query of the type string.
```

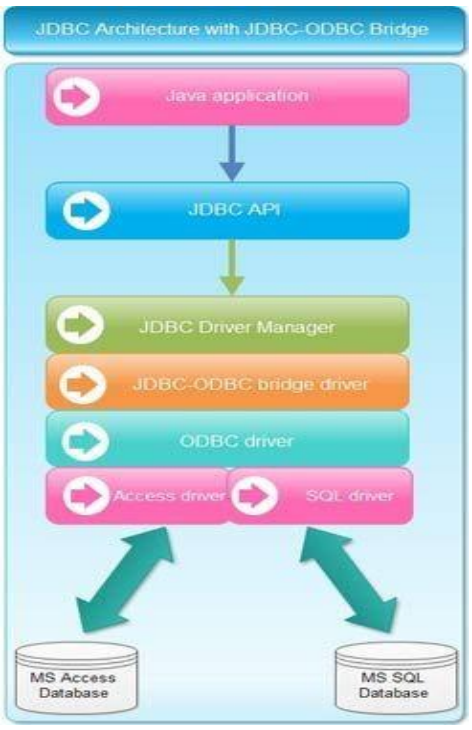
7. **Close the connection**

By closing connection, objects of Statement and Resultset will be closed automatically. The close() method of Connection interface is used to close the connection.

Example:

```
Con.close();
```

JDBC – ODBC Bridge (Remove from Java 8)

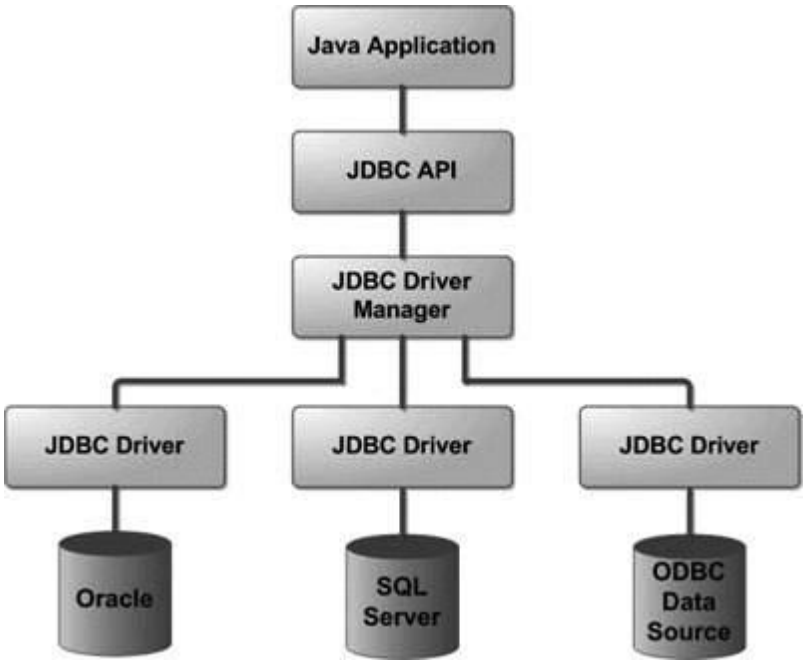


Microsoft's ODBC (Open Database Connectivity) is the most commonly used driver to connect to the database as it can connect to almost all databases on most of the platforms. However, ODBC uses the concept of pointers and the other constructs that are not supported by java.

Therefore, JDBC – ODBC bridge driver was developed which translates the JDBC API to the ODBC API and vice versa. The bridge acts as interface that enables all DBMS which support ODBC (Open Database Connectivity) to interact with java Application. JDBC – ODBC bridge is implemented as a class file and a native library. The name of the class file is JdbcOdbc.class.

Figure shows the JDBC application architecture in which a front – end application uses JDBC API for interacting with JDBC Driver Manager. Here, JDBC Driver Manager is the backbone of JDBC architecture. It acts as interface that connects a java application to the driver specified in the java program. Next, is the JDBC – ODBC bridge which helps the JDBC to access ODBC data sources.

JDBC Architecture



The JDBC API supports both two tier and three tier processing models for database access but in general, JDBC Architecture consists of two layers:

- JDBC API: This provides the application – to – JDBC Manager connection.
- JDBC Driver API: This supports the JDBC Manager – to – Driver Connection.

The JDBC API uses a driver manager and database – specific drivers to provide transparent connectivity to heterogeneous databases. The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

The JDBC API provides the following interfaces and classes:

- **DriverManager** − This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.
- **Driver** − This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection** − This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement** − You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet** − These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLExceptio**n - This class handles any errors that occur in a database application.

Example 1 (Get data from table)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class database
{
    public static void main(String args[])
    {
        getData();
    }
    4 usages
    public static Connection getConnection()
    {
        try
        {
            String driver = "com.mysql.cj.jdbc.Driver";
            String databaseUrl = "jdbc:mysql://localhost:3306/student";
            String username = "root";
            String password = "Nishanta";
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(databaseUrl, username, password);
            System.out.println("Database connected");
            return conn;
        }
        catch (Exception e)
        {
            System.out.println("Some error: " + e);
        }
        return null;
    }
```

```java
public static void getData()
{
    try
    {
        Statement statement = getConnection().createStatement();
        ResultSet result = statement.executeQuery( sql: "select * from student_detail");
        while (result.next()) {
            System.out.println(result.getString( columnLabel: "id"));
            System.out.println(result.getString( columnLabel: "name"));
            System.out.println(result.getString( columnLabel: "address"));
            System.out.println(result.getString( columnLabel: "phone"));
        }
    }
    catch (Exception e)
    {
        System.out.println("Error: " + e);
    }
}
```

Output

1

Nishanta

ktm

9843063710

2

Bipash

itahari

9841234568

3

Rajan Bhandari

illam

9845929158

Example 2 (Insert data from java to Mysql)

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class database
{
    public static void main(String args[])
    {
        insertData();
    }
    4 usages
    public static Connection getConnection() {
        try
        {
            String driver = "com.mysql.cj.jdbc.Driver";
            String databaseUrl = "jdbc:mysql://localhost:3306/student";
            String username = "root";
            String password = "Nishanta";
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(databaseUrl, username, password);
            System.out.println("Database connected");
            return conn;
        }
        catch (Exception e)
        {
            System.out.println("Some error: " + e);
        }
        return null;
    }
```

```java
public static void insertData()
{
    try
    {
        Statement statement = getConnection().createStatement();
        int result = statement.executeUpdate( sql: "insert into student_detail(id, name, address, phone) values(5, 'bijay', 'illam', '9844545456')");
        System.out.println(result);
        if (result == 1) {
            System.out.println("Data inserted");
        }
        else
        {
            System.out.println("Some error");
        }
    }
    catch (Exception e)
    {
        System.out.println("Error: " + e);
    }
}
```

Output

	id	name	address	phone
▶	1	Nishanta	ktm	9843063710
	2	Manas	Bkt	9840033123
	3	Bipash	itahari	9841234568
	4	Rajan	illam	9845929158
	5	bijay	illam	9845929123
*	NULL	NULL	NULL	NULL

Example 3 (Delete data from database)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class database
{
    public static void main(String args[])
    {
        Deletedata();
    }
    4 usages
    public static Connection getConnection()
    {
        try
        {
            String driver = "com.mysql.cj.jdbc.Driver";
            String databaseUrl = "jdbc:mysql://localhost:3306/student";
            String username = "root";
            String password = "Nishanta";
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(databaseUrl, username, password);
            System.out.println("Database connected");
            return conn;
        }
        catch (Exception e)
        {
            System.out.println("Some error: " + e);
        }
        return null;
    }
```

```
public static void Deletedata()
{
    try
    {
        Statement statement = getConnection().createStatement();
        int result = statement.executeUpdate( sql: "delete from student_detail where id = 5");
        if (result == 1) {
            System.out.println("Record deleted");
        }
        else
        {
            System.out.println("Error while deleting a detail");
        }

    }
    catch (Exception e)
    {
        System.out.println("Error: " + e);
    }
}
```

Output

id	name	address	phone
1	Nishanta	ktm	9843063710
2	Manas	Bkt	9840033123
3	Bipash	itahari	9841234568
4	Rajan	illam	9845929158
NULL	NULL	NULL	NULL

Example 4 (update data from the database)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
public class database
{
    public static void main(String args[])
    {
        updateData();
    }
    4 usages
    public static Connection getConnection() {
        try
        {
            String driver = "com.mysql.cj.jdbc.Driver";
            String databaseUrl = "jdbc:mysql://localhost:3306/student";
            String username = "root";
            String password = "Nishanta";
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(databaseUrl, username, password);
            System.out.println("Database connected");
            return conn;
        }
        catch (Exception e)
        {
            System.out.println("Some error: " + e);
        }
        return null;
    }
```

```java
    public static void updateData()
    {
        try{
            Statement statement =getConnection().createStatement();
            int result = statement.executeUpdate( sql: "update student_detail set name = 'Rajan Bhandari' where id = 4");
            if (result == 1)
            {
                System.out.println("Record updated");
            }
            else
            {
                System.out.println("Error while updating a data");
            }
        }
        catch(Exception e)
        {
            System.out.println("some errors " +e);
        }

    }
}
```

Output

id	name	address	phone
1	Nishanta	ktm	9843063710
2	Manas	Bkt	9840033123
3	Bipash	itahari	9841234568
4	Rajan Bhandari	illam	9845929158
NULL	NULL	NULL	NULL

Example 5 (Get record with input from Mysql)

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.Scanner;

public class database1 {
    public static void main(String args[])
    {
        get_from_user(); // Call the method to get user input and display data
    }

    2 usages
    public static Connection getConnection()
    {
        try
        {
            String driver = "com.mysql.cj.jdbc.Driver";
            String databaseUrl = "jdbc:mysql://localhost:3306/student";
            String username = "root";
            String password = "Nishanta";
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(databaseUrl, username, password);
            System.out.println("Database connected");
            return conn;
        }
        catch (Exception e)
        {
            System.out.println("Some error: " + e);
        }
        return null;
    }
```

```java
    public static void get_from_user()
    {
        try
        {
            Scanner scan = new Scanner(System.in);
            System.out.print("Enter student ID: ");
            int id = scan.nextInt();

            // Create a Statement from the connection
            Statement statement = getConnection().createStatement();

            // Execute the query using the created statement
            ResultSet result = statement.executeQuery( sql: "select * from student_detail where id = " +id);

            // Print the retrieved data
            while (result.next())
            {
                System.out.println(result.getString( columnLabel: "id"));
                System.out.println(result.getString( columnLabel: "name"));
                System.out.println(result.getString( columnLabel: "address"));
                System.out.println(result.getString( columnLabel: "phone"));
            }
            result.close();
            statement.close();
            getConnection().close();
            scan.close();
        } catch (Exception e) {
            System.out.println("Some error: " + e);
        }
    }
}
```

Output:

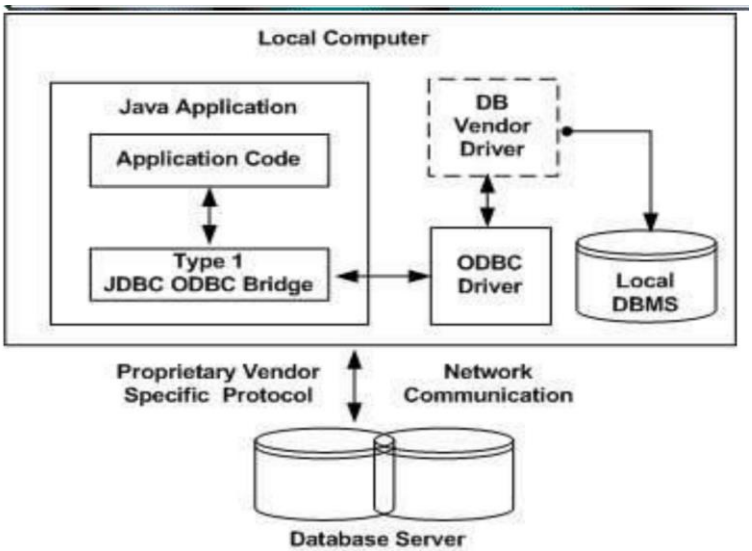
```
Enter student ID: 3
Database connected
3
Bipash
itahari
9841234568
Database connected
```

JDBC Drivers Types

JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types1, 2, 3 and 4 which is explained below.

