

JDBC

INTRODUCTION TO JDBC

JDBC stands for **J**ava **D**atabase **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. Java can be used to write different types of executables, such as –

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)
- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

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 –

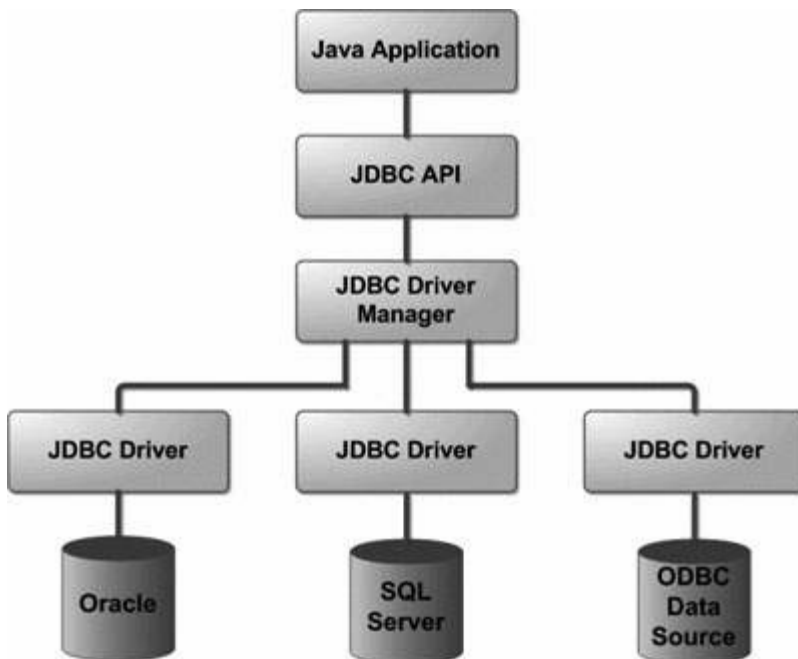
- **JDBC API:** This provides the application-to-JDBC Manager connection.

- **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 –



Common JDBC Components

The JDBC API provides the following interfaces and classes –

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using

communication sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.
- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.
- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.
- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.
- **SQLException:** This class handles any errors that occur in a database application.

JDBC Drivers

JDBC drivers are divided into four types or levels. The **different types of jdbc drivers** are:

Type 1: JDBC-ODBC Bridge driver (Bridge)

Type 2: Native-API/partly Java driver (Native)

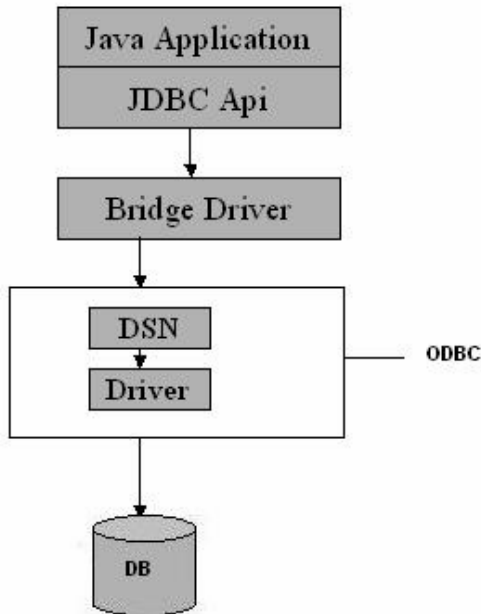
Type 3: AllJava/Net-protocol driver (Middleware)

Type 4: All Java/Native-protocol driver (Pure)

Type 1 JDBC Driver

JDBC-ODBC Bridge driver

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.



