Frame
A top-level window (that is, a window that is not contained inside another window) is called a frame in Java. The Abstract Window Toolkit (AWT) library has a class, called Frame, for this top level. The Swing version of this class is called JFrame and extends the Frame class. Thus, the decorations (buttons, title bar, icons, and so on) are drawn by the user'swindowing system, not by Swing.
What is JFrame?
JFrame is a class of javax.swing package extended by java.awt.frame, it adds support for JFC/SWING component architecture. It is the top level window, with border and a title bar**Creating a JFrame** JFrame class has many constructors used to create a JFrame.
Following is the description.
 JFrame(): creates a frame which is invisible
 JFrame(GraphicsConfiguration gc): creates a frame with a blank title and graphics configuration of
screen device.
 JFrame(String title): creates a JFrame with a title.
 JFrame(String title, GraphicsConfiguration gc): creates a JFrame with specific Graphics configuration and specified title.
Here is a simplest example just to create a JFrame.

```
import javax.swing.*;
import java.awt.*;
public class JFrameCenterPositionTest extends JFrame {
  public JFrameCenterPositionTest() {
    setTitle("JFrameCenter Position");
    add(new JLabel("JFrame set to center of the screen", SwingConstants.CENTER),
BorderLayout.CENTER);
    setSize(400, 300);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setLocationRelativeTo(null);
    setVisible(true);
  }
  public static void main (String[] args) {
    new JFrameCenterPositionTest();}}
```

Displaying Information in the Component

Here, we show how to display information (for example button, text field etc.) inside a frame. One approach to display such information is to use JPanel. The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class.

For example,

```
import javax.swing.*;
import java.awt.*;
public class SimpleFrame{
 public static void main(String[] args) {
 EventQueue.invokeLater(()->{
 JFrame f = new JFrame();
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
 f.setSize(300,200);
 SimplePanel p = new SimplePanel();
 f.add(p);
 f.setVisible(true);
 });}}
class SimplePanel extends JPanel
{
 private JButton b;
 private JTextField t;
 public SimplePanel()
 {
 b = new JButton("Ok");
 t = new JTextField(8);
 add(b);
 add(t);
 }
}
```

AWT vs Swing

1. Java AWT is an API to develop GUI applications in Java   Swing is a part of Java Foundation Classes and is used to create various applications.
2. The components of Java AWT are heavy weighted.   The components of Java Swing are light weighted.
3. Java AWT has comparatively less functionality as compared to Swing.   Java Swing has more functionality as compared to AWT.
4. The execution time of AWT is more than Swing. The execution time of Swing is less than AWT.
5. The components of Java AWT are platform dependent.   The components of Java Swing are platform independent.
6. MVC pattern is not supported by AWT.   MVC pattern is supported by Swing.
7. AWT provides comparatively less powerful components. Swing provides more powerful components.
8. AWT components require java.awt package   Swing components requires javax.swing package
9. AWT is a thin layer of code on top of the operating system.   Swing is much larger swing also has very much richer functionality.
10. AWT stands for Abstract windows toolkit .   Swing is also called as JFC(java Foundation classes). It is part of oracle's JFC.

Swing and its components:
In Java, Swing is a set of GUI (Graphical User Interface) components that allow you to create windows, buttons, menus, and other elements for building desktop applications. Swing provides a rich set of classes for creating a wide range of user interface components. Here are some of the commonly used Swing components:
1. **JFrame**:The main window of a Java Swing application. It can contain other components like buttons, labels, etc.
2. **JPanel:A container that can be used to group other components together.
3. **JButton**: A button that can trigger an action when clicked.
4. **JLabel**: A non-editable text component used for displaying information.
5. **JTextField**:A single-line text input field where the user can type.
6. **JTextArea**: A multi-line text input field.
7. **JCheckBox**:A checkbox that can be checked or unchecked.
8. **JRadioButton**: A radio button that is part of a group. Only one radio button in a group can be selected at a time.
9. **JComboBox**:A dropdown menu allowing the user to select one option from a list.
10. **JList**:A component that displays a list of items.
11. **JScrollPane**: A component that provides a scrollable view of another component.
12. **JSplitPane**: A container that contains two components separated by a divider, allowing the user to resize them.
13. **JMenuBar**: A bar that contains menus.
14. **JMenu**:A drop-down menu in a menu bar.
20. **JSlider**: A slider that allows the user to select a value from a range.

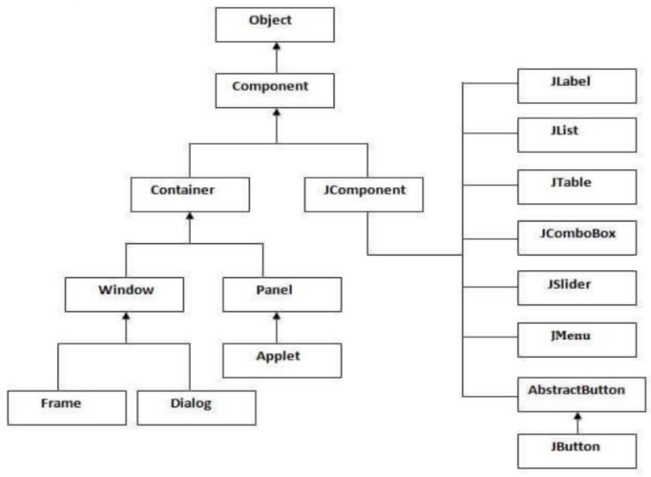
21. **JSpinner**: A component that allows the user to select a value from a predefined list.
22. **JDialog**:A popup window for showing alerts, prompts, or custom dialogs.

JButton Class
It is used to create a labelled button. Using the ActionListener it will result in some action when the button is pushed. It inherits the AbstractButton class and is platform independent.

Example:
Creating GUI by using swing
```
import javax.swing.*;
public class example{
public static void main(String args[]) {
JFrame a = new JFrame("example");
JButton b = new JButton("click me");
b.setBounds(40,90,85,20);
a.add(b);
a.setSize(300,300);
a.setLayout(null);
a.setVisible(true);
}
}
```

Swing class hierarchy

Swing is a GUI widget toolkit for Java. It is built on top of the AWT API. Also, it is a part of Oracle's Java Foundation Classes (JFC). Furthermore, Swing provides basic components such as labels, textboxes, buttons, etc. as well as advanced components such as tabbed panes, table, and, trees. Therefore, Swing provides more sophisticated components than AWT. Here, the programmer has to import javax.swing package to write a Swing application. This package provides a number of classes such as JButton, JTable, JList, JTextArea, and, JCheckBox. Swing is platform-independent and its components are lightweight. Furthermore, the components require minimum memory space. Therefore, Swing applications execute much faster. One common design pattern in development is the Model, View, Controller (MVC) pattern. Swing follows this pattern. It helps to maintain the code easily.



All the components in swing like JButton, JComboBox, JList, JLabel are inherited from the JComponent class which can be added to the container classes. Containers are the windows like frame and dialog boxes. Basic swing components are the building blocks of any gui application. Methods like setLayout override the default layout in each container. Containers like JFrame and JDialog can only add a component to itself. Following are a few components with examples to understand how we can use them.

Features of swing

Lightweight Components

Swing components are lightweight as they are written entirely in Java and do not depend on native peers (platform specific code resources). Rather, they use simple drawing primitives to render themselves on the screen. The look and the feel of the component is not controlled by the underlying operating system but by Swing itself. Thus, they are not restricted to platform-specific appearance like, rectangular or opaque shape.

Pluggable Look and Feel

The pluggable look arid feel feature allows us to tailor the look and feel of the application and applets to the standard looks like, Windows and Motif. We can even switch to different look and feel at runtime. Swing has the capability to support several look and feels, but at present, it provides support for the Windows and Motif. As the look and feel of components is controlled by Swing rather than by operating system, the feel of components can also be changed. The look and feel of a component can be separated form the logic of the component. Thus, it is possible to “plug in” a new look and feel for any given component without affecting the rest of the code.

Simple Java Swing Example

Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

File: FirstSwingExample.java

```
import javax.swing.*;
public class FirstSwingExample {
public static void main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
f.add(b);//adding button in JFrame
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
}
}
```

In this example, we draw six basic shapes on the panel: a square, a rectangle, a rounded rectangle, an ellipse, an arc, and a circle.

Working with 2d shapes

```
import java.awt.Color;
import java.awt.EventQueue;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.geom.Ellipse2D;
import javax.swing.JFrame;
import javax.swing.JPanel;
class Surface extends JPanel {
    private void doDrawing(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        g2d.setPaint(new Color(150, 150, 150));
        RenderingHints rh = new RenderingHints(
                RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
        rh.put(RenderingHints.KEY_RENDERING,
               RenderingHints.VALUE_RENDER_QUALITY);
   g2d.setRenderingHints(rh);
        g2d.fillRect(30, 20, 50, 50);
        g2d.fillRect(120, 20, 90, 60);
        g2d.fillRoundRect(250, 20, 70, 60, 25, 25);
        g2d.fill(new Ellipse2D.Double(10, 100, 80, 100));
        g2d.fillArc(120, 130, 110, 100, 5, 150);
        g2d.fillOval(270, 130, 50, 50);   }
    @Override
    public void paintComponent(Graphics g) {
       super.paintComponent(g);
        doDrawing(g);  }    }
public class BasicShapesEx extends JFrame {
public BasicShapesEx() {
 initUI();    }
    private void initUI() {
        add(new Surface());
        setTitle("Basic shapes");
        setSize(350, 250);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);    }
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable()
            @Override
            public void run() {
                BasicShapesEx ex = new BasicShapesEx();
                ex.setVisible(true);  }   });   }    }
```

Using Special Fonts for Text

To find out which fonts are available on a particular computer, call the getAvailableFontFamilyNames method of tGraphicsEnvironment class.The method returns an array of strings containing the names of all available fonts.To obtain an instance of the GraphicsEnvironment class that describes the graphics environment of the user's system, use the static getLocalGraphicsEnvironment method.

The following program prints the names of all fonts on your system:

```
import java.awt.*;

public class ListFonts
{
   public static void main(String[] args)
   {
      String[] fontNames = GraphicsEnvironment
         .getLocalGraphicsEnvironment()
         .getAvailableFontFamilyNames();
      for (String fontName : fontNames)
         System.out.println(fontName);
   }
}
```

Using Color (page. 569)

The setPaint method of the Graphics2D class lets you select a color that is used for all subsequent drawing operations on the graphics context. For example:
g2.setPaint(Color.RED);
g2.drawString("Warning!", 100, 100);
You can fill the interiors of closed shapes (such as rectangles or ellipses) with acolor. Simply call fill instead of draw:
Rectangle2D rect = . . .;
g2.setPaint(Color.RED);
g2.fill(rect); // fills rect with red
Define colors with the Color class. The java.awt.Color class offers predefined constantsfor the following 13 standard colors:
BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW
Here is an example of setting a custom color:
g2.setPaint(new Color(0, 128, 128)); // a dull blue-green
g2.drawString("Welcome!", 75, 125);
To set the background color, use the setBackground method of the Component class, an ancestor of JComponent.
MyComponent p = new MyComponent();
p.setBackground(Color.PINK);

Displaying Images

Once images are stored in local files or someplace on the Internet, you can read them into a Java application and display them on Graphics objects. There are many ways of reading images. Here, we use the ImageIcon class that you already saw:

Image img = new ImageIcon(filename).getImage();

Now the variable image contains a reference to an object that encapsulates the imagedata. You can display the image with the drawImage methods of the Graphics class asgiven below.

• boolean drawImage(Image img, int x, int y, ImageObserver observer) - drawsan unscaled image.

Here, img is the image to be drawn, x is the x coordinate of the top left corner, y isthe y coordinate of the top left corner, and observer is the object to notify of theprogress of the rendering process (may be null)

• boolean drawImage(Image img, int x, int y, int width, int height,
ImageObserver observer) – draws a scaled image. The system scales the imageto fit into a region with the given width and height.

Here, img is the image to be drawn, x is the x coordinate of the top left corner, y is the y coordinate of the top left corner, width is the desired width of image, heightis the desired height of image, and observer is the object to notify of the progress
of the rendering process (may be null). For example,

```
class SimplePanel extends JPanel
{
 public void paintComponent(Graphics g)
 {
 Image i = new ImageIcon("src\\simpleframe\\comp.gif","").getImage();
 g.drawImage(i, 10, 10,150,150, null);
 }
}
```

Event Handling

It is a mechanism to control the events and to decide what should happen after an event occur. To handle the events, Java follows the Delegation Event model.We can put the event handling code into one of the following places:

Within class Other class Anonymous class

Within class

Java event handling by implementing ActionListener

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
  b.addActionListener(this);//passing current instance
  add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);  }
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");  }
public static void main(String args[]){
new AEvent();  }  }
```

) Java event handling by anonymous class

```
import java.awt.*;
import java.awt.event.*;
class AEvent3 extends Frame{
TextField tf;
AEvent3(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(50,120,80,30);
b.addActionListener(new ActionListener(){
public void actionPerformed(){
tf.setText("hello");
}  });
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}  public static void main(String args[]){
new AEvent3();  }  }
```

Java event handling by anonymous class

```
import java.awt.*;
import java.awt.event.*;
class AEvent3 extends Frame{
TextField tf;
AEvent3(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(50,120,80,30);
b.addActionListener(new ActionListener(){
public void actionPerformed(){
tf.setText("hello");  }   });
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);  }
public static void main(String args[]){
new AEvent3();  }   }
```

Java event handling by outer class

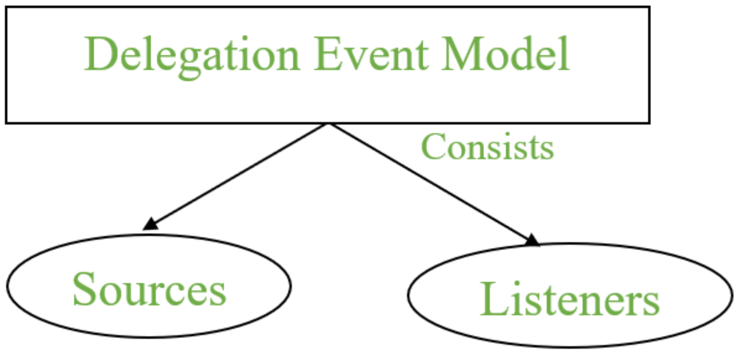
```
import java.awt.*;
import java.awt.event.*;
class AEvent2 extends Frame{
TextField tf;
AEvent2(){
//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
//register listener
Outer o=new Outer(this);
b.addActionListener(o);//passing outer class instance
//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);  }
public static void main(String args[]){
new AEvent2();  }  }
import java.awt.event.*;
class Outer implements ActionListener{
AEvent2 obj;
Outer(AEvent2 obj){
this.obj=obj;  }
public void actionPerformed(ActionEvent e){
obj.tf.setText("welcome");  }  }
```

Event Handling

It is a mechanism to **control the events** and to **decide what should happen after an event** occur. To handle the events, Java follows the ***Delegation Event model.***

Delegation Event model

- It has Sources and Listeners.



Delegation Event Model

- **Source:** Events are generated from the source. There are various sources like buttons, checkboxes, list, menu-item, choice, scrollbar, text components, windows, etc., to generate events.
- **Listeners:** Listeners are used for handling the events generated from the source. Each of these listeners represents interfaces that are responsible for handling events.

Swing and MVC Design Pattern

Swing and MVC Design Pattern Swing uses the model-view-controller architecture (MVC) as the fundamental design behind each of its components. Essentially, MVC breaks GUI components into three elements. Each of these elements plays a crucial role in how the component behaves. Model The model encompasses the state data for each component. There are different models for different types of components. For example, the model of a scrollbar component might contain information about the current position of its adjustable "thumb," its minimum and maximum values, and the thumb's width (relative to the range of values). A menu, on the other hand, may simply contain a list of the menu items the user can select from. Note that this information remains the same no matter how the component is painted on the screen; model data always exists independent of the component's visual representation. View The view refers to how you see the component on the screen. For a good example of how views can differ, look at an application window on two different GUI platforms. Almost all window frames will 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 older MacOS 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. Events come in many forms - a mouse click, gaining or losing focus, a keyboard event that triggers a specific menu command, or even a directive to repaint part of the screen. The controller decides how each component will react to the event – if it reacts at all.

Example of swing and mvc

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class EventTest extends JFrame implements ActionListener
{
private JTextfield t1, t2, t3;
JLabel 11, 12,13;
]Button b1, b2;
public EventTest ()
super ("Handling Action Event");
setDefaultcloseOperation(JFrame.EXIT_ON_CLOSE);
l1 = new JLabel("First Value:");
l2 = new jLabel("Second Value:");
l3 = new JLabel("Result:");
t1 = new TextField(10);
t2 = new JTextfield(10);
t3 = new JTextField(10);
b1 = new JButton("Add");
b2 = new Button("Subtract");
b1.addActionListener (this); b2. addActionListener (this);
setLayout (new FlowLayout (FlowLayout. LEFT, 150, 10)) ;
add (l1);
add(t1);
add (l2);
add (t2);
add (l3);
```

```
add(t3);
add(b1);
add(b2);
setsize (400, 300) ;
setVisible(true);
public void actionPerformed (ActionEvent ae)
{
int x, y, z;
x = Integer .parseInt (t1.getText ());
y = Integer .parseInt (t2 .getText ()) ;
if(ae. getActionCommand() == "Add")
z = x + У;
else
z = X - Y;
t3. setText (String.valueOf (2)) ;
Public static vold main(String a [1)
new EventTest();}}
```

