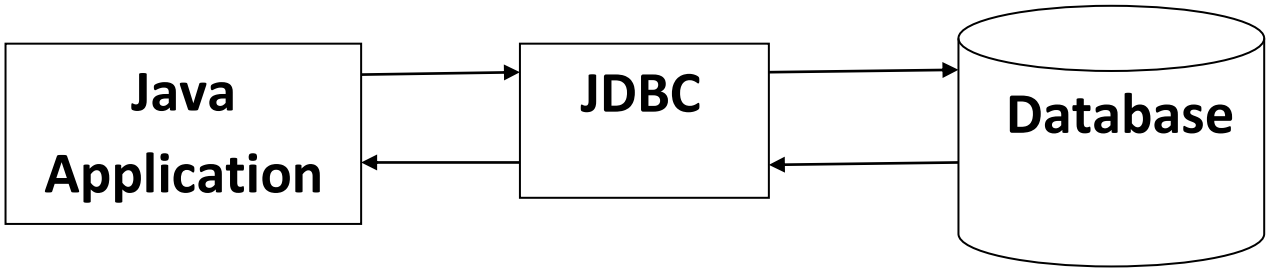
Unit-III

JDBC:

JDBC Overview – JDBC implementation – Connection class – Statements - Catching Database Results, handling database Queries. Networking– Inet Address class – URL class- TCP sockets – UDP sockets, Java Beans –RMI.

Introduction to JDBC

Java Database Connectivity (**JDBC**) is an application programming interface (API) for the programming language Java, which defines how a client may access a database. It is a Java-based data access technology used for Java database connectivity. It is part of the Java Standard Edition platform, from Oracle Corporation.



JDBC features

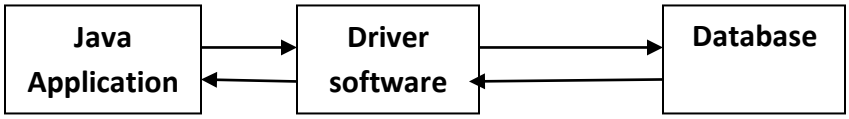
1. JDBC is a standard API. We can communicate with any database without rewriting our application i.e. it is Database independent API.
2. Most of JDBC drivers are developed in java and JDBC concepts can work for any platform. i.e. It is platform independent technology.
3. By using JDBC API. We can perform basic CRUD (Create, Read, Update, and Delete) operation very easily. we can also perform complex operations like(join, stored procedures) very easily.
4. Huge vendor support products based on JDBC API.

JDBC has four types of Drivers. They are

1. Jdbc-Odbc Bridge Driver. (Type-I driver);
2. Java-native ApI driver (Type-2 driver)
3. Java- net Driver (Type-3 driver)
4. Java native protocol driver (Type -4 driver)

JDBC Drivers

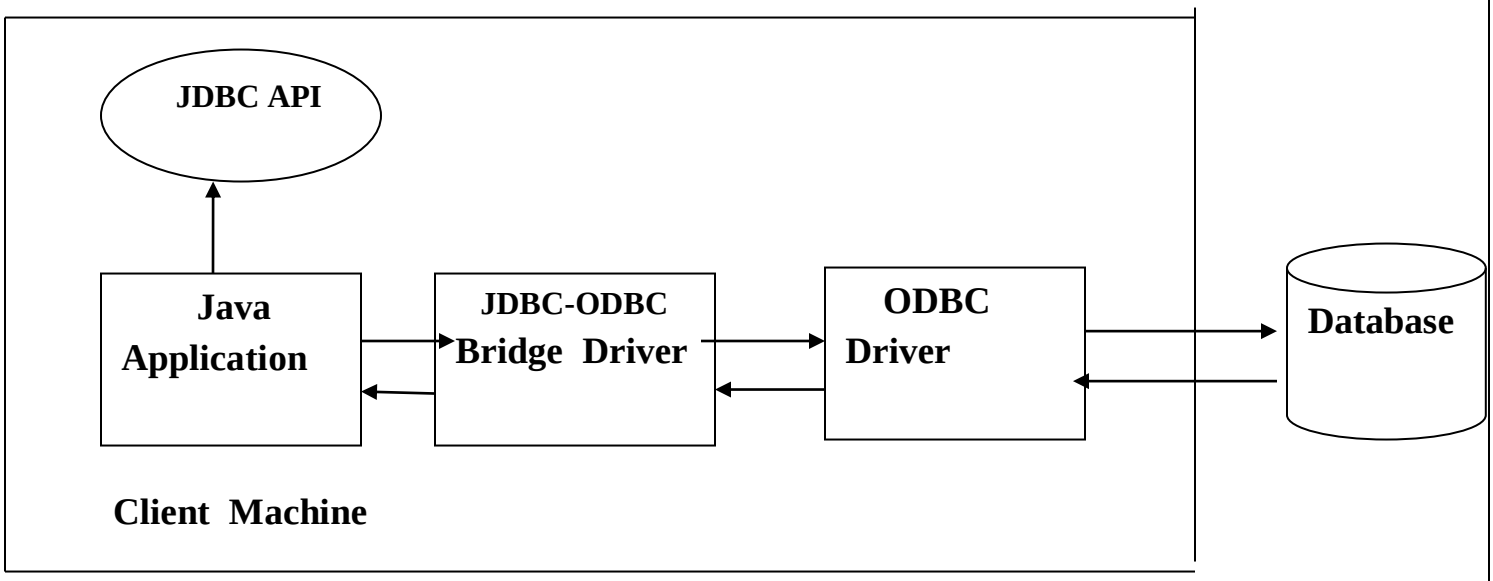
Need of Driver to convert java specific calls to Database specific call and vice versa we need Driver software.



JDBC drivers are classified into 4 types based on their functionality and architecture. Each of the driver are explained below.

Type-I Driver

1. This driver is also known as **JDBC-ODBC Bridge** (or) Bridge Driver.
2. This driver is provided by sun micro systems as part of JDK (**J**ava **D**evelopment **K**it).
3. Internally this driver will take support of ODBC Driver to communicate with DB.
4. Type-1 driver converts JDBC call to ODBC calls and ODBC driver converts ODBC calls into database specific calls.
5. Hence Type-1 driver acts as bridge between JDBC and ODBC.



Architecture of type-I driver

Advantages

1. It is Easy to use and maintain
2. We are not required to install because it is available as the part of JDK.
3. Type-1 driver won't communicate with DB directly hence it is DB independent driver because of this migrating from database to another database will become easy.

