# UNIT-3

# JDBC

JDBC stands for Java Database Connectivity.

JDBC is a Java API to connect and execute the query with the database.

It is a part of JavaSE (Java Standard Edition).

JDBC API uses JDBC drivers to connect with the database.

## JDBC Drivers

### 1) JDBC-ODBC bridge driver:

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.

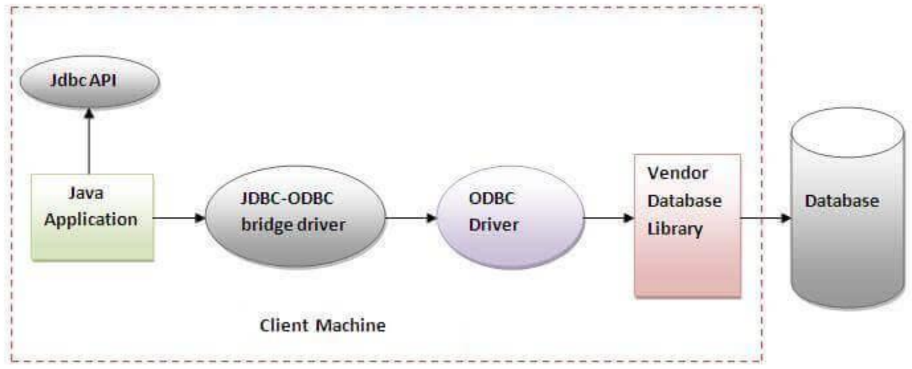


Figure- JDBC-ODBC Bridge Driver

In Java 8, the JDBC-ODBC Bridge has been removed.

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of 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.

**2) Native-API driver:** The Native API driver uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API.
It is not written entirely in java.

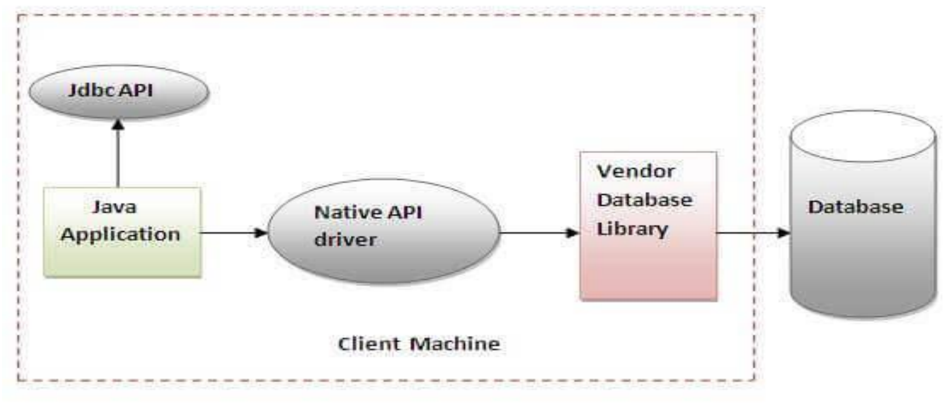


Figure- Native API Driver

Advantage: performance upgraded than JDBC-ODBC bridge driver.

Disadvantage: 1)The Native driver needs to be installed on the each client machine.

2)The Vendor client library needs to be installed on client machine.

## 3) Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

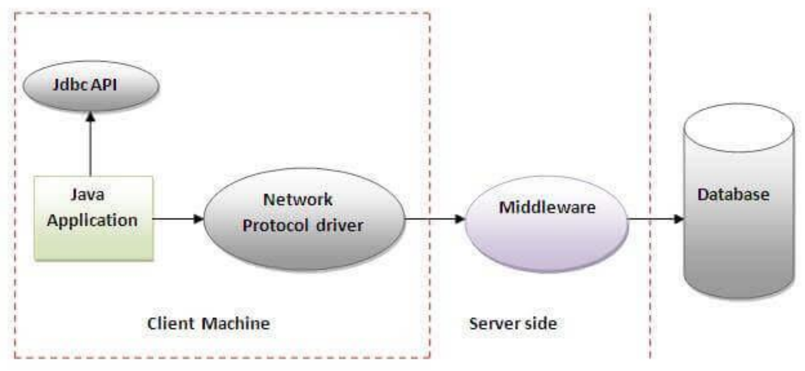


Figure- Network Protocol Driver

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

Disadvantages:

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

## 4) Thin driver

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

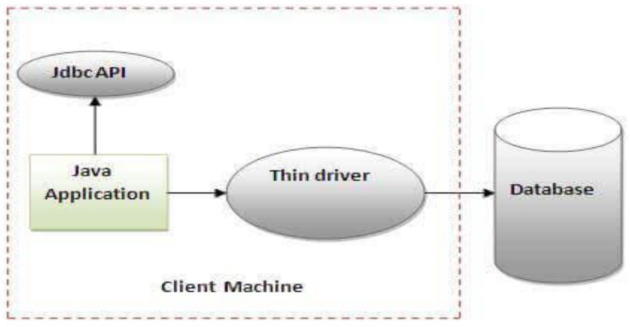


Figure- Thin Driver

Advantage: 1) Better performance than all other drivers.

2) No software is required at client side or server side.

Disadvantage: Drivers depend on the Database.

## Connection Interface

A Connection is the session between java application and database.

The Connection interface is a factory of Statement, PreparedStatement, and DatabaseMetaData i.e. object of Connection can be used to get the object of Statement and DatabaseMetaData.

The Connection interface provide many methods for transaction management like commit(), rollback() etc.

### Commonly used methods of Connection interface:

1) public Statement createStatement(): creates a statement object that can be used to execute SQL queries.

2) public void setAutoCommit(boolean status): is used to set the commit status. By default it is true.

3) public void commit(): saves the changes made since the previous commit/rollback permanent.

4) public void rollback(): Drops all changes made since the previous commit/rollback.

5) public void close(): closes the connection and Releases a JDBC resources immediately.

## Connecting to MYSQL Database:

Connection con;

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

DriverManager.registerDriver( new com.mysql.jdbc.Driver());

con = DriverManager.getConnection("jdbc:mysql://localhost:3306/kmit","root","");on con;

## Connecting to Oracle Database:

Connection con;

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

con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

## **Statement** interface

The Statement interface provides methods to execute queries with the database.

The statement interface is a factory of ResultSet i.e. it provides factory method to get the object of ResultSet.

## Commonly used methods of Statement interface:

1) public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sql): is used to execute queries that may return multiple results.

## Types of Statement interfaces:

1. **Statement**.
2. **PreparedStatement**: Helps to pass runtime parameters to the SQL Query.

3. **CallableStatement**: Helps us to invoke stored procedures/functions from java application.

## How to get Statement interface object:

Statement st= con.prepareStatement();

## How to get PrepareStatement interface object:

PreparedStatement pst;

pst = con.prepareStatement("SQL Query with Runtime parameters");

**Example:**

**Q)**Write an sql query that returns the returns all the records from employee table whose employee ID is greater than the runtime values supplied.

**A)** PreparedStatement pst = con.prepareStatement("select * from emp where empid>?");

pst.setInt(1,id); // where id is value of employee id given by user at runtime

ResultSet rs=pst.executeQuery(); // now rs conatins all the required records.

# Database Queries

## Inserting data into a Database Table:

int no_of_rows_effected =

Statement_reference.executeUpdate("

insert into student(rollno,name) values('17BD1A1235','SHIVA')");

## Updating data of the Database Table:

int no_of_rows_effected =

Statement_reference.executeUpdate("

update emp set salary=salary+5000 where empid='263'");

## Deleting data from the Database Table

int no_of_rows_effected =

Statement_reference.executeUpdate("

delete from emp where empid>'263'");

## Retrieving records from the Database Table:

```
ResultSet   rs = st.executeQuery("select * from student");
while(rs.next())
{
   System.out.print(rs.getInt("rno")+"    ");
   System.out.print(rs.getString(“name”)+"  ");
   System.out.println(rs.getString(“grade”)+"  ");
}
```

# NETWORKING

## InetAddress class

**Java InetAddress** class represents an numerical IP address & domain name of the host.

The java.net.InetAddress class provides methods to get the IP of any host name *for example* www.javatpoint.com, www.google.com, www.facebook.com, etc.

An IP address is represented by 32-bit(IP IV) or 128-bit(IP V6) unsigned number.

An instance of InetAddress represents the IP address with its corresponding host name.