Write a GUI application to calculate simple interest. Use text fields for inputs and output. Use mouseClick method to generate the result using jaava

```
import javax.swing.*;
import java.awt.event.*;
public class SimpleInterestCalculator {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Simple Interest Calculator");
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel();
        JLabel principalLabel = new JLabel("Principal Amount:");
        JTextField principalField = new JTextField(10);
        JLabel rateLabel = new JLabel("Rate (%):");
        JTextField rateField = new JTextField(10);
        JLabel timeLabel = new JLabel("Time (years):");
        JTextField timeField = new JTextField(10);
        JLabel resultLabel = new JLabel("Simple Interest: ");
        JButton calculateButton = new JButton("Calculate");
        calculateButton.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent e) {
           double principal Double.parseDouble(principalField.getText());
                double rate = Double.parseDouble(rateField.getText());
                double time = Double.parseDouble(timeField.getText());
                double interest = (principal * rate * time) / 100;
                resultLabel.setText("Simple Interest: " + interest);
            } });panel.add(principalLabel);
        panel.add(principalField);
        panel.add(rateLabel);
        panel.add(rateField);
        panel.add(timeLabel);
        panel.add(timeField);
        panel.add(calculateButton);
        panel.add(resultLabel);
        frame.add(panel);
        frame.setVisible(true);
    }}
```

write a program with three text boxes first number, second number, and result and four buttons add, subtract, multiply anddivide.handle the events to perform the required operation and display result

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Calculator {
 private JFrame frame;
 private JTextField firstNumberField, secondNumberField, resultField;
   public Calculator() {
       frame = new JFrame("Simple Calculator");
       frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       JPanel panel = new JPanel();
       panel.setLayout(new GridLayout(4, 2, 10, 10));
       JLabel firstNumberLabel = new JLabel("First Number:");
       JLabel secondNumberLabel = new JLabel("Second Number:");
       JLabel resultLabel = new JLabel("Result:");
       firstNumberField = new JTextField(10);
       secondNumberField = new JTextField(10);
       resultField = new JTextField(10);
       resultField.setEditable(false);
       JButton addButton = new JButton("Add");
       JButton subtractButton = new JButton("Subtract");
       JButton multiplyButton = new JButton("Multiply");
       JButton divideButton = new JButton("Divide");
       addButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
     double num1 = Double.parseDouble(firstNumberField.getText());
  double num2 =Double.parseDouble(secondNumberField.getText());
               double result = num1 + num2;
               resultField.setText(String.valueOf(result));
           }});
       subtractButton.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
               double num1 = Double.parseDouble(firstNumberField.getText());


double num2 = Double.parseDouble(secondNumberField.getText());
               double result = num1 - num2;
               resultField.setText(String.valueOf(result));
           }});
       multiplyButton.addActionListener(new ActionListener() {
           public void actionPerformed(ActionEvent e) {
     double num1 = Double.parseDouble(firstNumberField.getText());
  double num2 = Double.parseDouble(secondNumberField.getText());
               double result = num1 * num2;
               resultField.setText(String.valueOf(result));
           }  });//divide
```

```
    divideButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            double num1 = Double.parseDouble(firstNumberField.getText());
            double num2 = Double.parseDouble(secondNumberField.getText());
            if (num2 != 0) {
                double result = num1 / num2;
                resultField.setText(String.valueOf(result));
            } else {
                resultField.setText("Error: Division by zero");
            } } });
    panel.add(firstNumberLabel);
    panel.add(firstNumberField);
    panel.add(secondNumberLabel);
    panel.add(secondNumberField);
    panel.add(resultLabel);
    panel.add(resultField);
    panel.add(addButton);
    panel.add(subtractButton);
    panel.add(multiplyButton);
    panel.add(divideButton);
    frame.add(panel);
    frame.pack();
    frame.setVisible(true);
}
public static void main(String[] args) {
    new Calculator();
}}
```

write a program to display login form containing user id, password and ok button. use layout managers properly.

```
import javax.swing.*;import java.awt.*;import java.awt.event.*;
public class LoginForm {
    private JFrame frame;
    private JTextField userIdField;
    private JPasswordField passwordField;
    public LoginForm() {
        frame = new JFrame("Login Form");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        JPanel panel = new JPanel(new GridLayout(3, 2, 10, 10));
        JLabel userIdLabel = new JLabel("User ID:");
        userIdField = new JTextField(20);
        JLabel passwordLabel = new JLabel("Password:");
        passwordField = new JPasswordField(20);
        JButton okButton = new JButton("OK");
        okButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String userId = userIdField.getText();
                char[] passwordChars = passwordField.getPassword();
                String password = new String(passwordChars);
                JOptionPane.showMessageDialog(frame, "User ID: " + userId + "\nPassword: " +
password);
                 passwordField.setText("");
            } });
 panel.add(userIdLabel);
        panel.add(userIdField);
        panel.add(passwordLabel);
        panel.add(passwordField);
        panel.add(new JLabel()); // Empty label for spacing
        panel.add(okButton);
        frame.add(panel, BorderLayout.CENTER);
        frame.pack();
        frame.setVisible(true);
    } public static void main(String[] args) {
        new LoginForm();
    }
}
```

REMOTE METHOD INVOCATION

RMI, which stands for Remote Method Invocation, is a Java technology that allows objects in one Java Virtual Machine (JVM) to invoke methods on objects located in another JVM, possibly on a remote machine. The RMI is an API that provides a mechanism to create distributed applications in java. The RMI allows an object to invoke methods on an object running in another JVM. The RMI provides remote communication between the applications using two objects stub and skeleton.

Stub

The stub is an object, 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 Machine (JVM),

It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),

It waits for the result.

It reads the return value or exception, and

It finally returns the value to the caller.

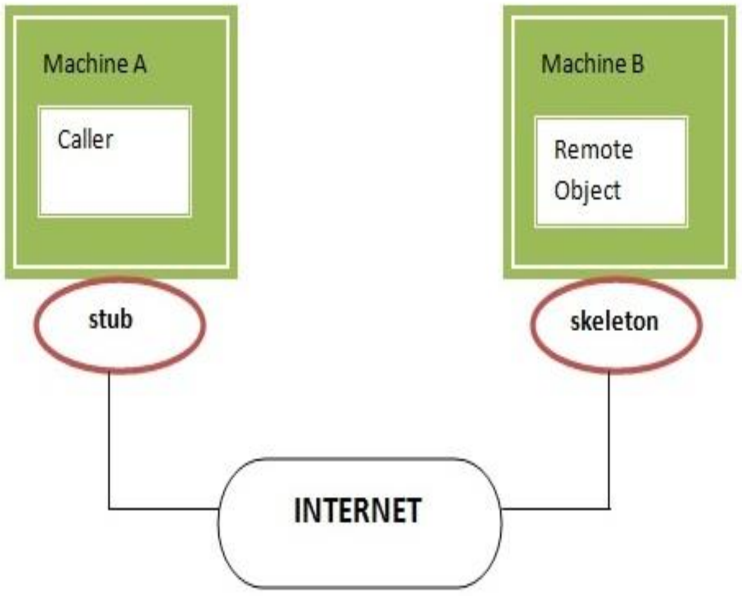
Skeleton

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

It reads the parameter for the remote method.

It invokes the method on the actual remote object, and

It writes and transmits (marshals) the result to the caller.



ROLES IN RMI APPLICATION

In an RMI application, there are typically two main roles: the client and the server. Let's explore the roles of these two components in Java RMI:

1.Server:

Implementation of Remote Objects: The server-side code contains implementations of remote objects. These are Java classes that define methods that can be invoked remotely by clients.

Remote Object Registration: The server is responsible for registering the remote objects with the RMI registry or naming service. This registration makes the objects accessible to clients. In Java RMI, you often use the Naming or Registry classes for this purpose.

Stub Generation: RMI generates stubs and skeletons for remote objects. The server provides the generated stub to the client. The stub acts as a proxy on the client side and forwards method calls to the actual remote object on the server.

Start RMI Service: The server typically starts the RMI service, which listens for incoming remote method invocations. This is usually done by calling java.rmi.registry.LocateRegistry.createRegistry(port) to create an RMI registry and bind remote objects to it.

Security Configuration: The server should also configure security settings as needed, such as defining security policies and providing security managers, to ensure secure communication between the client and server.

2.Client:

Lookup Remote Objects: The client is responsible for looking up remote objects in the RMI registry or naming service. This allows the client to obtain references to the remote objects it wants to interact with.

Invocation of Remote Methods: Clients use the stubs obtained from the server to invoke methods on remote objects as if they were local objects. These method calls are marshaled and sent over the network to the server.

Handling Exceptions: The client must handle exceptions that can occur during remote method invocations, such as RemoteException. These exceptions can occur due to network issues or server-side errors.

Security Configuration: Like the server, the client may also need to configure security settings to ensure secure communication with the server, including setting security managers and adhering to security policies.

Importance of RMI

RMI (Remote Method Invocation) is important to users and developers for several reasons, especially when building distributed systems and applications. Here are some key reasons why RMI is important and beneficial:

1.Simplicity and Familiarity: RMI is a part of the Java Standard Library, and it follows Java's object-oriented programming model. This makes it familiar and easy to use for Java developers who are already comfortable with Java classes and interfaces.

2.Seamless Remote Method Calls: RMI allows developers to invoke methods on remote objects as if they were local objects. This abstraction of remote communication makes it easier to create distributed applications without the need for complex low-level network programming.

3.Strong Typing: RMI enforces strong typing, which means that method calls and parameter types are checked at compile time. This helps catch errors early in the development process, reducing runtime errors.

4.Language Independence: While RMI is primarily used with Java, it can also be used to communicate with non-Java systems through technologies like CORBA and IIOP (Internet Inter-ORB Protocol), allowing interoperability with other programming languages.

5.Automatic Marshalling: RMI automatically handles the serialization and deserialization (marshalling and unmarshalling) of method parameters and return values. Developers do not need to write custom code for data conversion between client and server.

6.Scalability: RMI allows for the development of scalable distributed systems. Remote objects can be distributed across multiple servers to handle increased loads and improve system performance.

7.Object Activation: RMI provides an object activation mechanism, enabling remote objects to be created and destroyed on-demand. This feature helps manage system resources efficiently by activating objects when they are needed and deactivating them when they are not.

8.Security: RMI includes security mechanisms, such as code signing and access control, to ensure the integrity and security of remote method invocations.

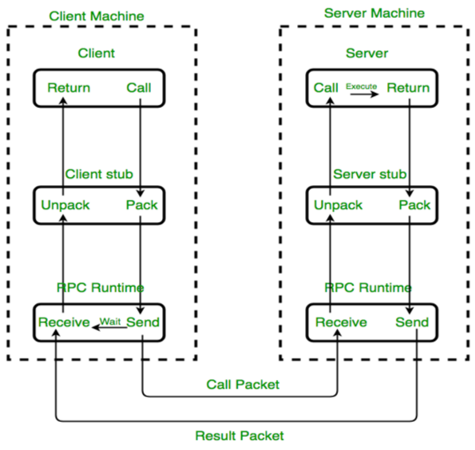
9.Standardization: RMI is a well-established and standardized technology within the Java ecosystem, ensuring compatibility across various Java platforms and environments.

10.Distributed System Development: RMI simplifies the development of distributed systems, allowing developers to focus on application logic rather than low-level networking details.

Remote Method Call

RMC is a communication technique that allows a program to invoke methods or functions on objects located in a remote address space, typically on a different machine or in a different process. RMC is commonly used in distributed computing environments to enable interactions between software components running on separate systems. Java RMI (Remote Method Invocation) is one of the implementations of remote method call in the Java programming language.

Working Process of remote method call:



Implementation of RPC mechanism

Interface Definition: In a remote method call scenario, you define an interface that declares the methods you want to invoke remotely. This interface should extend the java.rmi.Remote interface, and each method should declare the java.rmi.RemoteException in its throws clause.

Implementation: On the server side, you provide an implementation of this interface. This implementation defines the actual logic of the methods declared in the remote interface.

Server Setup: The server creates an instance of the implementation class, registers it with the RMI registry, and starts listening for incoming remote method calls.

Client: On the client side, you obtain a reference to the remote object from the RMI registry, and then you can call the remote methods on that object as if they were local methods.

Exception Handling: Both the server and the client should handle RemoteException and other exceptions that can occur during remote method calls, such as network errors.

This setup allows the client to invoke methods on objects located on the server as if they were local objects. The RMI system takes care of marshaling and unmarshaling parameters and return values, as well as handling network communication and exceptions.

Parameter Marshalling:

What is Marshalling: Marshalling is the process of converting complex data structures or objects into a format that can be transmitted over the network. In the context of RMI, marshalling is used to convert method parameters and return values into a format that can be sent from the client to the server and vice versa.

Role of Parameter Marshalling:

Client to Server: When the client invokes a remote method on a stub, the arguments passed to that method need to be marshalled into a format suitable for transmission over the network. This includes converting Java objects into a binary or textual representation.

Server to Client: Similarly, when the server processes the method call and returns a result or an exception, the result needs to be marshalled for transmission back to the client.

Java Serialization: In Java RMI, the default mechanism for parameter marshalling is Java's built-in serialization. Java objects can be serialized into a binary format and deserialized on the other end, allowing complex data structures to be transmitted over the network seamlessly. To support this, all objects in RMI must implement the java.io.Serializable interface.

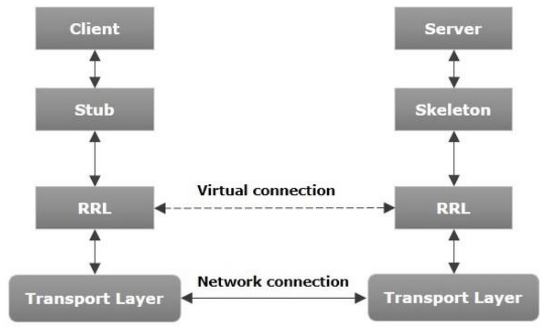
How parameter marshalling, and stubs work together:

- Client-side stub marshals method arguments into a network-friendly format and sends a request to the server.
- Server-side stub unmarshals the request, calls the actual method on the remote object, and marshals the result.
- The marshalled result is sent back to the client-side stub.
- Client-side stub unmarshals the result and returns it to the client application.

By using stubs and parameter marshalling, RMI enables transparent remote method invocation, making it seem as if the client and server objects are in the same JVM, even though they can be running on separate machines.

Architecture of RMI

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



- RMI system consists of the following components
- **Transport Layer** − This layer connects the client and the server. It manages the existing connection and also sets up new connections.
- **Stub** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- **Skeleton** − This is the object which resides on the server side. **stub** communicates with this skeleton to pass request to the remote object.
- **RRL(Remote Reference Layer)** − It is the layer which manages the references made by the client to the remote object.

• **The role of client and server/ Working of RMI application.**
  - When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
  - When the client-side RRL receives the request, it invokes a method called **invoke()** of the object **remoteRef**. It passes the request to the RRL on the server side.
  - The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
  - The result is passed all the way back to the client.

CREATING AND EXECUTING RMI APPLICATIONS

There are six steps to write the RMI applications.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

The client application needs only two files, remote interface and client application. In the RMI application, both client and server interact with the remote interface. The client application invokes methods on the proxy object; RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application.

Step 1: Create the Remote Interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and it declares RemoteException.