Disadvantages:

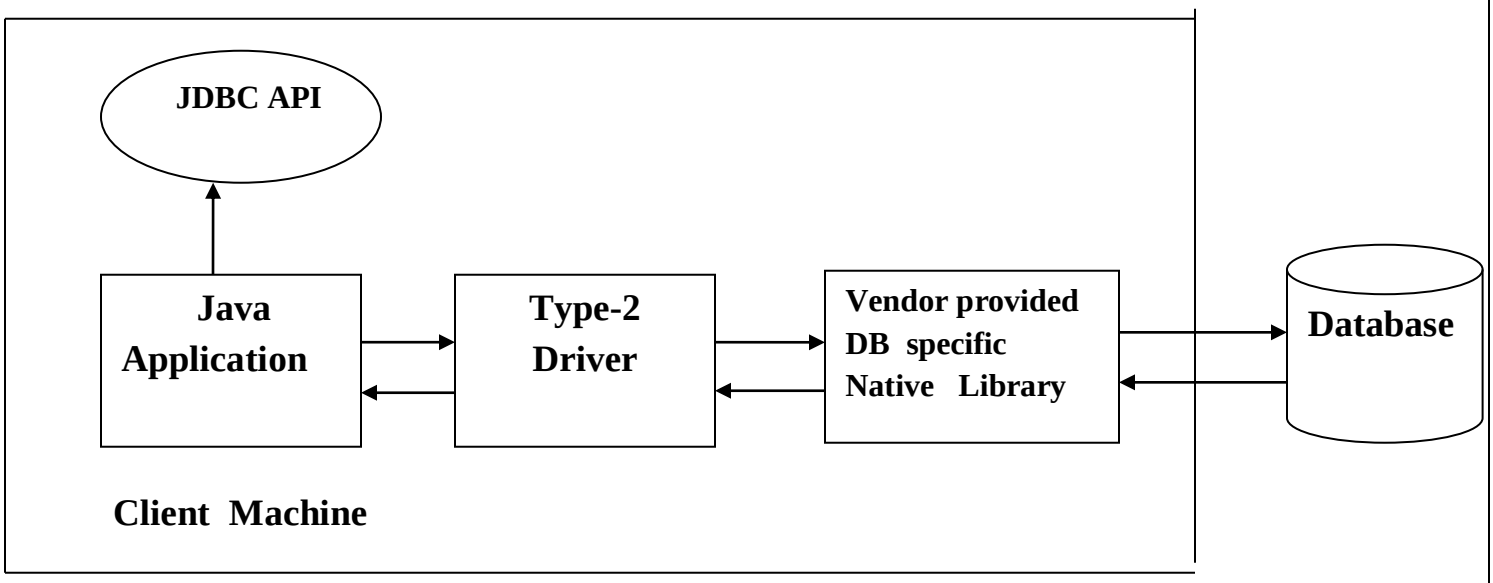
1. This driver internally depends on ODBC drivers which will work only for windows machines. Hence Type-1 driver is platform dependent.
2. Performance degraded because JDBC method call is converted into the ODBC function calls and ODBC calls into DB specific calls.
3. Restricted to connect to only local or local Network DB
4. Type-I driver support is available upto JDK1.7 only.

Other Details of Type-1 Driver

Driver Name	JDBC ODBC driver
Vendor	Sun & Intersolv
Software	For ODBC: Microsft front/back office product supplies ODBC drivers for few DBs like ACCESS, Oracle, SQL Server, Excel, Visual Faxpro etc and for DBs like MySQL, Pointbase, Interbase, Sybase, Ingres and DB2 the ODBC driver software will be supplied by the DB vendors only For JDBC: JDK 1.1 & above (JDK 1.1.1 - 1.1.8, J2SDK 1.2.0 - J2SDK 1.5.0)
Env Settings	
PATH	C:\WINNT;C:\WINNT\System32;D:\J2SDK1.4.2_03\bin (Note: WINNT\System32 directory containsODBC32.dll file)
CLASSPATH	d:\j2sdk1.4.2_04\jre\lib\rt.jar
Driver	sun.jdbc.odbc:JdbcOdbcDriver
URL	jdbc:odbc:mysqlDSN / jdbc:odbc:oracleDSN

Type-2 Driver

1. It is also known as **Native API partly java Driver.**
2. Type-2 driver is exactly same as Type-I driver except that ODBC driver is replaced with database vendor specific native libraries.
3. Native libraries means that set of functions written in non java (mostly C/C++).
4. We have to install vendor provided native libraries on the client machines.
5. Type-2 driver converts JDBC calls into vendor specific native calls which can be understandable directly database engine.



Architecture of type-II driver

Advantages:

1. When compared with type-I driver performed is high because it required only one level conversion from JDBC to native library calls
2. No need of arranging ODBC driver.

Disadvantage:

1. It is database dependent driver because it internally uses database specific native libraries hence migrating from one database to another database will become difficulty.
2. It is platform dependent driver.
3. We required to install native libraries on the client machine there is no guarantee every database vendor will provide these driver.

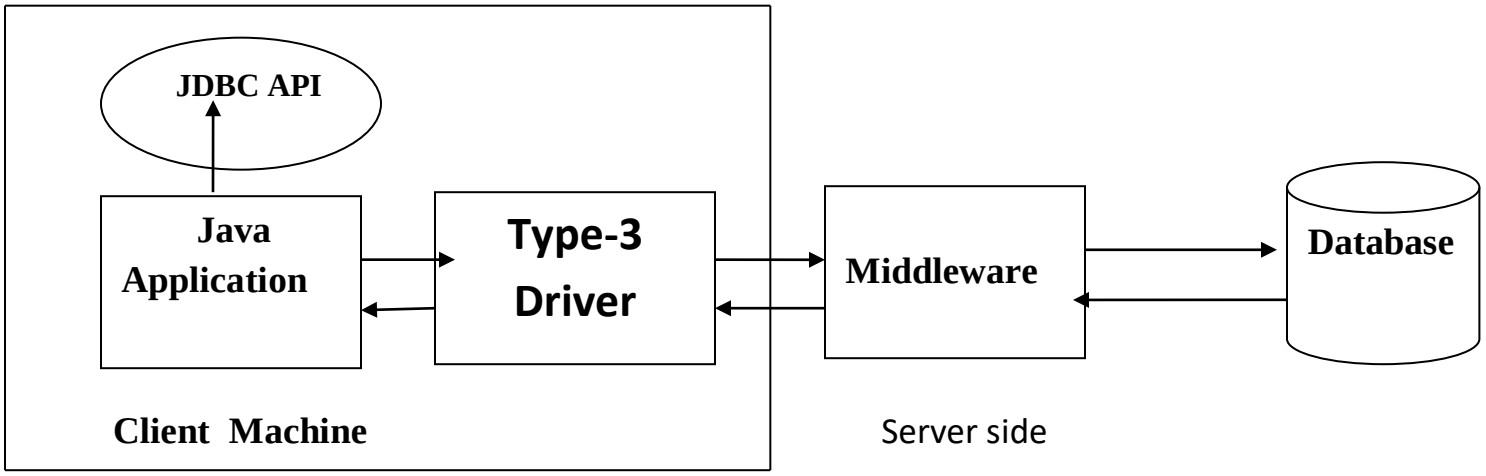
Other Details of Type-2 Driver

Driver Name	**Native API partly Java Driver (or) Part java, Part Native Driver**
Vendor	Database (Ex: Oracle)
Software	Database (Ex: Oracle) Oracle Server installation on Server systemOracle client installation on Client system
Env Settings	
PATH	d:\Oracle\Ora81\bin

CLASSPATH	d:\Oracle\Ora81\jdbc\lib\classes111.zip
Driver	oracle.jdbc.driver.OracleDriver
URL	jdbc:oracle:oci8:@tnsname

Type-3 driver

The name of this driver is **Middleware driver (or) Java Net Driver (or) Network Protocol Driver.** In this case, the JDBC driver forwards the JDBC calls to some middleware server using database independent network protocol. This intermediate server sends each client request to a specific database. The results are then sent back to the intermediate server, which in turn sends the result back to the client. This approach hides the details of connections to the database servers and makes it possible to change the database servers without effecting the client.



