What is JDBC?

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database.

JDBC Architecture

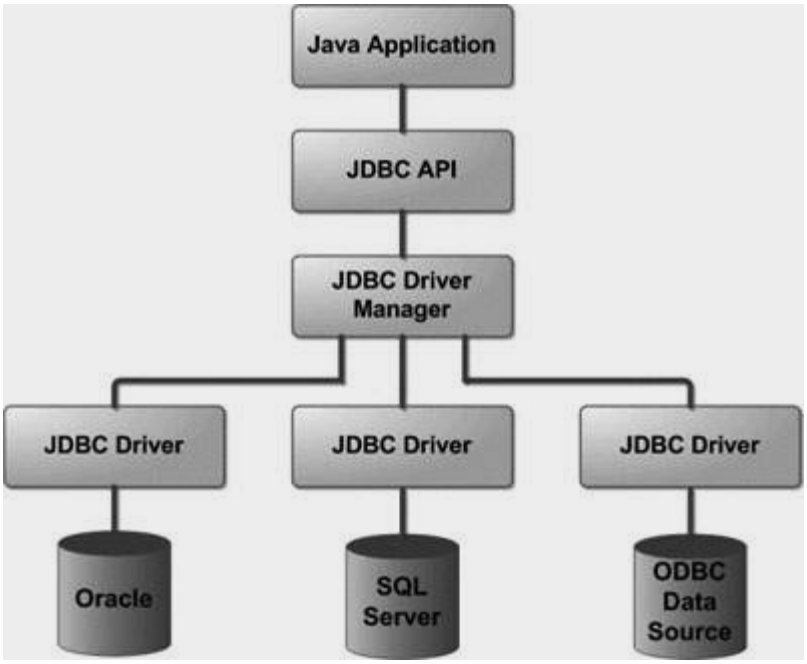
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

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

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application −



The JDBC API is comprised of two Java packages: java.sql and javax.sql. The following are core JDBC classes, interfaces, and exceptions in the java.sql package:

DriverManager

The DriverManager class (java.sql.DriverManager) is one of main components of JDBC. DriverManager manages database drivers, load database specific drivers and select the most appropriate database specific driver from the previously loaded drivers when a new connection is established. Managing, loading and selecting drivers are done automatically by the DriverManager from JDBC 4.0, when a new connection is created.

The DriverManager can also be considered as a connection factory class as it uses database specific drivers to create connection (java.sql.Connection) objects. DriverManager consist of only one private constructor and hence it cannot be inherited or initialized directly. All other members of DriverManager are static. DriverManager maintains a list of DriverInfo objects that hold one Driver object each.

Connection

Connection interface provides a standard abstraction to access the session established with a database server. JDBC driver provider should implement the connection interface. We can obtain a Connection object using the getConnection() method of the DriverManager as:

Connection con = DriverManager.getConnection(url, username, password);

Before JDBC 4.0, we had to first load the Driver implementation before getting the connection from the DriverManager as:

Class.forName(oracle.jdbc.driver.OracleDriver);

We can also use DataSource for creating a connection for better data source portability and is the preferred way in production applications.

Statement

The Statement interface provides a standard abstraction to execute SQL statements and return the results using the ResultSet objects.

A Statement object contains a single ResultSet object at a time. The execute method of a Statement implicitly close its current ResultSet if it is already open.

Two specialized Statement interfaces.

- **PreparedStatement**
- **CallableStatement**

You can create a Statement object by using the createStatement() method of the Connection interface as:

Statement st = con.createStatement();

You can then execute the statement, get the ResultSet and iterate over it:

```
ResultSet rs = st.executeQuery("select * from employeeList");
while (rs.next()) {
  System.out.println(rs.getString("empname"));
}
```

PreparedStatement

PreparedStatement is a sub interface of the Statement interface. PreparedStatements are pre-compiled and hence their execution is much faster than that of Statements. PreparedStatement object is got from a Connection object using the prepareStatement() method:

PreparedStatement ps1 = con.prepareStatement("insert into employeeList values ('Heartin',2)");

ps1.executeUpdate();

One difference here is that in a Statement you pass the sql in the execute method, but in PreparedStatement you have to pass the sql in the prepareStatement() method while creating the PreparedStatement and leave the execute method empty. You can even override the sql statement passed in prepareStatement() by passing another one in the execute method, though it will not give the advantage of precompiling in a PreparedStatement.

