Unit 5 : Interacting with Database

Introduction to ODBC and JDBC

Application1 ←- API -→ Application2
Application 1 <- ODBC → Database
Java Application<- JDBC → Database

ODBC stands for Open DataBase Connectivity.
It is an API (Application Programming Interface) to connect and execute the query with the database. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.
But, ODBC API uses **ODBC driver** which is **written in C language** (i.e. **platform dependent** and **unsecured**). That is why Java has defined its own API (**JDBC API**) that uses **JDBC drivers** (written in **Java language**).

Java Database Connectivity (JDBC)
JDBC (Java Database Connectivity) API is developed by Sun Microsystems specification. **Java Database Connectivity (JDBC)** is an application programming interface (API) that helps Java program to communicate with databases and manipulates their data. The JDBC API provides the methods that can be used to send SQL and PL/SQL statements to almost any relational database.

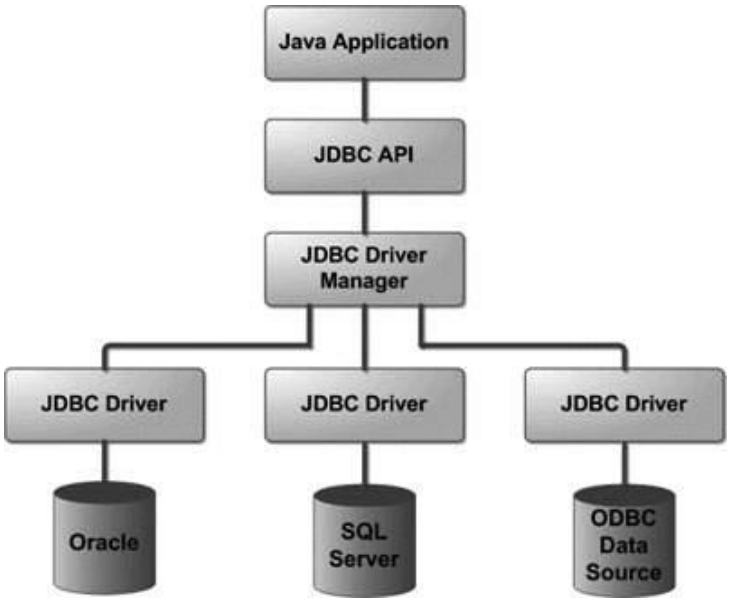
Applications of JDBC
JDBC enables you to create Java applications that handle the following three programming tasks:
- Make a connection to a data source, such as a database.
- Send database queries and update statements.
- Retrieve and process the database results that were returned in response to your query.

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 Manage**r-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 –



Two-tier Architecture

Two-tier Architecture provides **direct communication** between **Java applications** to the **database**. It requires a **JDBC driver** that can help to communicate with the particular database.

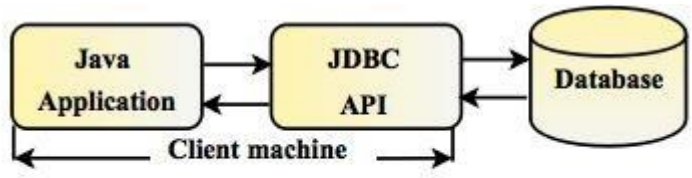


Fig: Two-tier Architecture of JDBC

Three-tier Architecture

In the three-tier model, commands are sent by the HTML browser to middle services i.e. Java application which can send the commands to the particular database. The middle tier has been written in **C or C++** languages. It can also provide better performance.

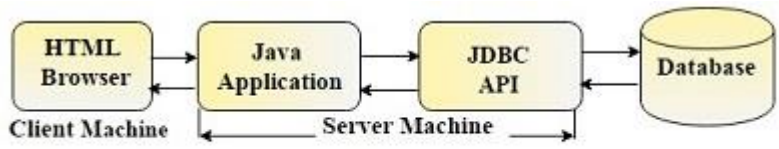


Fig: Three-tier Architecture of JDBC

HOW JDBC WORKS

1. First Java Application establish a **connection** with a data source.
2. The Java application calls **JDBC classes** and **interfaces** to **submit SQL statements** (Queries) to data source.
3. **JDBC driver** connects to **corresponding database** and **retrives the result**.
4. These results are based on SQL statements which are then returned to Java Application.
5. Java Application then uses retrieved information for further processing.