```
import java.rmi.*;

public interface Adder extends Remote{

 public int add (int x, int y) throws Remote Exception;    }
```

Step 2: Provide the Implementation of the Remote Interface

Now provide the implementation of the remote 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

In case, we extend the UnicastRemoteObject class, we must define a constructor that declares Remote Exception.

```
import java.rmi.*;
import java.rmi.server.*;
public class Adder Remote extends Unicast RemoteObject implements Adder{
AdderRemote() throws Remote Exception{
super();
}
public int add (int x, int y){
return x+y;
}}
```

Step 3: Create the Stub and Skeleton Objects Using the rmic Tool

Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

```
rmic AdderRemote
```

Step 4:Start the Registry Service by the rmiregistry Tool

Now start the registry service by using the rmiregistry tool. If we don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

```
rmiregistry 5000
```

Step 5:Create and Run the Server Application

Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. In this example, we are binding the remote object by the name rmiserver.

```
import java.rmi.*;

import java.rmi.registry.*; {

public class MyServer

{

public static void main(String args[ ])

{try{

Adder adderobject=new AdderRemote(); Naming.rebind("rmi://localhost:5000/rmiserver",

adderobject);

}

catch(Exception e)

{

System.out.println(e);

}}}
```

Step 6:Create and run the client application

At the client we are getting the adderobject object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If we want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

```
import java.rmi.*;

public class MyClient{

public static void main(String args[]) throws Exception{

Adder a=(Adder) Naming.lookup ("rmi://localhost: 5000/as");

int s=a.add(34,4);

System.out.println("Sum="+s);

}
```

RMI (Remote Method Invocation)

The RMI (Remote Method Invocation) programming model is a framework for building distributed applications in Java, allowing objects in one Java Virtual Machine (JVM) to invoke methods on objects located in another JVM, possibly on a different machine. The RMI programming model consists of several key concepts and steps:

1.Remote Interface: Define a Java interface that extends the java.rmi.Remote interface. This interface should declare the methods that you want to make remotely accessible.Each method in the remote interface should throw java.rmi.RemoteException to indicate that it can throw exceptions across network boundaries.

```
import java.rmi.Remote;

import java.rmi.RemoteException;

public interface MyRemoteInterface extends Remote {

    String remoteMethod() throws RemoteException;

}
```

2.Remote Implementation:

Create a class that implements the remote interface. This class provides the actual implementation for the methods declared in the interface.

```
import java.rmi.RemoteException;

import java.rmi.server.UnicastRemoteObject;

public class MyRemoteImplementation extends UnicastRemoteObject implements MyRemoteInterface {

    public MyRemoteImplementation() throws RemoteException {

        // Constructor

    }


    public String remoteMethod() throws RemoteException {

        // Actual implementation of the remote method

        return "Hello from the remote server!";

    }

}
```

3.Server Setup: In the server application, create an instance of the remote implementation class.Use the RMI registry (or naming service) to bind the remote object with a name so that clients can look it up.Start the RMI service on the server, usually by creating an RMI registry on a specific port.

```
import java.rmi.registry.Registry;

import java.rmi.registry.LocateRegistry;

public class MyRMIServer {
    public static void main(String[] args) {
        try {
            MyRemoteInterface remoteObject = new MyRemoteImplementation();
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.bind("MyRemoteObject", remoteObject);
            System.out.println("Server is ready.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

4.Client Application:

In the client application, obtain a reference to the remote object from the RMI registry by looking it up using its name.Use the remote object reference to call the remote methods as if they were local.

```
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
public class MyRMIClient {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.getRegistry("serverHostname", 1099);
            MyRemoteInterface remoteObject = (MyRemoteInterface) registry.lookup("MyRemoteObject");

            String result = remoteObject.remoteMethod();
            System.out.println("Result from remote method: " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

5.Exception Handling:

Both the server and the client should handle ‘RemoteException’ and other exceptions that can occur during remote method calls, such as network errors.

RMI Registry

The RMI Registry, often referred to as the RMI registry service or simply the registry, is a fundamental component of the Java Remote Method Invocation (RMI) framework. It serves as a naming service or directory service that allows RMI clients to locate and access remote objects hosted on a server. The RMI registry is a critical part of enabling distributed communication between Java applications.

Here are some key points about the RMI registry:

1.Binding and Lookup: The RMI registry provides a mechanism for binding remote objects to names and looking up remote objects by name. Each registered remote object is associated with a unique name within the registry.

2.Port Number: The RMI registry typically runs on a specific port (default is 1099). Clients and servers must agree on which port to use for the registry. You can create an RMI registry on a specific port using the LocateRegistry.createRegistry(port) method on the server side.

3.Naming Conventions: The names used to bind and look up remote objects in the registry are simply strings. They are used as identifiers and should be unique within the registry.

4.Client Lookup: RMI clients use the lookup method provided by the java.rmi.registry.Registry interface to find remote objects in the registry. The client specifies the name under which the remote object was bound.

5.Server Binding: On the server side, remote objects are bound to the registry using the rebind method provided by the Registry interface. The server specifies the name under which the remote object should be accessible.

Remote object activation

Remote object activation is a mechanism used in Java RMI (Remote Method Invocation) to manage the lifecycle of remote objects. It allows remote objects to be created on demand and deactivated when they are no longer in use. This helps conserve system resources in distributed applications by ensuring that remote objects are not kept alive indefinitely. Remote object activation is a powerful feature of RMI that helps manage resource utilization in distributed applications. It ensures that remote objects are only instantiated and kept in memory when they are needed, contributing to better system performance and scalability.

How remote object activation works:

1.Server-Side Configuration:

-On the server side, you configure an activation group and an activation system.

-You create activatable objects that extend Activatable and implement the remote interface.

2.Client Usage: Clients obtain references to activatable objects in a similar way to regular RMI objects by using the RMI registry or naming service.

3.Activation: When a client invokes a method on an activatable object for the first time, the activation system checks if the object is already active. If not, it activates the object by creating an instance in the appropriate activation group.

4.Deactivation: If an activatable object is not used for a specified period and is eligible for deactivation according to its activation descriptor, the activation monitor triggers the deactivation process. The object is removed from memory to free up system resources.

CORBA

It stands for Common Object Request Broker Architecture, is a middleware and distributed computing platform that enables communication and interaction between software objects, components, and services in a networked environment. CORBA was developed by the Object Management Group (OMG) and has been widely used in various industries for building distributed and interoperable applications. CORBA has been widely used in industries such as telecommunications, finance, and aerospace for building complex and distributed applications. However, in recent years, newer technologies like web services and RESTful APIs have gained popularity for distributed computing, and CORBA's usage has declined in favor of these more lightweight and web-centric approaches.

Key features and concepts of CORBA include:

1.Object Request Broker (ORB): The ORB is a central component of CORBA that acts as a middleware layer responsible for locating, invoking, and managing distributed objects.ORBs provide the necessary infrastructure for objects to communicate with each other regardless of their location or the programming languages they are implemented in.

2.Interface Definition Language (IDL):CORBA uses IDL to define the interfaces of distributed objects. IDL is a language-independent specification that allows developers to define object interfaces in a platform-neutral way.The IDL compiler generates language-specific stubs and skeletons that bridge the gap between different programming languages and the CORBA middleware.

3.Stubs and Skeletons: Stubs are client-side proxies that allow local objects to communicate with remote objects as if they were local.Skeletons are server-side components that receive incoming method invocations from clients and dispatch them to the appropriate remote objects.

4.Object Services:CORBA provides a set of standard object services, such as naming, trading, security, and transactions, that can be used to enhance the functionality and reliability of distributed applications.

5.Interoperability:CORBA promotes interoperability by allowing objects written in different programming languages (e.g., Java, C++, Python) to communicate seamlessly.This makes it suitable for building heterogeneous systems where components written in various languages need to work together.

6.Location Transparency:CORBA abstracts the location of remote objects, enabling clients to invoke methods on objects regardless of their physical location (e.g., on a different machine or in a different process).

7.Scalability and Extensibility:CORBA is designed to support the development of scalable and extensible distributed systems. New services and functionality can be added to an existing CORBA application without major modifications.

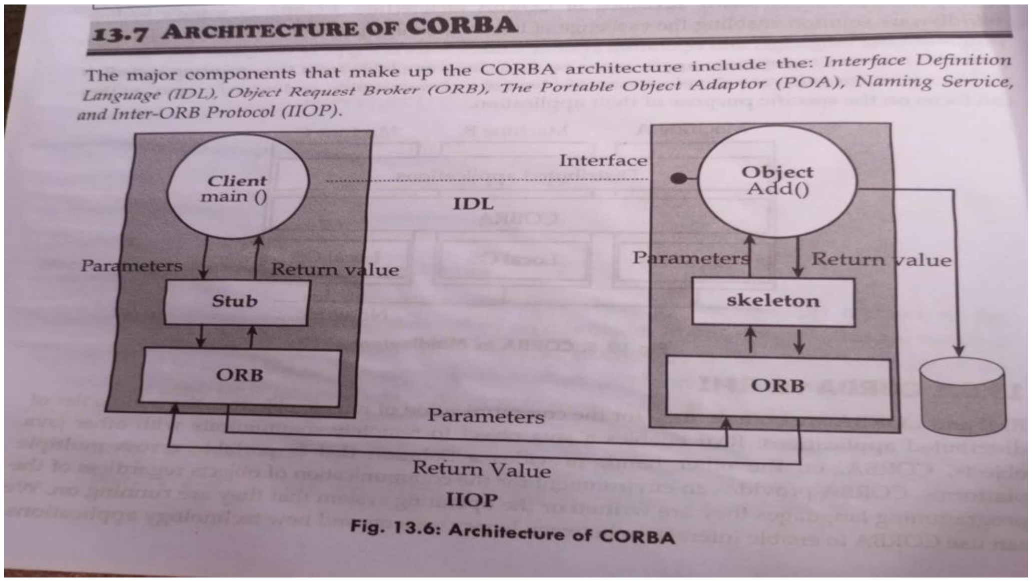
8.Middleware Services:CORBA provides middleware services, including object activation, object persistence, messaging, and event notification, to simplify the development of distributed systems.

Difference between RMI and CORBA :

RMI	CORBA
RMI is a Java-specific technology.	CORBA has implementation for many languages.
It uses Java interface for implementation.	It uses Interface Definition Language (IDL) to separate interface from implementation.
RMI objects are garbage collected automatically.	CORBA objects are not garbage collected because it is language independent and some languages like C++ does not support garbage collection.
RMI programs can download new classes from remote JVM's.	CORBA does not support this code sharing mechanism.
RMI passes objects by remote reference or by value.	CORBA passes objects by reference.
Java RMI is a server-centric model.	CORBA is a peer-to-peer system.
RMI uses the Java Remote Method Protocol as its underlying remoting protocol.	CORBA use Internet Inter- ORB Protocol as its underlying remoting protocol.
The responsibility of locating an object implementation falls on JVM.	The responsibility of locating an object implementation falls on Object Adapter either Basic Object Adapter or Portable Object Adapter.

Architecture of COBRA

The architecture of CORBA (Common Object Request Broker Architecture) is a comprehensive framework for building distributed and interoperable software systems. It defines a set of specifications, protocols, and components that allow objects written in different programming languages and running on different platforms to communicate and interact seamlessly. Below, I'll describe the key architectural components and concepts of CORBA:



13.7 ARCHITECTURE OF CORBA

The major components that make up the CORBA architecture include the: *Interface Definition Language (IDL), Object Request Broker (ORB), The Portable Object Adaptor (POA), Naming Service, and Inter-ORB Protocol (IIOP).*

Fig. 13.6: Architecture of CORBA

1.Object Request Broker (ORB): The Object Request Broker is the central component of the CORBA architecture. It acts as an intermediary or a broker between distributed objects, responsible for locating, invoking, and managing remote objects.The ORB abstracts the complexities of remote communication, including data marshalling, communication protocols, and object activation.

2.IDL (Interface Definition Language):CORBA uses IDL as a language-neutral interface specification language. IDL allows developers to define the interfaces of distributed objects in a platform-independent way.IDL interfaces are used to specify the methods and data structures that objects expose to other objects.

3.Portable Object Adapters (POA):POA is a component that manages object activation policies, allowing developers to control when and how objects are activated and deactivated.POA is particularly important for managing object lifecycles in a distributed environment.

4.Internet Inter-ORB protocol(IIOP):IIOP stands for "Internet Inter-ORB Protocol," and it is a key communication protocol used in the CORBA (Common Object Request Broker Architecture) distributed computing framework. IIOP enables interoperability between CORBA ORBs (Object Request Brokers) running on different platforms and implemented in different programming languages. It allows objects to communicate and exchange method calls across a network, making it a fundamental component of CORBA's distributed computing model.

5.Naming Service: In the context of CORBA (Common Object Request Broker Architecture) and distributed computing, a naming service, often referred to as a Naming Service or simply "Naming," is a fundamental component that provides a way to associate human-readable names with distributed objects. It allows clients to look up and locate remote objects by name rather than knowing their physical addresses or locations.

Interface Definition Language

IDL (interface definition language) is a generic term for a language that lets a program or object written in one language communicate with another program written in an unknown language. In distributed object technology, it's important that new objects be able to be sent to any platform environment and discover how to run in that environment. An Object Request Broker (ORB) is an example of a program that would use an interface definition language to "broker" communication between one object program and another one. An interface definition language works by requiring that a program's interfaces be described in a stub or slight extension of the program that is compiled into it. The stubs in each program are used by a broker program to allow them to communicate.IDL (Interactive Data Language) is a language for creating visualizations based on scientific or other data.IDL (interactive distance learning) is a general term for learning that takes place through remote telecommunication and that allows students to participate from a distance. Television has been used for many years for non-interactive distance learning. Teleconference classes are becoming more common where higher bandwidth and such technologies as ISDN and satellite communication per mit. The World Wide Web, with or without multimedia, offers new possibilities.

- **Write a program to add two numbers using RMI.**

  Source code:

  Adder.java

```
import java.rmi.*;

public interface Adder extends Remote {

    public int sum(int a, int b) throws RemoteException;

}
```

AddderImpl.java

```
import java.rmi.RemoteException;

public class AdderImpl implements Adder {

@Override

public int sum(int a, int b) throws RemoteException {

// TODO Auto-generated method stub

return a + b;     }

}
```

Client.java

```
import java.rmi.registry.LocateRegistry;

import java.rmi.registry.Registry;

public class Client {

private Client() {

}
```

```
   public static void main(String[] args) {
    try {
      Registry registry = LocateRegistry.getRegistry();
      Adder stub = (Adder) registry.lookup("Add");
      int sumValue = stub.sum(20, 40);
      System.out.println("Sum is " + sumValue);
    } catch (Exception e) {
      System.out.println("Client exception: " + e.toString());
      e.printStackTrace();
      // TODO: handle exception
    }
  }
}
```

Server.java

