JDBC: Introduction, Database Programming Using JDBC: How JDBC Works, JDBC Architecture, JDBC Driver Types; Studying javax.sql.*package, Accessing Database From JSP page: Use of Prepared Statement, ResultSet.

Introduction to 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.

Applications of JDBC

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 executable, such as −

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

All of these different executable 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.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas −

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.
- Annotations.

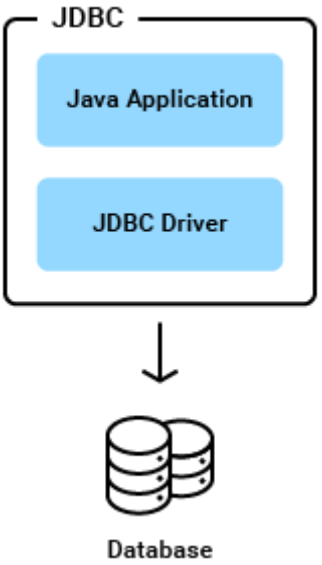
Database Programming using JDBC:

How JDBC Works?

JDBC works in the following way.

1. First of all java application establishes connection with the datasource.
2. Java application sends queries to the Datasource using JDBC driver.
3. The JDBC driver connects to the corresponding database and retrieve results
4. These results are based on sql statements which are then retured to the Java application.
5. Java application then uses the retrieved result for further processing.

The following figure shows the components of the\ JDBC model.



The Java application calls JDBC classes and interfaces to submit SQL statements and retrieve results.

The JDBC API is implemented through the JDBC driver. The JDBC Driver is a set of classes that implement the JDBC interfaces to process JDBC calls and return result sets to a Java application. The database (or data store) stores the data retrieved by the application using the JDBC Driver.

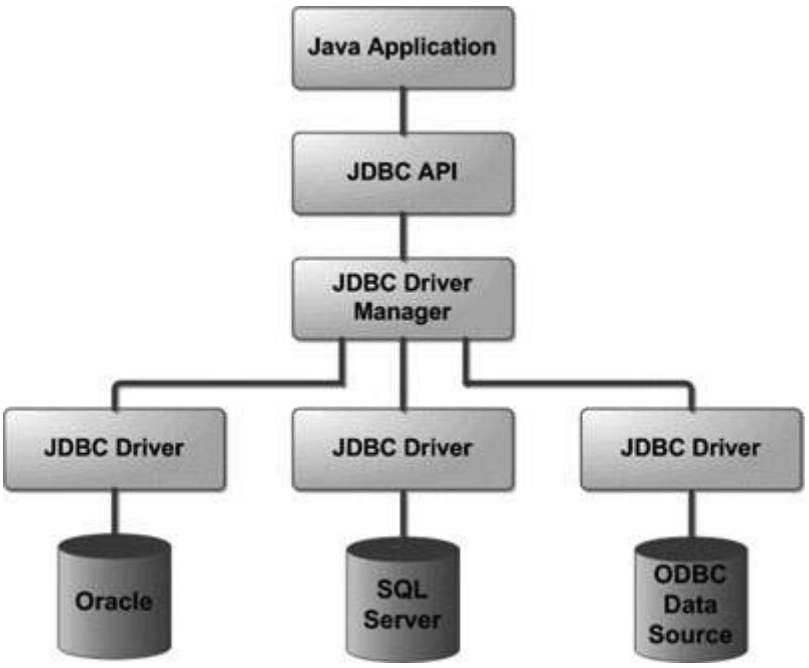
JDBC Architecture

The JDBC architecture consists of two major components

I. Java API

II. JDBC Driver types

The following diagram shows the JDBC architecture.



I. Java API :

Java API is set of various Interfaces Classes and Exception through which Java application can send sql statements to a database. JDBC API is present in **java.sql.*** and **javax.sql.*** packages. We use the following core JDBC classes and interfaces that belong to java.sql package

1. DriverManager: When a java application needs connection with the database it invokes the DriverManager class .This class loads the JDBC drivers in the memory. The DriverManager also attempts to open connection with the desired Database.