JDBC Drivers

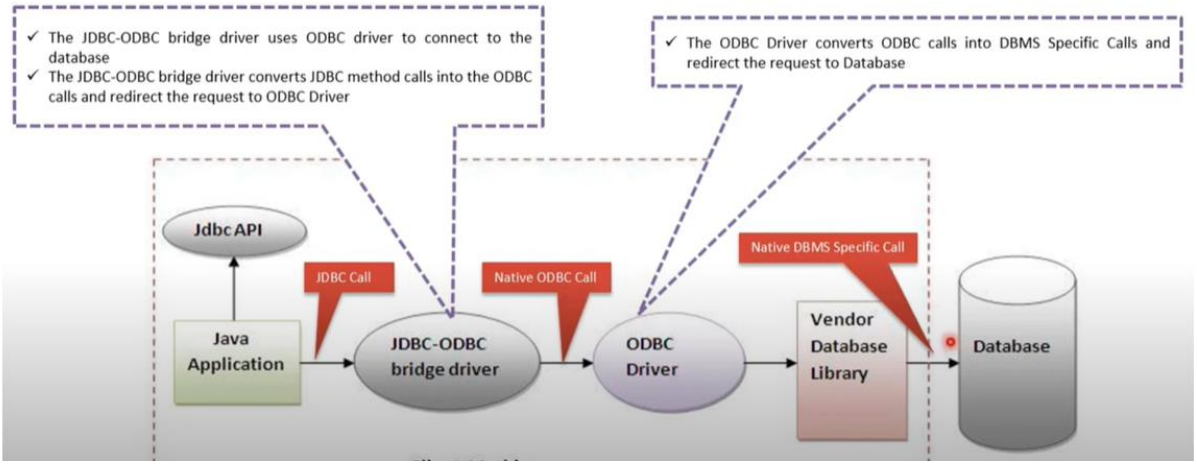
JDBC drivers are client-side adapters (installed on the client machine, not on the server) that **convert requests from Java programs** to a **protocol** that the **DBMS can understand**. There are 4 types of JDBC drivers:

1. Type-1 driver or JDBC-ODBC bridge driver
2. Type-2 driver or Native-API driver
3. Type-3 driver or Network Protocol driver
4. Type-4 driver or Thin driver

Type-1 driver

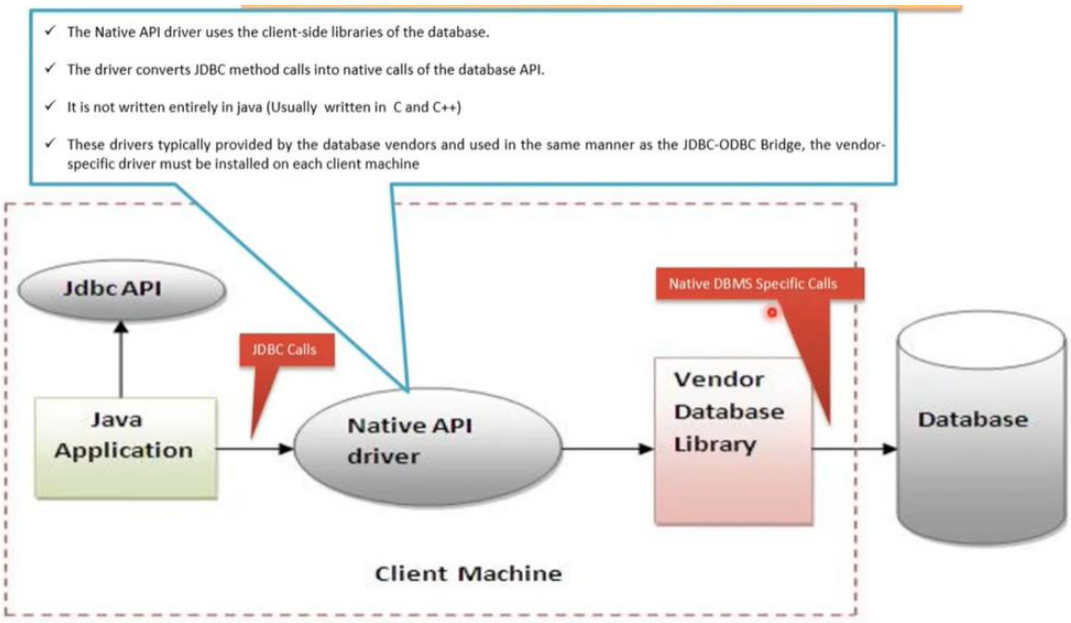
Type-1 driver or JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
The **JDBC-ODBC bridge driver** converts **JDBC method calls** into the **ODBC function calls.**
Type-1 driver is also called **Universal driver** because it can be used to connect to any of the databases.