```
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;
public class Server extends AdderImpl {
  public Server() {
  }
  public static void main(String[] args) {
    try {
      AdderImpl obj = new AdderImpl();
      Adder stub = (Adder) UnicastRemoteObject.exportObject(obj, 0);
      Registry registry = LocateRegistry.getRegistry();
      registry.bind("Add", stub);
      System.err.println("Server Ready");
    } catch (Exception e) {
      System.err.println("Server exception: " + e.toString());
      e.printStackTrace();
      // TODO: handle exception }   }   }
```

- **Write a program to multiply two numbers using RMI.**

  <u>Source code:</u>

  ProductInterface.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface ProductInterface extends Remote {
   int multiply(int a, int b) throws RemoteException;   }
```

  ProductImpl.java

```
import java.rmi.RemoteException;
public class ProductImpl implements ProductInterface {
   @Override
   public int multiply(int a, int b) throws RemoteException {
      return a * b;   }    }
```

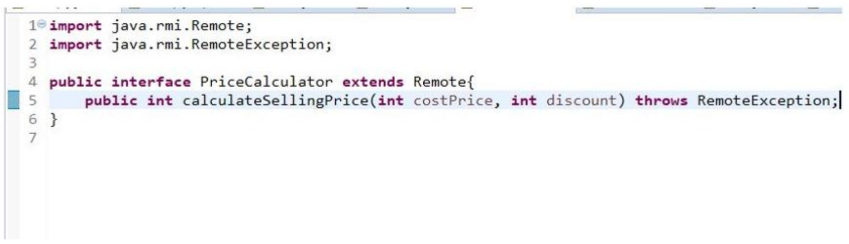
  Server.java

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
public class Server {
   public static void main(String[] args) {
      try {
         ProductImpl product = new ProductImpl();
         ProductInterface stub = (ProductInterface)
UnicastRemoteObject.exportObject(product, 0);
         Registry registry = LocateRegistry.createRegistry(9999);
         registry.bind("ProductService", stub);
         System.out.println("Server started!");
      } catch (Exception e) {
         System.err.println("Server exception: " + e.toString());
         e.printStackTrace();    }    }   }
```

  Client.java

```
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
public class Client {
   public static void main(String[] args) {
      try {
         Registry registry = LocateRegistry.getRegistry("localhost", 9999);
         ProductInterface stub = (ProductInterface) registry.lookup("ProductService");
         int a = 5;       int b = 7;
         int result = stub.multiply(a, b);
         System.out.println("Product: " + result);
      } catch (Exception e) {
         System.err.println("Client exception: " + e.toString());
         e.printStackTrace();    }    }   }
```

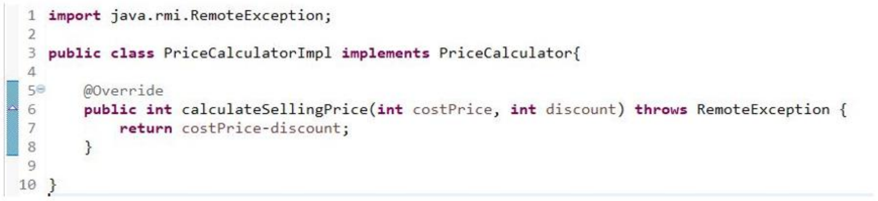
- Write a Client Server Application to find the selling price of a product with cost price of Rs.5000 after a discount of Rs.50.



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

public interface PriceCalculator extends Remote{
    public int calculateSellingPrice(int costPrice, int discount) throws RemoteException;
}

```

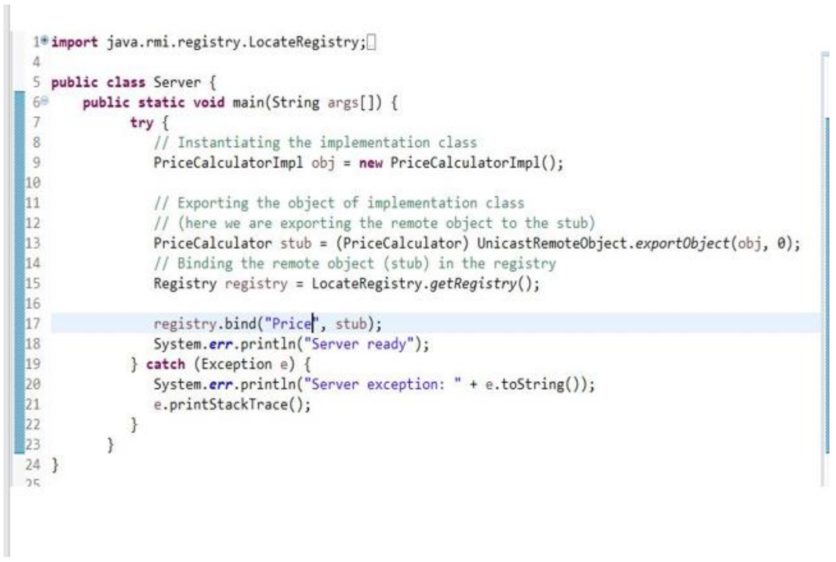


```java
import java.rmi.RemoteException;

public class PriceCalculatorImpl implements PriceCalculator{

    @Override
    public int calculateSellingPrice(int costPrice, int discount) throws RemoteException {
        return costPrice-discount;
    }

}
```

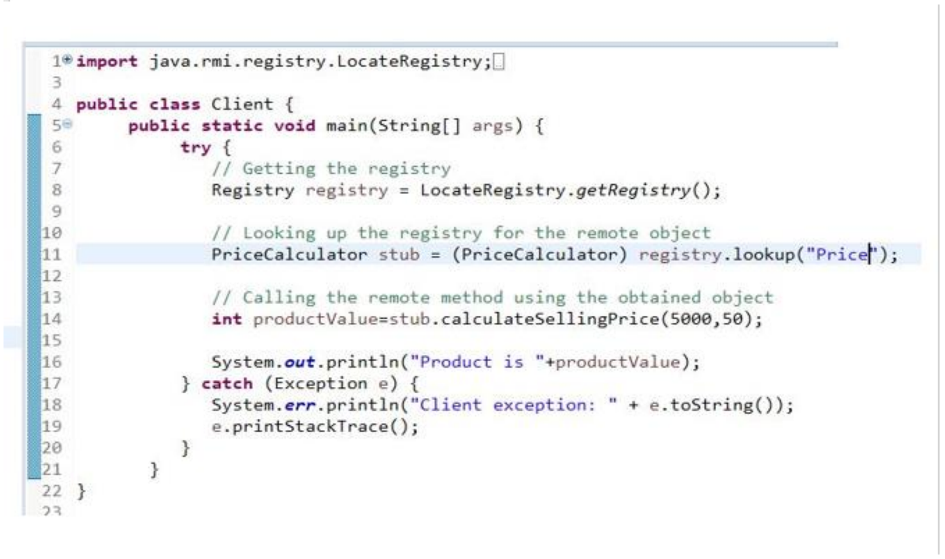


```java
import java.rmi.registry.LocateRegistry;

public class Server {
    public static void main(String args[]) {
        try {
            // Instantiating the implementation class
            PriceCalculatorImpl obj = new PriceCalculatorImpl();

            // Exporting the object of implementation class
            // (here we are exporting the remote object to the stub)
            PriceCalculator stub = (PriceCalculator) UnicastRemoteObject.exportObject(obj, 0);
            // Binding the remote object (stub) in the registry
            Registry registry = LocateRegistry.getRegistry();

            registry.bind("Price", stub);
            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```



```java
import java.rmi.registry.LocateRegistry;

public class Client {
    public static void main(String[] args) {
        try {
            // Getting the registry
            Registry registry = LocateRegistry.getRegistry();

            // Looking up the registry for the remote object
            PriceCalculator stub = (PriceCalculator) registry.lookup("Price");

            // Calling the remote method using the obtained object
            int productValue=stub.calculateSellingPrice(5000,50);

            System.out.println("Product is "+productValue);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

- **Write a Client Server Application in Java where the Server provides price of different laptop models in dollar. Client Program can ask for the price of a laptop model.**

LaptopService.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface LaptopService extends Remote {
   double getPrice(String model) throws RemoteException;
}
```

LaptopServiceImpl.java

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.HashMap;
import java.util.Map;
public class LaptopServiceImpl extends UnicastRemoteObject implements LaptopService
{
   private static final long serialVersionUID = 1L;
   private Map<String, Double> laptopPrices;
   public LaptopServiceImpl() throws RemoteException {
      super();
      laptopPrices = new HashMap<>();
      laptopPrices.put("Dell", 800.0);
      laptopPrices.put("HP", 750.0);
      laptopPrices.put("Lenovo", 850.0);
   }
   @Override
   public double getPrice(String model) throws RemoteException {
      return laptopPrices.getOrDefault(model, -1.0);
   }
}
```

LaptopServer.java

```
 import java.rmi.*;
 import java.rmi.registry.LocateRegistry;
 public class LaptopServer {
 public static void main(String[] args) {
 try {
LaptopService laptopService = new LaptopServiceImpl();
 LocateRegistry.createRegistry(9999);
 Naming.rebind("LaptopService", laptopService);
 System.out.println("Server started. Waiting for client requests...");
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

LaptopClient.java

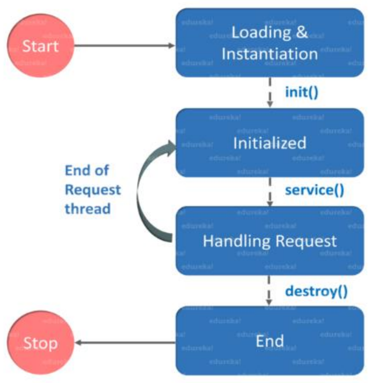
```
 import java.rmi.Naming;
 import java.rmi.registry.LocateRegistry;
 import java.rmi.registry.Registry;
 public class LaptopClient {
public static void main(String[] args) {
try {
   Registry registry = LocateRegistry.getRegistry("localhost");
   LaptopService laptopService = (LaptopService) Naming.lookup("rmi://localhost/LaptopService");
   String model = "Dell";
   double price = laptopService.getPrice(model);
   if (price != -1.0) {
       System.out.println("Price of " + model + " laptop model: $" + price);
   } else {
       System.out.println("Laptop model not found.");
   }
} catch (Exception e) {
   e.printStackTrace();
}
    }
}
```

- **LIFE CYCLE OF THE SERVLET**

  The entire life cycle of a Servlet is managed by the **Servlet container** which uses the **javax.servlet.Servlet** interface to understand the Servlet object and manage it.

  - **Servlet class is loaded:** The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.
  - Servlet instance is created: The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.
  - **Init method is invoked:** The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below: public void init(ServletConfig config) throws ServletException
  - **Service method is invoked**: The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. The syntax of the service method of the Servlet interface is given below: public void service(ServletRequesr request, ServletResponse response)
    throws ServletException, IOException
  - **Destroy method is invoked:** The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below: public void destroy().

  Servlet Life Cycle



- **Definition of servlet.**
  Servlet is a server-side Java program module that handles client requests and implements the servlet interface. Servlets are small programs that execute on the server side of a Web connection. Servlet is an API that provides many interfaces and classes including documentation. Servlets provide a component-based, platform-independent method for building Web-based applications.
- **Deployment Descriptor for Servlet**
  A web application's deployment descriptor maps the http request with the servlets. When the web server receives a request for the application, it uses the deployment descriptor to map the URL of the request to the code that ought to handle the request. **<u>The deployment descriptor should be named as web.xml.</u>** It resides in the application's WAR file under the WEB-INF/

directory. **Root element of web.xml should be <web-app>. <servlet> element maps a URL to a servlet using <servlet-mapping> element.**

To map a URL to a servlet, we declare the servlet with the <servlet> element, then define a mapping from a URL path to a servlet declaration with the <servlet-mapping> element.

- **The GenericServlet Class**

  The GenericServlet class implements the Servlet and ServletConfig interfaces. Since service() method is declared as an abstract method in GenericServlet class, it is an abstract class. The class extending this class must implement the service () method. **It is used to create servlets which are protocol independent.** GenericServlet class can handle any type of request so it is protocol-independent. The service() method accepts two arguments ServletRequest object and ServletResponse object. The request object tells the servlet about the request made by client while the response object is used to return a response back to the client.

  The GenericSerclet class pros and cons:

  Pros:

1. Generic Servlet is easier to write
2. Has simple lifecycle methods
3. To write Generic Servlet you just need to extend javax.servlet.GenericServlet and override the service() method

   Cons:

1. Working with Generic Servlet is not that easy because we don't have convenience methods such as doGet(), doPost(), doHead() etc in Generic Servlet that we can use in HttpServlet.

   We only need to override service() method for each type of request which is complicated.

- **ServletInputStream Class, ServletOutput class, Servlet exception class.**
  - **ServletInputStream** class provides stream to read binary data such as image etc. from the request object. It is an abstract class.

  The **getInputStream()** method of **ServletRequest** interface returns the instance of ServletInputStream class.

  ServletInputStream sin=request.getInputStream();

  There are only one method defined in the ServletInputStream class.

  int readLine(byte[] b, int off, int len) //reads the input stream.

  Here buffer is the array into which size bytes are placed starting at offset. The method returns the actual number of bytes read or -1 if an end-of-stream condition is encountered

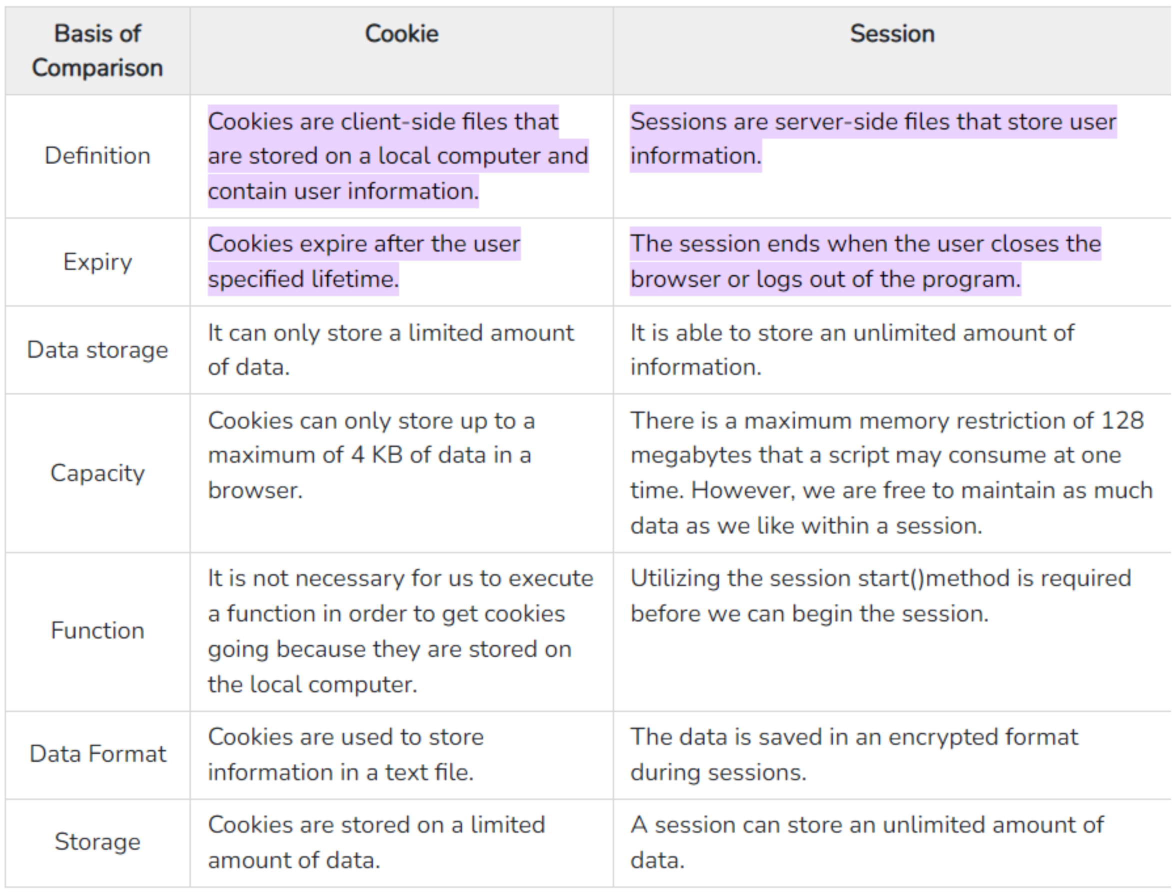
  - **ServletOutputStream** provides a stream to write binary data into the response.This class is an abstract class and extends **OutputStream**. The **getOutputStream()** method of **ServletResponse** interface returns the instance of ServletOutputStream class.

    ServletOutputStream out=response.getOutputStream(); It provides provides print() and println() methods that are overloaded and these methods output data to the stream.
  - javax.servlet defines two exceptions. The first one is **ServletException** which indicates that a servlet problem has occurred. The second one is **UnavailableException.** It indicates that a servlet is unavailable.

- **SESSION & COOKIES**

  - **A session is used to temporarily store the information on the server to be used across multiple pages of the website**. The session ends when the user closes the browsers or logs out of the program. It is able to store an unlimited amount of information. It can be used as an alternative to cookies for browsers that don't support cookies to store variables in a more secure way.

  - **A cookie is a small text file that is stored on the user's computer**. The maximum file size of a cookie is **4KB**. It is also known as an HTTP cookie, web cookie, or internet Cookie. Whenever a user visits a website for the first time, the site sends packets of data in the form of a cookie to the user's computer. We can enable or disable the cookies as per the requirement.



Basis of Comparison	Cookie	Session
Definition	Cookies are client-side files that are stored on a local computer and contain user information.	Sessions are server-side files that store user information.
Expiry	Cookies expire after the user specified lifetime.	The session ends when the user closes the browser or logs out of the program.
Data storage	It can only store a limited amount of data.	It is able to store an unlimited amount of information.
Capacity	Cookies can only store up to a maximum of 4 KB of data in a browser.	There is a maximum memory restriction of 128 megabytes that a script may consume at one time. However, we are free to maintain as much data as we like within a session.
Function	It is not necessary for us to execute a function in order to get cookies going because they are stored on the local computer.	Utilizing the session start()method is required before we can begin the session.
Data Format	Cookies are used to store information in a text file.	The data is saved in an encrypted format during sessions.
Storage	Cookies are stored on a limited amount of data.	A session can store an unlimited amount of data.

- **Cookies servlet code:**

CookieServlet.java