2. Connection: This is an interface which represents connectivity with the Data source. The connection is used for creating the Statement interface.

3. Statement: This interface is used for representing sql statements. Some examples of sql statements are

- Select *from students;
- Update students set name = ‘aaa’ where rollno = ‘12’;
- Delete from students where rollno = ‘10’

there are two specialized statement types **PreparedStatement** and **CallableStatement**. PreparedStatement represents pre compiled sql statements.

PreparedStatement Example:

Select *from students where name =?

The placeholder represented by? is used in the above sql statement. There are specific typesetter methods that assign value to the place holders before sql statements are executed.

CallableStatement: It is used for stored procedures. In this type of statements we can assign the methods for the type of output arguments.

4. ResultSet: This interface is used to represent the database ResultSet, after using select statement the result obtained from the database can be displayed using ResultSet.

5. SQLEXCEPTION: for handling SQLEXCEPTIONS this interface is used.

II.JDBC Driver Types

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

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. This is now discouraged because of thin driver.

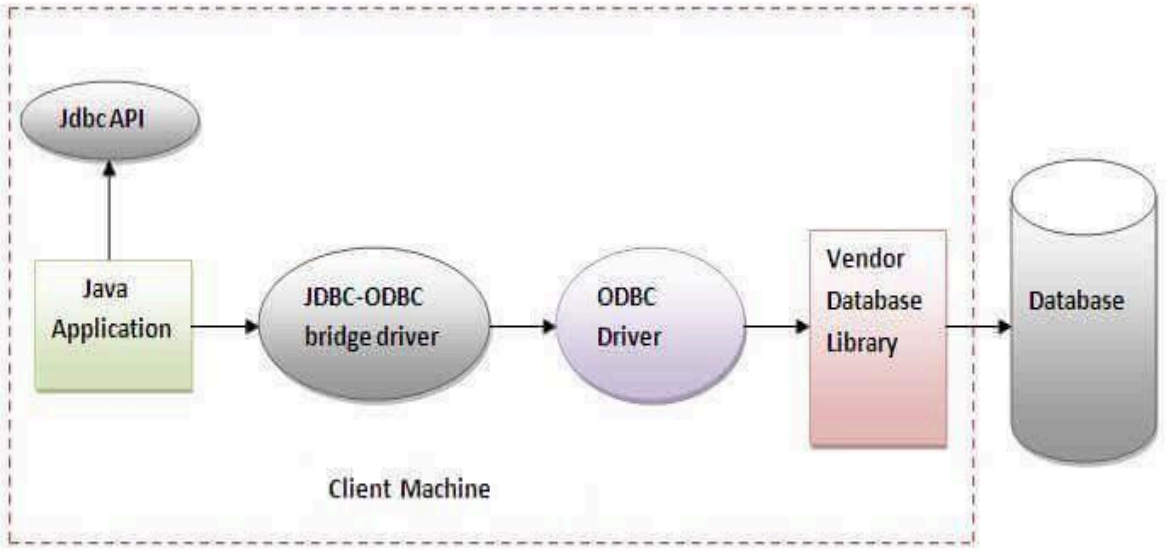


Figure- JDBC-ODBC Bridge Driver

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:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- 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.