Type 1: JDBC-ODBC Bridge

Advantage

The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

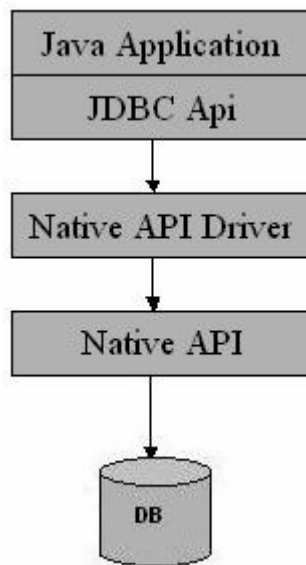
Disadvantages

1. Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
3. The client system requires the ODBC Installation to use the driver.
4. Not good for the Web.

Type 2 JDBC Driver

Native-API/partly Java driver

The distinctive characteristic of type 2 jdbc drivers are that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are shown below. Example: Oracle will have oracle native api.



Type 2: Native api/ Partly Java Driver

Advantage

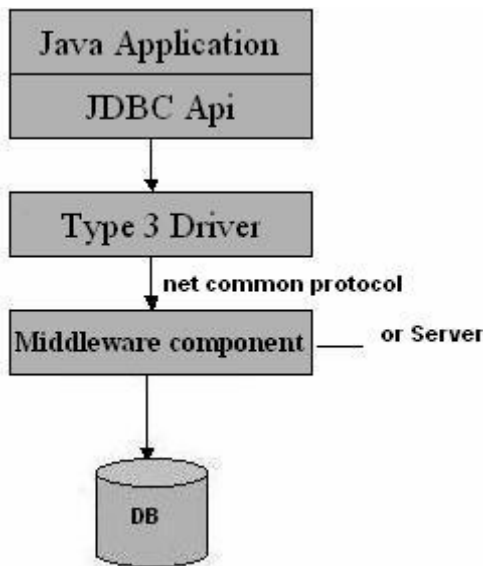
The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type 1 and also it uses Native api which is Database specific.

Disadvantage

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
2. Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
3. If we change the Database we have to change the native api as it is specific to a database
4. Mostly obsolete now
5. Usually not thread safe.

Type 3 JDBC Driver**All Java/Net-protocol driver**

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.

**Type 3: All Java/ Net-Protocol Driver****Advantage**

1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.
2. This driver is fully written in Java and hence Portable. It is suitable for the web.
3. There are many opportunities to optimize portability, performance, and scalability.
4. The net protocol can be designed to make the client JDBC driver very small and fast to load.
5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced system administration such as logging and auditing.
6. This driver is very flexible allows access to multiple databases using one driver.
7. They are the most efficient amongst all driver types.

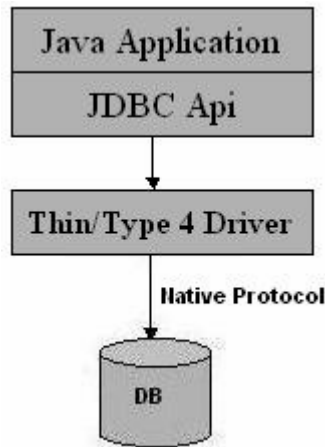
Disadvantage

It requires another server application to install and maintain. Traversing the recordset may take longer, since the data comes through the backend server.

Type 4 JDBC Driver

Native-protocol/all-Java driver

The Type 4 uses java networking libraries to communicate directly with the database server.



Type 4: Native-protocol/all-Java driver

Advantage

1. The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
2. Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
3. You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

With type 4 drivers, the user needs a different driver for each database.

Database Connectivity

JDBC stands for **J**ava **D**atabase **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.

Steps to connect database in java using JDBC are given below:

1. Load the JDBC driver.
2. Connection.
3. Statement.
4. Execute statement.
5. Close database connection.

1. Load the JDBC driver:

First step is to load or register the JDBC driver for the database. Class class provides `forName()` method to dynamically load the driver class.

Syntax:

```
Class.forName("driverClassName");
```

To load or register OracleDriver class:

Syntax:

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

To load or register MySQL class:

Syntax:

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

2. Create connection:

Second step is to open a database connection. `DriverManager` class provides the facility to create a connection between a database and the appropriate driver. To open a database connection we can call `getConnection()` method of `DriverManager` class.