PreparedStatement also has a set of setXXX() methods, with which you can parameterize a PreparedStatement as:

PreparedStatement ps1 = con.prepareStatement("insert into employeeList values (?,?)");

ps1.setString(1, "Heartin4");

ps1.setInt(2, 7);

ps1.executeUpdate();

This is very useful for inserting SQL 99 data types like BLOB and CLOB. Note that the index starts from 1 and not 0 and the setXXX() methods should be called before calling the executeUpdate() method.

CallableStatement

CallableStatement extends the capabilities of a PreparedStatement to include methods that are only appropriate for stored procedure calls; and hence CallableStatement is used to execute SQL stored procedures. Whereas PreparedStatement gives methods for dealing with IN parameters, CallableStatement provides methods to deal with OUT parameters as well.

Consider a stored procedure with signature as 'updateID(oldid in int,newid in int,username out varchar2)'. *Code to create this stored proc is given in the end.* We can use a CallableStatement for executing it as:

CallableStatement cs=con.prepareCall("{call updateID(?,?,?)}");

cs.setInt(1, 2);

cs.setInt(2, 4);

cs.registerOutParameter(3, java.sql.Types.VARCHAR);

cs.executeQuery();

System.out.println(cs.getString(3));

Database Code: updateID stored procedure:

create or replace procedure updateID(oldid in int,newid in int,username out varchar2)

is

begin

update student set id=newid where id=oldid;

select firstname into username from student where id=newid;

commit;

end updateID;

If there are no OUT parameters, we can even use PreparedStatement. But using a CallableStatement is the right way to go for stored procedures.

ResultSet

This interface represents a table of data representing a database result set, which is usually generated by executing a statement that queries the database.

ParameterMetaData

ParameterMetaData retrieves the type and properties of the parameters used in the PreparedStatement interface. For some queries and driver implementations, the data that would be returned by a ParameterMetaData object may not be available until the PreparedStatement has been executed. Some driver implementations may even not be able to provide information about the types and properties for each parameter marker in a CallableStatement object. The below example however worked fine for me with Oracle thin driver and Oracle XE database and printed 3.

CallableStatement cs=con.prepareCall("{call updateID(?,?,?)}");

cs.setInt(1, 2);

cs.setInt(2, 4);

```
cs.registerOutParameter(3, java.sql.Types.VARCHAR);

ParameterMetaData paramMetaData = cs.getParameterMetaData();

System.out.println("Count="+paramMetaData.getParameterCount());

cs.executeQuery();
```

We can use ParameterMetaData with CallableStatement as well as in this example because CallableStatement is also a PreparedStatement.

ResultSetMetaData

ResultSetMetaData is used to retrieve information about the count, types and the properties of columns used in a ResultSet object. Consider an example where you are passed a ResultSet to a method and you don't know the number or type of columns in the ResultSet. You can use ResultSetMetaData to get the column details and then find the corresponding row values using the column details.

```
public static void printColumnNames(ResultSet resultSet) throws SQLException {

  if (resultSet != null) {

    ResultSetMetaData rsMetaData = resultSet.getMetaData();

    int numberOfColumns = rsMetaData.getColumnCount();

    for (int i = 1; i < numberOfColumns + 1; i++) {

      String columnName = rsMetaData.getColumnName(i);

      System.out.println("column name=" + columnName);

    }

  }

}
```

RowId

RowId maps a java object with a RowId. The RowId is a built-in datatype and is used as the identification key of a row in a database table, especially when there are duplicate rows.

```
while (rs.next()) {
```

```
System.out.println(rs.getString("columnName"));
oracle.sql.ROWID rowid = (ROWID)rs.getRowId("columnName");
System.out.println(rowid);
}
```

We need to provide columnName or column index to getRowId same as rs.getString().