Architecture of type-III driver

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.

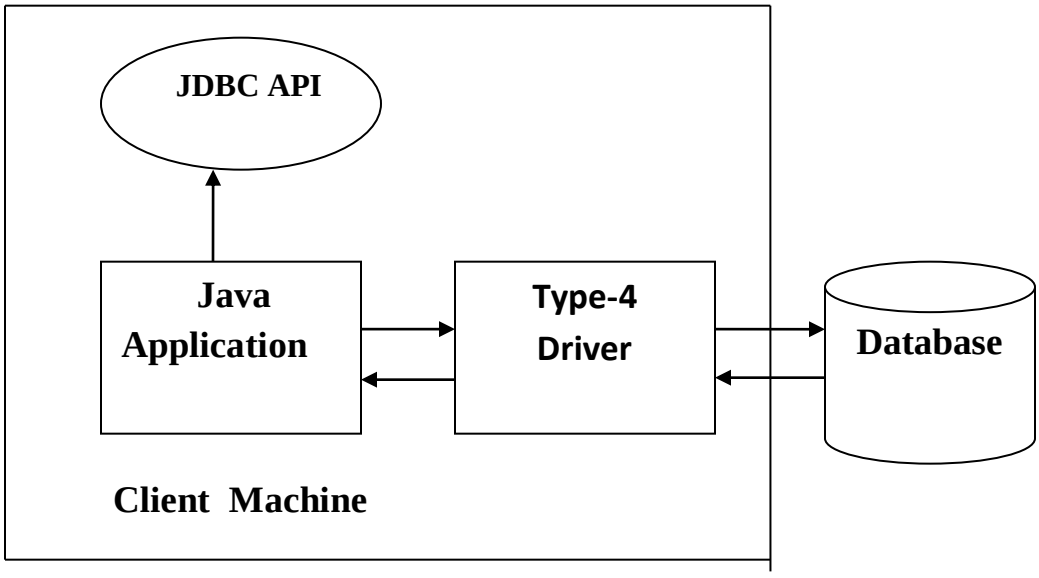
Other Details of Type-3 Driver

Driver Name	Java Net Driver

Vendor	Third party vendor like Intersolv (www.intersolv.com)
Software	IDS Server installation on DB server system IDS java libraries must be copied on to client system(jdk14drv.jar)
Env Settings	
PATH	
CLASSPATH	D:\IDSServer\classes\jdk14drv.jar
Driver	ids.sql.IDSDriver
URL	jdbc:ids://abc:12/conn?dsn='SysOracleDSN' (or) jdbc:ids://abc:12/conn?dsn='SysMySQLDSN'

Type-4 Driver

1. It is also known as driver is **Pure java Driver (or) Thin Driver (or) Java Native protocol driver.**
2. This driver uses Database specific native protocols to communicate with database.
3. This driver converts JDBC calls into Database specific calls directly.
4. This Driver won't require any ODBC Driver or native libraries at client side and hence it is called thin driver.



Advantage:

1. Better performance than all other drivers(only one conversion)
2. Platform independent driver

3. No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

Other Details of Type-4 Driver

Driver Name	Java Native protocol driver (or) Pure Java Driver
Vendor	DB (Oracle/MySQL)
Software	DB (Oracle/MySQL) Server software installation on server No client software installation is required on client system except required java libraries (classes111.zip)
Env Settings	
PATH	
CLASSPATH	d:\Oracle\Ora81\jdbc\lib\classes111.zip
Driver	oracle.jdbc.driver.OracleDriver
URL	jdbc:oracle:thin:@host:1521:SID

JDBC API

The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:

- Driver interface
- Connection interface
- Statement interface
- PreparedStatement interface
- ResultSet interface
- ResultSetMetaData interface
- DatabaseMetaData interface
- RowSet interface

A list of popular *classes* of JDBC API are given below:

- DriverManager class
- Blob class

- Clob class
- Types class

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 Statement createStatement(int resultSetType,int resultSetConcurrency): Creates a Statement object that will generate ResultSet objects with the given type and concurrency.

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

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

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

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

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:

The important methods of Statement interface are as follows:

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.

4) public int[] executeBatch(): is used to execute batch of commands.

PreparedStatement interface

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

Why use PreparedStatement?

Improves performance: The performance of the application will be faster if you use PreparedStatement interface because query is compiled only once.

The important methods of PreparedStatement interface are given below:

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int	sets the String value to the given parameter

paramIndex, String value)	index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

Commonly used methods of ResultSet interface

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in

	the ResultSet object.
6) public boolean relative(int row):	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Java DatabaseMetaData interface

DatabaseMetaData interface provides methods to get meta data of a database such as database product name, database product version, driver name, name of total number of tables, name of total number of views etc.

Commonly used methods of DatabaseMetaData interface

- **public String getDriverName()throws SQLException:**
  it returns the name of the JDBC driver.

- **public String getDriverVersion()throws SQLException:**
  it returns the version number of the JDBC driver.

- **public String getUserName()throws SQLException:**
  it returns the username of the database.

- **public String getDatabaseProductName()throws SQLException:**
  it returns the product name of the database.

- **public String getDatabaseProductVersion()throws SQLException:**
  it returns the product version of the database.

- **public ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String[] types)throws SQLException:**

it returns the description of the tables of the specified catalog. The table type can be TABLE, VIEW, ALIAS, SYSTEM TABLE, SYNONYM etc.

DriverManager class

The DriverManager class acts as an interface between user and drivers. It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver. The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method DriverManager.registerDriver().Useful methods of DriverManager class

Method	Description
1) public static void registerDriver(Driver driver):	is used to register the given driver with DriverManager.
2) public static void deregisterDriver(Driver driver):	is used to deregister the given driver (drop the driver from the list) with DriverManager.
3) public static Connection getConnection(String url):	is used to establish the connection with the specified url.

4) public static Connection getConnection(String url,String userName,String password):	is used to establish the connection with the specified url, username and password.

Steps required to write a JDBC application

There are seven basic steps while designing a JDBC application. They are

1. Import the java.sql.*;
2. Load and register the driver.
3. Establish a connection to the database server.
4. Create a statement.
5. Execute the statement.
6. Retrieve and process the results.
7. Close the statement and connection.

Step 1: Import java.sql.* package;

The JDBC API is set of classes and interfaces. The package name for these classes and interfaces is ***java.sql*** and is imported in the first line of JDBC program.

Step 2 : Load and Register the Driver

The driver can be loaded in different ways. But in general we use the method ***Class.forName()*** method to load the database driver.

For Example The following statement is used to load the JDBC bridge driver

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

The **Class.forName()** method takes the complete package name of the driver as its argument(dot separates the sub package).

when the driver is loaded , it automatically registers with DriverManager class. The driver itself calls the **DriverManager.registerDriver()** method at load time to ensure that the driver is properly registered.