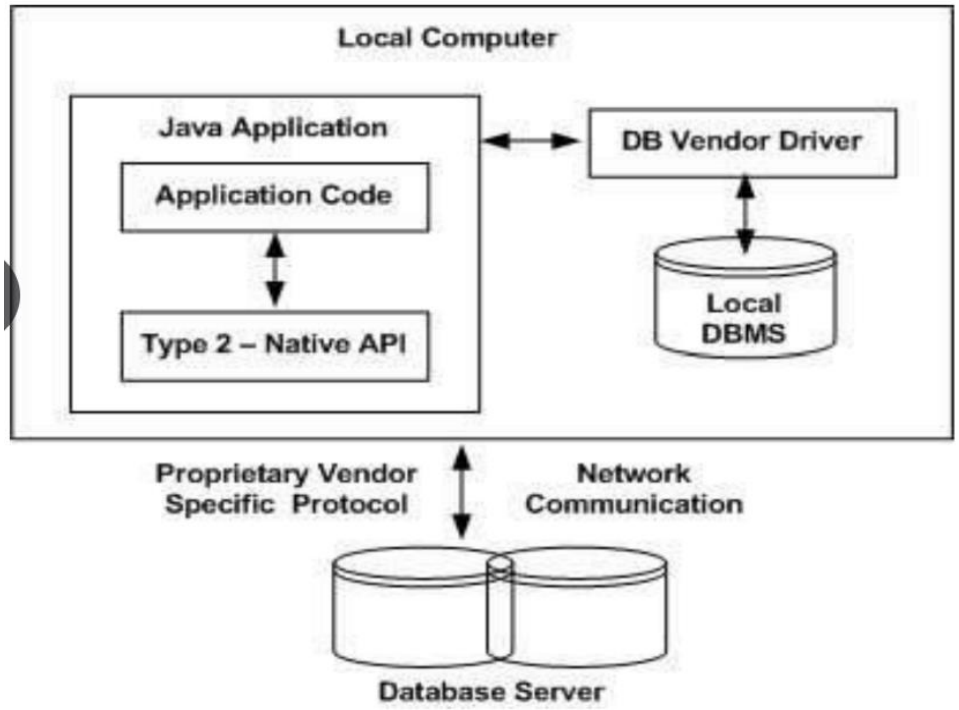
Type 1: JDBC - ODBC Bridge Driver

In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on our system a Data Source Name(DSN) that represents the target database. When java first came out, this was a useful driver because most databases only supported ODBC access but now **this type of driver is recommended only for experimental use or when no other alternative is available.**



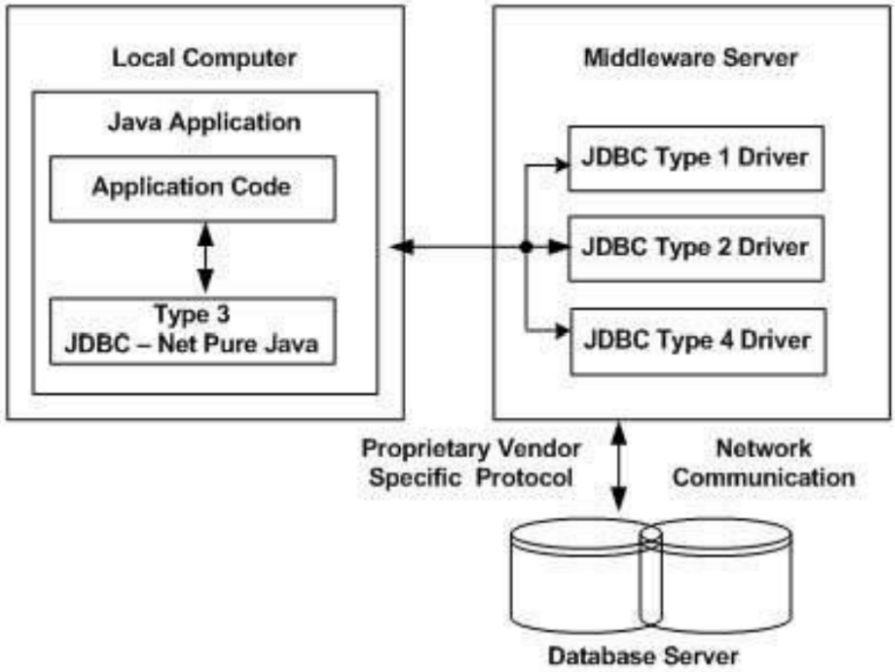
Type 2: JDBC – Native API

In this Type 2 driver, JDBC API calls are converted into native C / C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC – ODBC bridge. The vendor – specific driver must be installed on each client machine. If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but we may realize some speed increase with a Type 2 driver because it eliminates ODBC's overhead.



Type 3: JDBC – Net pure Java

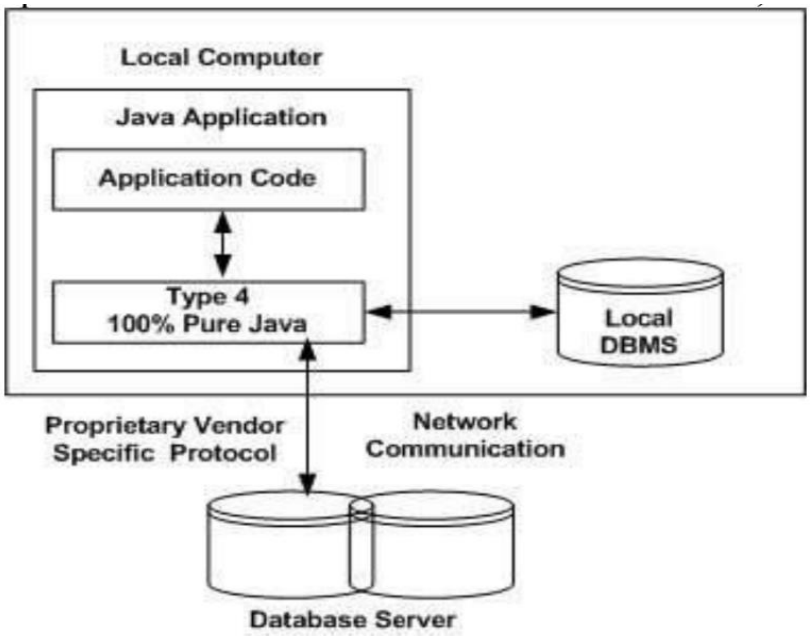
In a Type 3 driver, a three – tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server. This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.



We can think of the application server as a JDBC "proxy" meaning that it makes calls for the client application. As a result, we need some knowledge of the application server's configuration in order to effectively use this driver type.

Type 4 : 100% pure Java

In a Type 4 driver, a pure java – based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself. This kind of driver is extremely flexible; we don't need to install special software on the client or server. Further these drivers can be downloaded dynamically.



MySQL's Connector / J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Creating a Statement for Executing Queries

Once a connection is obtained we can interact with the database. The JDBC Statement, Callable statement and prepared statement interfaces define the methods and properties that enable us to send SQL or PL/SQL command and receive data from our database.They also define methods that help bridge data type differences between java and SQL data types used in database.

1. **Creating Statement object**

Before using a statement object to execute a SQL statement, we need to create one using the Connection object's **createStatement()** method.

```
Statement st = null;
try
{
    st = conn.createStatement()
    .....
}
catch(SQLException e)
{
    .....
}
Finally
{
    ......
}
```

2. **PreparedStatement Objects**

The PreparedStatement interface extends the statement interface, which gives us added functionality with a couple of advantages over a generic Statement object. This statement gives us the flexibility of supplying arguments dynamically.

```
PreparedStatement ps = null;
try
{
    String SQL = "Update Employees SET age =? WHERE id = ?";
    ps = conn.PreparedStatement(SQL);
}
catch(SQLException e)
{
    .........
}
Finally
```

```
{
    ..........
}
```

All the parameters is JDBC are represented by the ? Symbol, which is known as the parameter marker. We must supply values for every parameter before executing.

3. **CallableStatement object**

Just as a Connection object creates the Statement and PreparedStatement objects, it is also creates the CallableStatement object, which would be used to execute a call to a database store procedure.

```
CREATE OR REPLACE PROCEDURE getEmpName
(EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) as
BEGIN
    SELECT first INTO_FIRST
    FROM Employees
    WHERE ID = EMP_ID
END;
```

Above stored procedure has been written for Oracle, but we are working with MySQL database so, let us write same stored procedure for MySQL as follows to create it in EMP database.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS 'EMP'.'getEmpName'$$
CREATE PROCEDURE 'EMP'.'getEmpName'
(IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
    SELECT first INTO EMP_FIRST
    FROM Employees
    WHERE ID = EMP_ID;
END $$
DELIMITER
```

Three types of parameter exist: IN, OUT and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all the three.

Example:

```
CallableStatement cs = null;
try
{
    String SQL = "{call getEmpName(?,?)}";
    cs = conn.prepareCall(SQL);
}
catch(SQLException e)
{
    .........
```

```
}
Finally
{
    ..........
}
```

ResultSet:

The SQL statements that reads data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories

- **Navigational methods** − Used to move the cursor around.
- **Get methods** − Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods** − Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

JDBC provides the following connection methods to create statements with desired ResultSet :

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- **prepareCall(String sql, int RSType, int RSConcurrency);**

Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE	The cursor can scroll forward and backward and the result set is sensitive to changes made by others to the database that occur after the result was created.

Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	Description
Result.Set.CONCUR_READ_ONLY	Creates a read – only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

Navigating a Result Set

There are several methods in the ResultSet interface that involve moving the cursor, including:

Methods	Description
Public void beforeFirst() throws SQLException	Moves the cursor just before the first row.
Public void afterLast() throws SQLException	Moves the cursor just after the last row.
Public boolean first() throws SQLException	Moves the cursor to the first row
Public boolean last() throws SQLException	Moves the cursor to the last row
Public boolean absolute(int row) throws SQLException	Moves the cursor to the specified row.
Public boolean relative(int row) throws SQLException	Movess the cursor the given number of rows forward or backward, from where it is currently pointing.
Public boolean previous() throws SQLException	Moves the cursor to the previous row. This method returns false if there are no more rows in the result set.
Public boolean next() throws SQLException	Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
Public int getRow() throws SQLException	Return the row number that the cursor is pointing to.

Viewing a Result Set

The ResultSet interface contains dozen of methods for getting the data of current row. There is a get method for each of the possible data types and each get method has two version.

- One that takes in a column name.
- One that takes in a column index.

Method	Description
Public int getInt(String columnName) throws SQLException	Returns the int in the current row in the column named columnName.
Public int getInt(int columnIndex) throws SQLException	Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Updating a Result Set

The ResultSet interface contains a collection of update methods for updating the data of a result set. As with the get methods, there are two update methods for each data type.

- One that takes in a column name.
- One that takes in a column index.

Methods	Description
Public void updateString(int columnIndex, String s) throws SQLException	Changes the String in the specified column to the value of s.
Publuc void updateString(String columnName, String s) throws SQLException	Similar to the previous method, except that the column us specified by its name instead of its index.

ResultSetMetaData

The metadata means data about data i.e. we can get further information from the data.

If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

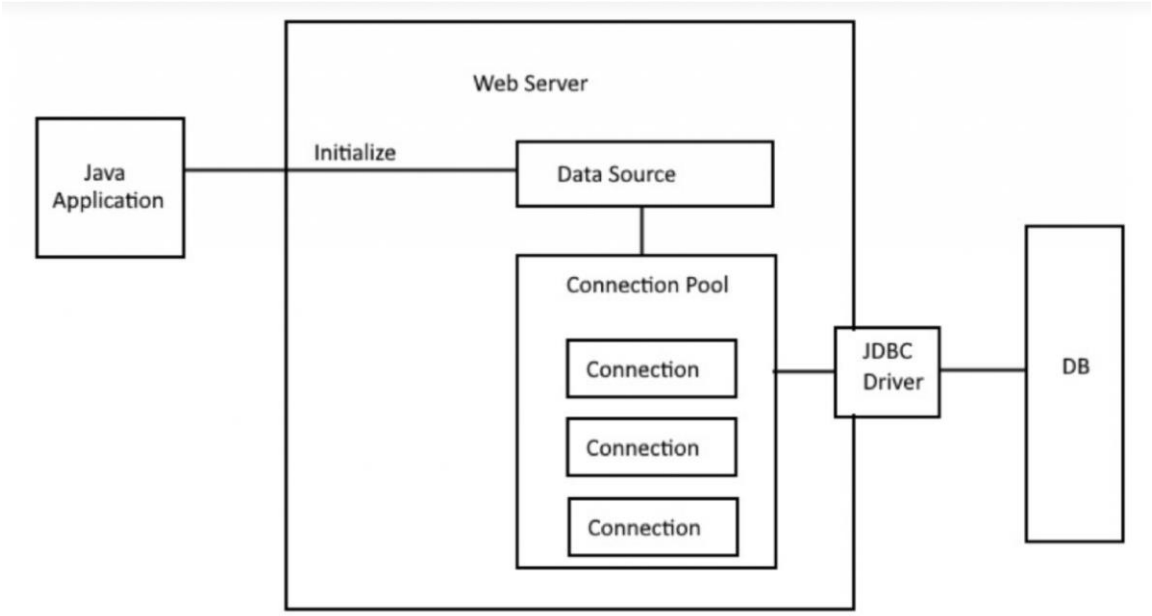
Method	Description
getColumnCount()	Retrieves the number of columns in the current ResultSet object.
getColumnLabel()	Retrieves the suggested name of the column for use.
getColumnName()	Retrieves the name of the column.
getTableName()	Retrieves the name of the table.

Example