SQLException

This class is an exception class that provides information on a database access error or other errors. JDBC 4.0 introduced following refined subclasses of SQLException:

- java.sql.SQLClientInfoException
- java.sql.SQLDataException
- java.sql.SQLFeatureNotSupportedException
- java.sql.SQLIntegrityConstraintViolationException
- java.sql.SQLInvalidAuthorizationSpecException
- java.sql.SQLSyntaxErrorException
- java.sql.SQLTransactionRollbackException
- java.sql.SQLTransientConnectionException

JDBC CODE:

Signin.html:

```
<html>
  <head>
    <title>Sign In</title>
  </head>
  <body>
    <form name="f1" action="Serv12" method="get">
      Enter empid:<input type="text" name="eid"/>
      Enter name:<input type="text" name="name"/>
      <input type="submit" value="click">
```

```
        </form>
    </body>
</html>
```

Serv12.java:

```
import java.io.IOException;
import java.io.PrintWriter;
import java.sql.DriverManager;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


@WebServlet(urlPatterns = {"/Serv12"})
public class Serv12 extends HttpServlet
{
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
IOException
    {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try
        {
            int e_id;
            String Name;
```

```
e_id=Integer.parseInt(request.getParameter("eid"));
Name=request.getParameter("name");

/* STATEMENT
String url="jdbc:mysql://localhost:3306/emp?zeroDateTimeBehavior=convertToNull";
Class.forName("com.mysql.jdbc.Driver");
java.sql.Connection connect=DriverManager.getConnection(url,"root","focus");
java.sql.Statement stmt=connect.createStatement();
String query="SELECT * FROM tab1e where empid="+e_id;
java.sql.ResultSet rs=stmt.executeQuery(query);*/

//PREPARED STATEMENT
String url="jdbc:mysql://localhost:3306/emp?zeroDateTimeBehavior=convertToNull";
String query2="SELECT * FROM tab1e where empid="+e_id;
Class.forName("com.mysql.jdbc.Driver");
java.sql.Connection connect=DriverManager.getConnection(url,"root","focus");
java.sql.PreparedStatement stmt2=connect.prepareStatement(query2);
java.sql.ResultSet rs=stmt2.executeQuery();
while(rs.next())
{
//out.print(rs.getInt(1));
//out.print(rs.getString(2));
if(rs.getInt(1)==e_id && rs.getString(2).equals(Name) )
{
out.println("<!DOCTYPE html>");
```

```
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet Serv12</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1> Welcome " + e_id +"</h1>");
        out.println("</body>");
        out.println("</html>");
        }
        else
        {
        out.println("<!DOCTYPE html>");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet Serv12</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1> Invalid User</h1>");
        out.println("</body>");
        out.println("</html>");
        }
    }
   }
  catch(Exception e)
  {
```

```
    out.println(e);
    }
  }
  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response)
  {

  }
}
```

NETWORKING:

- Networking Basics
  - Socket Overview
    - address plus port number
    - Connection Oriented (TCP/IP) Connectionless (Datagram)
  - Client/Server Model
    - Server has some resource that can be used
    - Client wants to gain access to server
    - Example—web server, print server, compute server, disk server
  - Reserved Sockets
    - HTTP:80, FTP:21, telnet:23, SMTP:25, finger:79, SSH:22, netnews:119 etc
  - Internet Addressing (IP Addressing)
    - Every host has a unique IP address
    - IPv4 and IPv^6

- Class A, B, C, D, E

- Domain Name System (DNS)
  - 203.197.107.107=www.jusl.ac.in
  - 203.197.107.107=www.itd.jusl.ac.in

INETADDRESS CLASS:

It is used to encapsulate IP addresses and domain names

Factory Methods

static InetAddress getLocalHost() throws UnknownHostException;

static InetAddress getByName(String hostName) throws UnknownHostException;

static InetAddress[] getAllByName(String hostName) throws UnknownHostException;

Instance Methods

boolean equals(Object inetAddress);

byte[] getAddress();

String getHostAddress();

String getHostName();

boolean isMulticastAddress();

boolean isLoopbackAddress();