There are two types of address types: Unicast and Multicast.

The Unicast is an identifier for a single interface whereas Multicast is an identifier for a set of interfaces.

### Commonly used methods of InetAddress class

1.InetAddress   getByName(String host) **:**

returns the instance of InetAddress containing LocalHost IP and name.

2. InetAddress  getLocalHost()

it returns the instance of InetAdddress containing local host name and address.

3. String getHostName()

it returns the host name of the IP address.

4. String getHostAddress()

it returns the IP address in string format.

### Example:

```
import java.io.*;  import java.net.*;
public class InetDemo {
public static void main(String[] args) { try {
InetAddress ip=InetAddress.getByName("www.google.com");
 System.out.println("Host Name: "+ip.getHostName());
System.out.println("IP Address: "+ip.getHostAddress());
}catch(Exception e){System.out.println(e);}  } }
```

**Output:**

```
Host Name: www.google.com

IP Address: 206.51.231.148
```

## URL class

The URL class is the gateway to any of the resource available on the internet.

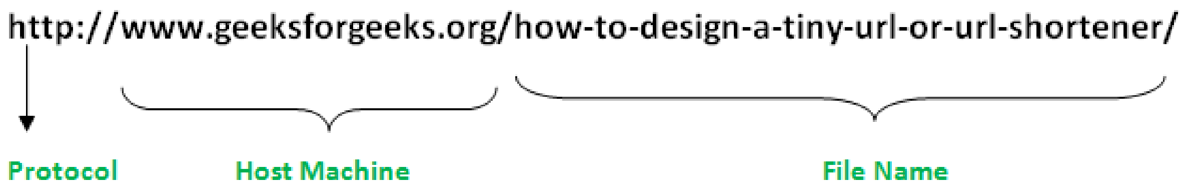
A Class URL represents a Uniform Resource Locator, which is a pointer to a "resource" on the World Wide Web.

A resource can point to a simple file or directory, or it can refer to a more complicated object, such as a query to a database or to a search engine

**What is a URL?**

As many of you must be knowing that Uniform Resource Locator-URL is a string of text that identifies all the resources on Internet, telling us the address of the resource, how to communicate with it and retrieve something from it.

A Simple URL looks like:


http://www.geeksforgeeks.org/how-to-design-a-tiny-url-or-url-shortener/
Protocol
Host Machine
File Name

**Components of a URL**:

A URL can have many forms. The most general however follows three-components system-

**Protocol**: HTTP is the protocol here

**Hostname**: Name of the machine on which the resource lives.

**File Name**: The path name to the file on the machine.

**Port Number**: Port number to which to connect (typically optional).

## TCP Sockets

**Network programming** refers to writing programs that execute across multiple devices (computers), in which the devices are all connected to each other using a network.

**TCP** − TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

The java.net package of the J2SE APIs contains a collection of classes and interfaces that provide the low-level communication details, allowing you to write programs that focus on solving the problem at hand.

### Socket Programming

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and the server can now communicate by writing to and reading from the socket.

The **java.net.Socket** class represents a socket, and the **java.net.ServerSocket** class provides a mechanism for the server program to listen for clients and establish connections with them.

**Establishing a TCP connection**:

1. The server instantiates a **ServerSocket object**, denoting which port number communication is to occur on.

2. The server invokes the **accept()** method of the ServerSocket class. This method waits until a client connects to the server on the given port.

3. After the server is waiting, a client instantiates a Socket object, specifying the server name and the port number to connect to.

4. The constructor of the Socket class attempts to connect the client to the specified server and the port number. If communication is established, the client now has a Socket object capable of communicating with the server.

5. On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a two-way communication protocol, hence data can be sent across both streams at the same time. Following are the useful classes providing complete set of methods to implement sockets.

## TCP Program:

// The server provides a random integer to the request client.