Step 3 : Establishing the Connection

Once driver is loaded, the standard method of establishing a connection to the database is to call one of the overloaded static method getConnection() of DriverManager class whose syntax's are

1) **public static** Connection getConnection(String url)**throws** SQLException
2) **public static** Connection getConnection(String url,String name,String password) **throws** SQLException

3) public static Connection getConnection(String url, properties);

Step 4: Create a Statement

Once a connection to the database is established, we can interact with the database. The connection interface provides methods for obtaining different statement objects that are used fire SQL statements via the established connection. The connection object can be used for other purposes such as gathering database information, and committing or rollback a transaction. some of the methods of Connection interface are

1) **public Statement createStatement():** creates a statement object that can be used to execute SQL queries.
2) **public PreparedStatement prepareStatement():** creates a PreparedStatement object to send parameterized SQL statements to the database.
3) **Public CallableStatement prepareCall():** creates a CallableStatement object for calling database stored procedures

These methods are called ***factory methods.***

Step-5 : Execute the Statement

To execute a query we can use executeQuery(), executeUpdate(), execute() methods of statement interface whose general form are

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.

Example ResultSet rs=s.executeQuery("select * from emp");

Step 6: Retreive and process the results

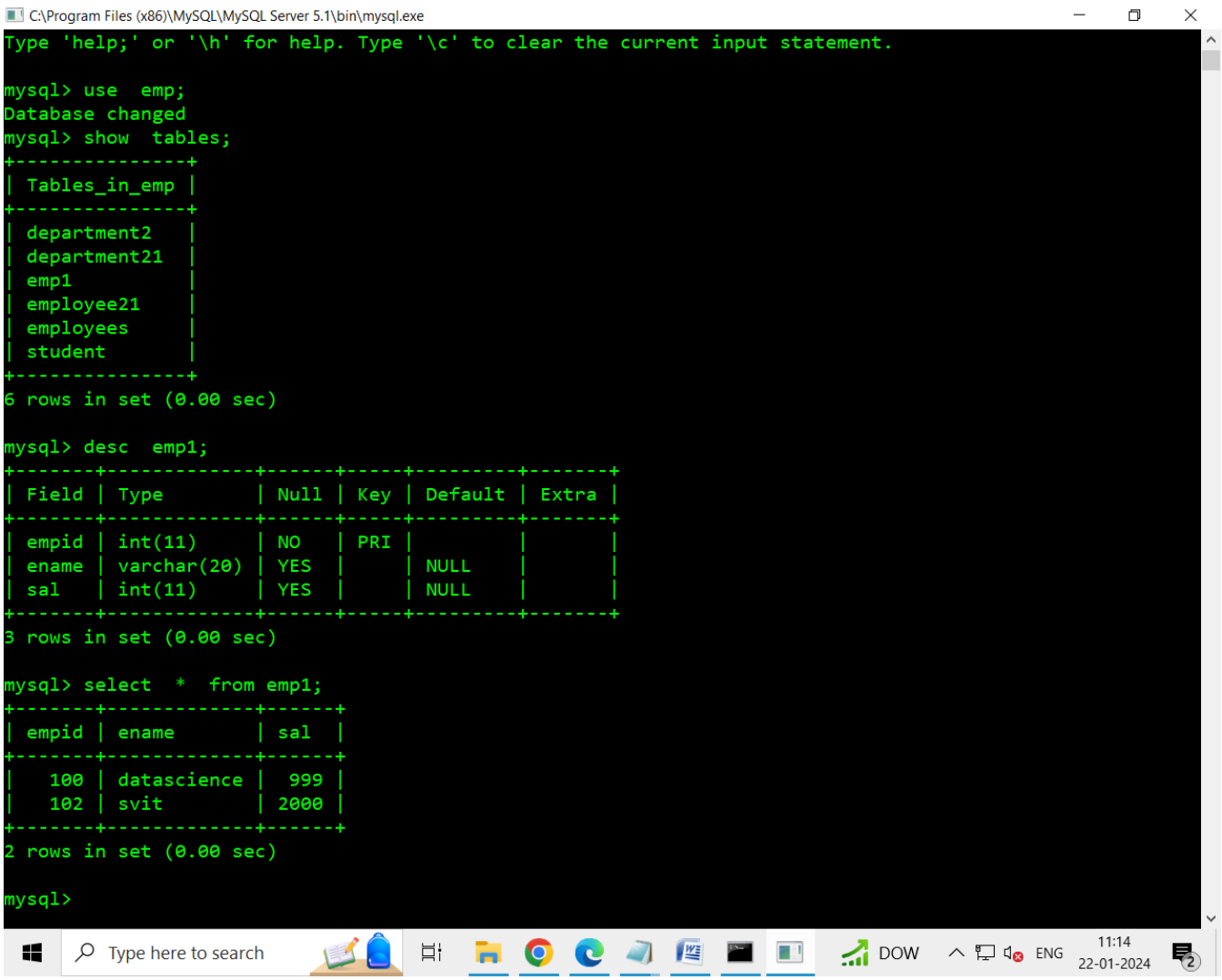
After the execution of the SQL statement, the next task is to retrieve the results. Results are stored in a ResultSet object. To retrieve the values from ResultSet object into our java program, we use **getxxx()** methods of ResultSet class.

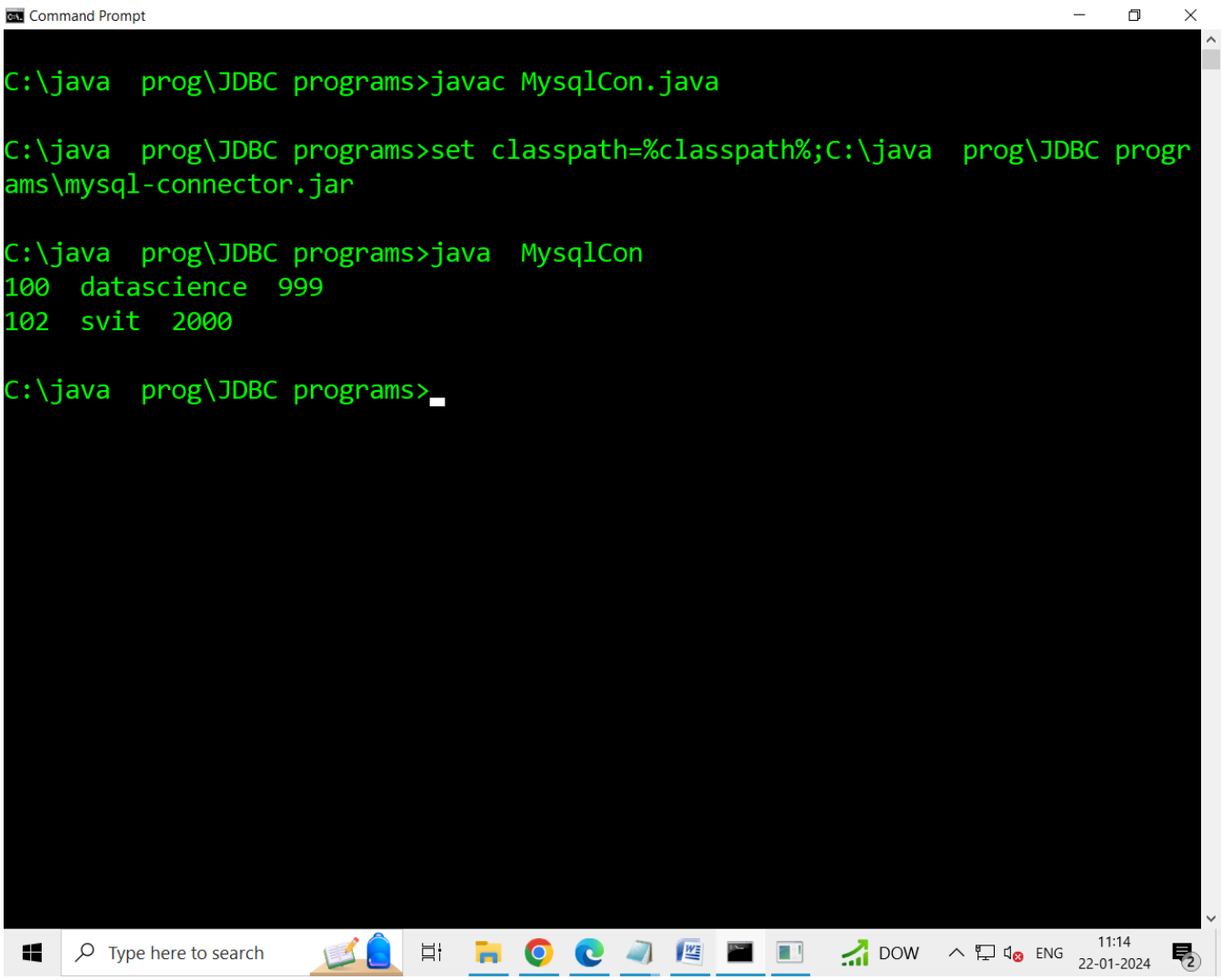
Step 7: Close the Connection and Statement