```
package bca.sixth;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class CookieServlet
 */
@WebServlet("/CookieServlet")
public class CookieServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;
    /**
     * @see HttpServlet#HttpServlet()
     */
    public CookieServlet() {
        super();
        // TODO Auto-generated constructor stub
    }
	/**
	 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		// TODO Auto-generated method stub
		response.getWriter().append("Served at: ").append(request.getContextPath());
	}
	/**
	 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		// TODO Auto-generated method stub
		PrintWriter writer = response.getWriter();
		String cookieValue = request.getParameter("cookieValue");
		//creating cookie object
		Cookie cookie = new Cookie("MyCookie", cookieValue);
		//adding cookie object to response
		response.addCookie(cookie);
		writer.append("Cookie added successfully"+ cookieValue);
	}
}
```

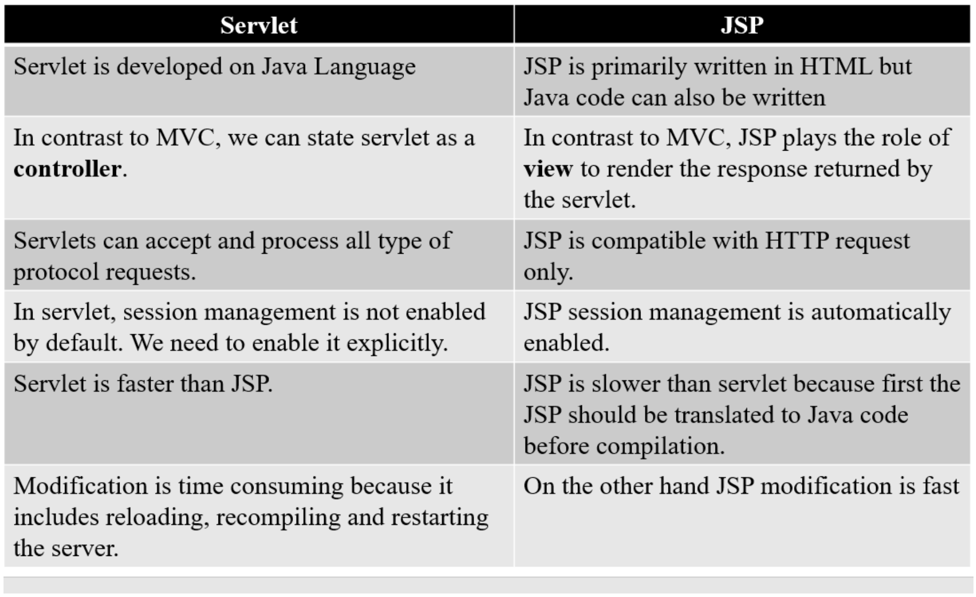
GetCookie.java

```
package bca.sixth;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class GetCookie
 */
@WebServlet("/GetCookie")
public class GetCookie extends HttpServlet {
	private static final long serialVersionUID = 1L;
    /**
     * @see HttpServlet#HttpServlet()
     */
    public GetCookie() {
        super();
        // TODO Auto-generated constructor stub
    }/**
	 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		// TODO Auto-generated method stub
		response.getWriter().append("Served at: ").append(request.getContextPath());
	}
	/**
	 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
		// TODO Auto-generated method stub
		doGet(request, response);
	}
}
```

GetCookieServlet.java

```
package bca.sixth;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class GetCookieServlet
 */
@WebServlet("/GetCookieServlet")
public class GetCookieServlet extends HttpServlet {
	private static final long serialVersionUID = 1L;
    /**
   * @see HttpServlet#HttpServlet()
   */
  public GetCookieServlet() {
    super();
    // TODO Auto-generated constructor stub
  }/**
	 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doGet(HttpServletRequest request, HttpServletResponse response) 
throws ServletException, IOException {
		// TODO Auto-generated method stub
		Cookie[] cookies = request.getCookies();
		PrintWriter writer = response.getWriter();
		for(int i=0; i<cookies.length; i++) {
			String cookieName = cookies[i].getName();
			String cookieValue = cookies[i].getValue();
	writer.append("Cookie Name is"+ cookieName+" Cookie value is "+ cookieValue);   }    }
	/**
	 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doPost(HttpServletRequest request, HttpServletResponse response) 
throws ServletException, IOException {
		// TODO Auto-generated method stub
		doGet(request, response);     }   }
```

- **Difference between Servlet and JSP.**

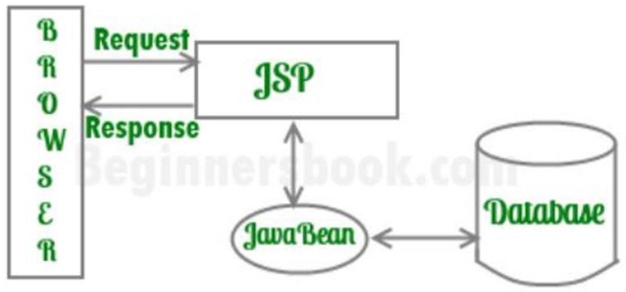


Servlet	JSP
Servlet is developed on Java Language	JSP is primarily written in HTML but Java code can also be written
In contrast to MVC, we can state servlet as a **controller**.	In contrast to MVC, JSP plays the role of **view** to render the response returned by the servlet.
Servlets can accept and process all type of protocol requests.	JSP is compatible with HTTP request only.
In servlet, session management is not enabled by default. We need to enable it explicitly.	JSP session management is automatically enabled.
Servlet is faster than JSP.	JSP is slower than servlet because first the JSP should be translated to Java code before compilation.
Modification is time consuming because it includes reloading, recompiling and restarting the server.	On the other hand JSP modification is fast

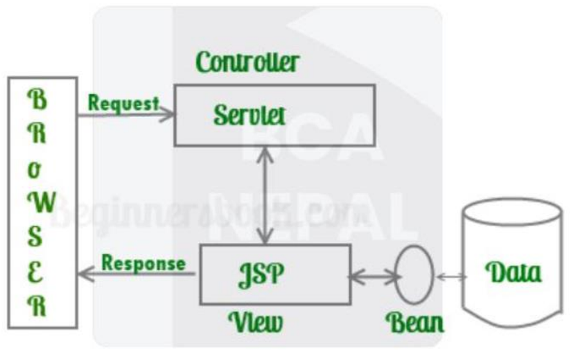
- **JSP and JSP element.**
  Java Server Pages (JSP) is a server-side programming technology that enables the creation of dynamic, platform-independent method for building Web-based applications. JSP have access to the entire family of Java APIs, including the JDBC API to access enterprise databases. JSP can be thought as an extension to servlet technology because it provides features to easily create user views.
  In JSP elements can be divided into 4 different types.
    - Expression
    - Scriptlets
    - Directives
    - Declarations

- JSP expression: We can use this tag to output any data on the generated page. These data are automatically converted to string and printed on the output stream.
  Syntax: JSP Expressions are : <%="Anything" %>
  JSP Expressions start with Syntax of JSP Scriptles are with <%=and ends with %>.
- JSP Scriptlets: In this tag we can insert any amount of valid java code and these codes are placed in the _jsp Service method by the JSP engine.
  Syntax: <%// java code %>
  JSP Scriptlets begins with <% and ends %> . We can embed any amount of Java code in the JSP Scriptlets.
- JSP Directives: JSP Directives are used to give special instructions to the container while JSP page is getting translated to servlet source code. A JSP "directive" starts with <%@ characters. In the directives, we can import packages, and define error-handling pages or the session information of the JSP page.
  Syntax: <%@directive attribute ="value"%>
- JSP declaration: A declaration declares one or more variables or methods that we can use in Java code later in the JSP file. We must declare the variable or method before you use it in the JSP file.

- **Architecture of JSP Application:**
  Model1 Architecture



In this model, JSP plays a key role and it is responsible for processing of the request made by client.
Model2 Architecture



In this model, servlet plays a key role and it is responsible for processing of the request made by client.

- **Passing data from servlet to JSP.**

1. **Using HttpServletRequest**
   The common way of passing data from servlet to JSP is through defining attributes in the HTTP request and then forwarding it to the corresponding JSP.
   request.setAttribute(Name, Value)
   At the server-side, set the attribute in the request then forward the request to the JSP page like the following:
   request.setAttribute("name", "Ram Bahadur");
   request.getRequestDispatcher("home.jsp").forward(request, response);
   //it forwards the request from one servlet to another resource (such as servlet, JSP, HTML file).
   At the jsp file, we can access the attribute as:
   <h3> My name is <%out.print(request.getAttribute("name"))%></h3>
2. **Redirecting to JSP using query string**
   The second way of passing data from servlet to JSP is through redirecting the response to the appropriate JSP and appending the attributes in the URL as a query string.
   response.sendRedirect("home.jsp?name=Ram Bahadur");
   At the jsp side, we can access the parameter as:
   <h3> My name is <%= request.getParameter("name") %></h3>

3. **Passing Objects from servlet to JSP**
   In order to pass a business object or POJO from servlet to JSP, you can pass it as an attribute using the **setAttribute()** method described above.

- **Why to use JSP?**
  - We can generate HTML response from servlets also but the process is cumbersome and error prone, when it comes to writing a complex HTML response, writing in a servlet will be a nightmare. JSP helps in this situation and provide us flexibility to write normal HTML page and include our java code only where it's required.
  - JSP provides additional features such as tag libraries, expression language, custom tags that helps in faster development of user views.
  - JSP pages are easy to deploy, we just need to replace the modified page in the server and container takes care of the deployment. For servlets, we need to recompile and deploy whole project again. Actually Servlet and JSPs compliment each other.
  - We should use Servlet as server side controller and to communicate with model classes whereas JSPs should be used for presentation layer.
- **Write a jsp program for multiplication table.**

```
<!DOCTYPE html>         <html>
<head>              <title>Multiplication Table</title>
</head>
<body>
  <h2>Multiplication Table</h2>
  <form method="post">
    Enter a number:
    <input type="number" name="number" required>
    <input type="submit" value="Generate Table">
  </form>
  <%-- JSP Scriptlet --%>
  <%
    if (request.getMethod().equals("POST")) {
      int number = Integer.parseInt(request.getParameter("number"));             %>
      <h3>Table for <%= number %></h3>
      <table border="1">
        <tr>
          <th>Number</th>
          <th>Result</th>
        </tr>
        <%
          for (int i = 1; i <= 10; i++) {
            int result = number * i;
        %>      <tr>
              <td><%= number %></td>
              <td><%= result %></td>            </tr>
        <% }%>
      </table>
  <% }   %>
</body>
</html>
```

- **Write a jsp program for adding 2 number.**

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Add Numbers</title>
</head>
<body>
  <h1>Add Numbers</h1>
  <form method="post">
    Number 1: <input type="text" name="num1"><br>
    Number 2: <input type="text" name="num2"><br>
    <input type="submit" value="Add">
  </form>

  <%
    String num1Str = request.getParameter("num1");
    String num2Str = request.getParameter("num2");
    if (num1Str != null && num2Str != null) {
      try {
        double num1 = Double.parseDouble(num1Str);
        double num2 = Double.parseDouble(num2Str);
        double result = num1 + num2;
  %>
        <p>Result: <%= result %></p>
  <%
      } catch (NumberFormatException e) {
  %>
        <p>Invalid input. Please enter valid numbers.</p>
  <%
      }
    }
  %>
</body>
</html>
```

- **What is framework in java? Explain the types of java framework available.**
  Java framework is the body or platform of pre-written codes used by Java developers to develop java applications or web applications. It acts like a skeleton that helps developers to develop an application by writing their own code.
  Some of the most popular type of java frame works are:
  - Spring
  - Hibernate
  - Grails
  - Play
  - JavaServer Faces(JSF)
  - Google Web Toolkits(GWT)
  - Quarkus

i. Spring: Its is a light-weighted, powerful java application development framework. It is used fro JEE. Its other modules are Spring security MVC, Spring Batch, Spring ORM, etc.
   Advantages:
   a. Loose coupling
   b. Lightweight
   c. Fast Development
   d. Easy to test

ii. Hibernate: Hibernate is ORM (Object-Relation Mapping) framework that allows us to establish communication between Java programming language and the RDBMS.
   Advantages
   a. Portability, productivity, maintainability
   b. Opensource framework

iii. Grails: It is a dynamic framework created by using Groovy programming language. It is an OOPs language.
   Advantages:
   a. Provides flexible profiles.
   b. Its object mapping features is easy to use.

iv. Play: It is unique java framework because it does not follow JEE standards. It follows MVC architecture pattern. It is used when we want to develop highly scalable java application.
   Advantages
   a. It also supports popular IDES.
   b. No configuration is required.
   c. It offers hot code reload and error message in the browser.

- **How to tracking session in Servlets? Explain**

  The HttpServletRequest interface provides two methods to get the object of HttpSession:
  public HttpSession getSession(): Returns the current session associated with this request, or if the request does not have a session, creates one.
  public HttpSession getSession(boolean create): Returns the current HttpSession associated with this request or, **if there is no current session and create is true**, returns a new session.
  A session is a way to store information (in variables) to be used across multiple pages.
  Session tracking is a way to maintain state of user. It is also known as session management in servlet.
  There are 4 technique used in session tracking.
  a. Cookies
  b. Hidden form field
  c. URL Rewriting
  d. HttpSession

  Commonly used method in session tracking:
  - **isNew():** Returns true is user does not know about the session. If cookies are disabled, then session is new.
  - **getId():** Returns string that contains unique identifier that is assigned to this session. Is used while using URL rewriting the session.
  - **getAttribute():** Returns the object bound in present session.
  - **setAttribute():** Binds object to present session, uses specified name.
  - **invalidate():** Expires current session and unbinds the object binded
  - **setMaxInactiveInterval():** Specifies time between client requests before servlet invalidates session. Negative time indicates session shouldn't timeout.