InetAddressTest.java:

```
import java.io.*;
import java.net.*;
class InetAddressTest {
  public static void main(String args[]) throws UnknownHostException {
        InetAddress address = InetAddress.getLocalHost();
```

```
        System.out.println(address);
        address = InetAddress.getByName("www.rediffmail.com");
        System.out.println(address);
        InetAddress[] addresses = InetAddress.getAllByName("www.yahoo.com");
        for(int i=0;i<addresses.length;i++)
        System.out.println(addresses[i]);
  }
}
```

Output:

```
        E:\UKR\java\socket>java InetAddressTest
        ukr/61.2.1.217
        www.rediffmail.com/202.54.124.154
        www.yahoo.com/68.142.197.87
        www.yahoo.com/68.142.197.88
        www.yahoo.com/68.142.197.90
        www.yahoo.com/68.142.197.64
        www.yahoo.com/68.142.197.77
        www.yahoo.com/68.142.197.67
        www.yahoo.com/68.142.197.84
        www.yahoo.com/68.142.197.80
```

Java.net package:

- Provides classes and interfaces for writing networking applications

- Important classes are
  - URL
    - » Used to represent a resource that is uniquely identified by the provided information
  - ServerSocket
    - » Used to create TCP server application
  - Socket
    - » Used to write TCP client applications
  - DatagramSocket
    - » Used to write UDP application
  - DatagramPacket
    - » Used to create datagram packets in UDP application

Creating TCP—based Applications(TCP SOCKETS):

Create Server application

- » Create server socket

  ServerSocket(int port)
- » Wait for client request
- » Provides accept() method that blocks the server for client request.
- » Returns a Socket object representing the connection with a client.
- » On receipt of the client request,
  - ▸ Get Input/Output streams to read from/write to client
  - ▸ Read client request, process and write back

Create Client application

1. Create a socket

Available Constructors Connecting to a Server Socket:

Socket(String host, int port)

Socket(InetAddress address, int port)

Socket(String host, int port, InetAddress localAddr, int localPort)

Socket(InetAddress address, int port, InetAddress localAddr, int localPort)

Examining a Socket

InetAddress getInetAddress()	*address to which the socket is connected*
int getPort()	*remote port to which this socket is connected*
int getLocalPort()	*local port to which this socket is bound*

2. Obtain Input/Output streams to read form/write to server

   Read/write data from/to the server

   Receiving and sending data through Socket:

InputStream getInputStream()	*Returns an input stream for this socket*
OutputStream getOutputStream()	*Returns an output stream forthis socket*
void close()	*Closes this socket*

TCPServer.java:

```
package tcp;
import java.io.*;
import java.net.*;
import java.util.Scanner;
public class TCPServer {
  public static void main(String argv[]) throws Exception {
```

```
    String clientSentence, capitalizedSentence;
    ServerSocket welcomeSocket = new ServerSocket(6789);
    System.out.println ("TCP Server Waiting for client on port 6789");
    while(true) {
        Socket connectionSocket = welcomeSocket.accept();
        System.out.println ("Socket:"+connectionSocket.getPort());
       /* BufferedReader inFromClient =
          new BufferedReader(new  InputStreamReader(
                connectionSocket.getInputStream())); */

        Scanner inFromClient=new Scanner(connectionSocket.getInputStream());
        clientSentence = inFromClient.nextLine();
        System.out.println ("From Client:"+clientSentence);

        capitalizedSentence = clientSentence.toUpperCase() + '\n';

        DataOutputStream  outToClient =
          new DataOutputStream(connectionSocket.getOutputStream());

        outToClient.writeBytes(capitalizedSentence);
      }
   }
}
```

TCPClient.java:

```
package tcp;
```

```
import java.io.*;
import java.net.*;
import java.util.Scanner;
class TCPClient {
   public static void main(String argv[]) throws Exception {
      String sentence, modifiedSentence;

      /*BufferedReader inFromUser =
         new BufferedReader(new InputStreamReader(System.in)); */

      Socket clientSocket = new Socket("localhost", 6789);

      System.out.println("Enter the sentence to convert");
      Scanner inFromUser=new Scanner(System.in);
      sentence = inFromUser.nextLine();

      DataOutputStream outToServer =
         new DataOutputStream(clientSocket.getOutputStream());
      outToServer.writeBytes(sentence + '\n');

      Scanner inFromServer=new Scanner(clientSocket.getInputStream());
      /*BufferedReader inFromServer =
         new BufferedReader(new InputStreamReader(clientSocket.getInputStream())); */

      modifiedSentence = inFromServer.nextLine();
```

```
        System.out.println("FROM SERVER: " + modifiedSentence);
        clientSocket.close();
    }
}
```