Like we close all I/O stream objects at the end of input/output operations, we also must close all the objects involved in the database connection. The objects, generally are of class Statement, Connection and ResultSet etc.

JDBC Program to retrieve the records from table

```
import java.sql.*;
class MysqlCon{
public static void main(String args[]){
try{
Class.forName("com.mysql.jdbc.Driver");
Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/emp","root","manager");
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("select * from student");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}}
```


C:\Program Files (x86)\MySQL\MySQL Server 5.1\bin\mysql.exe
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mysql> use emp;
Database changed
mysql> show tables;
+---------------+
| Tables_in_emp |
+---------------+
| department2 |
| department21 |
| emp1 |
| employee21 |
| employees |
| student |
+---------------+
6 rows in set (0.00 sec)
mysql> desc emp1;
+-------+-------------+------+-----+---------+-------+
| Field | Type | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
empid	int(11)	NO	PRI		
ename	varchar(20)	YES		NULL	
sal	int(11)	YES		NULL	
+-------+-------------+------+-----+---------+-------+
3 rows in set (0.00 sec)
mysql> select * from emp1;
+-------+-------------+------+
| empid | ename | sal |
+-------+-------------+------+
| 100 | datascience | 999 |
| 102 | svit | 2000 |
+-------+-------------+------+
2 rows in set (0.00 sec)
mysql>
Type here to search
DOW
ENG
11:14
22-01-2024



JDBC program to perform DML operations on employee table using menu driven program

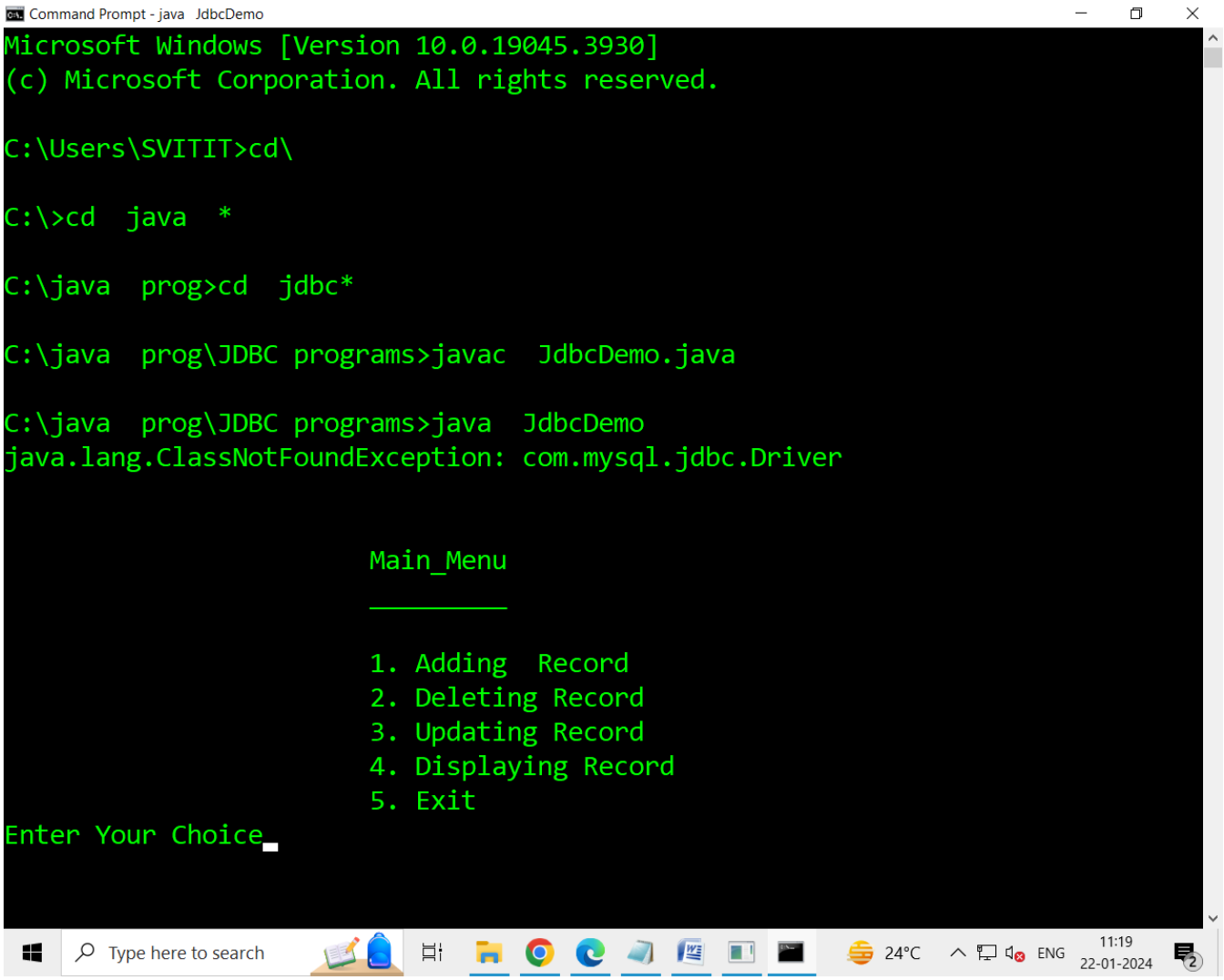
```
import java.util.*;
import java.sql.*;
public class JdbcDemo
{
Connection con;
PreparedStatement pst;
ResultSet rs;
Scanner s;
JdbcDemo()
{
try
{
```

```
Class.forName("com.mysql.jdbc.Driver");
con=DriverManager.getConnection("jdbc:mysql://localhost:3306/emp","root","manager");
s=new Scanner(System.in);

}catch(Exception e)
{
System.out.println(e);
}}

public void insertRecord() throws Exception
{

int eno,sal,count;
String name;
System.out.println("enter Emp No");
eno=s.nextInt();

System.out.println("enter Emp Name");
name=(String)s.next();

System.out.println("enter Emp sal");
sal=s.nextInt();
String query="insert into emp1 values(?,?,?)";
pst=con.prepareStatement(query);
pst.setInt(1,eno);
pst.setString(2,name);
pst.setInt(3,sal);
System.out.println(query);
count=pst.executeUpdate();
if(count>0)
System.out.println("Record sucessfully Inserted");
else
System.out.println("Sorry Record not Inserted");
}

public void DeleteRecord() throws Exception
{
int eno;
System.out.println("enter EmpId to delete");
eno=s.nextInt();

pst=con.prepareStatement("delete from emp1 where empid=?");
pst.setInt(1,eno);
```