- As a common driver is used in order to interact with different databases, the data transferred through this driver is not so secured.
- The ODBC bridge driver is needed to be installed in individual client machines.
- Type-1 driver isn't written in java, that's why it isn't a portable driver.
- This driver software is built-in with JDK so no need to install separately.
- **It is a database independent driver.**



Type 2 – JDBC-Native API

In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database.
These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge.
The vendor-specific driver must be installed on each client machine.
If we change the Database, we have to change the native API, as it is specific to a database.
They are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.

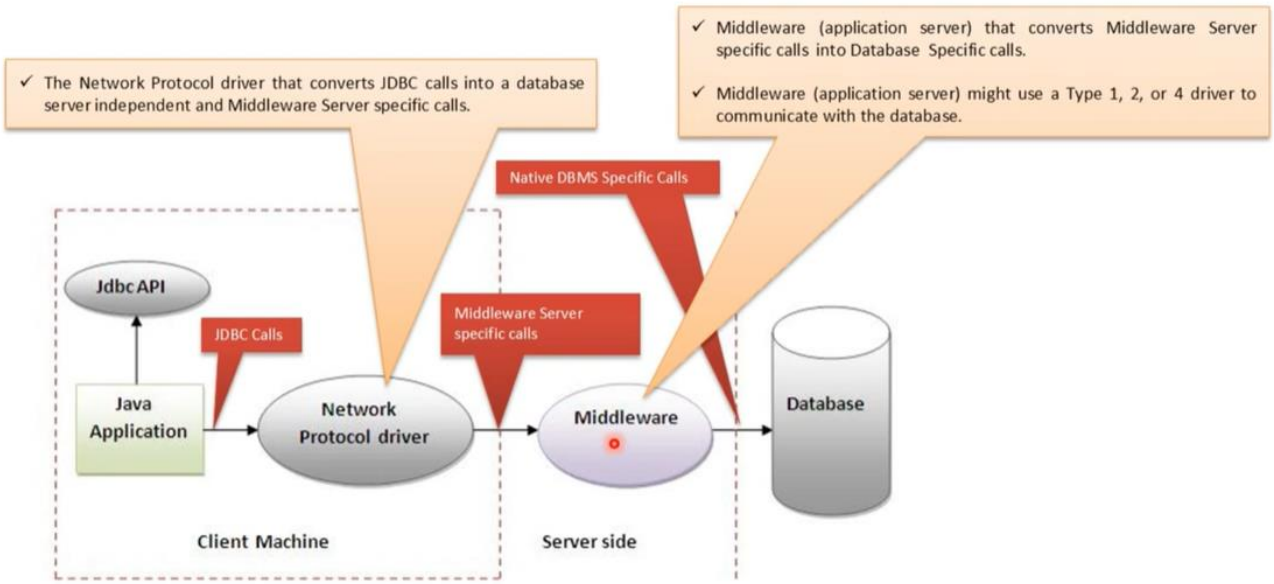


Type 3 – Network Protocol driver

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

Here all the database connectivity drivers are present in a single server, hence no need of individual client-side installation.