### Server.java

```
import java.net.*; import java.util.*; import java.io.*;

class server

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

      ServerSocket  ss=new  ServerSocket(5555);

      Random  r=new  Random();

      while(true)       {

          Socket   s = ss.accept();

          OutputStream os=s.getOutputStream();

          DataOutputStream dos=new DataOutputStream(os);

          dos.writeInt(r.nextInt());      s.close();

       }  }  }
```

### Client.java

```
import java.net.*; import java.util.*; import java.io.*;

class Client{

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

     Socket  s= new  Socket("localhost",5555);

     InputStream is=s.getInputStream();

     DataInputStream dis=new DataInputStream(is);

     int  i=dis.readInt();

     System.out.println("random value received is="+i);

     s.close();  }}
```

# UDP Sockets

**UDP** − UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications.

DatagramPacket and DatagramSocket are the two main classes that are used to implement a UDP client/server application.

**DatagramPacket** is a data container.

**DatagramSocket** is a mechanism to send and receive DatagramPackets.

## 1. DatagramPacket

In UDP's terms, data transferred is encapsulated in a unit called datagram.

A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed.

In Java, DatagramPacket represents a datagram.

You can create a DatagramPacket object by using one of the following constructors:

```
DatagramPacket(byte[] buf, int length)

DatagramPacket(byte[] buf, int length, InetAddress address, int port)
```

As you can see, the data must be in the form of an array of bytes. The first constructor is used to create a DatagramPacket to be received.

The second constructor creates a DatagramPacket to be sent, so you need to specify the address and port number of the destination host.

The parameter length specifies the amount of data in the byte array to be used, usually is the length of the array (buf.length).

## 2. DatagramSocket

You use DatagramSocket to send and receive DatagramPackets.

DatagramSocket represents a UDP connection between two computers in a network.

In Java, we use DatagramSocket for both client and server. There are no separate classes for client and server like TCP sockets.

So you create a DatagramSocket object to establish a UDP connection for sending and receiving datagram, by using one of the following constructors:

```
DatagramSocket()

DatagramSocket(int port)
```

The no-arg constructor is used to create a client that binds to an arbitrary port number. The second constructor is used to create a server that binds to the specific port number, so the clients know how to connect to.

## Methods of the DatagramSocket:

send(DatagramPacket p): sends a datagram packet.

receive(DatagramPacket p): receives a datagram packet.

close(): closes the socket.

## UDP Program:

// The Receiver receives the incoming data grams and display the content on monitor.

### Receiver.java

```
import java.net.*; import java.util.*; import java.io.*;
class  Receiver{
  public static void main(String args[])  throws Exception  {
     DatagramSocket  ds=new DatagramSocket(4444);
     byte  buf[]=new byte[60];
    while(true)     {
        DatagramPacket dp=new DatagramPacket(buf,buf.length);
        ds.receive(dp);
        String s=new String(dp.getData());
        System.out.println(s);    }  }}
```

### Sender.java

```
import java.net.*; import java.util.*; import java.io.*;
class sender{
  public static void main(String args[])  throws Exception  {
      DatagramSocket ds=new DatagramSocket();
      InetAddress i=InetAddress.getByName("localhost");
      byte buf[]=args[0].getBytes();
      DatagramPacket dp=new DatagramPacket(buf,buf.length,i,4444);
ds.send(dp);  }}
```

# JAVA BEANS

A JavaBean is a specially constructed Java class written in the Java and coded according to the JavaBeans API specifications.

## Characteristics

It may have a number of properties (fields) which can be read or written.

For each property we must have corresponding accessor methods ie getXXX() and setXXX(), where XXX is the name of the property.

In the accessor methods, the first letter of the field must be capital as per the java naming conventions.

Apart from accessor methods the java bean class can also have other business methods(methods for application specific functionality).

It should be serializable, ie the bean class has to implement the Serializable interface.

It provides a default, no-argument constructor.

## JavaBeans Properties

A JavaBean property is a named attribute that can be accessed by the user of the object. The attribute can be of any Java data type, including the classes that you define.

A JavaBean property may be read, write, read only, or write only. JavaBean properties are accessed through two methods in the JavaBean's implementation class −

1. getPropertyName()

For example, if property name is firstName, your method name would be getFirstName() to read that property. This method is called accessor.

2. setPropertyName()

For example, if property name is firstName, your method name would be setFirstName() to write that property. This method is called mutator.

A read-only attribute will have only a getPropertyName() method, and a write-only attribute will have only a setPropertyName() method.