Syntax:

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

To create a connection with Oracle database:

Syntax:

```
Connection connection =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe",
user,"password");
```

To create a connection with **MySQL** database:

Syntax:

```
Connection connection =
DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb","ro
ot","");
```

3. Create statement:

The statement object is used to execute the query against the database. Connection interface acts as a factory for statement object. A statement object can be any one of the *Statement*, *CallableStatement*, and *PreparedStatement* types. To create a statement object we have to call `createStatement()` method of Connection interface. Connection interface also provides the transaction management methods like `commit()` and `rollback()` etc.

Syntax:

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

4. Execute statement:

Statement interface provides the methods to execute a statement.

To execute a statement for select query use below:

Syntax:

```
ResultSet resultSet = stmt.executeQuery(selectQuery);
```

5. Close database connection:

After done with the database connection we have to close it. Use close() method of Connection interface to close database connection. The statement and ResultSet objects will be closed automatically when we close the connection object.

Syntax:

```
connection.close();
```

Example

```
<%@page import="java.sql.*" %>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
out.println("<h1>");
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
out.println("Connect set");
}
catch(Exception e)
{
out.println(e.toString());

}
%>
```

Studying Javax.sql.* package

Provides the API for server side data source access and processing from the Java™ programming language. This package supplements the java.sql package.

The javax.sql package provides for the following:

1. The DataSource interface as an alternative to the DriverManager for establishing a connection with a data source
2. Connection pooling and Statement pooling
3. Distributed transactions
4. Rowsets

Applications use the DataSource and RowSet APIs directly, but the connection pooling and distributed transaction APIs are used internally by the middle-tier infrastructure.

CRUD OPERATIONS.

Basic database operations (CRUD - Create, Retrieve, Update and Delete) using JDBC (Java Database Connectivity) API. These CRUD operations are equivalent to the INSERT, SELECT, UPDATE and DELETE statements in SQL language.

Create a table

```
<%@page import="java.sql.*" %>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
out.println("<h1>");
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
```

```

out.println("Connect set");
Statement stmt=c.createStatement();
String sql="CREATE TABLE studentbalu " + "(sno INTEGER not
Null,"+
                " sname VARCHAR2(20)," +
                " age VARCHAR2(20))";

stmt.executeUpdate(sql);
System.out.println("Create table in given database...");

}
catch(Exception e)
{
out.println(e.toString());
}
%>

```

Insert operation

```

<%@page import="java.sql.*" %>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
out.println("<h1>");
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
out.println("Connect set");
Statement stmt=c.createStatement();
String sql= sql = "INSERT INTO studentbalu VALUES (101, 'balu',
25)";
stmt.executeUpdate(sql);

```

```
System.out.println("Inserted records into the table...");
```

```

}
catch(Exception e)
{
out.println(e.toString());
}
%>

```

Update operation

```

<%@page import="java.sql.*" %>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
out.println("<h1>");
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
out.println("Connect set");
Statement stmt=c.createStatement();
String sql= "UPDATE studentbalu " +
           "SET age = 42 WHERE sno=101";
stmt.executeUpdate(sql);
System.out.println("Updated records into the table...");

}
catch(Exception e)