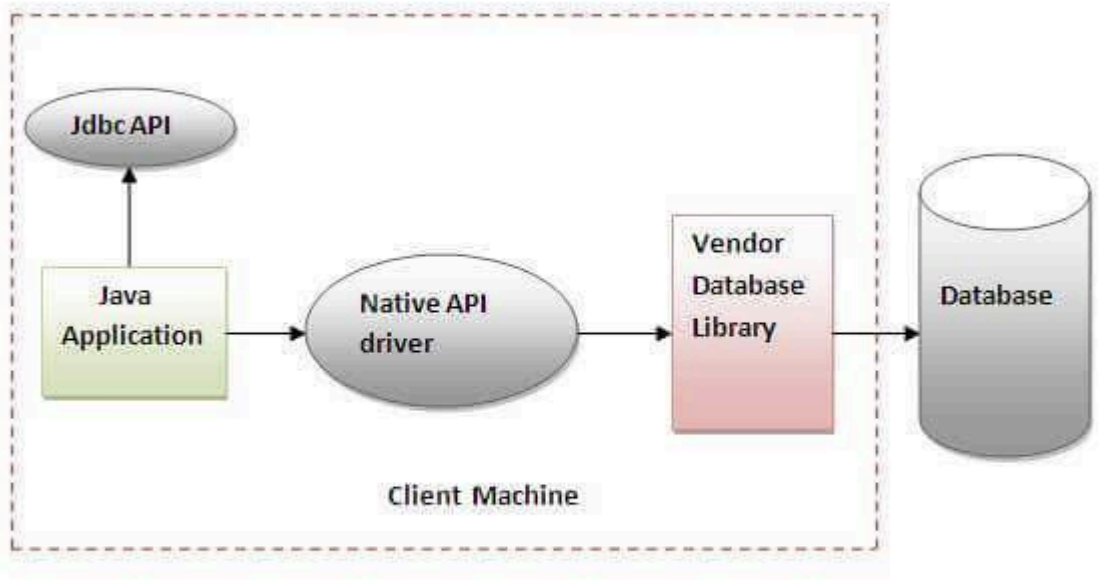


Figure- Native API Driver

Advantage:

- Performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- 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.

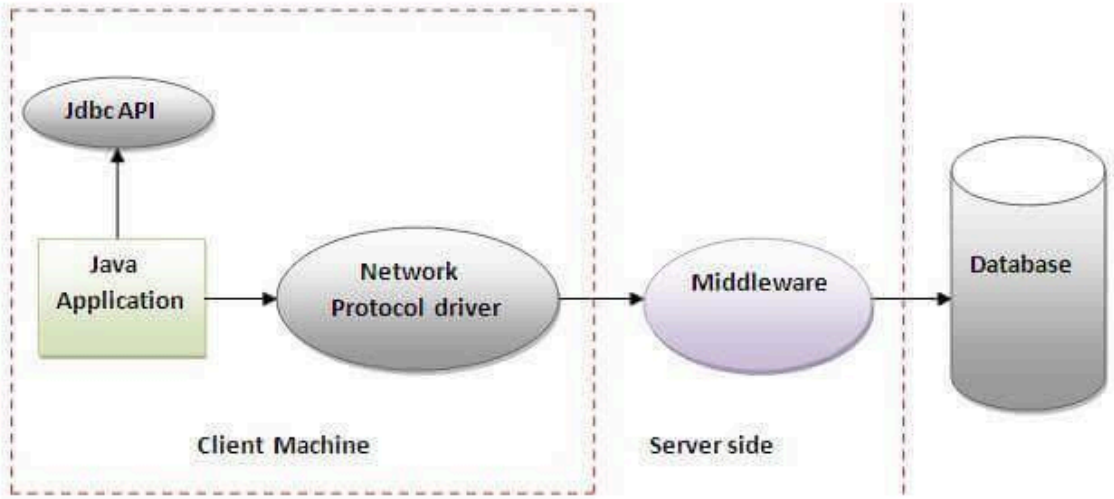


Figure- Network Protocol Driver

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.