Creating UDP—based Applications(UDP SOCKETS):

Create UDP Server application:

- Create DatagramSocket
- Wait until a DatagramPacket from a client is received
- On receipt a DatagramPacket read, process and send response DatagramPacket (if any) back to the client

Create UDP Client application:

- Create a DatagramSocket
- Create a DatagramPacket and sent it through the socke
- Wait until the response DatagramPacket is received from the server

DatagramSocket Class

- Constructor needs to know the port number where the datagram socket is created
  - DatagramSocket(int port)
- **send() and receive()** methods are provided to send and receive DatagramPacket respectively
  - Both these methods need a DatagramPacket as an argument

DatagramPacket Class:

- Provides many overloaded constructors; most commonly used one is:
  - DatagramPacket(byte[] data, int len, InetAddress addr, int port)

  data-contains data as an array of bytes

  len-specifies number of bytes to be sent

  addr-destination host address where the packet will be delivered

  port-specifies the port number of the destination host where the packet will be delivered

- **getAddress() and getPort()** methods are used to obtain the address and port where the packet is delivered

UDPServer.java:

```
package udp;
import java.net.*;
class UDPServer {
  public static void main(String args[]) throws Exception  {
      byte[] receive_data = new byte[1024],send_data = new byte[1024];
      int recv_port;
      int len;

      DatagramSocket server_socket = new DatagramSocket(5000);
      System.out.println("UDPServer Waiting for client on port 5000");
      while(true)
      {
       DatagramPacket receive_packet = new DatagramPacket(receive_data,receive_data.length);
       server_socket.receive(receive_packet);

       String data = new String(receive_packet.getData());
       InetAddress IPAddress = receive_packet.getAddress();
       recv_port = receive_packet.getPort();
       len=receive_packet.getLength();

      System.out.println("( " + IPAddress + " , Length of data:" +len+ " Port no:" + recv_port+") said :"+data );
```

```
        String sentence = new String(receive_packet.getData());
        String capitalizedsentence=sentence.toUpperCase();

        send_data=capitalizedsentence.toUpperCase().getBytes();
        InetAddress IPAddress1 = receive_packet.getAddress();
        recv_port = receive_packet.getPort();
        len=receive_packet.getLength();

        DatagramPacket send_packet = new
DatagramPacket(send_data,len,IPAddress1,recv_port);
        server_socket.send(send_packet);

       }
      }
     }
```

UDPClient.java:

```
package udp;
import java.net.*;
import java.io.*;
import java.util.Scanner;
class UDPClient {
  public static void main(String args[]) throws Exception
```

```
{
        byte[] receive_data = new byte[1024],send_data = new byte[1024];
     DatagramSocket client_socket = new DatagramSocket(5001);


   while(true)
   {
                System.out.println("Enter the sentence to convert");
                Scanner infromuser=new Scanner(System.in);
                String data = infromuser.nextLine();

                send_data = data.getBytes();
                InetAddress IPAddress =  InetAddress.getByName("localhost");

                DatagramPacket send_packet = new
DatagramPacket(send_data,send_data.length,IPAddress, 5000);
                client_socket.send(send_packet);



                DatagramPacket receive_packet = new
DatagramPacket(receive_data,receive_data.length);
                client_socket.receive(receive_packet);

                String data1 = new String(receive_packet.getData());
                InetAddress IPAddress1 = receive_packet.getAddress();
```

```
            int recv_port = receive_packet.getPort();

            int len=receive_packet.getLength();

            System.out.println("( " + IPAddress1 + " , Length of data:" +len+ " Port no:" +
recv_port+") said :"+data1 );

        }

  }

}
```

URL CLASS:

- URL (Uniform Resource Locator)
  - Used to locate resources on the net.
  - Every resource can be identified by an URL
- Format
  - protocol://hostname:[port]/path
  - Example
    - http://www.yahoo.com
    - http://uroy.itd.jusl.ac.in:80/web/publication.html
    - ftp://rediffmail.com
    - file://e:\ukr\java\test.java

Connecting an URL(Available Constructors)

- URL(String *urlSpecifier*)
- URL(String *protocol*, String *host*, int *port*, String *file*)
- URL(String *protocol*, String *host*, String *file*)

Instance Methods

- String getProtocol() *protocol of this URL*
- int getPort() *port number of this URL*

- String getHost() *host name of this URL*
- String getFile() *file name of this URL*

Example code:

```
package URL;
import java.net.MalformedURLException;
import java.net.URL;
public class UrlTest {
   public static void main(String args[]) throws MalformedURLException {
        URL ukr = new URL("http://localhost:8080/DB_jsp/Page1.html");
        System.out.println("Protocol: "+ukr.getProtocol());
        System.out.println("Port: "+ukr.getPort());
        System.out.println("Host: "+ukr.getHost());
        System.out.println("File: "+ukr.getFile());
        System.out.println("Ext: "+ukr.toExternalForm());
  }

}
```

Output:

```
Protocol: http
Port: 8080
Host: localhost
File: /DB_jsp/Page1.html
Ext: http://localhost:8080/DB_jsp/Page1.html
```

The openConnection() method:

- Opens a connection to this URL
- Returns URLConnection object
    - Once the connection is opened, you can read/write on this connection using getInputStream() and getOutputStream() methods on this object

Reading From URL:

- Essential Steps
    - Construct the URL Object
    - Open the connection using openConnection() method
    - Obtain the InputStream on this connection
        - If requires, create specific stream, for example DataInputStream, BufferedInputStream, BufferedReader etc.
    - Read from stream till end of the stream

Code:

```
package URL;

import java.io.InputStream;

import java.net.URL;

import java.net.URLConnection;

import java.util.Date;

public class Url1 {

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

                                int c;

      URL ukr = new URL("http://localhost:8080/DB_jsp/Page1.html");

                                URLConnection con = ukr.openConnection();

                                System.out.println("Date: "+new Date(con.getDate()));
```

```
                                System.out.println("Content-type: "+con.getContentType());
                                System.out.println("Expires: "+con.getExpiration());
                                System.out.println("Last-Modified: "+new
Date(con.getLastModified()));
                                int len = con.getContentLength();
                                if(len > 0) {
                                  InputStream in = con.getInputStream();
                                  System.out.println("=== Content ===");
                                  while(((c = in.read()) != -1) )
                                  System.out.print((char)c);
                                  in.close();
     }
                                else System.out.println("no content available");
  }
}
```

OUTPUT:

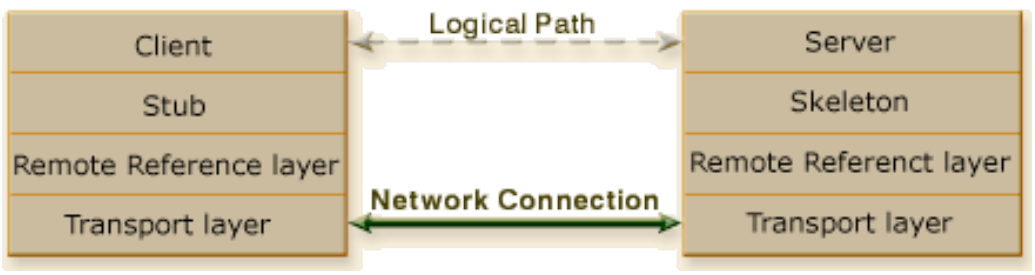
```
Date: Tue Aug 18 19:27:10 IST 2015
Content-type: text/html
Expires: 0
Last-Modified: Tue Aug 18 14:17:44 IST 2015
=== Content ===
    <!DOCTYPE html>
<html>
   <head>
      <title>Page1</title>
```

```
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <form name="f1" action="Serv12" method="post">
      Enter empid:<input type="text" name="eid"/>
      Enter the name:<input type="text" name="name"/>

      <input type="submit" value="click">
    </form>
  </body>
</html>
```