```

```
{
out.println(e.toString());

}
%>
```

Delete operation

```
<%@page import="java.sql.*" %>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
out.println("<h1>");
out.println("Loaded the Driver");
Connection
c=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe
","system","manager");
out.println("Connect set");
Statement stmt=c.createStatement();
String sql = "DELETE FROM studentbalu " +
        "WHERE sno = 101";
stmt.executeUpdate(sql);
System.out.println("deleted records into the table...");

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

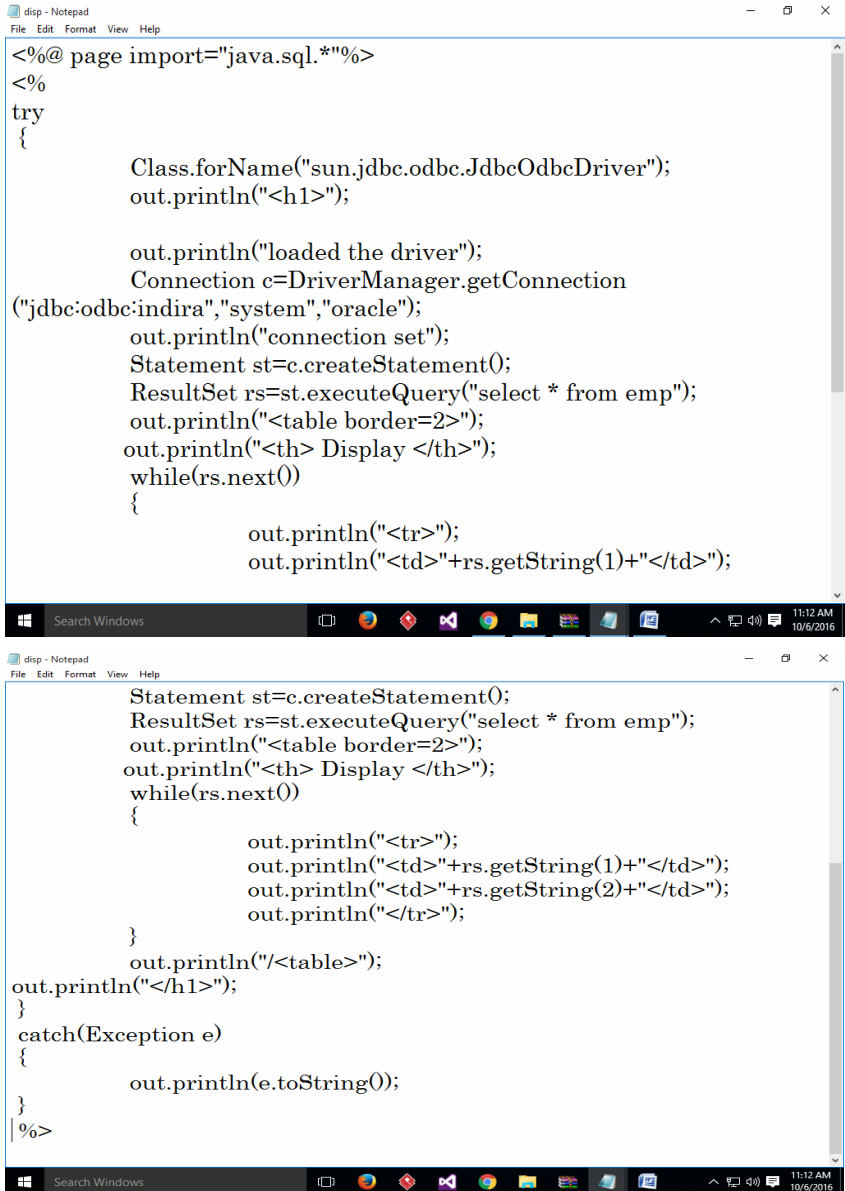
}
}
```

```
%>
```

Accessing a database from a JSP Page

Sample programs to access database from JSP:

Write a JSP to display employee number and name from emp table

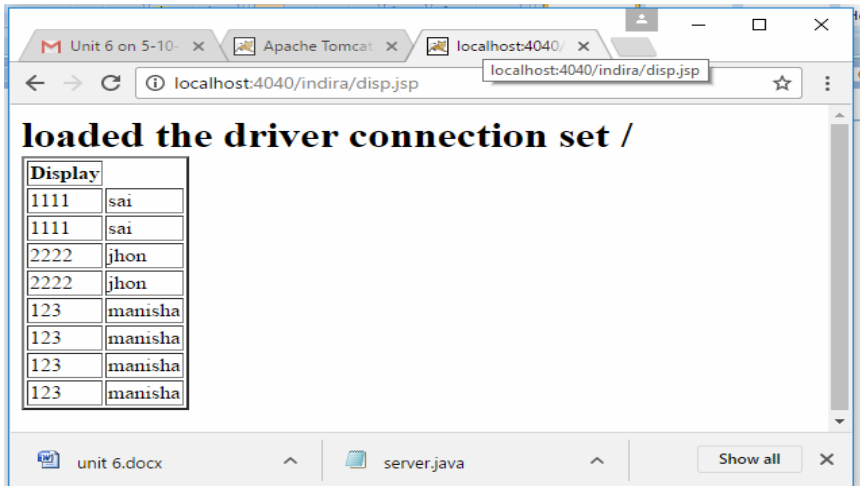


```

disp - Notepad
File Edit Format View Help
<%@ page import="java.sql.*"%>
<%
try
{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    out.println("<h1>");

    out.println("loaded the driver");
    Connection c=DriverManager.getConnection
("jdbc:odbc:indira","system","oracle");
    out.println("connection set");
    Statement st=c.createStatement();
    ResultSet rs=st.executeQuery("select * from emp");
    out.println("<table border=2>");
    out.println("<th> Display </th>");
    while(rs.next())
    {
        out.println("<tr>");
        out.println("<td>" +rs.getString(1)+"</td>");

        Statement st=c.createStatement();
        ResultSet rs=st.executeQuery("select * from emp");
        out.println("<table border=2>");
        out.println("<th> Display </th>");
        while(rs.next())
        {
            out.println("<tr>");
            out.println("<td>" +rs.getString(1)+"</td>");
            out.println("<td>" +rs.getString(2)+"</td>");
            out.println("</tr>");
        }
        out.println("/<table>");
    }
    out.println("</h1>");
}
catch(Exception e)
{
    out.println(e.toString());
}
}%>
  
```


Output:

- **Write a JSP to Insert one record into dept table and Display them.**

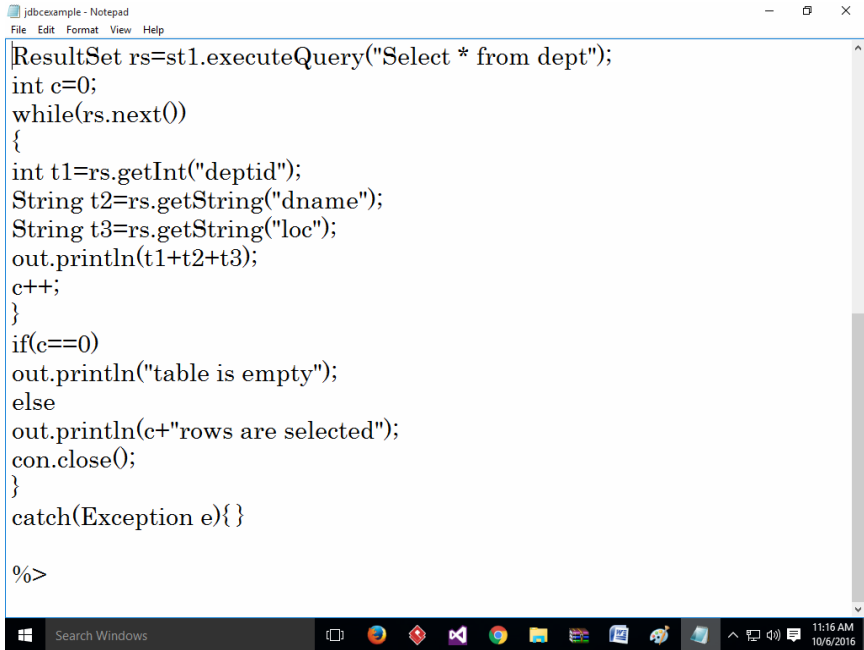
```

jdbcxample - Notepad
File Edit Format View Help
<%@page import="java.sql.*" %>

<%

try
{
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con=DriverManager.getConnection
    ("Jdbc:Odbc:ora","system","system");
    Statement st=con.createStatement();
    Statement st1=con.createStatement();
    String str="insert into dept values(1,'sales','kakinada)";
    int n=st.executeUpdate(str);
    if(n>0)
    out.println("Inserted");
    else
    out.println("error");
    ResultSet rs=st1.executeQuery("Select * from dept");
    int c=0;