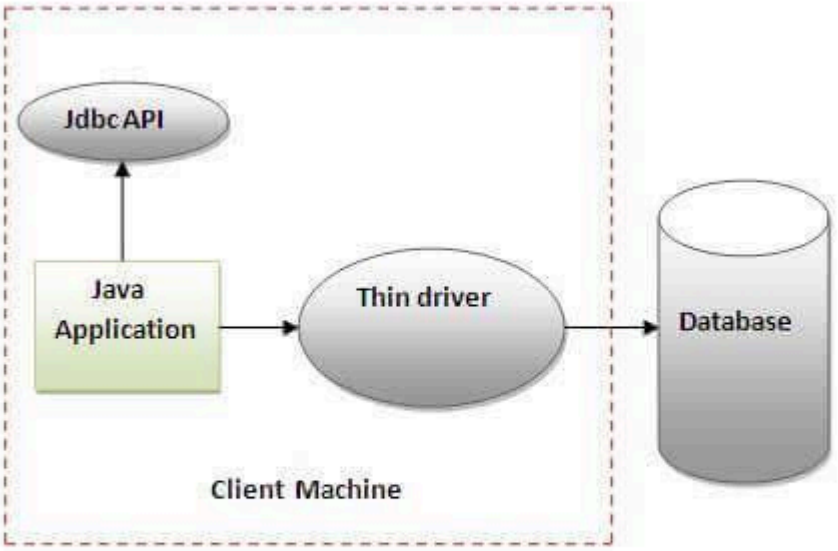


Figure- Thin Driver

Advantage:

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

Disadvantage:

- Drivers depend on the Database.

javax.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.

Interface Summary

Interface	Description
CommonDataSource	Interface that defines the methods which are common between DataSource, XADataSource and ConnectionPoolDataSource.
ConnectionEventListener	An object that registers to be notified of events generated by a PooledConnection object.
ConnectionPoolDataSource	A factory for PooledConnection objects.
DataSource	A factory for connections to the physical data source that this DataSource object represents.
PooledConnection	An object that provides hooks for connection pool management.
RowSet	The interface that adds support to the JDBC API for the JavaBeans™ component model.
RowSetInternal	The interface that a RowSet object implements in order to present itself to a RowSetReader or RowSetWriter object.
RowSetListener	An interface that must be implemented by a component that wants to be notified when a significant event happens in the life of a RowSet object.
RowSetMetaData	An object that contains information about the columns in a RowSet object.
RowSetReader	The facility that a disconnected RowSet object calls on to populate itself with rows of data.
RowSetWriter	An object that implements the RowSetWriter interface, called a *writer*.
StatementEventListener	An object that registers to be notified of events that occur on PreparedStatements that are in the Statement pool.
XAConnection	An object that provides support for distributed transactions.
XADataSource	A factory for XAConnection objects that is used internally.

Class Summary

Class	Description
ConnectionEvent	An Event object that provides information about the source of a connection-related event.
RowSetEvent	An Event object generated when an event occurs to a RowSet object.
StatementEvent	A StatementEvent is sent to all StatementEventListeners which were registered with a PooledConnection.

JDBC Database Connection:

After you've installed the appropriate driver, it is time to establish a database connection using JDBC.

The programming involved to establish a JDBC connection is fairly simple. Here are these simple four steps −

- **Import JDBC Packages** − Add **import** statements to your Java program to import required classes in your Java code.
- **Register JDBC Driver** − this step causes the JVM to load the desired driver implementation into memory so it can fulfil your JDBC requests.
- **Database URL Formulation** − this is to create a properly formatted address that points to the database to which you wish to connect.
- **Create Connection Object** − finally, code a call to the *DriverManager* object's *getConnection ( )* method to establish actual database connection.

Import JDBC Packages

The **Import** statements tell the Java compiler where to find the classes you reference in your code and are placed at the very beginning of your source code.

To use the standard JDBC package, which allows you to select, insert, update, and delete data in SQL tables, add the following *imports* to your source code −

```
import java.sql.* ;  // for standard JDBC programs
import java.math.* ; // for BigDecimal and BigInteger support
```

Register JDBC Driver

You must register the driver in your program before you use it. Registering the driver is the process by which the Oracle driver's class file is loaded into the memory, so it can be utilized as an implementation of the JDBC interfaces.

You need to do this registration only once in your program. You can register a driver in one of two ways.