```java
import java.sql.*;

public class metadata
{
    public static void main(String args[])
    {
        metadataDemo();
    }

    1 usage
    public static Connection getConnection()
    {
        try
        {
            String driver = "com.mysql.cj.jdbc.Driver";
            String databaseUrl = "jdbc:mysql://localhost:3306/student";
            String username = "root";
            String password = "Nishanta";
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(databaseUrl, username, password);
            System.out.println("Database connected");
            return conn;
        }
        catch (Exception e)
        {
            System.out.println("Some error: " + e);
        }
        return null;
    }
```

```java
    public static void metadataDemo()
    {
        try
        {
            Statement statement = getConnection().createStatement();
            ResultSet result = statement.executeQuery( sql: "select * from student_detail");
            ResultSetMetaData RSMD = result.getMetaData();
            System.out.println("no of columns: " +RSMD.getColumnCount());
            System.out.println("Column 3 name is: "+RSMD.getColumnName(3));
            System.out.println("Column 3 type is: "+RSMD.getColumnTypeName(3));
        }
        catch (Exception e)
        {
            System.out.println("Error: " + e);
        }
    }
}
```

Connection Pooling



Connection pooling means a pool of Connection Objects. Connection pooling is based on an object pool design pattern. Object pooling design pattern is used when the cost (time & resources like CPU, Network, and IO) of creating new objects is higher. As per the Object pooling design pattern, the application creates an object in advance and place them in Pool or Container. Whenever our application requires such objects, it acquires them from the pool rather than creating a new one.

An application that uses a connection pooling strategy has already DB connection objects which can be reused. So, when there is a need to interact with the database, the application obtains connection instances from Pool. Connection pooling improves application performance that interacts with the database.

We can create our own implementations of Connection pooling. Any connection pooling framework needs to do three tasks.

- Creating Connection Objects
- Manage usage of created Objects and validate them
- Release/Destroy Objects

With Java, we have great set of libraries which are readily available. We only need to configure few properties to use them

Chapter – 3

Java Beans

- A java Bean is a software component that has been designed to be reusable in a variety of different environments.
- There is no restriction on the capability of a bean. It may perform a simple function, such as obtaining an inventory value, or a complex function such as forecasting the performance of a stock portfolio.
- A bean may be visible to end user. One of the example of this is a button on a graphical user interface. A bean may also be invisible to a user. Software to decode a stream of multimedia information in real time is an example of this type of building block.
- A bean may be designed to work autonomously on a user's workstation or to work in cooperation with a set of other distributed components.
- Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally.
- However, a Bean that provides real – time price information from a stock or commodities exchange would need to work in cooperation with other distributed software to obtain its data.

Advantages of java beans

- A bean obtain all the benefits of java's "write once, run anywhere" paradigm.
- The properties, events and methods of a Bean that are exposed to another application can be controlled.
- Auxiliary software can be provided to help configure a Bean. This software is only needed when the design – time parameters for that components are being set. It does not need to be included in the run – time environment.
- The state of a Bean can be saved in persistent storage and restored at a later time.
- A bean may register to receive events form other objects and can generate events that are sent to other objects.

The standard to develop a java bean class are follows:

1. Class must be taken as public.
2. It is recommended to implement java.io.Serializable to send data over the network.
3. All bean properties (member variable must be taken as private and non static).
4. Every bean property should have one public setter method and one public getter method. Here setter method will be used to set the values to the property (member variable) and getter method will be used to get those values from variables.
5. There must be one zero(0) parameter constructor explicitly (given by programmer) or implicitly (given by java compiler as default constructor).

Introspection

At the core of java beans in introspection. This is the process of analyzing a Bean to determine its capabilities. This is an essential features of the java beans API because it allows another application such as a design tools, obtain information about a component. Without introspection, the java beans technology could not operate.

There are two ways in which the developer of a bean can indicate which of its properties, events and methods should be exposed. With the first method, simple naming conventions are used. These allow the introspection mechanism to infer information about a Bean. In the second way, an additional class that extends the **Beaninfo** interface is provided that explicitly supplies this information. Both approaches are examined here.

Design patterns for properties

A property is a subset of a beans state. The value assigned to the properties determine the behavior and appearance of that component. A property is set through is set through a setter method. A property is obtained by getter method.

There are two types of properties

Simple properties

A simple property has a single value. It can be identified by the following design patterns where N is the name of the property and T is the type:

Public T getN()

Public void setN(T arg)

A read / write property has both of these method to access its values. A read – only property has only a get method. A write – only property has only a set method.

Example

```
private double depth, height, width;

public double getDepth()
{
    return depth;
}
public void setDepth(double d)
{
    depth = d;
}


public double getHeight()
{
    return height;
}
public void setHeight(double h)
{
    height = h;
}


public double getWidth()
{
    return width;
}
public void setWidth(double w)
{
    width = w;
}
```

Indexed Properties

An indexed property consists of multiple values. It can be identified by the following design patterns, where N is the name of the property and T is its type.

Public T getN(int index);

Public void setN(int index, T value);

Public T[] getN();

Public void setN(T values[]);

```java
private double data[]

public double getData(int index)
{
    return data(index);
}

public void setData(int index, double value)
{
    data[index] = value;
}

public double [] getData()
{
    return data;
}

public void setData(double[] values)
{
    data = new double[values.length];
    System.arraycopy (values, 0, data, 0, values.length);
}
```

Design Patterns for Events

Beans can generate events and send them to other objects. These can be identified by the following design patterns, where T is the type of event:

Public void addTListener(TListener eventListener)

Public void addTListener(TListener eventListener) throws java.util.TooManyListenerException

Public void removeTListener(TListener eventListener)

- These methods are used to add or remove a listener for the specified event.
- The version of **addTListener()** that does not throw an exception can be used to multicast an event, which means that more than one listener can register for the event notification.
- The version that throws **TooManyListenerException** unicast the event, which means that the number of listener can be restricted to one.
- The **removeTListener()** is used to remove the listener.

For example, assuming an event interface type called **TemperatureListener**, a bean that monitors temperature might supply the following methods:

Public void addTemperatureListener(TemperatureListener tl)

{

……..

}

Public void removeTemperatureListener(TemperatureListener tl)

{

……..

}

Simple Bean example

```java
import java.io.Serializable;
public class Bean2 implements Serializable  3 usages
{
    private int roll_no;  no usages
    private String name;  2 usages
    private String faculty;  no usages

    public void setName(String n)  1 usage
    {
        name = n;
    }

    public String getName()  1 usage
    {
        return name;
    }
}
```

```java
class Test
{
    public void register(Bean2 B)  1 usage
    {
        System.out.println(B.getName());
    }

    public static void main(String[] a)
    {
        Bean2 B = new Bean2();
        B.setName("Nishanta");

        Test t = new Test();
        t.register(B);
    }
}
```

Methods and Design patterns

Design patterns are not used for naming non property methods. The introspection mechanism finds all of the public methods of a Bean. Protected and Private methods are not presented.

Using the Beaninfo Interface

As the preceding discussion shows, design patterns implicitly determine what information is available to the user of a Bean. The **BeanInfo** interface enables you to explicitly control what information is available. The **BeanInfo** interface defines several methods, including these

PropertyDescriptor[] getPropertyDescriptors()

EventDescriptor[] getEventDescriptors()

MethodDescriptor[] getMethodDescriptors()

They return array of objects that provide information about the properties, events and methods of a Bean. The classes **PropertyDescriptor, EventDescriptor** and **MethodDescriptor** are defined within the **java.beans** package and they describe the indicated elements.

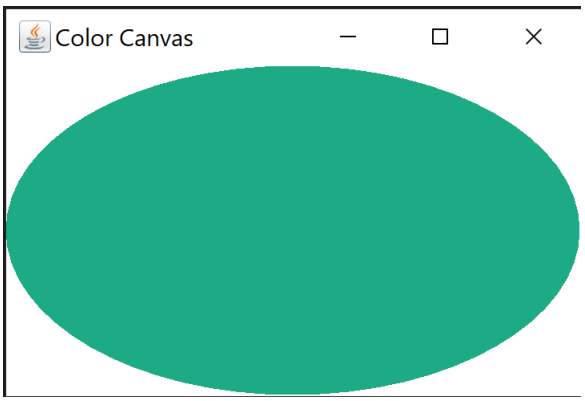
When creating a class that implements BeanInfo, you must call that class ***bnameBeanInfo,*** where bname is the name of the Bean. For example, if the Bean is called MyBean, then the information class must be called **MyBeanBeanInfo**.

Example:

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

public class Color extends Canvas implements Serializable
{
    private transient java.awt.Color color;  2 usages

    public Color()  1 usage
    {
        addMouseListener(new MouseAdapter()
        {
            public void mousePressed(MouseEvent e)
            {
                change();
            }
        });
    }

    public void change()  1 usage
    {
        color = randomColor();
        repaint();
    }
```

```java
    private java.awt.Color randomColor()  1 usage
    {
        int r = (int) (255 * Math.random());
        int g = (int) (255 * Math.random());
        int b = (int) (255 * Math.random());
        return new java.awt.Color(r, g, b);
    }

    public void paint(Graphics g)
    {
        Dimension d = getSize();
        int h = d.height;
        int w = d.width;
        g.setColor(color);
        g.fillOval( x: 0,  y: 0,  width: w - 1,  height: h - 1);

    }
    public static void main(String[] args) {
        Frame frame = new Frame( title: "Color Canvas");
        Color canvas = new Color();
        frame.add(canvas);
        frame.setSize( width: 300,  height: 200);
        frame.setVisible(true);
    }
}
```

Output

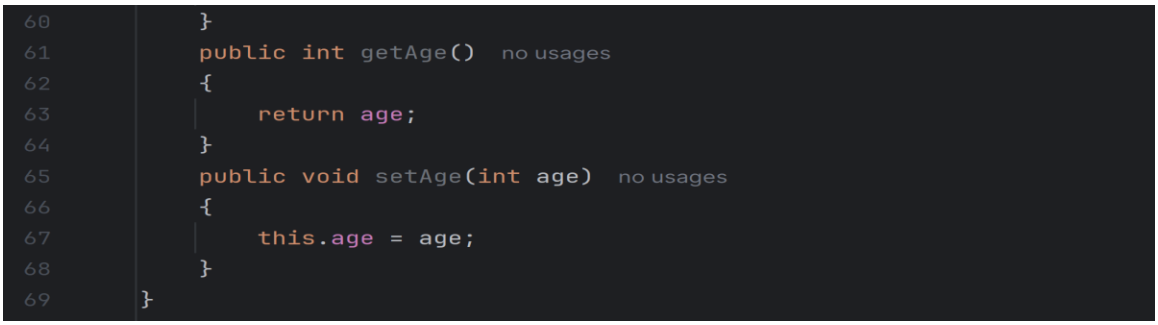


BeanInfo Interface

The BeanInfo interface defines the methods that a class must implement in order to export information about a Java bean. The Introspector class knows how to obtain all the basic information required about a bean. A bean that want to be more “programmer - friendly” may provide a class that implements this interface, however in order to provide additional information about itself (such as an icon and description strings for each of its properties, events and methods).

Example

```java
import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.lang.reflect.Method;

public class bean_interface
{
    public static void main(String[] args)
    {
        try
        {
            // Get the BeanInfo for the Person class
            BeanInfo beanInfo = Introspector.getBeanInfo(Person.class);

            // Print the properties
            PropertyDescriptor[] properties = beanInfo.getPropertyDescriptors();
            for (PropertyDescriptor property : properties)
            {
                // Ignore the "class" property
                if (property.getName().equals("class"))
                {
                    continue;
                }
                System.out.println("Property Name: " + property.getDisplayName());
                System.out.println("  Type: " + property.getPropertyType().getSimpleName());

                // Print the read and write methods
                Method readMethod = property.getReadMethod();
                if (readMethod != null)
                {
                    System.out.println("  Read Method: " + readMethod.getName());
                }
                Method writeMethod = property.getWriteMethod();
                if (writeMethod != null)
                {
                    System.out.println("  Write Method: " + writeMethod.getName());
                }
                System.out.println();
            }
        }
        catch (IntrospectionException e)
        {
            e.printStackTrace();
        }
    }
}

class Person {  1 usage
    private String name;  2 usages
    private int age;  2 usages

    public String getName()  no usages
    {
        return name;
    }
    public void setName(String name)  no usages
    {
        this.name = name;
```



```
60          }
61          public int getAge()  no usages
62          {
63              return age;
64          }
65          public void setAge(int age)  no usages
66          {
67              this.age = age;
68          }
69      }
```

Bound and Constrained Properties

A bean that have a bound property generates an event when the property is changed. The event is of type PeopertyChangeEvent and is sent to objects that previously registered an interest in receiving such notifications. A class that handles this event must implement the PropertyChangeListener interface.