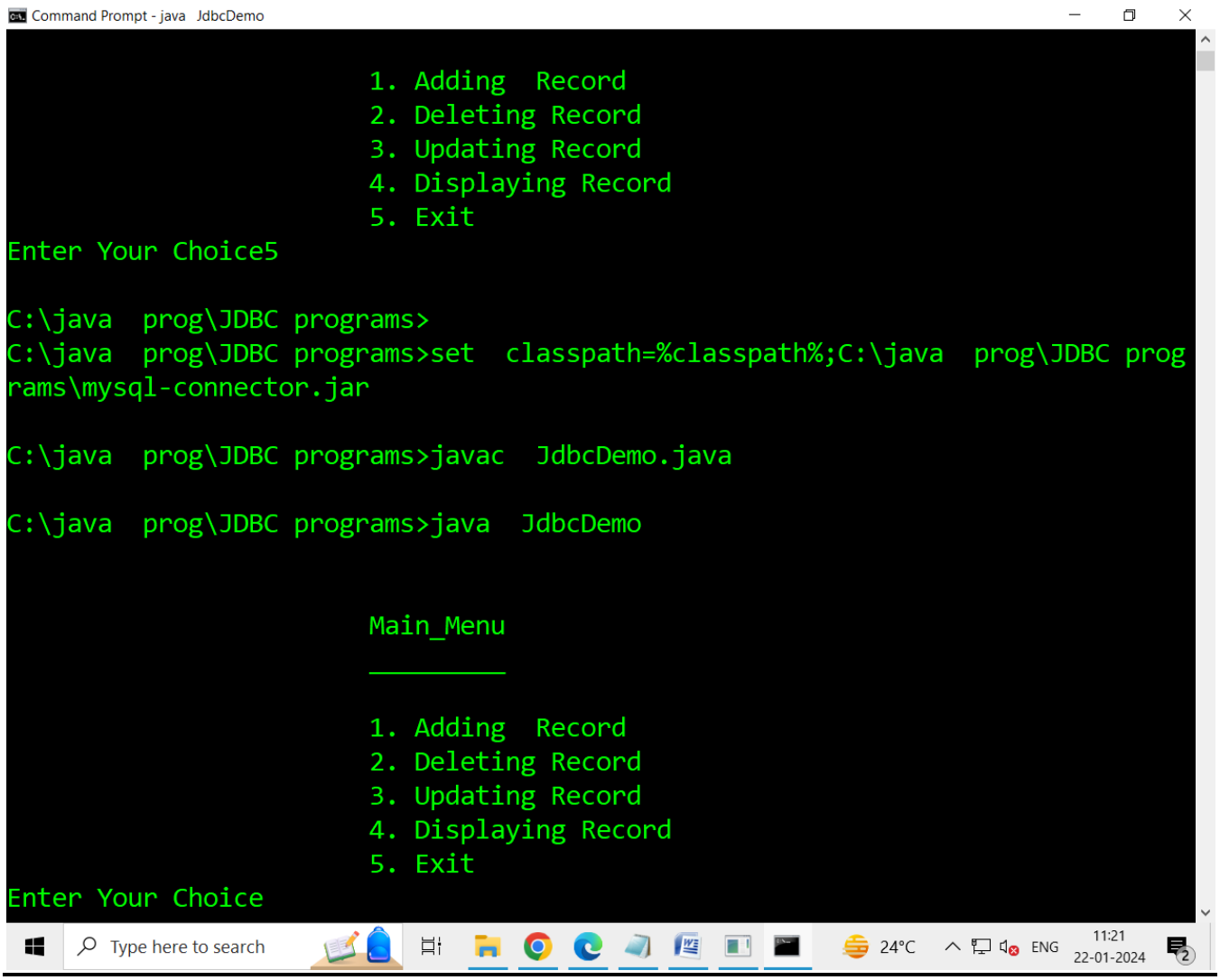
```
int  x=pst.executeUpdate();
if(x>0)
System.out.println("Record  sucessfully  Deleted");
else
System.out.println("Sorry! Record not deleted");
}

public  void display()
{
try
{
Statement stmt1=con.createStatement();
ResultSet rs1=stmt1.executeQuery("select * from emp1");
while(rs1.next())
{
System.out.println(rs1.getInt(1)+"  "+rs1.getString(2)+"  "+rs1.getInt(3));
}
}catch(Exception e)
{
System.out.println(e);
}
}

public  void UpdateRecord()  throws  Exception
{
int eno,sal1,count;
String name;
System.out.println("Enter Emp  No.  to  update");
eno=s.nextInt();
pst=con.prepareStatement("select  * from emp1 where  empid=?");
pst.setInt(1,eno);
rs=pst.executeQuery();
if(rs.next())
{
System.out.println("enter  Emp  Name");
name=(String)s.next();
System.out.println("enter  Emp  sal");
sal1=s.nextInt();
String query="update  emp1  set ename=?, sal=?  where empid=?";
pst=con.prepareStatement(query);
pst.setString(1,name);
pst.setInt(2,sal1);
pst.setInt(3,eno);
```

```
count=pst.executeUpdate();
if(count>0)
System.out.println("Record  Updated Sucessfully");
else
System.out.println("Record  Updated Sucessfully");
}
else
System.out.println("Sorry  Record  not Existing");
}
public  static  void main(String  args[]) throws Exception
{
int n;
Scanner  s1=  new Scanner(System.in);
JdbcDemo  s=new  JdbcDemo();
do
{

System.out.println( "\n\n\t\t\tMain_Menu");
System.out.println("\t\t\t_________\n");
System.out.println( "\t\t\t1. Adding  Record");
System.out.println( "\t\t\t2. Deleting Record");
System.out.println( "\t\t\t3. Updating Record");
System.out.println( "\t\t\t4. Displaying Record");
System.out.println( "\t\t\t5. Exit");

System.out.print( "Enter Your Choice");
n=s1.nextInt();
switch(n)
{
case 1:         s.insertRecord();
          System.out.println("Hello");
          break;

case 2: s.DeleteRecord();
          break;
case 3: s.UpdateRecord();
          break;

case 4:  s.display();
          break;
case 5:  System.exit(0);
          break;
default: System.out.println("Invalid  choice. pls Try  again");
```

```
}
}while(n<=5);
}}
```



To overcome the above Exception set the **classpath for mysql driver**



Check out the options during the executions on your PC

Networking

TCP/IP Sockets

TCP/IP Sockets are used to implement reliable, bidirectional, persistent, point-to-point stream-based connections between hosts on the internet. A ***socket***[1] is one endpoint that can be used to connect other programs that may reside either on the local machine or on any other machine on the internet.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The **ServerSocket** class is designed to be a "listener," which waits for clients to connect before doing anything. Thus, ServerSocket is for servers. The **Socket** class is for clients. It is designed to connect to server sockets and initiate protocol exchanges.