Approach I - Class.forName()

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses Class.forName( ) to register the Oracle driver

```
try {
   Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows −

```
try {

Class.forName("oracle.jdbc.driver.OracleDriver").newInstance()
;
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
catch(IllegalAccessException ex) {
   System.out.println("Error: access problem while loading!");
   System.exit(2);
```

```
catch(InstantiationException ex) {
   System.out.println("Error: unable to instantiate driver!");
   System.exit(3);
}
```

Approach II - DriverManager.registerDriver()

The second approach you can use to register a driver, is to use the static **DriverManager.registerDriver()** method.

You should use the *registerDriver()* method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

The following example uses registerDriver() to register the Oracle driver −

```
try {
   Driver myDriver = new oracle.jdbc.driver.OracleDriver();
   DriverManager.registerDriver( myDriver );
}
catch(ClassNotFoundException ex) {
   System.out.println("Error: unable to load driver class!");
   System.exit(1);
}
```

Database URL Formulation

After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method.

- getConnection(String url)
- getConnection(String url, Properties prop)
- getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database.

Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format

MySQL	com.mysql.jdbc.Driver	**jdbc:mysql:**//hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:@**hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

Using a Database URL with a username and password

- The most commonly used form of getConnection() requires you to pass a database URL, a *username*, and a *password* −
- Assuming you are using Oracle's **thin** driver, you'll specify a ***host:port:databaseName*** value for the database portion of the URL.
- If you have a host at TCP/IP address 192.0.0.1 with a host name of aruns, and your Oracle listener is configured to listen on port 1521, and your database name is Student, then complete database URL would be −

jdbc:oracle:thin:@aruns:1521:Student

1. **Now you have to call getConnection() method with appropriate username and password to get a Connection object as follows −**

String URL = "jdbc:oracle:thin:@aruns:4040:Student";

String USER = "username";

String PASS = "password"

Connection conn = DriverManager.getConnection(URL, USER, PASS);

Using Only a Database URL

2. **A second form of the DriverManager.getConnection( ) method requires only a database URL −**

DriverManager.getConnection(String url);

- However, in this case, the database URL includes the username and password and has the following general form −

jdbc:oracle:driver:username/password@database

So, the above connection can be created as follows −

String URL = "jdbc:oracle:thin:username/password@aruns:1521:EMP";

Connection conn = DriverManager.getConnection(URL);

Using a Database URL and a Properties Object

3. A third form of the DriverManager.getConnection( ) method requires a database URL and a Properties object −

DriverManager.getConnection(String url, Properties info);

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code −

```
import java.util.*;
String URL = "jdbc:oracle:thin:@aruns:1521:Student";
Properties info = new Properties( );
info.put (“user", "username”);
info.put (“password", "password”);
Connection conn = DriverManager.getConnection(URL, info);
```

Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, you could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not.

To close the above opened connection, you should call close() method as follows −

conn.close();

Explicitly closing a connection conserves DBMS resources, which will make your database administrator happy.

Accessing Database from JSP:

The PreparedStatement Objects:

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which starts at 0.

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can input the parameters.

Closing PreparedStatement Object

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first, it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   pstmt.close();
}
```

Example: (Registration Form)

index.html

```
<html>
<head>
</head>
<body>
<form action="store.jsp" method="post">
<table align="center">
<tr>
<th colspan="2" style="padding:20px;"> Registration Page </th>
</tr>
<tr>
<td>Name</td>
<td><input type="text" name="rname"></td>
</tr>
<tr>
<td>Email</td>
```

```
<td><input type="email" name="remail"></td>
</tr>
<tr>
<td>Gender</td>
<td>Male <input type="radio" name="rgender" value="Male">
              Female   <input   type="radio"   name="rgender"
value="Female"></td>
</tr>
<tr>
<td>Enter Password</td>
<td><input type="password" name="renterpass"></td>
</tr>

<tr>
<td>Confirm Password</td>
<td><input type="password" name="rconfirmpass"></td>
</tr>
<tr>
<td></td>
<td style="padding:20px;">
<input type="reset" value="Clean">
<input type="submit" value="Signup">
</td>
</tr>
</table>
</form>
</body>
</html>
```