```



```

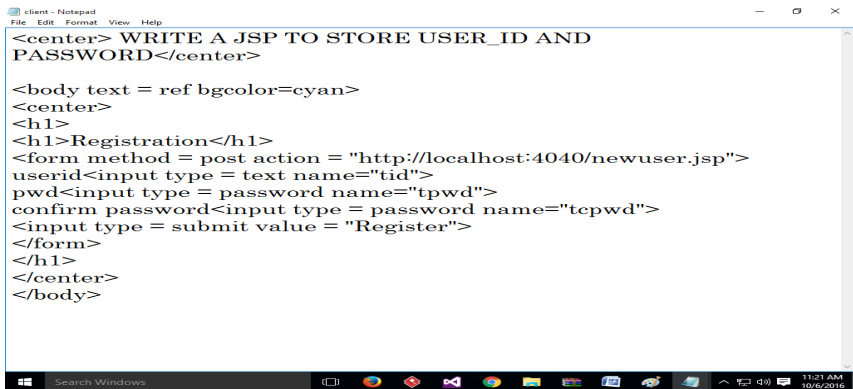
ResultSet rs=st1.executeQuery("Select * from dept");
int c=0;
while(rs.next())
{
int t1=rs.getInt("deptid");
String t2=rs.getString("dname");
String t3=rs.getString("loc");
out.println(t1+t2+t3);
c++;
}
if(c==0)
out.println("table is empty");
else
out.println(c+"rows are selected");
con.close();
}
catch(Exception e){}

%>

```

WRITE A JSP TO STORE USER_ID AND PASSWORD and Display

User.html



```

<center> WRITE A JSP TO STORE USER_ID AND
PASSWORD</center>

<body text = ref bgcolor=cyan>
<center>
<h1>
<h1>Registration</h1>
<form method = post action = "http://localhost:4040/newuser.jsp">
userid<input type = text name="tid">
pwd<input type = password name="tpwd">
confirm password<input type = password name="tcpwd">
<input type = submit value = "Register">
</form>
<h1>
</center>
</body>

```

Newuser.jsp

```

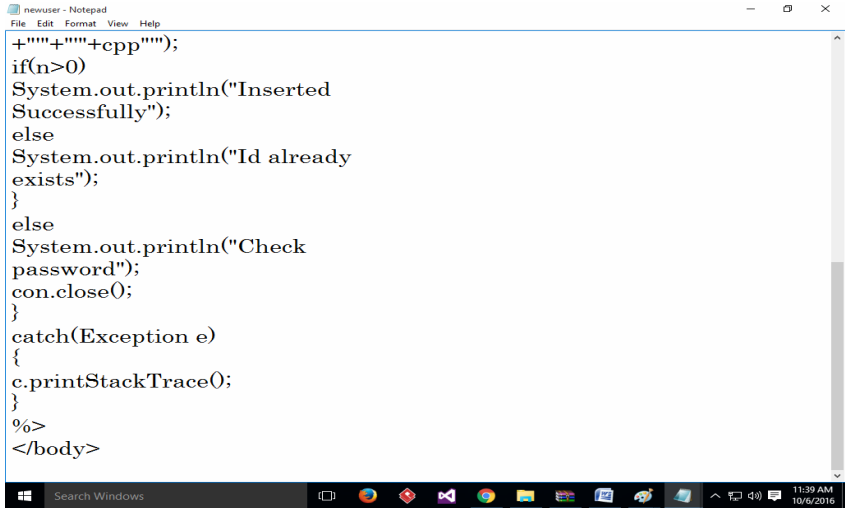
newuser - Notepad
File Edit Format View Help
<% @ page import = "java.sql.*"%>
<body>
<%
try
{
class.forName
("sun.jdbc.odbc.JdbcOdbcDriver");
Connection con=DriverManager.getConnection
("jdbc:odbc:indira","system","oracle");
Statement st=con.createStatement();
String id,pwd,cp;
id=request.getParameter("tid");
pwd=request.getParameter("tpwd");
cp=request.getParameter("tcpwd");
if(pwd.equals(cp))
ResultSet rs=st.executeQuery("select * from webusers where
userid="+id);
if(rs.next()
{