A Bean that has a constrained property generates an event when an attempt is made to change its value. It also generates an event of type PropertyChangeEvent. However those other object have the ability to veto the proposed change by throwing a PropertyVetoException. This capability allow a Bean to operate differently according to its run time environment. A class that handles this event must implement the VetoableChangeListener interface.

Persistence

Persistence is the ability to save the current state of a Bean, including the values of a Bean's properties and instance variable, to non volatile storage and to retrieve them at a later time.The object serialization capabilities provided by the java class libraries are used ti provide persistence for Beans.

The easiest way to serialize a Bean is to have it implement the java.io.Serializable interface, which is simply a marker interface. Implementing java.io.Serializable makes serialization automatic. Your Bean need taje no other action. Automatic serialization can also be inherited. Therefore, if any superclass of a Bean implements java.io.Serializable then automatic serialization is obtained.

When using automatic serialization, you can selectively prevent a field from being saved through the use of the transient keyword. Thus, data members of a Bean specified as transient will not be serialized.

Customizers

A bean developer can provide a customizer that helps another developer configure the Bean. A customizer can provide a step – by – step guide through the process that must be followed to use the component in a specific context. Online documentation can also be provided. A bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.

The Java Beans API

The java Beans functionality is provided by a set of classes and interface in the java.beans package. Beginning with JDK 9, this package is in the java.desktop module. This section provides a brief overview of its contents. The fig – 1 lists the interfaces in java.beans and provides a brief description of their functionality. The fig – 1 lists the classes in java.beans.

Interface	Description
AppletInitializer	Methods in this interface are used to initialize Beans that are also applets. (Deprecated by JDK 9.)
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred.
PropertyChangeListener	A method in this interface is invoked when a bound property is changed.
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

Fig 1: The interface in java.beans

Class	Description
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.
DefaultPersistenceDelegate	A concrete subclass of **PersistenceDelegate**.
Encoder	Encodes the state of a set of Beans. Can be used to write this information to a stream.
EventHandler	Supports dynamic event listener creation.
EventSetDescriptor	Instances of this class describe an event that can be generated by a Bean.
Expression	Encapsulates a call to a method that returns a result.
FeatureDescriptor	This is the superclass of the **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** classes, among others.

Class	Description
IndexedPropertyChangeEvent	A subclass of **PropertyChangeEvent** that represents a change to an indexed property.
IndexedPropertyDescriptor	Instances of this class describe an indexed property of a Bean.
IntrospectionException	An exception of this type is generated if a problem occurs when analyzing a Bean.
Introspector	This class analyzes a Bean and constructs a **BeanInfo** object that describes the component.
MethodDescriptor	Instances of this class describe a method of a Bean.
ParameterDescriptor	Instances of this class describe a method parameter.
PersistenceDelegate	Handles the state information of an object.
PropertyChangeEvent	This event is generated when bound or constrained properties are changed. It is sent to objects that registered an interest in these events and that implement either the **PropertyChangeListener** or **VetoableChangeListener** interfaces.
PropertyChangeListenerProxy	Extends **EventListenerProxy** and implements **PropertyChangeListener**.
PropertyChangeSupport	Beans that support bound properties can use this class to notify **PropertyChangeListener** objects.
PropertyDescriptor	Instances of this class describe a property of a Bean.
PropertyEditorManager	This class locates a **PropertyEditor** object for a given type.
PropertyEditorSupport	This class provides functionality that can be used when writing property editors.
PropertyVetoException	An exception of this type is generated if a change to a constrained property is vetoed.
SimpleBeanInfo	This class provides functionality that can be used when writing **BeanInfo** classes.
Statement	Encapsulates a call to a method.
VetoableChangeListenerProxy	Extends **EventListenerProxy** and implements **VetoableChangeListener**.
VetoableChangeSupport	Beans that support constrained properties can use this class to notify **VetoableChangeListener** objects.
XMLDecoder	Used to read a Bean from an XML document.
XMLEncoder	Used to write a Bean to an XML document.

Fig 2: The classes in java.beans

Introspector

The introspector class provides several static methods that support introspection. Of most interest is getBeanInfo(). This method returns a BeanInfo object that can be used to obtain information about the Bean. The getBeanInfo() method has several forms, including the one shown here.

Static BeanInfo getBeanInfo(Class<?> bean) throws IntrospectionException

The returned object contain information about the Bean Specified by bean.

PropertyDescriptor

The propertyDescriptor class describes the characteristics of a Bean property. It supports several methods that manage and describe properties. For example, you can determine if a property is bound by calling isBound(). To determine if a property is constrained, call isConstrained(). You can obtain the name of a property by calling getName().

EventSetDescriptor

The EventSetDescriptor class represent a set of Bean events. It supports several methods that obtain the methods that a Bean uses to add or remove event listeners, and to otherwise manage events. For example, to obtain the method used to add listener, call getAddListenerMethod(). To obtain the method used to remove listeners, call getRemoveListenerMethod(). To obtain the type of a listener, call getListenerType(). You can obtain the name of an event set by calling getName.

MethodDescriptor

The methodDescriptor class represents a Bean method. To obtain the name of the method call getName(). You can obtain information about the method by calling getMethod(), shown below

Method getMethod()

An object of type Method that describes the method is returned.

Example:

```java
import java.beans.EventSetDescriptor;
import java.beans.IntrospectionException;
import java.beans.MethodDescriptor;
import java.beans.PropertyDescriptor;
import java.beans.BeanInfo;
import java.beans.Introspector;
import java.lang.reflect.Method;

public class color1 {
    public static void main(String[] args) {
        try {
            // Get the BeanInfo for the java.awt.Button class
            BeanInfo beanInfo = Introspector.getBeanInfo(java.awt.Button.class);

            // Print the property descriptors
            System.out.println("Properties:");
            PropertyDescriptor[] properties = beanInfo.getPropertyDescriptors();
            for (PropertyDescriptor property : properties) {
                System.out.println("Name: " + property.getName());
                System.out.println("Type: " + property.getPropertyType());
                System.out.println("Read method: " + property.getReadMethod());
                System.out.println("Write method: " + property.getWriteMethod());
                System.out.println();
            }
```

```java
            // Print the event set descriptors
            System.out.println("Event Sets:");
            EventSetDescriptor[] events = beanInfo.getEventSetDescriptors();
            for (EventSetDescriptor event : events) {
                System.out.println("Name: " + event.getName());
                System.out.println("Event types: " + event.getListenerType());
                System.out.println("Add listener method: " + event.getAddListenerMethod());
                System.out.println("Remove listener method: " + event.getRemoveListenerMethod());
                System.out.println();
            }

            // Print the method descriptors
            System.out.println("Methods:");
            MethodDescriptor[] methods = beanInfo.getMethodDescriptors();
            for (MethodDescriptor method : methods) {
                System.out.println("Name: " + method.getName());
                System.out.println("Return type: " + method.getMethod().getReturnType());
                Method m = method.getMethod();
                System.out.println("Parameter types: " + m.getParameterTypes());
                System.out.println();
            }
        } catch (IntrospectionException e) {
            e.printStackTrace();
        }
    }
}
```

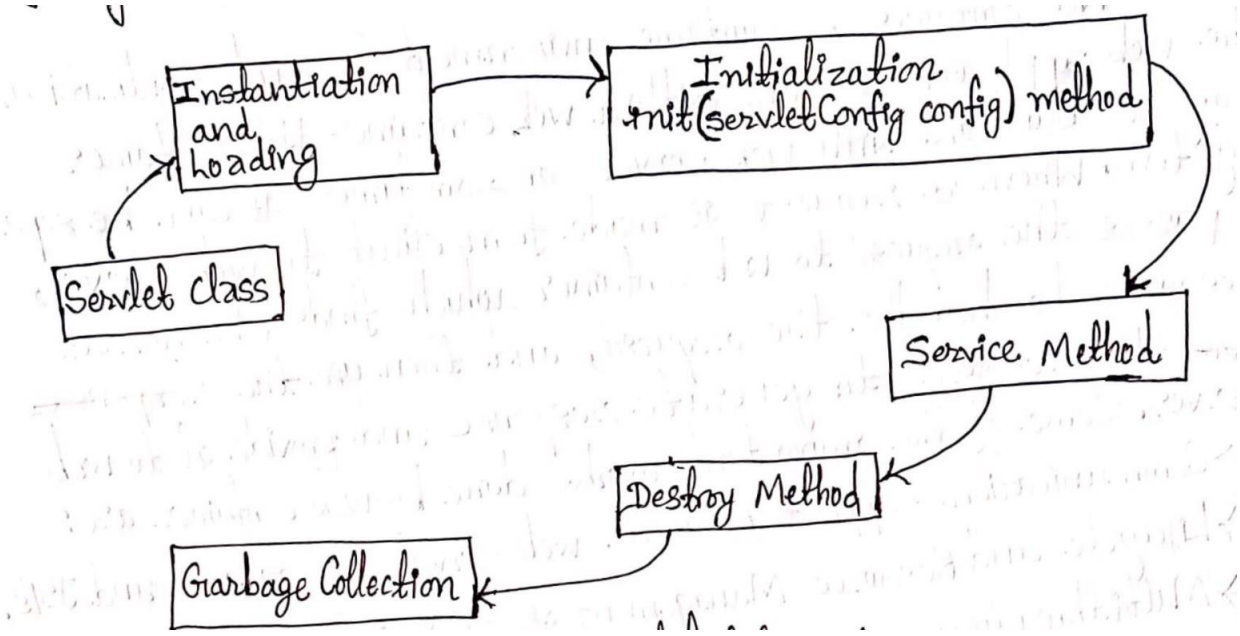
Chapter – 4

Servlets and JSP

Introduction to Servlets

- Servlets are small programs that execute on the server side of a web connection.
- Just as applets dynamically extend the functionality of a web browser servlets dynamically extend the functionality of a Web server.
- It is a java programming language class used to extend the capabilities of servers that host applications accessed via a request – response programming model.
- Although servlets can respond to any type of request they are commonly used to extend the applications hosted by web servers.
- For such applications, java servlet technology define HTTP – Specific servlet classes.
- The javax.servlet and javax.servlet.http package provide interfaces and classes for writing servlets .
- All servlets must implement the servlet interface, which defines life – cycle methods.

Life cycle of a servlets



The servlet life cycle had 5 stages in general. Among them three methods are central to the life cycle of a servlet. These are int(), service() and destroy().

1. Loaded and Instantiated

The class loader is responsible to load the servlet class. Then it creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

2. Initialized by calling the int() method

The init method is used to initialize the servlet.

Syntax of the init() method is as follows

Public void init(ServletConfig config) throws ServletException

3. Process a clients request by invoking service() method

The web container calls the service method each time when request for the servlet is received. The syntax is as follows:

Public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException

4. Terminated by calling the destroy() method

It gives the servlet an opportunity to clean up any resource for example memory, thread etc

Syntax

Public void destroy()

5. Garbage collected by the JVM

Once the servlet is destroyed, garbage collection component of JVM is responsible collecting the garbage's.

Servlet APIs

- Two packages contain the classes and interfaces that are required to build servlets. These are javax.servlet and javax.servlet.http .
- They constitute the servlet API.
- These packages are not part of the java core packages.
- Instead, they are standard extensions. Therefore, they are not included in the Java software Development kit.
- We must download TomcaT or Glass Fish server to obtain their functionality.

```java
import java.io.*;
import jakarta.servlet.*;
public class ServletAPI implements Servlet {
    private ServletConfig config = null;

    public void init(ServletConfig config) throws ServletException {
        this.config = config;
        System.out.println("Servlet is initialized");
    }
    public void service(ServletRequest request, ServletResponse response) throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.println("<b> Hello simple servlet </b>");
        out.println("</body></html>");
    }
    public void destroy() {
        System.out.println("Servlet is destroyed");
    }
    public ServletConfig getServletConfig() {
        return config;
    }
    public String getServletInfo() {
        return "Implementing Servlet Interface";
    }
}
```

The javax.servlet package

This package contains a number of interfaces and classes that establish the framework in which servlets operate. Following are some of the interfaces and classes.

Interface	Description
Servlet	Declares life cycle methods for a servlet.
ServletConfig	Allows servlets to get initialization parameters.
ServletRequest	Used to read data from a client request.
ServletResponse	Used to write data to a client response.

Class	Description
GenericServlet	Implements the Servlet and ServletConfig interfaces.
ServletInputStream	Provides an input stream for reading requests.
ServletOutputStream	Provides an output stream for writing response.
ServletException	Indicates a servlet error occurred.

The javax.servlet.http package

This package contains a number of interfaces and classes that are commonly used by servlet developers. Following are some of the interfaces and classes.

Interface	Description
HttpServletRequest	Enables servlets to read data from HTTP request.
HttpServletResponse	Enables servlets to write data to an HTTP response.
HttpSession	Allows session data to be read and write.