  For example:

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;
import java.io.IOException;
@WebServlet("/SessionExample")
public class SessionExampleServlet extends HttpServlet {
   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
      // Get or create a session
      HttpSession session = request.getSession(true);
      // Set session attributes
      String username = "john_doe";
      session.setAttribute("username", username);
      String storedUsername = (String) session.getAttribute("username");
      if (session.isNew()) {
         response.getWriter().println("New session created.");
      } else {
         response.getWriter().println("Session already exists.");
         response.getWriter().println("Username from session: " + storedUsername);   }  }  }
```

- **How to access database in JSP? Explain with suitable code.**

  We can 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.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ page import="java.sql.ResultSet" %>
<%@ page import="yourpackage.DatabaseBean" %>
<!DOCTYPE html>
<html>          <head>
  <meta charset="UTF-8">
  <title>Database Access in JSP</title>
</head>         <body>
  <h1>Database Access in JSP</h1>
  <%
    // Create an instance of the DatabaseBean
    DatabaseBean db = new DatabaseBean();
    // Example query
    String sql = "SELECT * FROM your_table";
    ResultSet resultSet = db.executeQuery(sql);
  %>
  <table border="1">
    <tr>
      <th>ID</th>
      <th>Name</th>
    </tr>
    <% while (resultSet.next()) { %>
      <tr>
        <td><%= resultSet.getInt("id") %></td>
        <td><%= resultSet.getString("name") %></td>
      </tr>
    <% } %>
  </table>
  <%    // Close the database connection when done
    db.close();
  %>
</body>         </html>
```

Html code:

```
<!DOCTYPE html>
<html>          <head>
  <meta charset="UTF-8">
  <title>Database Access in JSP</title>
</head>
<body>
  <h1>Database Access in JSP</h1>

  <table border="1">
    <tr>        <th>ID</th>
                <th>Name</th>        </tr>
  </table>      </body>        </html>
```

- **servelet application using servelet interface in java**

  <u>SimpleServlet</u>

```
import javax.servlet.*;
import java.io.IOException;
import java.io.PrintWriter;

public class SimpleServlet implements Servlet {
   private ServletConfig config;

   public void init(ServletConfig config) throws ServletException {
      this.config = config;
   }

  public void service(ServletRequest request, ServletResponse response) throws ServletException,
IOException {
      response.setContentType("text/html");
      PrintWriter out = response.getWriter();
      out.println("<html><body>");
      out.println("<h1>Hello, Servlet World!</h1>");
      out.println("</body></html>");
   }
 public void destroy() {
      // Cleanup resources here
   }

   public ServletConfig getServletConfig() {
      return this.config;
   }

   public String getServletInfo() {
      return "SimpleServlet";
   }}
```

Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee  http://xmlns.jcp.org/xml/ns/javaee/web-
app_4_0.xsd"
        version="4.0">
  <servlet>
      <servlet-name>SimpleServlet</servlet-name>
      <servlet-class>SimpleServlet</servlet-class>
   </servlet>
  <servlet-mapping>
      <servlet-name>SimpleServlet</servlet-name>
      <url-pattern>/SimpleServlet</url-pattern>
   </servlet-mapping>
</web-app>
```

- **servelet application using GenericServelet interface in java**

<u>SimpleGenericServlet</u>

```
import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import java.io.IOException;
import java.io.PrintWriter;
public class SimpleGenericServlet extends GenericServlet {
   @Override
   public void service(ServletRequest request, ServletResponse response) throws
ServletException, IOException {
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();
       out.println("<html><body>");
       out.println("<h1>Hello, GenericServlet World!</h1>");
       out.println("</body></html>");
   }
}
```

Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
        version="4.0">
 <servlet>
       <servlet-name>SimpleGenericServlet</servlet-name>
       <servlet-class>SimpleGenericServlet</servlet-class>
   </servlet>
<servlet-mapping>
       <servlet-name>SimpleGenericServlet</servlet-name>
 <url-pattern>/SimpleGenericServlet</url-pattern> </servlet-mapping>
</web-app>
```

- **write a servelet that receives data from HTML form and inserts form into the database in java**

<u>DatabaseConnectionServlet</u>

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
public class DatabaseConnectionServlet extends HttpServlet {
   private Connection connection;
public void init() throws ServletException {
       String jdbcURL = "jdbc:mysql://localhost:3306/your_database";
       String jdbcUser = "your_username";
       String jdbcPassword = "your_password";
 try {
          Class.forName("com.mysql.jdbc.Driver");
          connection = DriverManager.getConnection(jdbcURL, jdbcUser, jdbcPassword);
       } catch (Exception e) {
          throw new ServletException(e);
       }
   }
   public void destroy() {
       try {
          if (connection != null) {
             connection.close();
          }
       } catch (SQLException e) {
          e.printStackTrace();
       }
   }
}
```

<u>success.html</u>

```
<!DOCTYPE html>
<html>
<head>
          <title>Insert Data</title>
</head>
<body>
   <form action="/InsertDataServlet" method="post">\
  <label for="name">Name:</label>
       <input type="text" id="name" name="name" required><br>
  <label for="email">Email:</label>
       <input type="email" id="email" name="email" required><br>
<input type="submit" value="Submit">
   </form>
</body>
</html>
```

InsertDataServlet

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class InsertDataServlet extends HttpServlet {
    private Connection connection;

    public void init() throws ServletException {
        // Database connection setup code (as shown in the first program)
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        String name = request.getParameter("name");
        String email = request.getParameter("email");

        try {
            String sql = "INSERT INTO your_table (name, email) VALUES (?, ?)";
            PreparedStatement preparedStatement = connection.prepareStatement(sql);
            preparedStatement.setString(1, name);
            preparedStatement.setString(2, email);
            preparedStatement.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }

        response.sendRedirect("/success.html");
    }

    public void destroy() {
        // Database connection cleanup code (as shown in the first program)
    }
}
```

Explain the different implicit objects in JSP.
There are implicit objects supported by JSP. These objects are created by the container, not by the code author. These are the java objects to implement the Servlet and JSP API interfaces. There are nine (9) JSP implicit objects available.
JSP implicit objects are as follows:
1. request : An object that implements javax.servlet.http.HttpServletRequest, to represent request given by the user.
2. response : An object that implements javax.servlet.http.HttpServletResponse, to represent the response directed to the client.
3. out : An object that is the instance of javax.servlet.jsp.Jsp'Writer class, to represent the output content that is to be sent to the client. The out implicit object is used to write the output content.
4. session: An object that is an instance of a class that implements the javax.servlet.http.HttpSession object that represents a client specific conversation.
5. application: An object that is an instance of a class that implements javax.servlet.ServletContext interface, to provide the facility of obtaining the web application to the JSP.
6. exception: An instance of java.lang.Throwable class, to handle exceptions in a JSP. The exception is represented as an error page.
7. config: An implicit config object that implements javax.servlet.ServletConfig interface, to provide the facility to obtain the parameters information.
8. page: An instance of java.lang.Object class, to represent the current JSP itself.
9. pageContext: An implicit pageContext page, to represent the context information.

- **What are Directives in JSP.**
  In JavaServer Pages (JSP), directives are special instructions that provide information to the JSP container and control various aspects of the JSP page's behavior. There are three types of directives in JSP:
- **Page Directives (<%@ page %>):** Page directives provide instructions related to the entire JSP page, such as error handling, session management, and content type. They typically appear at the top of the JSP file.
  Syntax:
  <%@ page attribute="value" %>
  Example:<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
- **Include Directives (<%@ include %>):** Include directives allow you to include external files (usually other JSP files or HTML files) within your JSP page at translation time. The content of the included file becomes part of the generated servlet code.
  Syntax:
  <%@ include file="filename" %>
  Example: <%@ include file="header.jsp" %>

- **Taglib Directives (<%@ taglib %>):** Taglib directives are used to declare and define custom JSP tag libraries for use in your JSP page. These directives specify the location of the tag library descriptor (TLD) and the prefix to use when referencing the tags.
- Syntax: <%@ taglib uri="URI" prefix="prefix" %>

  Example: <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

Simple program using directive

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8" %>
<%@ include file="header.jsp" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
   <title>My JSP Page</title>
</head>
<body>
   <h1>Welcome to my JSP Page</h1>
   <p>The current date and time is: <c:out value="${now}"/></p>
  <%-- This is a JSP comment --%>
    <%
      // This is a scriptlet that generates server-side Java code
      String message = "Hello, JSP!";
      out.println(message);
 %>
  <%@ include file="footer.jsp" %>
</body>
</html>
```

The Design of JDBC

Java provides a pure Java API for SQL access along with a driver manager to allow third-party drivers to connect to specific databases. Database vendors provide their own drivers to plug in to the driver manager. There is a simple mechanism for registering third-party drivers with the driver manager.

Programs written according to the API talk to the driver manager, which, in turn, uses a driver to talk to the actual database. This means the JDBC API is all that most programmers will ever have to deal with.

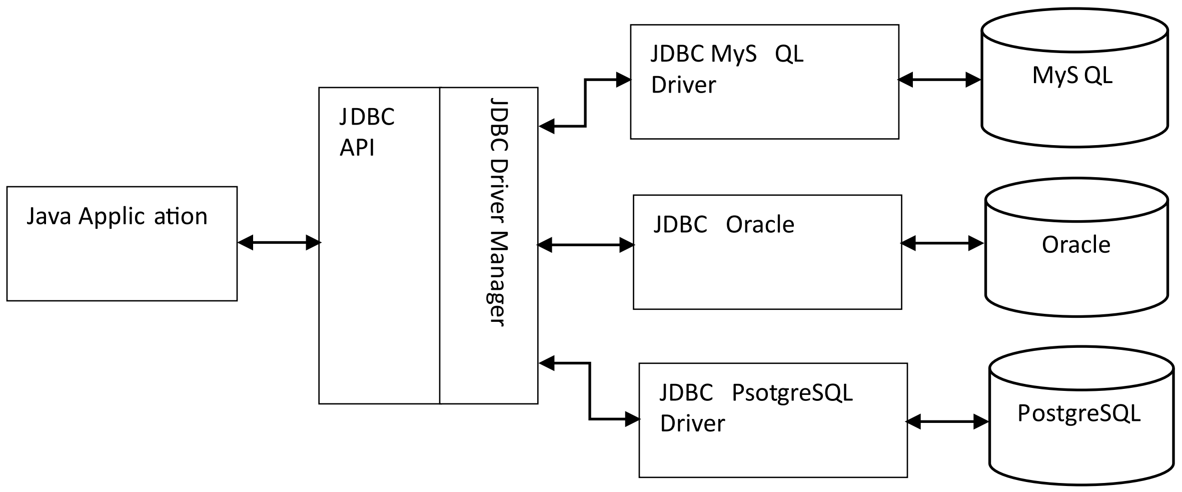


Fig: The Design of JDBC

JDBC Driver Types:
The JDBC specification classifies drivers into the following types:

Type 1: A type 1 driver translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Early versions of Java included one such driver, the JDBC/ODBC bridge. Java 8 no longer provides the JDBC/ODBC bridge.

Advantages:

1. easy to use.
2. can be easily connected to any database.

Disadvantages:

1. Performance degraded because JDBC method call is converted into the ODBC function calls.
2. The ODBC driver needs to be installed on the client machine.

- **Type 2:** A type 2 driver is written partly in Java and partly in native code. It communicates with the client API of a database. When using such a driver, you must install some platform-specific code onto the client in addition to a Java library.

Advantage: performance upgraded than JDBC-
ODBC bridge driver.

Disadvantage:

1. The Native driver needs to be installed on each client machine.
2. The Vendor client library needs to be installed on client machine.

- **Type 3:** A type 3 driver is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This simplifies deployment because the platform-specific code is located only on the server.

Advantage:

1. No client-side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

1. Network support is required on client machine.
2. Requires database-specific coding to be done in the middle tier.
3. Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

- **Type 4:** A type 4 driver is a pure Java library that translates JDBC requests directly to a database-specific protocol.

Advantage:

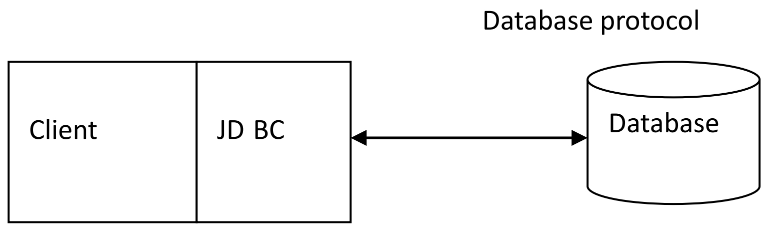
1. Better performance than all other drivers.
2. No software is required at client side or server side.

Disadvantage:

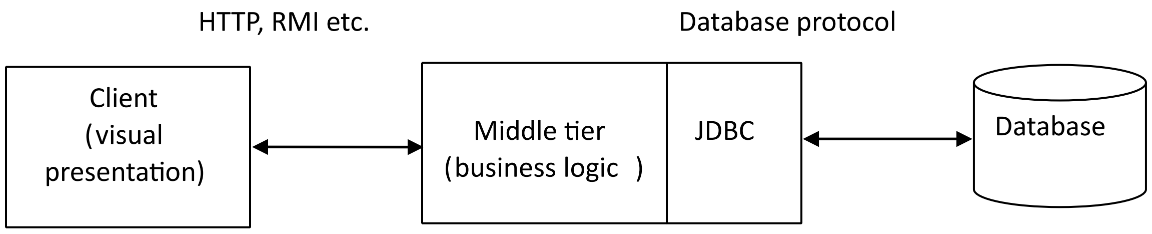
1. Drivers depend on the Database.

Typical uses of JDBC

The JDBC supports both **two-tier** and **three-tier** processing models for database access. The two-tier model is the traditional client/server model. This model has a rich GUI on the client and a database on the server. In this model, a JDBC driver is deployed on the client.



However, the world is moving away from client/server toward a three-tier model or even more advanced n-tier model. In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates visual presentation (on the client) from the business logic (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients. Communication between the client and middle tier can occur through HTTP or another mechanism such as RMI. JDBC manages the communication between the middle tier and the back-end database.



Configuring JDBC and Executing SQL Statements

Of course, you need a database program for which a JDBC driver is available. There are many excellent choices, such as SQL Server, MySQL, Oracle, and PostgreSQL. You must also create a database for your experimental use.

You need to gather a number of items before you can write your first database program.

Database URLs

When connecting to a database, you must use various database-specific parameters such as host names, port numbers, and database names. JDBC uses syntax similar to that of ordinary URLs to describe data sources. The JDBC URL for MySQL database is given below.

"jdbc:mysql://localhost:3306/college","root","" The general syntax is jdbc:*subprotocol*:*other stuff* where a subprotocol selects the specific driver for connecting to the database. The format for the *other stuff* parameter depends on the subprotocol used.

Driver JAR Files

You need to obtain the JAR file in which the driver for your database is located. If you use MySQL, you need the file *mysql-connector.jar*. Include the driver JAR file on the class path when running a program that accesses the database.

Starting the Database

The database server needs to be started before you can connect to it. The details depend on your database.

Import JDBC Libraries: Make sure you have the JDBC library for your specific database installed and included in your project. For example, if you're using MySQL, you'd need the MySQL JDBC driver.

Load JDBC Driver: Load the JDBC driver class using Class.forName().

Establish Database Connection: Create a connection to the database using DriverManager.getConnection().

Create Statement: Create a Statement or a PreparedStatement object to execute your SQL statement.

Execute Statement: Use the appropriate method (executeQuery() for SELECT statements, executeUpdate() for INSERT, UPDATE, DELETE, etc.) to execute the SQL statement.

Process Results: If you're executing a SELECT statement, process the result set using the ResultSet object.

Close Resources: Close the Statement, ResultSet, and the database connection in reverse order.

A Complete Example to connect with mysql