## JavaBeans Example

Consider a student class with few properties −

```
package edu.kmit;

public class StudentsBean implements java.io.Serializable {
   private String firstName = null;
   private String lastName = null;
   private int age = 0;
   public StudentsBean() {    }
   public String getFirstName(){
      return firstName;
   }
   public String getLastName(){
      return lastName;
   }
   public int getAge(){
      return age;
   }
   public void setFirstName(String firstName){
      this.firstName = firstName;
   }
   public void setLastName(String lastName){
      this.lastName = lastName;
   }
   public void setAge(Integer age){
      this.age = age;
   }
}
```

## RMI

RMI stands for Remote Method Invocation.

It is a mechanism that allows an object residing in one system (JVM) to access/invoke an object running on another JVM.

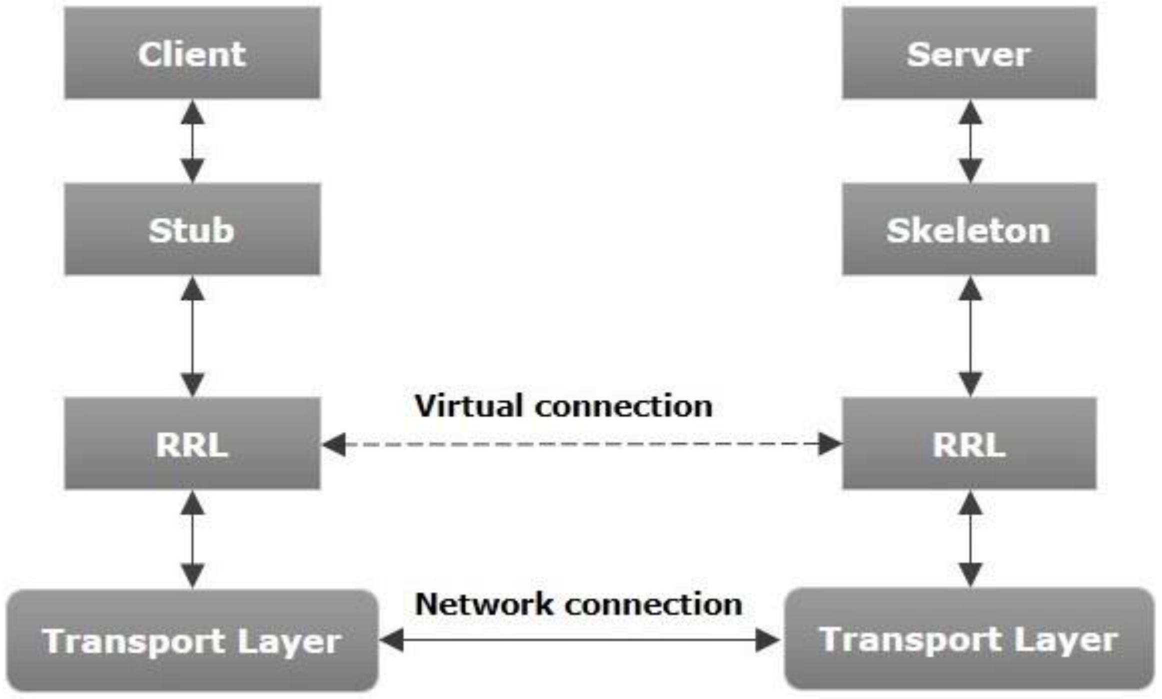
RMI is used to build distributed applications. It provides remote communication between Java programs. It is provided in the package java.rmi.

## Architecture of an RMI Application

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.

The following diagram shows the architecture of an RMI application.



Let us now discuss the components of this architecture.

- **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.

## Working of an RMI Application

The following points summarize how an RMI application works −

- 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.

## Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as marshalling.