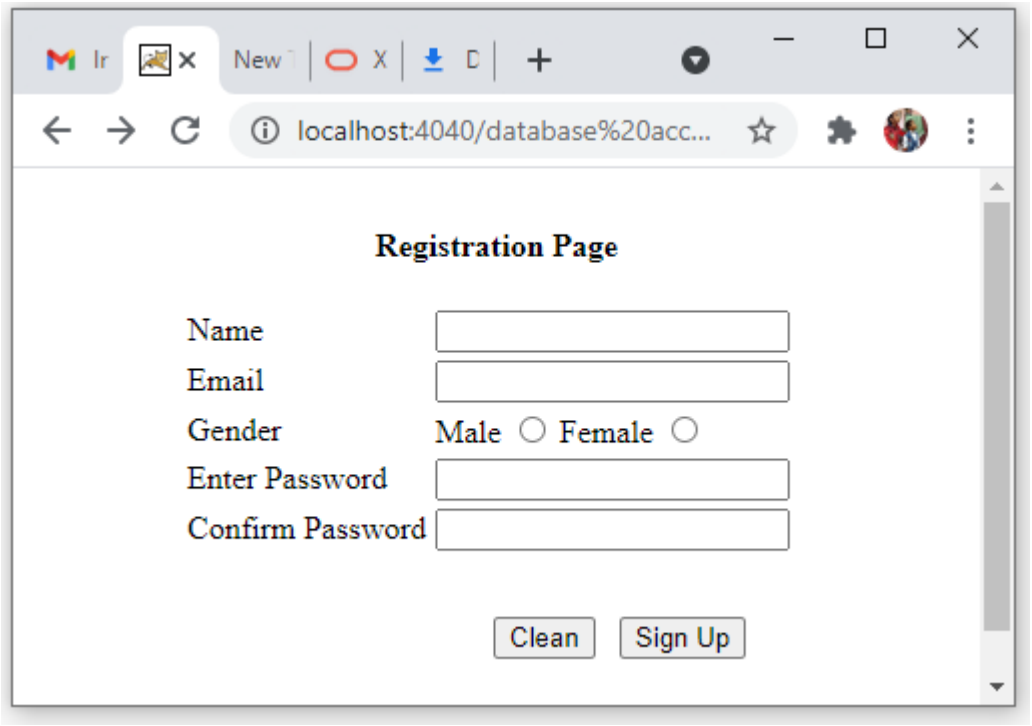
store.jsp

```
<%@ page import="java.sql.*"%>
<%
```

```
String name=request.getParameter("rname");
String email=request.getParameter("remail");
String gender=request.getParameter("rgender");
String pass=request.getParameter("renterpass");
String pass1=request.getParameter("rconfirmpass");
//initialize driver class
if(pass.equals(pass1))
{
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection
conn=DriverManager.getConnection("jdbc:oracle:thin:@localhost:
1521:XE","system","aruns");
PreparedStatement ps=conn.prepareStatement("insert into
registration values(?,?,?,?)");
ps.setString(1,name);
ps.setString(2,email);
ps.setString(3,gender);
ps.setString(4,pass);
int x=ps.executeUpdate();
if(x!=0)
{
out.print("Sign up done successfully");
}
else
{
out.print("Something went wrong");
}
}
catch(Exception ex) {
   System.out.println("Error: unable to load driver class!");
```

```
    }
}
else
{
out.print("Password not matching");
}
%>
```

Output:



ResultSet interface

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

But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

1. Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,

   ResultSet.CONCUR_UPDATABLE);

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.

Example:

retrieve.jsp:

```
<%@ page import="java.sql.*"%>
<%
try
{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection conn
=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521
:XE","system","aruns");
Statement st=con.createStatement();
ResultSet rs=st.executeQuery("select * from registration");
while(rs.next())
{
out.println(rs.getString("name"));
out.println(rs.getString("email"));
out.println(rs.getString("gender"));
out.println(rs.getInt("password"));
}
}
catch(Exception e)
{
out.println(e);
} %>
```