Class	Description
Cookie	Allow state information to be stored on client machine.
HttpServlet	Provides methods to handle HTTPrequests and responses.
HttpSessionEvent	Encapsulates a session – changed event

Writing Servlet Programs

The servlet program can be created by three ways:

I. By implementing servlet interface
II. By inheriting GenericServlet class
III. By inhenriting HttpServlet class

The mostly used approach is by extending HttpServlet because it provides http request specific method such as doGet(), doPost(), doHead() etc.

There are six steps to write Servlet Program (Here we are using **apache tomcat server**)

I. Create a directory structure
II. Create a Servlet
III. Compile the Servlet
IV. Create a deployment descriptor
V. Start the server and deploy the project
VI. Access the servlet

Simple servlet program

```
import java.io.*;
import jakarta.servlet.*;

public class HelloServlet extends GenericServlet {
    public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException {
        // Set content type for the response
        response.setContentType("text/html");

        // Get PrintWriter object to write HTML response
        PrintWriter pw = response.getWriter();
        pw.println("<B>Hello students. I am Nishanta</B>");
        pw.close();
    }
}
```

Reading Servlet Parameter

The **servletRequest** interface includes method that allow you to read the names and values of parameter that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in **Register.html** and a servlet is defined in **RegisterServlet.java.**

The HTML source code for **Register.html** is shown below. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is phone. There is also a submit button.

Register.html

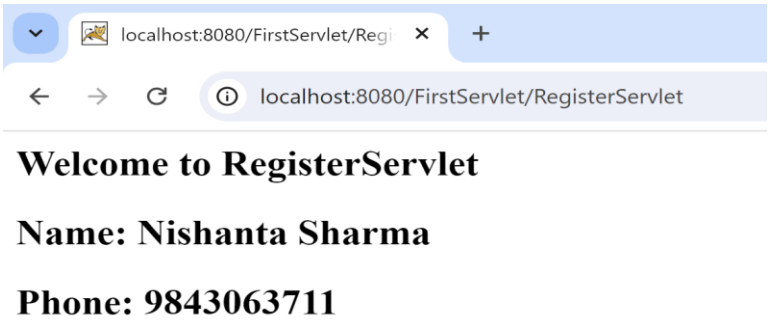
```
<html>
<body>
    <center>
        <form name="Form1" method="post" action="RegisterServlet">
            <table>
                <tr>
                    <td><b>Employee:</b></td>
                    <td><input type="text" name="employee" size="25"></td>
                </tr>
                <tr>
                    <td><b>Phone:</b></td>
                    <td><input type="text" name="phone" size="25"></td>
                </tr>
            </table>
            <input type="submit" value="Submit">
        </form>
    </center>
</body>
</html>
```

The source code for **RegisterServlet.java** is shown below. The **service()** method is overridden to process client requests. The **getParameterNames()** method returns an enumeration of the parameter names. There are processed in a loop. You can see that the parameter name and value are output to the client. The parameter value is obtained via the **getParameter()** method.

RegisterServlet.java

```
import java.io.*;
import java.util.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class RegisterServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String name=request.getParameter("employee");
        String phone= request.getParameter("phone");

        out.println("<h2>Welcome to RegisterServlet</h2>");
        out.println("<h2>Name: "+ name +"</h2>");
        out.println("<h2>Phone: "+ phone +"</h2>");
    }
}
```

Output



Handline HTTP Request and Response

The HTTPServlet class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are **doDelete(), doGet(), doHead(), doOption(), doPost()** and **doTrace().** However, the GET and POST requests are commonly used when handling forms input.

Handling HTTP Get Requests

Here we develop a servlet that handles an HTTP Get request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorGet.html** and a servlet is defined in **ColorGetServlet.java**. The source code for **ColorGet.html** is shown below. It defines a form that contain a select element and a submit button.

ColorGet.html

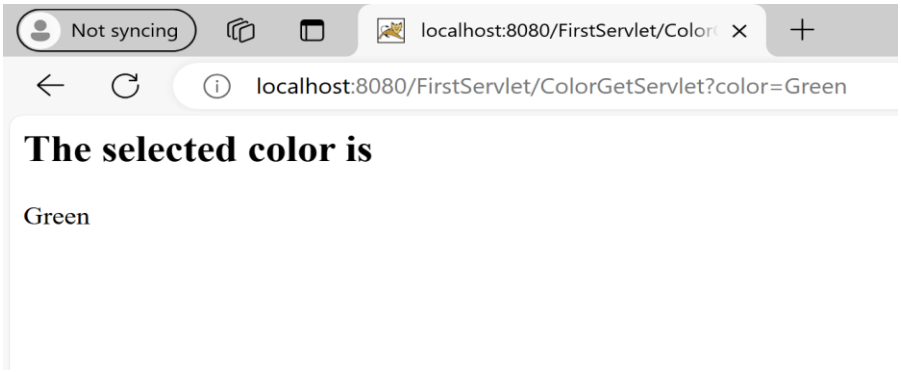
```
<html>
<body>
    <center>
        <form name="Form1" method="get" action="ColorGetServlet">
            <b>Color:</b>
            <select name="color" size="1">
                <option value="Red">Red</option>
                <option value="Green">Green</option>
                <option value="Blue">Blue</option>
            </select>
            <input type="submit" value="Submit">
        </form>
    </center>
</body>
</html>
```

the source code for ColorGetServlet.java is shown below. The doGet() method is overridden to process any HTTP GET request that are sent to this servlet. It uses getParameter() method of HttpServletRequest to obtain the selection that was made by the user.

ColorGetServlet.java

```
import java.io.*;
import java.util.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class ColorGetServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String color=request.getParameter("color");

        out.println("<h2>The selected color is</h2>");
        out.println(color);
        out.close();
    }
}
```

Output



Handling HTTP POST Request

Here we will develop a servlet that handles an **HTTP POST** request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorPost.html** and a servlet is defined in **ColorPostServlet.java.**

The HTML source code for **ColorPost.html** is shown below. It is identical to **ColorGet.html** except that the method parameter for the form tag explicitly specifies that the **POST** method should be used and the action parameter for the form tag specifies a different servlet.

ColorGet.html

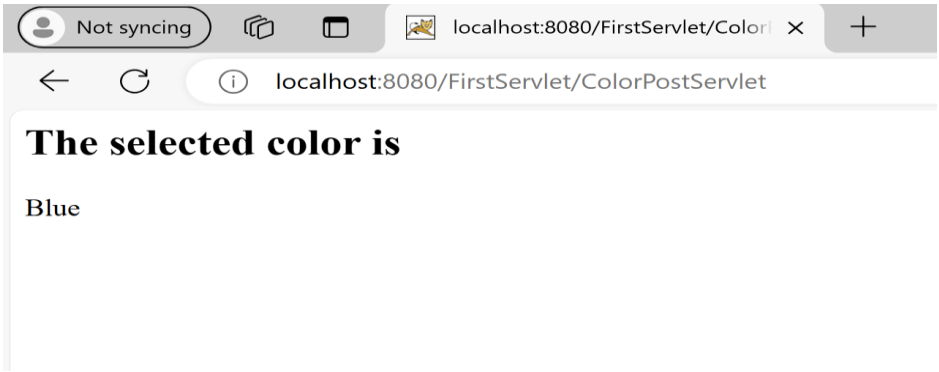
```
<html>
<body>
    <center>
        <form name="Form1" method="post" action="ColorPostServlet">
            <b>Color:</b>
            <select name="color" size="1">
                <option value="Red">Red</option>
                <option value="Green">Green</option>
                <option value="Blue">Blue</option>
            </select>
            <input type="submit" value="Submit">
        </form>
    </center>
</body>
</html>
```

The source code for **ColorPostServlet.java** is shown below. The **doPost()** method is overridden to process any HTTP POST request that are sent to this servlet. It uses the **getParameter()** method of **HTTPServletRequest** to obtain the selection that was made by the user. A response is then formulated.

ColorPostServlet.java

```
1  import java.io.*;
2  import java.util.*;
3  import jakarta.servlet.*;
4  import jakarta.servlet.http.*;
5
6  public class ColorPostServlet extends HttpServlet
7  {
8      public void doPost(HttpServletRequest request, HttpServletResponse response)
9              throws ServletException, IOException
10     {
11        response.setContentType("text/html");
12        PrintWriter out = response.getWriter();
13
14        String color=request.getParameter("color");
15
16        out.println("<h2>The selected color is</h2>");
17        out.println(color);
18        out.close();
19     }
20 }
```

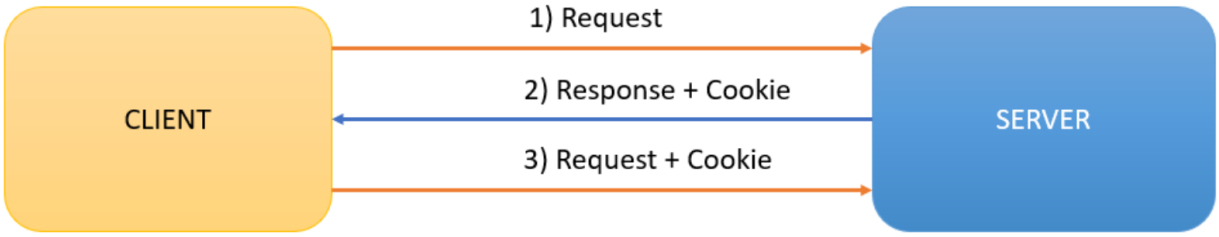
Output



Handling cookie in Servlet

A cookie is a small piece of information that is persisted between the multiple client request. A cookie has a name, a single value and optional attributes such as comment, path and domain qualifiers, a maximum age and a version number. There are three steps involved in identifying returining user:

- Server script sends a set of cookies to the browser. For example name, age or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server use that information to identify the user.



Advantage of cookie

- Simplest techniques of maintaining the state.
- Cookies are maintained at a client side.

Disadvantage of cookie

- It will not work if cookie is disabled from the browser.
- Only textual information can be set in cookie object.

Constructor of cookie class

Constructor	**Description**
Cookie()	Construct a cookie.
Cookie(String name, string val)	Construct a cookie with a specified name and value.

Useful methods of cookie class

Method	**Description**
setMaxAge(int expiry)	Sets the maximum age of the cookie in seconds
String getName()	Returns the name of the cookie
String getValue()	Returns the value of the cookie
Void setName(String name)	Change the name of the cookie.
Void setValue(String value)	Change the value of the cookie

Methods required for using cookie

- **Public void addCookie(Cookie ck)**

Method of HttpServletResponse interface is used to add cookie in response object.

- **Public cookie[] getCookies()**

Method of HttpServletRequest interface is used to return all the cookie from the browser.

For example

The HTML source code **for Cookie.html** is show below. This page contains text fields in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to **SetServlet.java**

Cookie.html

```
<html>
<body>
    <center>
        <form name="Form1" method="post" action="SetServlet">
            Name:<input type="text" name="unname">
            <input type="submit" value="Submit">
        </form>
    </center>
</body>
</html>
```

The source code for **SetServlet.java** is shown below. It gets the value of a parameter named **"unname".** Then it creates a Cookie object that has the name **"name"** and contains the value of the **"unname"** parameter. The cookie is the added to the header of HTTP response via the **addCookie()** method. A feedback message is then written to the browser.