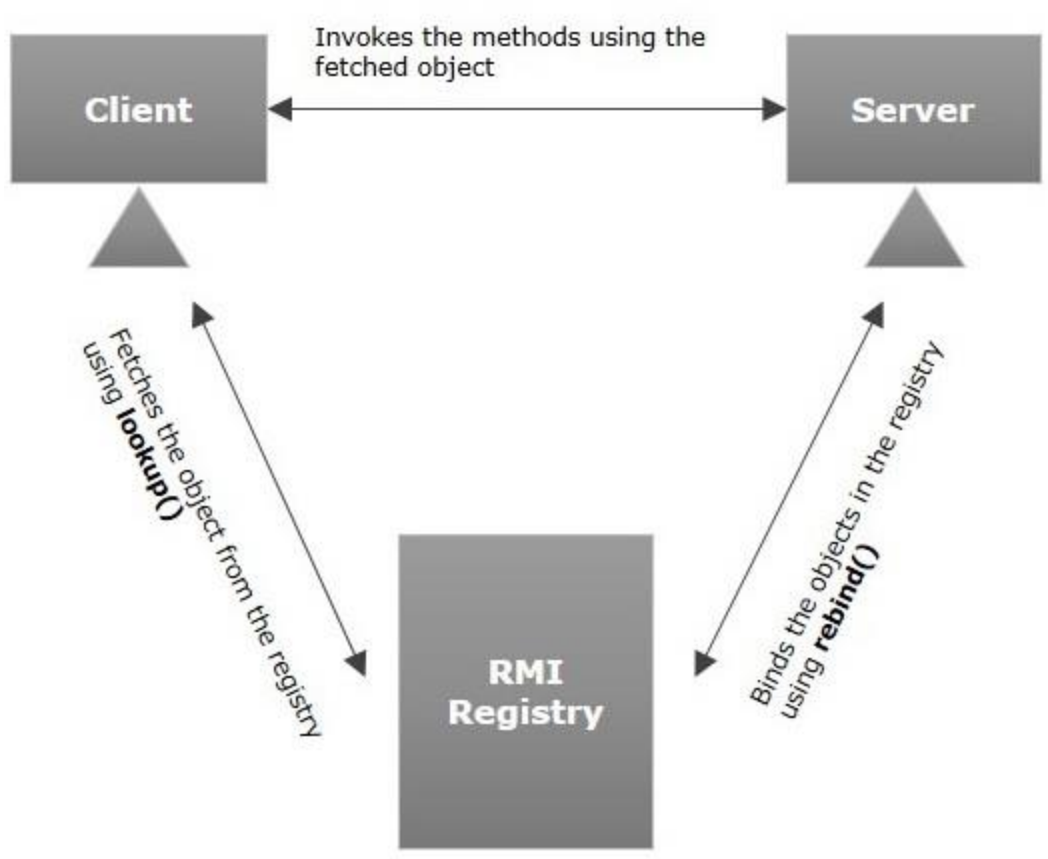
At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as unmarshalling.

## RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using bind() or reBind() methods). These are registered using a unique name known as bind name.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using lookup() method).

The following illustration explains the entire process −


Client
Invokes the methods using the fetched object
Server
Fetches the object from the registry using lookup()
Binds the objects in the registry using rebind()
RMI Registry

## Goals of RMI

Following are the goals of RMI −

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

## RMI Example Program

### AddServerIntf.java

```
import java.rmi.*;

public   interface    AddServerIntf    extends    Remote

{

   public   double   add(double  x,double   y) throws   RemoteException;

}
```

## AddServerImpl.java

```
// The UnicastRemoteObject class defines a non-replicated remote object whose references are
valid only while the server process is alive.

import  java.rmi.*;

import java.rmi.server.*;

public   class   AddServerImpl   extends   UnicastRemoteObject   implements   AddServerIntf

{     public   AddServerImpl()   throws   RemoteException  {   }

  public   double   add(double   x,double   y)  throws   RemoteException   {

      return (x+y);  }

}
```

## AddServer.java

```
// Registering(Binding) the service in RMI Registry

import java.rmi.*;

import java.net.*;

public   class  AddServer

{   public  static  void  main(String  args[])

    {   try {

        AddServerImpl   obj=new   AddServerImpl();

        Naming.rebind("rmi://localhost/MyAddServer",obj);

        }  catch(Exception e)  {

            System.out.println(e);

        }

    }

}
```

## Client.java

```
import   java.rmi.*;

public  class  Client
```

```
{  public   static   void   main(String   args[])  {
       try {   String   url="rmi://"+args[0]+"/MyAddServer";
              AddServerIntf    ob=(AddServerIntf)Naming.lookup(url);
            double    n1=Double.parseDouble(args[1]);
           double    n2=Double.valueOf(args[2]).doubleValue();
          System.out.println(ob.add(n1,n2));
      }  catch(Exception    e) {
           System.out.println(e);  }
    }  }
```