```

```

newuser - Notepad
File Edit Format View Help
{
int n=executeUpdate("insert into webusers values('"+id+"','"+pwd
+"','"+cp+"')");
if(n>0)
System.out.println("Inserted
Successfully");
else
System.out.println("Id already
exists");
}
else
System.out.println("Check
password");
con.close();
}
catch(Exception e)
{
c.printStackTrace();
}
}

```

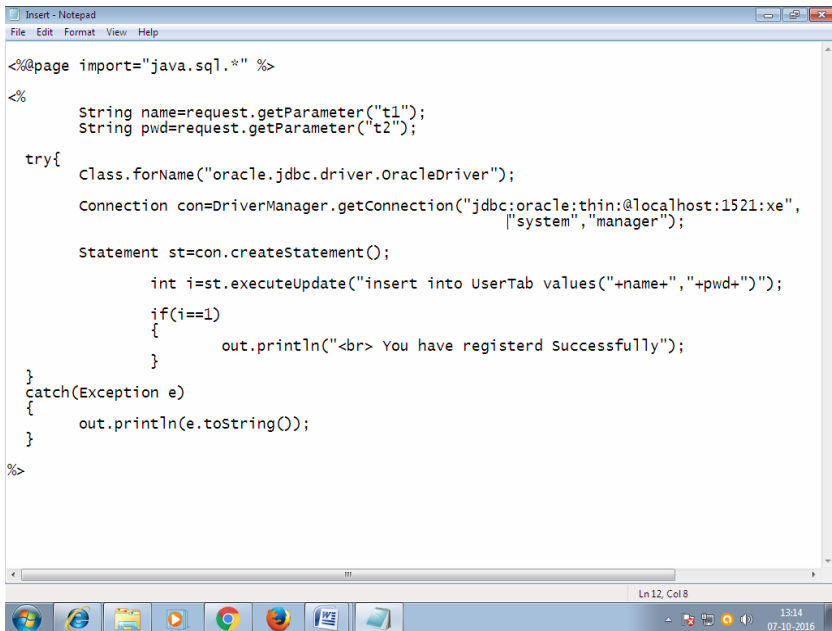


```

newuser - Notepad
File Edit Format View Help
+""+""+cpp""");
if(n>0)
System.out.println("Inserted
Successfully");
else
System.out.println("Id already
exists");
}
else
System.out.println("Check
password");
con.close();
}
catch(Exception e)
{
c.printStackTrace();
}
%>
</body>

```

Accessing Database using Type 4 Driver



```

Insert - Notepad
File Edit Format View Help
<%@page import="java.sql.*" %>
<%
    String name=request.getParameter("t1");
    String pwd=request.getParameter("t2");

    try{
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe",
            "system","manager");

        Statement st=con.createStatement();

        int i=st.executeUpdate("insert into UserTab values("+name+","+pwd+")");

        if(i==1)
        {
            out.println("<br> You have registerd Successfully");
        }
    }
    catch(Exception e)
    {
        out.println(e.toString());
    }
%>

```