SetServlet.java

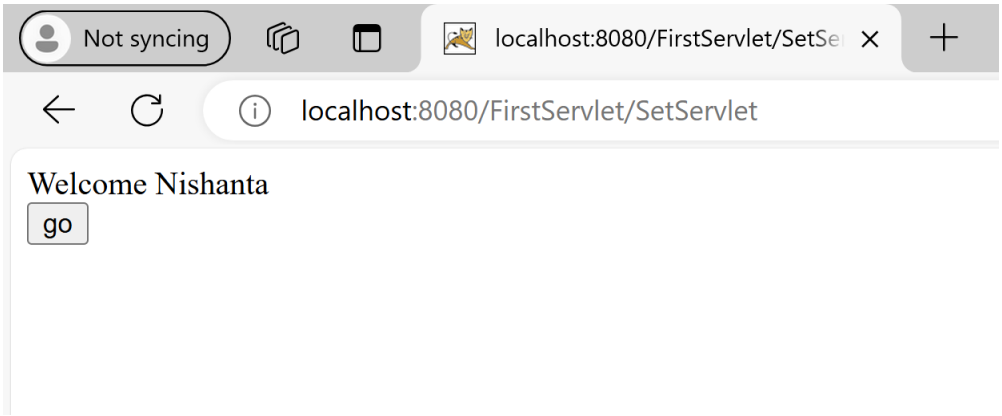
```
1  import java.io.*;
2  import java.util.*;
3  import jakarta.servlet.*;
4  import jakarta.servlet.http.*;
5
6  public class SetServlet extends HttpServlet
7  {
8      public void doPost(HttpServletRequest request, HttpServletResponse response)
9              throws ServletException, IOException
10     {
11        response.setContentType("text/html");
12        PrintWriter out = response.getWriter();
13
14        String name = request.getParameter("unname");
15        out.println("Welcome " + name);
16
17        //creating submit button
18         out.println("<form action='ReadServlet' method='post'>");
19         out.println("<input type='submit' value='go'>");
20         out.println("</form>");
21
22         out.println("</body>");
23         out.println("</html>");
24
25         //create cookie
26        Cookie ck = new Cookie("name", name);
27        //add cookie to HTTP respomse
28        response.addCookie(ck);
29
30        out.close();
31    }
32 }
```

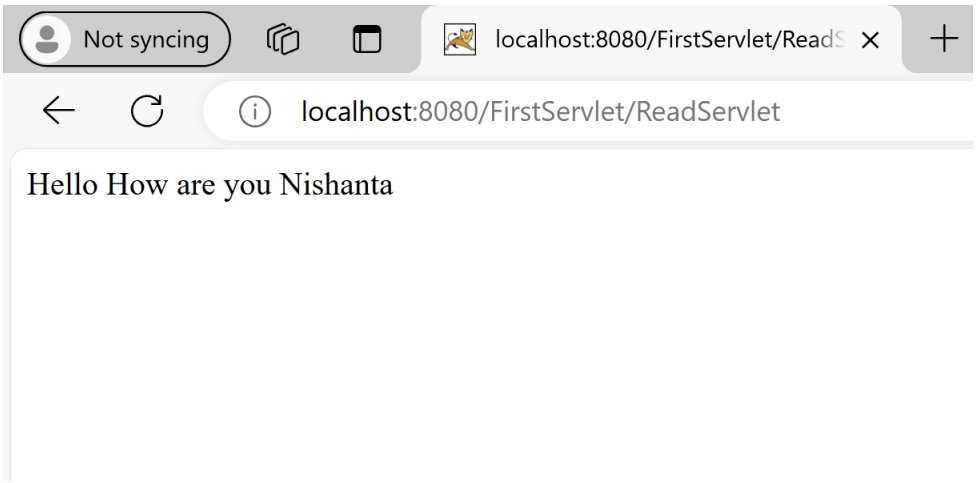
The source code for **ReadServlet.java** is shown below. It involves the **getCookies()** method to read any cookies that are included in the HTTP Get request. The names and value of these cookies are then written to the HTTP response.

ReadServlet.java

```java
import java.io.*;
import java.util.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class ReadServlet extends HttpServlet
{
    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
       response.setContentType("text/html");
       PrintWriter pw = response.getWriter();

       Cookie[] cook = request.getCookies();

        if(cook!=null)
        {
          pw.println("Hello How are you "+cook[0].getValue());
          pw.close();
        }
    }
}
```

Output





Session Tracking

HTTP is a "stateless" protocol which means each time a client retrieves a Web page, the client opens a separate connection to the web server and the server automatically does not keep any record of previous client request. Thus, each time user request to the server, server treats the request as the new request. So we need to maintain the state of an user to recognize particular user. Session simply means a particular interval of time.

Session tracking is a way to maintain state(data) of an user. It is also known as session management in servlet. There are four techniques used in session tracking.

1. Cookies
2. Hidden Form field
3. URL rewriting
4. HttpSession

HttpSession object

Servlet provides HttpSession interface which provides a way to identify a user across more than one page request or visit to a web site and to store information about the user. The servlet container uses this interface to create a session between a HTTP client and an HTTP server. The session persists for a specified time period, across more than one connection or page request from the user. We can get **HttpSession** object by calling the public method **getSession()** of **HttpServletRequest** as below

HttpSession session = request.getSession();

You need to call **request.getSession()** before you send any document content to the client. Commonly used methods of HttpSession interface.

1. **Public String getId():** Returns a string containing the unique Identifier value.
2. **Public long getCreationTime():** Returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.

3. **Public long getLstAccessedTime():** Returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
4. **Public void invalidate():** Invalidates this session then unbinds any objects bound to it.

Session.html

```
<html>
<body>
    <center>
        <form name="Form1" action="SetSessionServlet">
            Name:<input type="text" name="unname">
            <input type="submit" value="Submit">
        </form>
    </center>
</body>
</html>
```

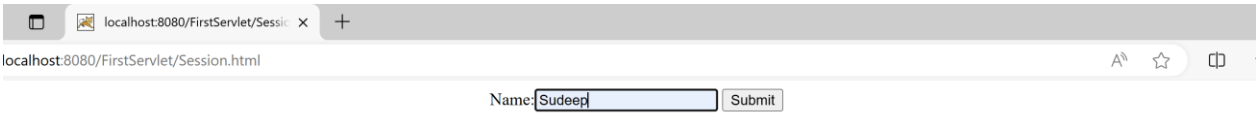
SetSessionServlet.java

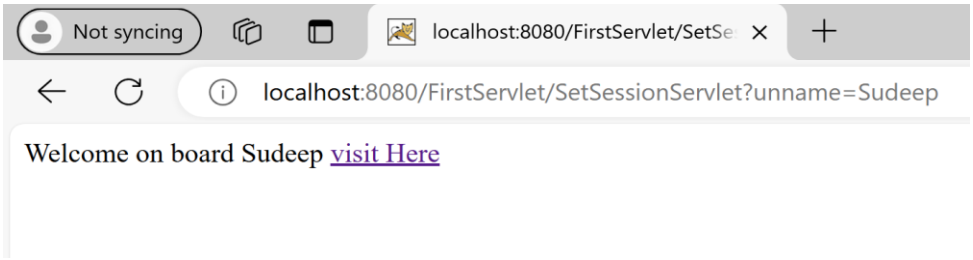
```
import java.io.*;
import java.util.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class SetSessionServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String name = request.getParameter("unname");
        out.println("Welcome on board " + name);

        // get the HttpSession object
        HttpSession hs = request.getSession();
        hs.setAttribute("unname",name);

        out.println("<a href ='ReadSessionServlet'> visit Here </a>");
        out.close();
    }
}
```

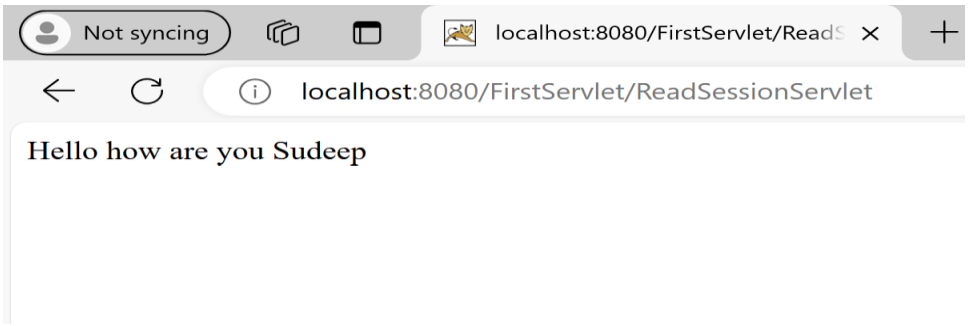
ReadSessionServlet.java

```java
import java.io.*;
import java.util.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class ReadSessionServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
       response.setContentType("text/html");
       PrintWriter pw = response.getWriter();

       HttpSession session = request.getSession(false);
       String name = (String)session.getAttribute("unname");
       pw.print("Hello how are you "+name);
       pw.close();
    }
}
```

Output







Database access with servlets

Database access with servlets is same as JDBC , that we have discussed earlier. Only the difference is that we have to put mysql-connector.jar file in libraries directory.

Insert.html

```
<html>
<body>
    <center>
        <form name="Form1" method="get" action="InsertServlet">
            <table>
                <tr>
                    <td><b>Roll_no</b></td>
                    <td><input type="text" name="roll_no" size="25"></td>
                </tr>
                <tr>
                    <td><b>Name</b></td>
                    <td><input type="text" name="name" size="25"></td>
                </tr>
                <tr>
                    <td><b>Email</b></td>
                    <td><input type="text" name="email" size="25"></td>
                </tr>
                <tr>
                    <td><b>Address</b></td>
                    <td><input type="text" name="address" size="25"></td>
                </tr>
            </table>
            <input type="submit" value="Submit">
            <input type="submit" value="reset">
        </form>
    </center>
</body>
</html>
```

InsertServlet.java

```java
import java.io.*;
import java.sql.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class InsertServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException
    {
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();

       String Roll_no = request.getParameter("roll_no");
       String Name = request.getParameter("name");
       String Email = request.getParameter("email");
       String Address = request.getParameter("address");

        try {
            String driver = "com.mysql.cj.jdbc.Driver";
            String databaseUrl = "jdbc:mysql://localhost:3306/nist_6th";
            String username = "root";
            String password = "admin";
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(databaseUrl, username, password);
            System.out.println("Database connected");

            Statement statement = conn.createStatement();
            statement.executeUpdate("insert into nist_student values('"+Roll_no+"','"
                    + "'"+Name+"', '"+Email+"','"+Address+"' )");
            out.println("<h1>your data is successfully submitted</h1>");
        }
        catch(Exception e)
        {
            System.out.println("Some error: " + e);
        }
    }
}
```

Output

Result Grid | Filter Rows: | Export:

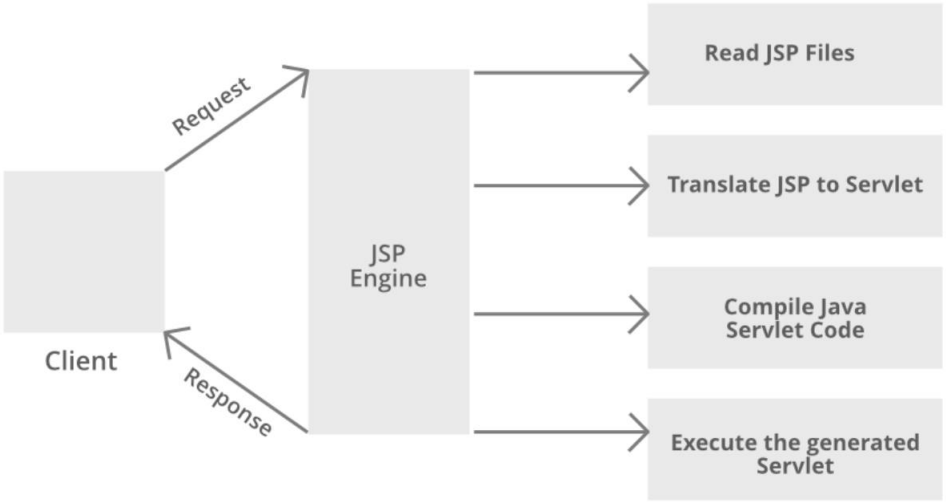
roll_no	name	email	address
1	Nishanta	nishanta@gmail.com	chapagaun
2	eliza	eliza@gmail.com	mulpani
3	sarada	sarada@gmail.com	Tahakhel
4	pradip	pradip@gmail.com	Palpa
21	Shshir	shishir@gmail.com	Raniban
21	Shshir	shishir@gmail.com	Raniban
6	Bishal	Bishal@gmail.com	Sarlahi

JSP(Java Server Page)

Java Server Page (JSP) is a server side programming technology that enables the creation of dynamic, platform – independent method for building web based applications. JSP have access to the entire family of java APIs, including the JDBC API to access enterprise databases. It can be thought of as an extension to servlet because it provides more functionality than servlet. A JSP page consists of HTML tags and JSP tags. The JSP pages are easier to maintain than servlet because we can separate designing and development.

Web developers write JSPs as text files that combine HTML or XHTML code, XML elements and embedded JSP actions and commands.

JSP Process



- The client navigates to a file ending with the ***.jsp extension*** and the browser initiates an HTTP request to the webserver. For example, the user enters the login details and submits the button. The browser requests a status.jsp page from the webserver.
- If the compiled version of JSP exists in the web server, it returns the file. Otherwise, the request is forwarded to the JSP Engine. This is done by recognizing the URL ending with ***.jsp*** extension.
- The JSP Engine loads the JSP file and translates the JSP to Servlet(Java code). This is done by converting all the template text into println() statements and JSP elements to Java code. This process is called **translation.**
- The JSP engine compiles the Servlet to an executable **.class** file. It is forwarded to the Servlet engine. This process is called **compilation** or **request processing phase.**
- The **.class** file is executed by the Servlet engine which is a part of the Web Server. The output is an HTML file. The Servlet engine passes the output as an HTTP response to the webserver.
- The web server forwards the HTML file to the client's browser.

JSP Access Model

In JSP, the access model refers to the way in which the built – in objects and other resources are made available to the JSP page. There are three main access models in JSP, which are used according to the need of application.

i. **Page – Base Access**

In this model, the built-in objects are made available to the JSP page through local variables.

Syntax: request.getParameter("ParaName")

ii. **Action – Based Access**

In this mode, the built – in objects are made available to the JSP through the use of custom tags.

iii. **Bean – Based Access**

In this model, the built – in objects are made available to the JSP page through JavaBeans.

Syntax: beanName.propertyName

JSP Syntax

There are five main tags that all together form JSP syntax. These tags are Declaration tags, Expression tags, Directive tags, Scriptlet tags, and Action tags.