- Type-3 drivers are fully written in Java, hence they are portable drivers.
- No client side library is required because of **application server** that can perform many tasks like auditing, load balancing, logging etc.
- Network support is required on client machine.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.



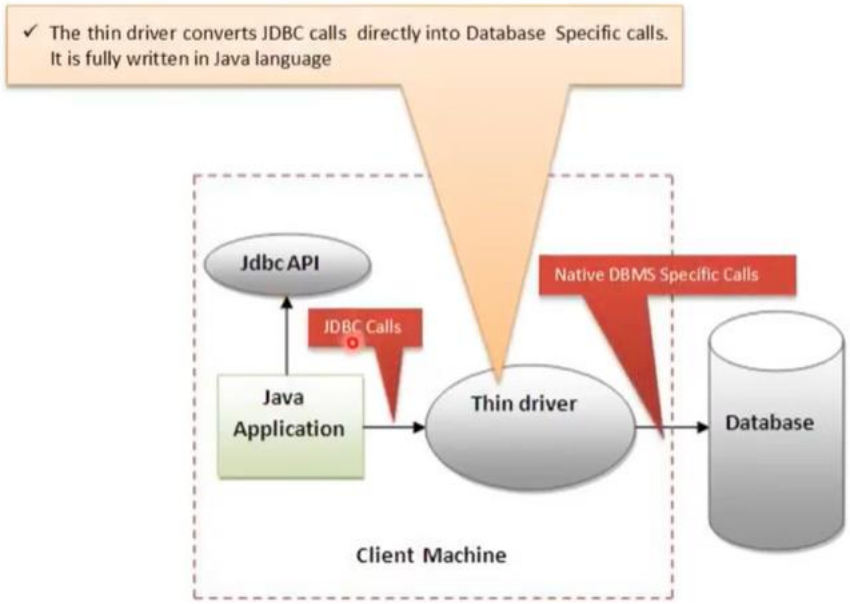
Type-4 driver: **Native protocol driver (Pure Java Driver)**

Type-4 driver is also called native protocol driver.

This driver interact directly with database.

It does not require any **native database library**, that is why it is also known as **Thin Driver**.

- Does not require any native library and Middleware server, so no client-side or server-side installation.
- It is fully written in Java language, hence they are portable drivers.



Java Database Connectivity

There are 5 steps to connect any java application with the database using JDBC.
These steps are as follows:

- Register the Driver class
- Create connection
- Create statement
- Execute queries
- Close connection

1) Register the driver class

We first need to **load** the **driver** or **register** it before using it in the program.
The **forName()** method of **Class** class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

```
public static void forName(String className)throws ClassNotFoundException
```

Example to register the OracleDriver class

Here, Java program is loading oracle driver to esteblish database connection.

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

2) Create the connection object

The **getConnection()** method of **DriverManager class** is used to establish connection with the database.

Syntax of getConnection() method

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

- user: username from which sql command prompt can be accessed.
- password: password from which sql command prompt can be accessed.
- con: reference to Connection interface.
- url : Uniform Resource Locator. We can create it as follows:
- String url = " jdbc:oracle:thin:@localhost:1521:xe"

Where **oracle** is the **database**, **thin** is the **driver**, **@localhost** is the **IP Address** where the database is stored, **1521** is the **port number**, and **xe is the service provider**. All three parameters are of String type and the programmer should declare them before calling the function

Example to establish connection with the Oracle database

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

3) Create the Statement object

Once you establish a connection, you can interact with the database.
The JDBC **Statement**, **CallableStatement**, and **PreparedStatement interfaces** define the **methods that allow us to send the SQL commands and receive data from the database.**
The **createStatement()** method of **Connection** interface is used to create statement.
The object of **Statement** is responsible to **execute queries** with the database.