TCP/IP client sockets

1) Socket(String hostName, int port) throws UnknownHostException, IOException	Creates a socket connected to the named host and port.
2) Socket(InetAddress ipAddress, int port) throws IOException	Creates a socket using a preexisting InetAddress object and a port.

Important methods

1) InputStream getInputStream( ) throws IOException	Returns the InputStream associated with the invoking socket.
2) OutputStream getOutputStream( ) throws IOException	Returns the OutputStream associated with the invoking socket
3) synchronized void close()	closes this socket

TCP/IP Sever Sockets

1) ServerSocket(int port) throws IOException	Creates server socket on the specified port with a queue length of 50.
2) ServerSocket(int port, int maxQueue) throws IOException	Creates a server socket on the specified port with a maximum queue length of maxQueue.

[1] A socket is a point connection between server and a client on a network

3) ServerSocket(int port, int maxQueue, InetAddress localAddress) throws IOException	Creates a server socket on the specified port with a maximum queue length of maxQueue.

Important methods

1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Working of TCP/IP sockets

The TCP/IP protocol suite allows us to identify a machine uniquely all over the world using a 32-bit ***IP address***[2]. Any process may run on a single computer. each of these process is assigned a locally unique positive integer called the ***port***[3] number. Any process can now be identified uniquely all over the world by this port number together with the ip address of the machine in which it is running. This IP address and port number combination can be thought of as an address called the socket address. The socket address makes the communication between the process possible.

Communication using TCP sockets consists of the following steps:

1. The server creates a ServerSocket object, specifying the port number it listens on.
2. The server invokes the accept() method on this object. This method makes the server waiting until a request comes from the client.
3. The client creates a socket object, specifying the server address and port number to connect.
4. The constructor of the Socket class attempts to establish a connection to the specified server and port number. If the connection is established, it returns a Socket object that represents the logical connection to the client. The client use this Socket object communicate with the server. The accept() method on the ServerSocket object also returns a Socket Object to the server that is connected to the client's socket.

[2] An IP address is unique identification number allotted to every computer on a network or internet.
[3] Port number is a 2 byte number which is used identify socket uniquely.

Each socket has both an InputStream and an OutputStream. The client InputStream is connected to the server's OutputStream, and the Client's OutputStream is connected to the server's InputStream. Now both client and the server can communicate using I/O streams.

Example program on TCP/IP Sockets

TCPFactClient.java

```
import java.io.*;
import java.net.*;
public class TCPFactClient
{
public static void main(String args[])throws Exception
{
String fact;
Socket socket=new Socket("localhost",6789);
PrintWriter toserver=new PrintWriter(socket.getOutputStream(),true);
BufferedReader fromserver=new BufferedReader(new
InputStreamReader(socket.getInputStream()));
BufferedReader fromuser=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter an integer");
String n=fromuser.readLine();
toserver.println(n);
System.out.println("sent to server :" +n);
fact=fromserver.readLine();
System.out.println("Recieved from server :"+fact);
socket.close();
}
}
```

TCPFactServer.java

```
import java.io.*;
import java.net.*;
public class TCPFactServer
{
public static void main(String args[])throws Exception
{
ServerSocket serversocket=new ServerSocket(6789);
System.out.println("server is listening on port 6789");
Socket socket=serversocket.accept();
System.out.println("Request accepted");
```

```
BufferedReader fromclient=new BufferedReader(new
InputStreamReader(socket.getInputStream()));
PrintWriter  toclient=new PrintWriter(socket.getOutputStream(),true);
int n=Integer.parseInt(fromclient.readLine());
System.out.println("Received from client :"+n);
int fact=1;
for(int i=2;i<=n;i++)
{
fact*=i;
}
toclient.println(fact);
System.out.println("sent  to client :"+fact);
}}
```

InetAddress

The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address. You interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The **InetAddress** class hides the number inside. **InetAddress** can handle both IPv4 and IPv6 addresses.

Factory Methods

The **InetAddress** class has no visible constructors. To create an **InetAddress** object, you have to use one of the available factory methods. *Factory methods* are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer. Three commonly used **InetAddress** factory methods are shown here:

```
static InetAddress getLocalHost( )
  throws UnknownHostException

static InetAddress getByName(String hostName)
  throws UnknownHostException

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

The **getLocalHost( )** method simply returns the **InetAddress** object that represents the local host. The **getByName( )** method returns an **InetAddress** for a host name passed to it. If these methods are unable to resolve the host name, they throw an **UnknownHostException**.

On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The **getAllByName( )** factory method returns an array of **InetAddresses** that represent all of the addresses that a particular name resolves to. It will also throw an **UnknownHostException** if it can't resolve the name to at least one address.

InetAddress also includes the factory method **getByAddress( )**, which takes an IP address and returns an **InetAddress** object. Either an IPv4 or an IPv6 address can be used.

The following example prints the addresses and names of the local machine and two well-known Internet web sites:

```
// Demonstrate InetAddress.
import java.net.*;

class InetAddressTest
{
  public static void main(String args[]) throws UnknownHostException {
    InetAddress Address = InetAddress.getLocalHost();
    System.out.println(Address);
    Address = InetAddress.getByName("osborne.com");
    System.out.println(Address);
    InetAddress SW[] = InetAddress.getAllByName("www.nba.com");
    for (int i=0; i<SW.length; i++)
      System.out.println(SW[i]);
  }
}
```

Here is the output produced by this program. (Of course, the output you see may be slightly different.)

```
default/206.148.209.138
osborne.com/198.45.24.162
www.nba.com/64.5.96.214
www.nba.com/64.5.96.216
```

Datagrams

TCP/IP-style networking is appropriate for most networking needs. It provides a serialized, predictable, reliable stream of packet data. This is not without its cost, however. TCP includes many complicated algorithms for dealing with congestion control on crowded networks, as well as pessimistic expectations about packet loss. This leads to a somewhat inefficient way to transport data. Datagrams provide an alternative.

Datagrams are bundles of information passed between machines. They are somewhat like a hard throw from a well-trained but blindfolded catcher to the third baseman. Once the datagram has been released to its intended target, there is no assurance that it will arrive or even that someone will be there to catch it. Likewise, when the datagram is received, there is no assurance that it hasn't been damaged in transit or that whoever sent it is still there to receive a response.

Java implements datagrams on top of the UDP protocol by using two classes: the **DatagramPacket** object is the data container, while the **DatagramSocket** is the mechanism used to send or receive the **DatagramPacket**s. Each is examined here.

DatagramSocket

DatagramSocket defines four public constructors. They are shown here:

DatagramSocket( ) throws SocketException

DatagramSocket(int *port*) throws SocketException

DatagramSocket(int *port*, InetAddress *ipAddress*) throws SocketException

DatagramSocket(SocketAddress *address*) throws SocketException

Differences between TCP and UDP

Parameter	TCP	UDP
Type of Service	Connection-oriented.	Connectionless.
Reliability	It is reliable as it guarantees that the data reaches the destination address.	Unreliable.
Error-Checking	TCP uses robust error-checking methods and ensures error-free data is transmitted.	UDP used basic error-checking mechanisms using checksums.
Sequence Control	Sequencing of data is done in TCP, i.e., packets arrive in order at the receiver host.	There is no sequencing in UDP.
Retransmission of lost data	Retransmission of lost or incorrect packets is possible in TCP.	No retransmission of lost packets in UDP.
Speed	Due to error-control and flow-control, there is processing overhead in TCP, and it is slow.	UDP is faster and more efficient than TCP.
Header Size	Varies between 20-60 bytes.	Has header of fixed size(8 bytes).
Broadcasting	Not supported	Supports broadcasting.
Protocols	Used by HTTP, HTTPs, FTP, SMTP, Telnet etc.	Used by DNS, DHCP, TFTP, SNMP, etc.
Overhead	Higher than UDP.	Very low.

URL(**U**niform **R**esource **L**ocator) represents the address that is specified to access some information (or) resource on internet.

URL represents a class in java.net package

example:

http://www.abc.com:80/index.html
Here protocol --->http
server name or ip address of the server---->www.abc.com
port no--->80
the file that is referred is index.html

Java's URL class has several constructors; each can throw a MalformedURLException.

1. URL(String urlSpecifier) throws MalformedURLException
2. URL(String protocolName, String hostName, int port, String path) throws MalformedURLException
3. URL(String protocolName, String hostName, String path) throws MalformedURLException
4. URL(URL urlObj, String urlSpecifier) throws MalformedURLException

/* Demo Program on URL*/