Declaration tag: This tag is used to declare variable and methods

Syntax: < % ! Variable or method declaration %>

Example

```
< % !
        private int counter = 0;
        private String getAccount(int accountNO);
 % >
```

Expression tag: It is used to insert a single java expression directly into the response message by placing inside a out.print() method.

Syntax: < % = Java Expression % >

Example

Date : < % = new java.util.Date() % >

Directive tags: The directive tags gices special information about the page to the JSP Engine.

Syntax: < %@ directive attribute = "value" %>

Example:

< % @ page language = "java" % >

< % @page import = "java.util.* " % >

Scriptlet tag: This tag is used to execute java code in JSP.

Syntax: < % java code % >

Example:

```
< %
        String username = "Nishanta";
        out.println(username);
% >
```

Action tag: The action tag are used to control the flow between pages and to use java beans. Some of the action tags supported by JSP are : jsp:forward, jsp:include, jsp:param.

Syntax: <jsp : forward page = "printdate.jsp" />

JSP comments: The JSP comment is used when we want to hide or "comment out" part of the JSP page.

Syntax: <% -- comment -- %>

Example:

< % - - This is JSP Comment - - % >

JSP Implicit Objects

In JSP, implicit objects are objects that are automatically available to the JSP page and do not need to be explicitly created. Some of the implicit objects in JSP are as follows:

Request object: It represents the HTTP request received by the web server and provides access to request – specific information such as parameters, headers and session attributes.

Response object: It represents the HTTP response that will be sent back to the client and allows the JSP to control the content and headers of the response.

Page context object: Provides access to various components of the JSP environment, such as the ServletContext, HttpSession and JspWriters.

Session object: Represents the HTTP session for a user and allows the JSP to access and modify session – level data.

Page object: Represents the current JSP page and provides access to the pages attributes and methods.

These implicit objects can be accessed using standard java syntax within a JSP page. For example, to access the request object, we would use 'request' and to access a parameter passed in the request we would use ***'request.getParameter("paramName")'***.

Object Scope

In JSP, object scope refers to the availability of an object within the page, application or session. The scope of an object determines where it can be accessed and for how long it is available. There are four object scopes in JSP:

Page Scope: an object with page scope is available only within the current JSP page and is not accessible from other pages.

< jsp:useBean id = "employee" class="EmployeeBean" scope = "page" />

Request Scope: An object with request scope is available within the current request and response cycle and is not accessible from other requests

< jsp:useBean id = "employee" class="EmployeeBean" scope = "request" />

Session Scope: An object with session scope is available to all pages within a user's session and is not accessible from other sessions.

<jsp:useBean id = "employee" class="EmployeeBean" scope = "session" />

Application Scope: An object with application scope is available to all pages in the web application and is not accessible from other applications.

<jsp:useBean id = "employee" class="EmployeeBean" scope = "application" />

Simple JSP program

```
<html>
    <head>
        <title>Simple JSP Page</title>
    </head>
    <body>
        <h1>Displaying "Tribhuwan University" 10 times </h1>
         <table>
            <%
             for(int i = 1; i<=10; i++)
            { %>
                 <tr>
                <td> Tribhuwan University</td>
                </tr>
            <% } %>
        </table>
    </body>
</html>
```

Displaying "Tribhuwan University" 10 times

Tribhuwan University
Tribhuwan University
Tribhuwan University
Tribhuwan University
Tribhuwan University
Tribhuwan University
Tribhuwan University
Tribhuwan University
Tribhuwan University
Tribhuwan University

Form Processing in JSP

Form processing in JSP is similar to form processing in servlets. We can use request object to read parameter values from HTML file to another or another JSP file. Methods supported by request object are similar to methods used in servlets.

processFile.html

```
<html>
<body>
    <center>
        <form name="Form1" method="post" action="ProcessFile.jsp">
            <table>
                <tr>
                    <td><b>Employee:</b></td>
                    <td><input type="text" name="employee" size="25"></td>
                </tr>
                <tr>
                    <td><b>Phone:</b></td>
                    <td><input type="text" name="phone" size="25"></td>
                </tr>
            </table>
            <input type="submit" value="Submit">
        </form>
    </center>
</body>
</html>
```

ProcessFile.jsp

```
<html>
    <body>
        <%
            String Name = request.getParameter("employee");
            String Phone = request.getParameter("phone");
        %>
    <center>
        <h2> Hello , <%=Name+" your phone number is " +Phone %> </h2>
    </center>
    </body>
</html>
```

Output

Database Access with JSP

We also access database by using JSP. All JDBC APIs can be accessed from JSP. We have put all codes related to database access within the expression tag,

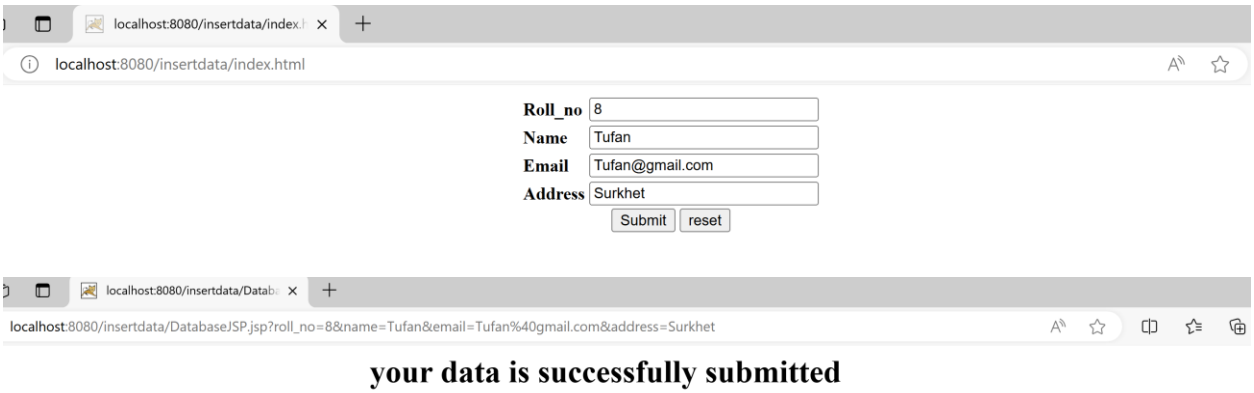
Index.html

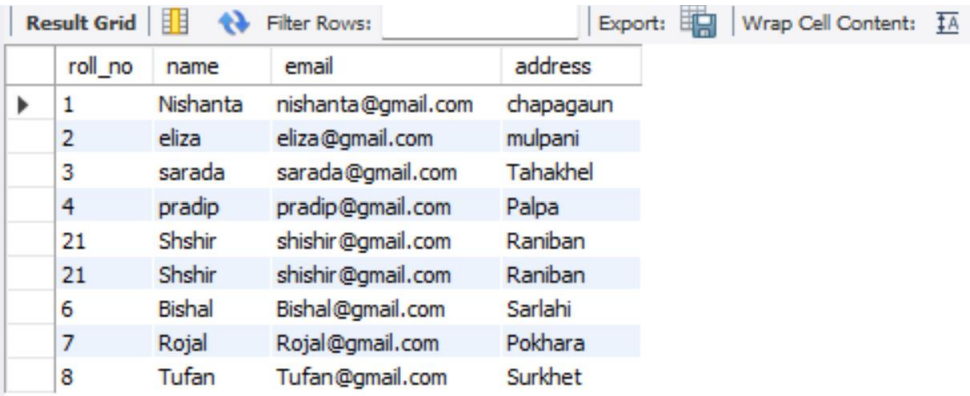
```
<html>
<body>
    <center>
        <form name="Form1" method="get" action="DatabaseJSP.jsp">
            <table>
                <tr>
                    <td><b>Roll_no</b></td>
                    <td><input type="text" name="roll_no" size="25"></td>
                </tr>
                <tr>
                    <td><b>Name</b></td>
                    <td><input type="text" name="name" size="25"></td>
                </tr>
                <tr>
                    <td><b>Email</b></td>
                    <td><input type="text" name="email" size="25"></td>
                </tr>
                <tr>
                    <td><b>Address</b></td>
                    <td><input type="text" name="address" size="25"></td>
                </tr>
            </table>
            <input type="submit" value="Submit">
            <input type="submit" value="reset">
        </form>
    </center>
</body>
</html>
```

DatabaseJSP.jsp

```jsp
<html>
    <body>
        <%
            String Roll_no = request.getParameter("roll_no");
            String Name = request.getParameter("name");
            String Email = request.getParameter("email");
            String Address = request.getParameter("address");
        %>
    <center>
        <%@page import = "java.sql.*"%>
        <%
            String driver = "com.mysql.cj.jdbc.Driver";
            String databaseUrl = "jdbc:mysql://localhost:3306/nist_6th";
            String username = "root";
            String password = "admin";
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(databaseUrl, username, password);
            System.out.println("Database connected");

            Statement statement = conn.createStatement();
            statement.executeUpdate("insert into nist_student values('"+Roll_no+"',"
                    + "'"+Name+"', '"+Email+"','"+Address+"' )");
            out.println("<h1>your data is successfully submitted</h1>");
        %>
    </center>
    </body>
</html>
```

Output





Result Grid | Filter Rows: | Export: | Wrap Cell Content:

roll_no	name	email	address
1	Nishanta	nishanta@gmail.com	chapagaun
2	eliza	eliza@gmail.com	mulpani
3	sarada	sarada@gmail.com	Tahakhel
4	pradip	pradip@gmail.com	Palpa
21	Shshir	shishir@gmail.com	Raniban
21	Shshir	shishir@gmail.com	Raniban
6	Bishal	Bishal@gmail.com	Sarlahi
7	Rojal	Rojal@gmail.com	Pokhara
8	Tufan	Tufan@gmail.com	Surkhet

Difference between Servlet and JSP

Servlets	JSP
Servlet is a pure java code.	JSP is a tag based approach.
We write HTML in Servlet code.	We write JSP code in HTML.
Servlet is faster than JSP.	JSP is slower than Servlet.
Writing code for servlet is harder.	Writing code for JSP is easier.
Servlet can accept all protocol request.	JSP only accept http request.
Modification in Servlet is a time consuming task because it includes reloading.	JSP modification is fast, we just need to click refresh button.
Servlet do not have implicit objects.	JSP have implicit objects.
In MVC pattern, servlet act as a controller.	In MVC pattern, JSP act as a view.

Introduction to java Web Framework

A web framework is a collection of libraries and tools that make it easier to develop web applications. Web framework provide a standard way to build and deploy web applications and they can help to save time and effort. There are many java web frameworks available and each has its own strengths and limitations. Some popular java web frameworks include:

1. **Spring**

Spring is widely used framework for building Java applications. It provides a range of features including dependency injection, data access and web support.

2. **Hibernate**

Hibernate is an object – relational mapping (ORM) framework that simplifies the process of storing and retriving data from a database. It is often used with spring.

3. **Structs**

Structs is an older web framework that is still used by some developers. It uses a MVC architecture and provides support for actions, forms and validation.

4. **JSF**

It is a framework for building user interfaces for java web applications. It provides support for actions, forms, validation. JSF stands for JavaServer Faces.

5. **Play**

Play is a modern web framework that is designed to be easy to use and scalable. It uses an event – driven architecture and supports real time web applications.

Chapter 5

Remote Method Invocation

- Java Remote Method Invocation (RMI) is an object – oriented Remote Procedure Call (RPC) techniques. It allows us to invoke methods on an object that exists in a different address space.
- This address space may exist on the same computer or even on a different computer connected to the source computer by a network.
- So, it enable objects, distributed in different computers, to communicate with one another. Although it provides a simple but elegant model, it is remarkably powerful and can be used to invoke remote methods easily and in a natural way.

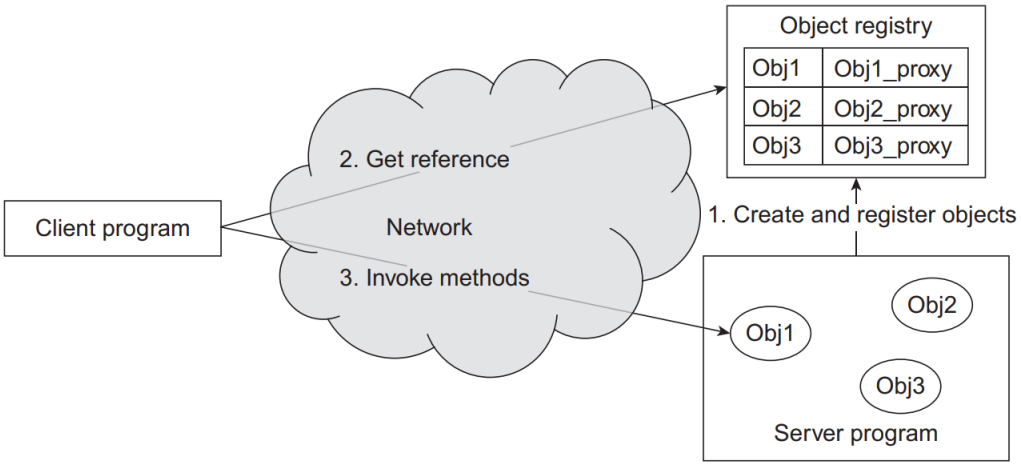


Fig: RMI programming model

- The underlying communication between clients and server in this model takes place seamlessly using sockets. It means that the message from the client is not method invocation in the OO sense. Instead, it is a stream of data that must be interpreted by the server before it can invoke method on the target object.
- In general, RMI architecture was developed taking the following goals into consideration
    - A primary goal of java RMI technology was to allow programmers to develop distributed java programs with the same syntax and semantics used for non – distributes programs.
    - Another goal was to create a distributed object model that fits the java programming language and the local object model naturally. RMI architects finally succeeded in creating such a powerful system that extends the safety and robustness of the java architecture to the distributed computing world.