```
import java.sql.*;

public class MyDatabaseApplication  {

   public static void main(String[] args)  {

              try

       {

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

          Connection
conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/college","root","");

          Statement stmt=conn.createStatement();

          String sqlStmt = "SELECT * FROM student";           ResultSet
rs=stmt.executeQuery(sqlStmt);

          while(rs.next())

             System.out.println(rs.getString(1)+"  "+rs.getString(2)+"  "+rs.getInt(3));          rs.close();
stmt.close();          conn.close();
       }

       catch(Exception e)

       {

          System.out.println(e); }   }   }
```

Executing other SQL Statements

Creating Tables

String sqlStmt = "CREATE TABLE Student (SID VARCHAR(10), SName VARCHAR(50), Roll_no INTEGER)";
stmt.executeUpdate(sqlStmt);

Entering Data into a Table

String sqlStmt = "INSERT INTO Student VALUES ('S101', 'Nawaraj',4)"; stmt.executeUpdate(sqlStmt);

Updating Data into a Table

String sqlStmt = "UPDATE Student SET Roll_no = 5 WHERE SID = 'S101'"; stmt.executeUpdate(sqlStmt);

Deleting Data from a Table

String sqlStmt = "DELETE FROM Student WHERE Roll_no = 5"; stmt.executeUpdate(sqlStmt);

Dropping a Table

String sqlStmt = "DROP table Student"; stmt.executeUpdate(sqlStmt);

Prepared Statement

The Prepared Statement interface, a sub-interface of Statement interface is used to create prepared statement. Prepared statement is used to execute parameterized query. We can prepare a query with a host variable and use it many times. Each host variable in a prepared query is indicated with a ?. If there is more than one variable, you must keep track of the positions of the ? when setting the values.

You can execute the prepared statement as given in the program below:

```
ResultSet rs = stat.executeQuery();
String sid = "S112"; int roll = 15;
String sqlStmt = "SELECT * FROM Student WHERE SID = ? AND Roll_no = ?";
PreparedStatement stmtPrepared = conn.prepareStatement(sqlStmt);
stmtPrepared.setString(1, sid); stmtPrepared.setInt(2, roll);
ResultSet rs = stmtPrepared.executeQuery(); while(rs.next())
        System.out.println(rs.getString(2)); rs.close();
stmtPrepared.close(); conn.close();
```

Prepared Statement works with an example:

```
import java.sql.*;
public class PreparedStatementShortExample {
  public static void main(String[] args) {
    String jdbcUrl = "jdbc:mysql://localhost:3306/mydatabase";
    String username = "yourusername";
    String password = "yourpassword";
    try {
      Class.forName("com.mysql.cj.jdbc.Driver");
      Connection connection = DriverManager.getConnection(jdbcUrl, username, password);
      // Using PreparedStatement to insert data
      String insertQuery = "INSERT INTO employees (name, salary) VALUES (?, ?)";
      PreparedStatement preparedStatement = connection.prepareStatement(insertQuery);
      preparedStatement.setString(1, "John Doe");
      preparedStatement.setDouble(2, 50000.0);
      int rowsAffected = preparedStatement.executeUpdate();
      System.out.println(rowsAffected + " row(s) inserted.");
      preparedStatement.close();
      connection.close();
    } catch (ClassNotFoundException | SQLException e) {
      e.printStackTrace();   }   }  }
```

Working with JDBC Statements
How to use the JDBC Statement to execute SQL statements, obtain results, and deal with errors.

1 Executing SQL Statements

To execute a SQL statement, you first create a Statement object. To create statement objects, use the Connection object that you obtained from the call to DriverManager.getConnection.

```
 Statement stat = conn.createStatement();
```

Next, place the statement that you want to execute into a string, for example

```
String command = "UPDATE Books" + " SET Price = Price - 5.00" + " WHERE Title NOT LIKE
'%Introduction%'";
```

Then call the executeUpdate method of the Statement interface: stat.executeUpdate(command);
The executeUpdate method returns a count of the rows that were affected by the SQL statement, or zero for statements that do not return a row count.
When you execute a query, you are interested in the result. The executeQuery object returns an object of type ResultSet that you can use to walk through the result one row at a time.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books");
```

The basic loop for analyzing a result set looks like this:

```
while (rs.next())
{
look at a row of the result set
}
```

A large number of accessor methods give you this information.

```
String isbn = rs.getString(1); double price = rs.getDouble("Price");
```

2 Managing Connections, Statements, and Result Sets

Every Connection object can create one or more Statement objects.
We can use the same Statement object for multiple unrelated commands and queries. However, a statement has at most one open result set.
When we are done using a ResultSet, Statement, or Connection, we should call the close method immediately.
To make absolutely sure that a connection object cannot possibly remain open, use a trywith-resources statement:

```
try (Connection conn = . . .)
{
Statement stat = conn.createStatement();
ResultSet result = stat.executeQuery(queryString); process query result
}
```

3 Analyzing SQL Exceptions

Each SQLException has a chain of SQLException objects that are retrieved with the getNextException method.
We can simply use an enhanced for loop:

```
 for (Throwable t : sqlException)
{ do something with t
}
```

We can call getSQLState and getErrorCode on a SQLException to analyze it further.
We can retrieve warnings from connections, statements, and result sets.
To retrieve all warnings, use this loop:

```
SQLWarning w = stat.getWarning(); while (w != null)
```

{ do something with w w = w.nextWarning(); }

4 Populating a Database

Specifically, the program reads data from a text file in a format such as

CREATE TABLE Publishers (Publisher_Id CHAR(6), Name CHAR(30), URL CHAR(80));

INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley', 'www.aw-bc.com'); INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons', 'www.wiley.com');

Make sure that your database server is running, and run the program as follows:

java -classpath driverPath:. exec.ExecSQL Books.sql java -classpath driverPath:. exec.ExecSQL Authors.sql java -classpath driverPath:. exec.ExecSQL Publishers.sql java -classpath driverPath:. exec.ExecSQL BooksAuthors.sql

The following steps briefly describe the ExecSQL program:

1. Connect to the database. The getConnection method reads the properties in the file database.properties and adds the jdbc.drivers property to the system properties. The getConnection method uses the jdbc.url, jdbc.username, and jdbc.password properties to open the database connection.
2. Open the file with the SQL statements. If no file name was supplied, prompt the user to enter the statements on the console.
3. Execute each statement with the generic execute method. If it returns true, the statement had a result set.
4. If there was a result set, print out the result.
5. If there is any SQL exception, print the exception and any chained exceptions that may be contained in it.
6. Close the connection to the database. Scrollable and Updatable Result Sets

Compare JDBC with ODBC in detail.
JDBC (Java Database Connectivity) and ODBC (Open Database Connectivity) are both technologies used to connect applications with databases. However, they differ in various aspects. Let's compare JDBC and ODBC in detail:

1. Platform:

- **JDBC:** It's a Java-based API that provides a standard way for Java applications to interact with databases. It's platform-independent and tightly integrated with the Java programming language.
- **ODBC:** It's a C/C++-based API that allows applications written in different languages to communicate with databases. ODBC provides a bridge between the application and the database through a driver manager.

2. Language:

- **JDBC:** Primarily used with Java applications. Provides Java classes and interfaces for database interaction.
- **ODBC:** Originally designed for C/C++, but can be used with other languages like Java through language bindings.

3. Architecture:

- **JDBC:** Utilizes Java's native database access mechanisms, making it more integrated with the Java environment.
- **ODBC:** Requires a driver manager and database drivers to mediate communication between applications and databases.

4. Portability:

- **JDBC:** Provides better portability due to Java's "Write Once, Run Anywhere" principle.
- **ODBC:** Portability can be an issue as ODBC drivers and configurations might vary across different platforms.

5. Driver Mechanism:

- **JDBC:** Uses JDBC drivers to communicate with databases. There are four types of JDBC drivers: Type-1 (JDBC-ODBC bridge), Type-2 (Native-API), Type-3 (Network Protocol), and Type-4 (Thin driver).
- **ODBC:** Utilizes ODBC drivers, which are specific to each database management system. ODBC drivers interact with the ODBC driver manager.

6. Connection String:

- **JDBC:** Connection strings are typically URL-based, and they contain details like database location, driver class, and credentials.
- **ODBC:** Connection strings are configured differently for each ODBC driver. They can be complex and sometimes need to be set up on the system level.

7. Performance:

- **JDBC:** Generally, offers better performance due to its tighter integration with the Java runtime and JVM optimizations.
- **ODBC:** Performance might be influenced by the language binding and translation between languages, although ODBC drivers are typically well-optimized.

8. Security:

- **JDBC:** Inherits Java's security features, including the ability to use Java's security manager for fine-grained control.
- **ODBC:** Security measures might vary based on the specific programming language used for the application.

Scrollable Result Sets

By default, result sets are not scrollable or updatable. To obtain scrollable result sets from queries, we must obtain a different Statement object with the method:

```
Statement stat = conn.createStatement(type, concurrency);
```

For a prepared statement, use the call

```
PreparedStatement stat = conn.prepareStatement(command, type, concurrency);
```

For example, if we simply want to be able to scroll through a result set but don’t want to edit its data, use:

```
Statement stat = conn.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

All result sets that are returned by method calls

```
ResultSet rs = stat.executeQuery(query)
```

are now scrollable. A scrollable result set has a cursor that indicates the current position.

Updatable Result Sets

If we want to edit the result set data and have the changes automatically reflected in the database, create an updatable result set.

To obtain updatable result sets, create a statement as follows:

```
Statement stat = conn.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

The result sets returned by a call to executeQuery are then updatable.

For example, suppose we want to raise the prices of some books, but you don’t have a simple criterion for issuing an UPDATE statement. Then, you can iterate through all books and update prices, based on arbitrary conditions.

```
String query = "SELECT * FROM Books"; ResultSet rs = stat.executeQuery(query); while (rs.next())
 { if (. . .)
{ double increase = . . . double price = rs.getDouble("Price"); rs.updateDouble("Price", price + increase);
rs.updateRow(); // make sure to call updateRow after updating fields } }
```

When you are done inserting, call moveToCurrentRow to move the cursor back to the position before the call to moveToInsertRow. Here is an example:

```
rs.moveToInsertRow();
rs.updateString("Title", title); rs.updateString("ISBN", isbn); rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price); rs.insertRow();
rs.moveToCurrentRow();
```

We can delete the row under the cursor: rs.deleteRow();

Row Sets

Scrollable result sets are powerful, but they have a major drawback. You need to keep the database connection open during the entire user interaction. However, a user can walk away from the computer for a long time, leaving the connection occupied. That is not good – database connections are scarce resources. In this situation, use a row set. The Row Set interface extends the Result Set interface, but row sets don't have to be tied to a database connection. Row sets are also suitable if you need to move a query result to a different tier of a complex application, or to another device such as a cell phone. You would never want to move a result set – its data structures can be huge, and it is tethered to the database connection.

Constructing Row Sets

The javax.sql.rowset package provides the following interfaces that extend the RowSet interface:

- A CachedRowSet allows disconnected operation.
- A WebRowSet is a cached row set that can be saved to an XML file. The XML file can be moved to another tier of a web application where it is opened by another WebRowSet object.
- The FilteredRowSet and JoinRowSet interfaces support lightweight operations on row sets that are equivalent to SQL SELECT and JOIN operations. These operations are carried out on the data stored in row sets, without having to make a database connection.
- A JdbcRowSet is a thin wrapper around a ResultSet. It adds useful methods from the RowSet interface.

There is a standard way for obtaining a row set:

```
RowSetFactory factory = RowSetProvider.newFactory(); CachedRowSet crs =
factory.createCachedRowSet();
```

There are similar methods for obtaining the other row set types.

Cached Row Sets

A cached row set contains all data from a result set. Since CachedRowSet is a subinterface of the ResultSet interface, you can use a cached row set exactly as you would use a result set. Cached row sets confer an important benefit: You can close the connection and still use the row set. Each user command simply opens the database connection, issues a query, puts the result in a cached row set, and then closes the database connection. For example,

```
Statement stmt = conn.createStatement();

String sqlStmt = "SELECT * FROM Student";

ResultSet rs = stmt.executeQuery(sqlStmt);

RowSetFactory factory = RowSetProvider.newFactory(); CachedRowSet crs =
factory.createCachedRowSet(); crs.populate(rs); conn.close(); while(crs.next()) {

System.out.println(crs.getString(1)+"  "+crs.getString(2)+"  "+crs.getInt(3));

} rs.close(); crs.close(); stmt.close();
```

It is even possible to modify the data in a cached row set. Of course, the modifications are not immediately reflected in the database; you need to make an explicit request to accept the accumulated changes. The CachedRowSet then reconnects to the database and issues SQL statements to write the accumulated changes. You can inspect and modify the row set with the same methods you use for result sets. If you modified the row set contents, you must write it back to the database by calling crs.acceptChanges(conn);

For example,

```
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
String sqlStmt = "SELECT * FROM Student";
ResultSet rs = stmt.executeQuery(sqlStmt);
RowSetFactory factory = RowSetProvider.newFactory(); CachedRowSet crs =
factory.createCachedRowSet(); crs.populate(rs); while(crs.next())
{
            if(crs.getString(2).equals("Anuja"))  {
                    crs.updateString("SName", "Nawaraj");
                    crs.updateRow();
            }    }
crs.acceptChanges(conn); rs.close(); crs.close(); stmt.close();
conn.close();
```

RowSet	**ResultSet**
RowSet is present in the javax.sql package	ResultSet is present in the java.sql package
A Row Set can be connected, disconnected from the database.	A ResultSet always maintains the connection with the database.
RowSet is scrollable providing more flexibility	ResultSet by default is always forward only
A Row Set object can be serialized.	It cannot be serialized.
You can pass a Row Set object over the network.	ResultSet object cannot be passed other over the network.
Result Set Object is a JavaBean object. RowSet using the RowSetProvider.newFactory().createJdbcRowSet() method.	Result Set object is not a JavaBean object result set using the executeQuery() method

Database CRUD operation

ConnectionTest.java

```
package jdbc;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class ConnectionTest {
   public static void main(String[] args) {
      // Step 1: Establishing a Connection
      try {
         // Step 2: Load the driver
         Connection connection = DriverManager
         .getConnection("jdbc:mysql://localhost:3306/test?characterEncoding=latin1", "root",
"password");
         if (connection != null) {
            System.out.println("connected");
         } else {
            System.out.println("connection closed");   }   }
      catch (SQLException e) {
         e.printStackTrace();    }    }   }
```

CreatingTableDemo.java

```
package jdbc;
// import packages
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;
public class CreatingTableDemo {
        String dbURL = "jdbc:mysql://localhost:3306/test?characterEncoding=latin1";
        String username = "root";
        String password = "password";
        CreatingTableDemo() {
                // raw query
                // CREATE TABLE users (
                // user_id int(11) NOT NULL AUTO_INCREMENT,
                // username varchar(45) NOT NULL,
                // password varchar(45) NOT NULL,
                // fullname varchar(45) NOT NULL,
                // email varchar(45) NOT NULL,
                // PRIMARY KEY (user_id)
                // );
                try {
                        // Load or register the driver
                        try {
                                Class.forName("com.mysql.cj.jdbc.Driver");
                        } catch (ClassNotFoundException e) {
                                e.printStackTrace();   }
                        // create the connection
```

```
            Connection conn = DriverManager.getConnection(dbURL, username, password);
                        if (conn != null) {
                              System.out.println("Connected");
                              String sql = "CREATE TABLE users " +
"(user_id INTEGER not NULL AUTO_INCREMENT, " +
                                          " username VARCHAR(255), " +
                                          " password VARCHAR(255), " +
                                          " fullname VARCHAR(255), " +
                                          " email VARCHAR(255), " +
                                          " PRIMARY KEY ( user_id ))";
                              // create statement
                              Statement stmt = conn.createStatement();
                              // execute statement
                              stmt.executeUpdate(sql);
                              // process the result
                              System.out.println("Created table in given database...");
                              // closing the statement and connection
                              stmt.close();
                              conn.close();  }
                  } catch (SQLException ex) {
                        ex.printStackTrace();      }   }
            public static void main(String[] args) {
                  new CreatingTableDemo();   }   }
```