REMOTE METHOD INVOCATION(RMI):

- Java RMI is a mechanism that allows one to invoke a method on an object that exists in another address space.
- The RMI mechanism is basically an object-oriented RPC mechanism
- There are three processes that participate in supporting remote method invocation
  - The *Client* is the process that is invoking a method on a remote object.
  - The *Server* is the process that owns the remote object.
  - The *Object Registry* is a name server that relates objects with names



Remote Interface

- The interface must be public.
- The interface must extend the interface java.rmi.Remote.
- Every method in the interface must declare that it throws java.rmi.RemoteException. Other exceptions may also be thrown.

Remote Classes

- It must implement a Remote interface.
- It should extend the java.rmi.server.UnicastRemoteObject class.
- Define at least one explicit constructor
- Provide implementation for the methods that can ve invoked remotely
- It can have methods that are not in its Remote interface. These can only be invoked locally.
- Export the object so that it can accept incoming method calls.

HelloInterface.java:

```
package Rmi;

import java.rmi.*;

public interface HelloInterface extends Remote {

public String say() throws RemoteException;

}
```

Hello.java:

```
package Rmi;

import java.rmi.*;

import java.rmi.server.*;

public class Hello extends UnicastRemoteObject implements HelloInterface {
```

```
                                private String message;
                                 public Hello (String msg) throws RemoteException

    {
       message = msg;
    }

    @Override
                                public String say() throws RemoteException
    {
       return message;
    }
}
```

Server.java:

```
package Rmi;
import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.RemoteException;

public class Server {
 public static void main (String[] argv) {
                                try {
                                  Naming.rebind ("Hello", new Hello ("Hello, world!"));
                                  System.out.println ("Hello Server is ready.");
```

```
                    } catch (RemoteException | MalformedURLException e) {
System.out.println ("Hello Server failed: " + e); }
}
}
```

Client.java:

```
package Rmi;


import java.rmi.Naming;
public class Client {
public static void main (String[] argv) {
                    try {
                      HelloInterface hello = (HelloInterface) Naming.lookup ("Hello");
                      System.out.println (hello.say());
                    } catch (Exception e) { System.out.println ("HelloClient exception:
" + e); }
}


}
```

JAVABEANS CLASSES

- JSTL Core actions are designed to be used for simple, presentation-oriented programming tasks
- More sophisticated programming tasks should still be performed with a language such as Java
- JavaBeans technology allows a JSP document to call Java methods

Requirements:

JavaBeans class must

- Be public and not abstract
- Contain at least one *simple property design pattern* method .
    - Simple property design patterns
        - Two types: getter and setter
            - Both require that the method be public
            - getter:
                - no arguments
                - returns a value (i.e., cannot be void)
                - name begins with get (or is, if return type is boolean) followed by upper case letter
            - setter:
                - one argument (same type as getter return value)
                - Void (i.e., must not return a value)
                - name begins with set followed by upper case letter
- Class must have a default (no-argument) constructor to be instantiated by useBean .
- Class should belong to a package

```
public class LoginBean

{

private String name,password;


public String getName()

{

return name;

}

public void setName(String name)

{

this.name = name;
```

```
}
public String getPassword()
{
return password;
}
public void setPassword(String password)
{
this.password = password;
}
public boolean validate()
{
if(password.equals("admin"))
{
return true;
}
else
{
return false;
}
}
}
```

For JSP:

```
<jsp:useBean id="bean12" class="com.LoginBean"/>
    <c:set target="${bean12}" property="name" value="Howdy" />
    Welcome: <c:out value="${bean12.name}"/>
```

For JAVA:

```
LoginBean bean=new LoginBean();
bean.setName ("Howdy");
HttpSession session=request.getSession();
session.setAttribute("bean",bean);

<%
LoginBean bean=(LoginBean)session.getAttribute("bean");
out.print("Welcome, "+bean.getName());
%>
```

(Or)

```
Welcome: <c:out value="${sessionScope.bean.name}"/>
```

Output:

Howdy