Application Components

Three entities are often involved in an RMI application: a server program, a client program and object registry.

Server

This is a program that typically creates a remote object to be used for method invocation. This object is an ordinary object except that its class implements a Java RMI interface. Upon creation, the object is exported and registered with a separate application called object registry.

Client

A client program typically consults the object registry to get a reference (handle) to a remote object with a specified name. It can then invoke methods on the remote object using this reference (handle) as if the object is stored in the client's own address space. The RMI handles the details of communication (using sockets) between the client and the server and passes information back and forth. Note that this complex communication procedure is absolutely hidden to the client and server applications.

Object Registry

It is essentially a table of objects. Each entry of the table maps the object name to its proxy known as stub. The server registers the stub buy name to the object registry. Once the stub is registered to the object registry successfully, the object is said to be available for other's use. Clients can now get a reference (handle) to the remote object (actually stub) from this registry and can invoke methods on it.

Goals of RMI

- To minimize the complexity of the application.
- To preserve type safety.
- Distributed garbage collection.
- Minimize the difference between working with local and remote objects.

RMI architecture

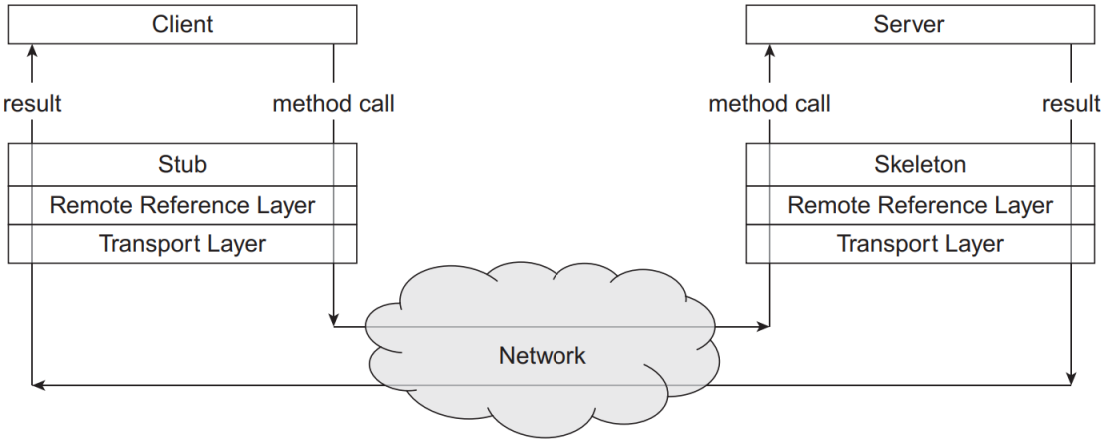


Fig: RMI Architecture

In an RMI application, we write two programs , a server programs and a client program. Inside the server program, a remote object is created and reference of that object is made available for the client. The client program requests the remote objects on the server and tries to invoke its methods.

Application Layer

In this layer client and server are involved in communication. The java program on client side communicates with the java program on server side.

Stub

The stub is an object that acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

- It initiates a connection with remote Virtual Machines (JVM).
- It writes and transmits the parameter to remote Virtual Machine.
- It waits for the result.
- It reads the return value.
- Finally returns the value to the caller.

Skeleton

The skeleton is an object, that acts as a gateway for the server side object. All the incoming requests are routed through it, when the skeleton receives the incoming requests, it does the following tasks:

- It reads the parameter for the remote method.
- It involves the method on the actual remote object.
- It writes and transmits the result to the caller.

Remote Reference Layer (RRL)

The remote reference layer defines and supports the invocation semantics of the RMI connection. This layer maintains the session during the method call. This layer provides a **RemoteRef** object that represents the link to the remote service implementation objects. The stub objects use the **invoke()** method in **RemoteRef** to forward the method call. The **RemoteRef** object understands the invocation semantics for remote service.

Transport layer

It is responsible for setting up communication between two JVMs. All connections are stream – based network connections that use TCP/IP. Even if two JVMs are running on the same physical computer, they connect through their hosts computer's TCP/IP network protocol stack.

Steps of writing RMI application

Step 1: Create the remote interface

For creating the remote interface, extend the Remote interface and declare the RemoteExeption with all the methods of the remote interface.

Step 2: Provide the implementation of the Remote Interface

There are two ways for the implementation of the remote interface

i. By writing implementation class separately
ii. By directly make the server program implement the interface.

For providing the implementation of the Remote interface, we need to

- Either extend the UnicastRemoteObject class
- Or use the exportObject() method of the UnicastRemoteObject class.

Step 3: Create, Define and Run Server application

Create a client class from where to invoke the remote object. Create a remote object. Get the RMI registry using the getRegistry() method of the LocateRegistry class which belongs to the package java.rmi.registry. Bind the remote object created to the registry using the Rebind() method of the class named Registry.

Step 4: Create, Define and Run Client application

Create a client class. Get the RMI registry using the getRegistry() method of the LocateRegistry class which belongs to the package java.rmi.registry. Fetch the object from the registry using the method lookup() of the class Registry which belongs to the package java.rmi.registry. Invoke the required method using the obtained remote object.

Example

Adder.java

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Adder extends Remote
{
    int add(int n1, int n2) throws RemoteException;
}
```

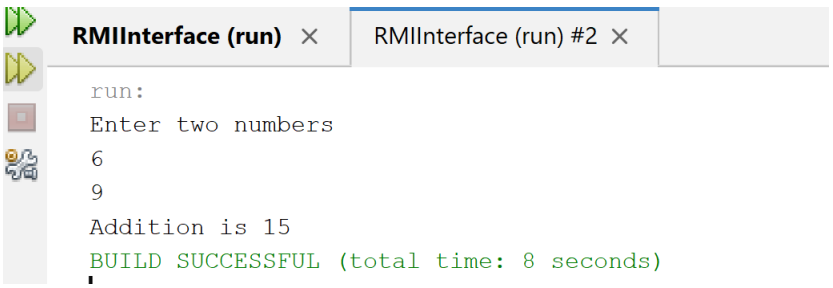
RMIServer.java

```java
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;

public class RMIServer extends UnicastRemoteObject implements Adder
{
    public RMIServer() throws RemoteException
    {
        super();
    }

    @Override
    public int add(int n1, int n2) throws RemoteException
    {
        return n1 + n2;
    }

    public static void main(String[] args)
    {
        try
        {
            Registry reg = LocateRegistry.createRegistry(9999);
            reg.rebind("hi server", new RMIServer());
            System.out.println("Server is ready");
        }
        catch (RemoteException e)
        {
            System.out.println("Exception: " + e);
        }
    }
}
```

RMIClient.java

```java
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Scanner;

public class RMIClient
{
    public static void main(String[] args) throws RemoteException
    {
        RMIClient rc = new RMIClient();
        rc.connectRemote();
    }

    private void connectRemote() throws RemoteException
    {
        try
        {
            Scanner sc = new Scanner(System.in);
            Registry reg = LocateRegistry.getRegistry("localhost", 9999);
            Adder ad = (Adder) reg.lookup("hi server");
            System.out.println("Enter two numbers");
            int a = sc.nextInt();
            int b = sc.nextInt();
            System.out.println("Addition is " + ad.add(a, b));
        }
        catch (RemoteException | NotBoundException e)
        {
            System.out.println("Exception: " + e);
        }
    }
}
```

Output



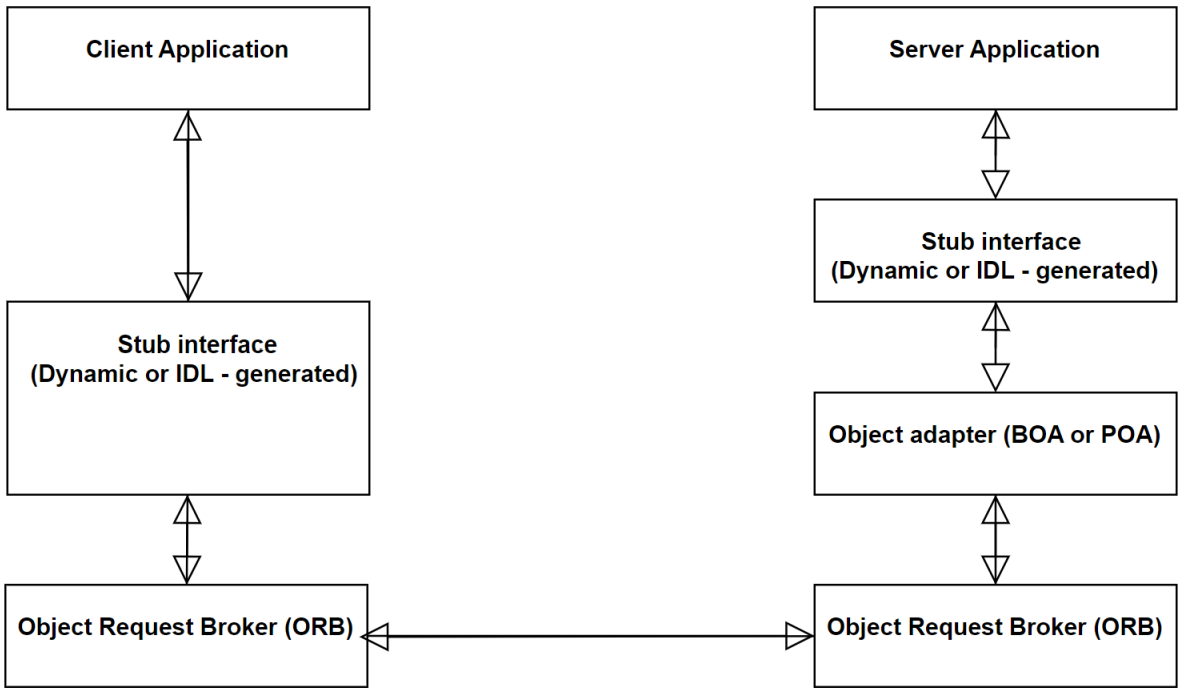
Introduction to CORBA

- The Common Object Request Broker Architecture (CORBA) is a standard developed by the Object Management Group (OMG) to provide interoperability among distributed objects.
- CORBA is the world's leading middleware solution enabling the exchange of information, independent of hardware platforms, programming languages and operating systems.
- CORBA is often described as a "software bus" because it is a software – based communications interface through which objects are located and accessed.

RMI vs CORBA

RMI	CORBA
RMI is a Java Specific technology	CORBA is independent of programming language.
It uses java interface for implementation.	It uses Interface Definition Language(IDL) to separate interface from implementation.
RMI programs can download new classes from remote JVMs.	CORBA does not have this code sharing mechanism.
RMI passes objects by remote reference or by value.	CORBA passes objects by reference.
Distributed garbage collection is available.	Distributed garbage collection is not available.
Generally simpler to use.	More complicated.
It is free of cost.	Cost money according to vendor.

CORBA Architecture



The CORBA architecture consists of following components

Object Request Broker (ORB)

The ORB is the core component of the CORBA architecture. It is responsible for handling communication between objects and for locating and activating remote objects.

IDL compiler

The IDL compiler is used to generate code from IDL interfaces. It generates code for the client – side and server side stubs and skeletons, which are used to communicate with the remote object.

Client

The client is a program that invokes operations on a remote object. It uses the client – side stub to communicate with the ORB, which in turn communicate with server – side skeletons.

Server

The server is a program that implements the operations of a remote object. It uses the server – side skeleton to communicate with the ORB, which in turn communicates with client – side stub.

Interface Definition Language(IDL)

The Interface Definition Language (IDL) is a language that is used to define interfaces for distributed objects in the CORBA. IDL is a language – neutral way of specifying the interface to a remote object and it allows software components written in different programming languages to work together.

In IDL an interface is defined is a set of operations that can be invoked on a remote object. Each operation has a name return type, and a list of parameters. Here is an example of an IDL interface that defines a single operation, 'sayHello' which takes no parameter and returns a string.

```
interface Hello
{
        String sayHello();
}
```

Simple CORBA Program

```
Module HelloApp
{
        interface Hello
        {
                String sayHello(“Hello”);
        };
}
```