CreateDemo.java

```
package jdbc;
// Step 1 : Import the packages
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class CreateDemo {
            String dbURL = "jdbc:mysql://localhost:3306/test?characterEncoding=latin1";
            String username = "root";
            String password = "password";
            CreateDemo(){
                  try {
                        // Step 2 : Load or register the driver
                        try {
                              Class.forName("com.mysql.cj.jdbc.Driver");
                        } catch (ClassNotFoundException e) {
                              e.printStackTrace();   }
                        // Step 3 : Establish a connection
            Connection conn = DriverManager.getConnection(dbURL, username, password);
                     if (conn != null) {
                        System.out.println("Connected");
String sql = "INSERT INTO Users (username, password, fullname, email) VALUES (?, ?, ?, ?)";
                        // Step 4 : Creating a statement
                        PreparedStatement statement = conn.prepareStatement(sql);
                        statement.setString(1, "ram");
                        statement.setString(2, "password");
                        statement.setString(3, "Ram Bahadur");
                        statement.setString(4, "ram@gmail.com");
                        // Step 5 : executing the statement
                        int rowsInserted = statement.executeUpdate();
                        // Step 6 : processing the result
                        if (rowsInserted > 0) {
                          System.out.println("A new user was inserted successfully!");  }
                        // Step 7 : closing the statement and connection
                        statement.close();
                        conn.close();   }
                } catch (SQLException ex) {
                      ex.printStackTrace();   }    }
        public static void main(String[] args) {
                new CreateDemo();    }    }
```

Update.java

```
package jdbc;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class UpdateDemo {
        String dbURL = "jdbc:mysql://localhost:3306/test?characterEncoding=latin1";
        String username = "root";
        String password = "password";
        UpdateDemo(){
                try {
        Connection conn = DriverManager.getConnection(dbURL, username, password);
                      if (conn != null) {
                        System.out.println("Connected");
String sql = "UPDATE Users SET password=?, fullname=?, email=? WHERE username=?";
                        PreparedStatement statement = conn.prepareStatement(sql);
                        statement.setString(1, "updatedpassword");
                        statement.setString(2, "Shyam Bahadur");
                        statement.setString(3, "shyam@gmail.com");
                        statement.setString(4, "ram");
                        int rowsUpdated = statement.executeUpdate();
                        if (rowsUpdated > 0) {
                          System.out.println("An existing user was updated successfully!"); }}
                }catch(SQLException e) {
                          e.printStackTrace();   }    }
        public static void main(String[] args) {
                new UpdateDemo();   }   }
```

ReadDemo.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
public class ReadDemo {
        String dbURL = "jdbc:mysql://localhost:3306/test?characterEncoding=latin1";
        String username = "root";
        String password = "password";
        ReadDemo(){
                try {
                        // load or register the driver
                        try {
                                Class.forName("com.mysql.cj.jdbc.Driver");
                        } catch (ClassNotFoundException e) {
                                e.printStackTrace();   }
                        // create the db connection
Connection conn = DriverManager.getConnection(dbURL, username, password);
                   if (conn != null) {
                     System.out.println("Connected");
                     String sql = "SELECT * FROM Users";
                     // create statement
                     Statement statement = conn.createStatement();
                     // execute statement
                     ResultSet result = statement.executeQuery(sql);
                     // process result
                     while (result.next()){
                       String name = result.getString("username");
                       String pass = result.getString(3);
                       String fullname = result.getString("fullname");
                       String email = result.getString("email");
                       System.out.println("Username:"+name+" Password:"+pass+"
FullName:"+fullname+" email:"+email);  }
                     // close the statement and connection
                     statement.close();
                     conn.close();  }
                } catch (SQLException ex) {
                   ex.printStackTrace();   }   }
        public static void main(String[] args) {
                new ReadDemo();   }   }
```

DeleteDemo.java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
public class DeleteDemo {
        String dbURL = "jdbc:mysql://localhost:3306/test?characterEncoding=latin1";
        String username = "root";
        String password = "password";
        DeleteDemo(){
                try {
                        Connection conn = DriverManager.getConnection(dbURL, username, password);
                  if (conn != null) {
                        System.out.println("Connected");
                        String sql = "DELETE FROM Users WHERE username=?";
                        PreparedStatement statement = conn.prepareStatement(sql);
                        statement.setString(1, "ram");
                        int rowsDeleted = statement.executeUpdate();
                        if (rowsDeleted > 0) {
                          System.out.println("A user was deleted successfully!");  } }
                }catch(SQLException e) {
                        e.printStackTrace();  }   }
        public static void main(String[] args) {
                new DeleteDemo();  }  }
```

Java Beans

A JavaBean is a reusable software component that follows certain conventions to make it easy to integrate into various Java-based applications. JavaBeans are typically used in graphical user interface (GUI) programming, but they can be used in various other contexts as well.

Advantages of JavaBeans:

1. **Reusability:** JavaBeans are designed to be reusable components. Once you create a JavaBean, you can easily use it in different parts of your application or even in entirely different applications.
2. **Encapsulation:** JavaBeans encapsulate their state and behavior, following the principles of object-oriented programming (OOP). This helps in maintaining clean and modular code.
3. **Integration:** JavaBeans can be easily integrated into various development environments and frameworks, such as IDEs and GUI builders, making it simpler to design user interfaces.
4. **Event Handling:** The event-handling mechanism in JavaBeans allows for the creation of interactive and responsive user interfaces by allowing beans to generate and handle events.
5. **Serialization:** The built-in support for serialization makes it easy to store JavaBeans' state persistently or transmit them over a network, which is useful for saving user preferences or exchanging data between client and server applications.

Characteristics:

1. **Serializable:** JavaBeans are typically serializable, which means they can be easily stored, retrieved, and transmitted across a network. This is important in GUI development when saving the state of a component or when sending data between client and server applications.
2. **Properties:** JavaBeans have properties, which are accessed using getter and setter methods. Properties are essentially the attributes of the bean that can be read and modified. For example, a **Person** bean might have properties like **name**, **age**, and **address**.
3. **Events:** JavaBeans can generate events and allow other components to listen for and respond to these events. This is crucial in GUI programming to handle user interactions and trigger appropriate actions.
4. **No-Argument Constructor:** JavaBeans must have a no-argument constructor, which allows for easy instantiation of the bean using the default constructor. This is important when beans are dynamically created, often during GUI construction.

Java Bean vs Java Classes:

Aspect	JavaBean	Java Class
Serializable	Typically serializable for easy data transport.	May or may not be serializable.
Properties	Contains properties with getter and setter methods.	May have fields, but properties not required.
Events	Can generate events and handle event listeners.	Events and listeners are not inherent.
No-Arg Constructor	Must have a no-arg constructor for instantiation.	Not required to have a no-arg constructor.
Naming Conventions	Follows naming conventions for properties/methods.	No specific naming conventions enforced.
Use Cases	Often used for creating reusable GUI components.	Used for various programming tasks.
Reusability	Designed for easy reusability in different apps.	Can be reused but may require adaptation.
Encapsulation	Emphasizes encapsulation of state and behavior.	Encapsulation depends on developer choices.
Event Handling	Supports event handling with listeners and events.	Event handling needs to be manually coded.
Platform Independence	Can be used on any platform supporting Java.	Can be used on any platform supporting Java.

Java Beans and Its Types:

A bean property is a named attribute of a bean that can affect its behavior or appearance. Example of bean properties include color, label, font, font size and display size. The JavaBeans specification defines the following types of bean Properties:

Simple Property: A bean property with a single valued whose changes are independent of changes in any other property is called simple Property. To add simple properties to a bean, add appropriate getxxx and setxxx methods(or isxxx and setxxx methods for Boolean property).The names of these methods follow specifics rules called design patterns.

Indexed Property: A bean property that supports a range of valued instead of a single valued is called indexed property. An indexed property is an array of properties or objects that supports a range of valued and enable the access or to specify an element of a property to read or write.

Bound Property: A bean property for which a change to the property results in a notification being sent to some other bean is called bound property. Whenever a bound property changes, notification of the change is sent to interested listeners. The access methods for a bound property are defind in the same way ad those for simple properties. However you also need to provide the event listener registration methods for PropertyChangeListener classes and fire a PropertyChange event to the PropertyChangeListerner by calling their propertyChange methods.

Constrained Property: A bean property for which a change to the property results in a validation by another bean. The other bean may reject the change if it is not appropriate.

Bean Writing Process:

The term "bean writing process" typically refers to the process of creating and defining JavaBeans. JavaBeans are reusable software components in Java that follow certain conventions, making them easy to integrate into various Java-based applications.

1. **Create a directory for the new Bean:**
    - Start by creating a directory or folder where you will organize all the files related to your JavaBean project. This directory will contain your Java source code, manifest file, and any other necessary resources.
2. **Create the Java source file(s):**
    - Write the Java code for your JavaBean. This code should follow the JavaBean conventions, including having a no-argument constructor and providing getter and setter methods for properties.
3. **Compile the source file:**
    - Use a Java compiler (e.g., **javac**) to compile your Java source code. This step generates **.class** files from your **.java** source files.
4. **Create a manifest file:**
    - If you intend to package your JavaBean into a JAR (Java Archive) file, you'll need a manifest file. The manifest file specifies details about your JAR file, such as the main class (if applicable) and other metadata.

plaintextCopy code Manifest-Version: 1.0 Created-By: 1.8.0_301 (Oracle Corporation) Main-Class: com.example.MyBeanMainClass

5. **Generate a JAR File:**
    - Package your compiled **.class** files, manifest file, and any other resources into a JAR file using the **jar** utility or a build tool like Apache Maven. This JAR file will contain your JavaBean.

bashCopy code jar cfm MyBean.jar Manifest.txt com/example/MyBean.class

6. **Start the BDK (Bean Development Kit):**
    - The BDK is a development environment for designing, developing, and testing JavaBeans. You can start the BDK to work with your JavaBean visually and integrate it into other applications.
7. **Test the Bean:**
    - In the BDK or your chosen development environment, you can test your JavaBean by placing it on a visual design canvas (if applicable) and interacting with it. The BDK provides tools for testing and inspecting JavaBeans to ensure they behave as expected.

Introspection and its working methods

Introspection is a mechanism in Java that allows you to examine the properties, methods, and events of a JavaBean at runtime. It provides the ability to discover and manipulate a JavaBean's characteristics dynamically. Introspection is commonly used in JavaBeans to enable design-time tools, such as JavaBean builders and IDEs, to work with JavaBeans effectively. Here's how introspection works in JavaBeans:

1. Class Discovery: Introspection starts with the discovery of the JavaBean's class. Typically, this is done using the `java.lang.Class` object, which allows you to examine the structure of a class, including its fields, methods, and constructors.

2. Property Identificatio: Introspection identifies the properties of the JavaBean by examining its methods. JavaBeans conventionally follow a naming convention for getter and setter methods. By searching for methods with specific naming patterns (e.g., `getPropertyName()` and `setPropertyName()`), introspection can identify the properties and their data types.

3. Access to Properties: Once the properties are identified, introspection provides access to these properties through their getter and setter methods. This allows you to read and modify the values of the bean's properti**3. Access to Properties:**es programmatically.

4. Event Handling: Introspection can also identify and work with events in JavaBeans. It allows you to discover event listener registration methods, event firing methods, and other related event mechanisms.

5. Annotations: Modern JavaBeans may also use annotations to provide metadata about their properties, methods, and events. Introspection can take advantage of annotations to gain insights into the JavaBean's characteristics.

6. BeanInfo: JavaBeans can provide additional information through a special class called `BeanInfo`. The `BeanInfo` class is a way for JavaBean authors to provide metadata about the bean's properties, methods, and events explicitly.

7. Dynamic Bean Manipulation: Introspection enables dynamic manipulation of JavaBeans. You can programmatically inspect a bean's properties and methods, set property values, and invoke methods without having prior knowledge of the bean's structure at compile time.

Introspection is primarily used by design-time tools like Integrated Development Environments (IDEs) to provide a visual design environment for working with JavaBeans. For example, when you're designing a graphical user interface using a visual editor in an IDE, the IDE uses introspection to list and allow you to interact with the properties of a JavaBean visually.

Customizer in java bean

A customizer, in the context of JavaBeans, is a user interface component or class that allows you to visually customize the properties of a JavaBean at design-time (i.e., during the development of a graphical user interface). Customizers are typically used in Integrated Development Environments (IDEs) and visual development tools to provide a user-friendly way to set the properties of a JavaBean without having to write code manually.

Here are some key points about customizers:

1. **Visual Interface**: Customizers provide a visual interface for interacting with a JavaBean's properties. They may include forms, dialogs, or other graphical elements that allow you to input or select property values.

2. **Integration with IDEs**: Customizers are often integrated into IDEs to enhance the user experience when designing GUI-based applications. Developers can interact with a JavaBean's properties directly within the IDE's design view.

3. **Property Editing**: Customizers allow you to edit and configure the properties of a JavaBean without writing code. This can include setting text for labels, specifying colors, configuring layout parameters, and more.

4. **Property Validation**: Customizers can perform validation on the values entered or selected for properties to ensure they adhere to constraints or rules defined by the JavaBean.

5. **Event Handling**: Customizers can handle events related to property changes, allowing for dynamic updates of the user interface when a property's value changes.

6. **Persistence**: Customizers often work in conjunction with property editors and persistence mechanisms to ensure that the customized property values are saved properly, allowing the JavaBean to be reconfigured with the specified values at runtime.

7. **Customization Scenarios:** Customizers are particularly useful when dealing with complex JavaBeans that have numerous properties or properties with intricate dependencies. They simplify the process of configuring such beans.

8. **Standard JavaBeans AP**I: Customizers adhere to the JavaBeans API standards and conventions, making them compatible with various JavaBeans-compliant components and frameworks.

9. **Accessibility:** Good customizers are designed with accessibility in mind, ensuring that they can be used effectively by developers who rely on screen readers or other assistive technologies.

To create a customizer for a JavaBean, you typically implement the `java.beans.Customizer` interface, which defines methods for initializing the customizer and applying changes to the bean's properties. Customizers can be associated with JavaBeans through the use of the `beanInfo` class or annotations.

Design patterns

Design patterns are recurring solutions to common design problems in software engineering. They represent best practices, proven over time, for designing flexible, maintainable, and scalable software systems. Design patterns provide a common vocabulary and a set of well-defined solutions that help software developers communicate and make informed design decisions. There are several categories of design patterns, including creational, structural, and behavioral patterns. Here are some examples of design patterns in each category:

1. **Creational Design Patterns:**

- Singleton: Ensures that a class has only one instance and provides a global point of access to it.

- Factory Method: Defines an interface for creating an object but allows subclasses to alter the type of objects that will be created.

- Abstract Factory: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

2. **Structural Design Patterns**:

- Adapter: Allows objects with incompatible interfaces to work together by providing a wrapper that converts one interface into another.

- Decorator: Attaches additional responsibilities to an object dynamically, extending its behavior without altering its class.

- Composite: Composes objects into tree structures to represent part-whole hierarchies. Clients can treat individual objects and compositions of objects uniformly.

3. **Behavioral Design Patterns:**

- Observer: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

- Strategy: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

- Command: Encapsulates a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.

4. **Architectural Patterns:**

- MVC (Model-View-Controller): Separates an application into three interconnected components to separate the user interface, data, and control flow.

- MVVM (Model-View-ViewModel): Similar to MVC but emphasizes the separation of the presentation logic and user interface.