```
import java.net.*;
class URLTest
{
public static void main(String args[]) throws Exception
{
URL obj=new URL("https://ceotelangana.nic.in/Mlc2024.html");
System.out.println("Protocol:"+obj.getProtocol());
System.out.println("Host:"+obj.getHost());
System.out.println("File:"+obj.getFile());
System.out.println("Port:"+obj.getPort());
System.out.println("URL:"+obj.toExternalForm());
}}
```

Introduction to Java Beans

What Is a Java Bean?

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

Advantages of Java Beans

The following list enumerates some of the benefits that Java Bean technology provides for a component developer:

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

Introspection

At the core of Java Beans is *introspection*. This is the process of analyzing a Bean to determine its capabilities. This is an essential feature of the Java Beans API because it allows another application, such as a design tool, to obtain information about a component. Without introspection, the Java Beans technology could not operate.

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

Introduction to 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**

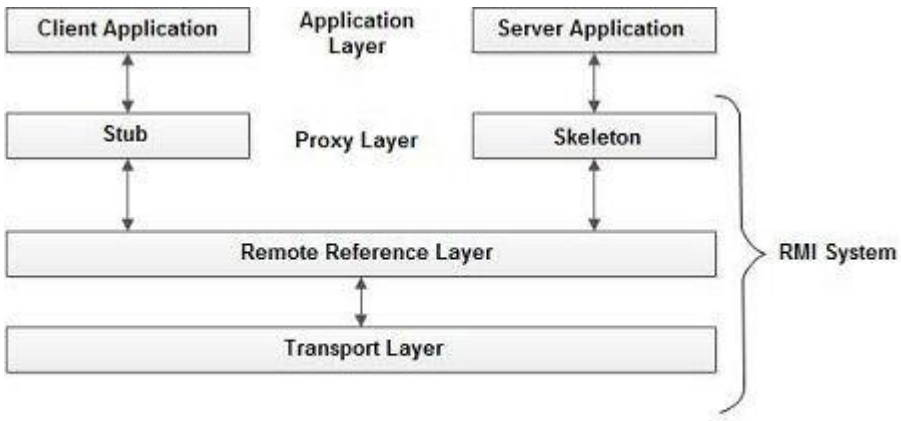
(or)

Allows objects-to-object communication between different java virtual machines (JVM).JVMs can be distinct entities located on the same or separate computers. yet one JVM can invoke methods belonging to an object stored in another JVM. This enables applications to call object methods located remotely, sharing resources and processing load across systems. Methods can even pass objects that a foreign virtual machine has never encountered before ,allowing dynamic loading of new classes as required. This really is quite a powerful feature.

Architecture of RMI

The RMI implementation is essentially built on three layers.

1. The stub/Skeleton(proxy) Layer
2. The Remote Reference Layer
3. The transport Layer



Archilecture of RMI

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:

1. It initiates a connection with remote Virtual Machine (JVM) containing the remote object,
2. It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
3. It waits for the result
4. It reads (unmarshals) the return value or exception, and
5. 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:

1. Unmarshals (reads)the parameter for the remote method
2. It invokes the method on the actual remote object implementation
3. Marshals (writes and transmits) the result to the caller.

Remote Reference Layer

1. The proxy layer is connected to the RMI mechanism through the Remote Reference Layer. This layer is responsible for the communication and transfer of objects between client and server. The invocation semantics of the RMI connection is defined and supported by this layer.
2. The remote Reference Layer is responsible for maintaining the session during the method call. i.e. It manages the references made by the client to the remote server object. This layer is also responsible for handling duplicated objects.

Transport Layer

The transport layer makes the stream based network connection over TCP/IP between the JVMs, and is responsible for setting and managing those connections. even if two JVMs are running on the same physical computer, they connect through their host computer's TCP/IP network protocol stack

RMI API

1. public static java.rmi.Remote lookup(java.lang.String) throws
   java.rmi.NotBoundException,
   java.net.MalformedURLException,
   java.rmi.RemoteException;

   It returns the reference of the remote object.

2. public static void bind(java.lang.String, Java.rmi.Remote) throws
   java.rmi.AlreadyBoundException,
   java.net.MalformedURLException,
   java.rmi.RemoteException;

   It binds the remote object with the given name.

3. public static void unbind(java.lang.String) throws java.rmi.RemoteException,
   java.rmi.NotBoundException,
   java.net.MalformedURLException;

   It destroys the remote object which is bound with the given name.

4. public static void rebind(java.lang.String, java.rmi.Remote) throws
   java.rmi.RemoteException,
   java.net.MalformedURLException;

   It binds the remote object to the new name.

5. public static java.lang.String[] list(java.lang.String) throws
   java.rmi.RemoteException,
   java.net.MalformedURLException;

   It returns an array of the names of the remote objects bound in the registry.

Steps required to develop RMI Application

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

RMI Application to print Hello Message

//Remote Interface

```
import java.rmi.*;
public interface Hello extends Remote
{
public abstract String sayHello() throws RemoteException;
}
```

//Implementation of Remote interface

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloImple extends UnicastRemoteObject implements Hello
{
public HelloImple() throws RemoteException
{}
public String sayHello() throws RemoteException
{
return "Hello SVIT";
}
}
```

//Server Application

```
import java.rmi.*;
public class HelloServer
{
public static void main(String args[]) throws Exception
{
HelloImple hello=new HelloImple();
Naming.rebind("hello",hello);
System.out.println("Object Bound");
}
}
```

//Client application

```
import java.rmi.*;
public class HelloClient
{
public static void main(String args[])throws Exception
{
Hello hello=(Hello)Naming.lookup("hello");
System.out.println(hello.sayHello());}}
```

For running **this** rmi Program,

1) compile all the java files

C:\RMI>javac *.java

2)create stub and skeleton object by rmic tool

C:\RMI>rmic HelloImple

3)start rmi registry in one command prompt

C:\RMI>start rmiregistry

4)start the server in another command prompt

C:\RMI>java HelloServer

5)start the client application in another command prompt

C:\RMI>java HelloClient

Pictorial Representation of RMI Program Execution