Syntax of createStatement() method

```
public Statement createStatement()throws SQLException
```

Example to create the statement object

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

Here, con is a reference to the Connection interface that we used in the previous step.

4) Execute the query

The most crucial part is executing the query. Here, Query is an SQL Query.
Now, as we know that we can have multiple types of queries. Some of them are as follows:

- The query for **updating** or **inserting tables** in a **database**.
- The query for **retrieving data** from the **database**.

The **executeQuery()** method of **Statement interface** is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

```
public ResultSet executeQuery(String sql)throws SQLException
```

Example to execute query

```
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next()){
System.out.println(rs.getInt(1)+" "+rs.getString(2));
}
```

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

```
public void close()throws SQLException
```

Example to close connection

```
con.close();
```

It avoids explicit connection closing step.

DriverManager class

The DriverManager class is the component of JDBC API and also a member of the *java.sql* package. The DriverManager class acts as an interface between users and drivers.

It keeps track of the drivers that are available and handles establishing a connection between a database and the appropriate driver.

It contains all the appropriate methods to register and deregister the database driver class and **to create a connection between a Java application and the database.**

The DriverManager class maintains a list of Driver classes that have registered themselves by calling the method **DriverManager.registerDriver**().

Note that before interacting with a Database, it is a mandatory process to register the driver; otherwise, an exception is thrown.

Register JDBC Driver

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!");
}
```

Approach II - DriverManager.registerDriver()

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

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!");
}
```

Methods of the DriverManager Class

Method	Description
1) void registerDriver(Driver driver):	is used to register the given driver with DriverManager. No action is performed by the method when the given driver is already registered.
2) void deregisterDriver(Driver driver):	is used to deregister the given driver (drop the driver from the list) with DriverManager. If the given driver has been removed from the list, then no action is performed by the method.
3) public static Connection getConnection(String url) throws SQLException:	is used to establish the connection with the specified url. The SQLException is thrown when the corresponding Driver class of the given database is not registered with the DriverManager.
4) public static Connection getConnection(String url,String userName,String password) throws SQLException:	is used to establish the connection with the specified url, username, and password. The SQLException is thrown when the corresponding Driver class of the given database is not registered with the DriverManager.
5) public static Driver getDriver(String url)	Those drivers that understand the mentioned URL (present in the parameter of the method) are returned by this method.
7) pubic static void setLoginTimeout(int sec)	The method provides the time in seconds. sec mentioned in the parameter is the maximum time that a driver is allowed to wait in order to establish a connection with the database.

Connection interface

A Connection is a session between a Java application and a database. It helps to establish a connection with the database.

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 commit(): saves the changes made since the previous commit/rollback is permanent.

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

4) 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.

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.

ResultSet interface

The object of ResultSet maintains a cursor pointing to a row of a table.

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 int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
7) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
8) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
9) public String getString(String columnName):	is used to return the data of specified column name of the current row as String.

Steps to run jdbc Programs

To connect java application with the Oracle database ojdbc14.jar file is required to be loaded.

Step 1 : How to set the permanent classpath:

Go to environment variable then click on new tab.
In variable name write **classpath** and in variable value paste the path to ojdbc14.jar by appending ojdbc14.jar;
as C:\oraclexe\app\oracle\product\10.2.0\server\jdbc\lib\ojdbc14.jar;

Step 2 : - First open command prompt and type sqlplus

Enter Username : - System
Enter Password: - 123

Step 3 : - Create a Table

Before establishing connection, let's first create a table in oracle database. Following is the SQL query to create a table.

```
CREATE TABLE table_name
(
   column1 datatype,
   column2 datatype,
   column3 datatype,
  ....
);
```

Example:

```
SQL>  CREATE TABLE emp (
  id int,
   fname varchar(255),
   lname varchar(255),
  );
```

Step 4: Insert some values in Table

Syntax:

```
INSERT INTO table_name
VALUES (value1, value2, value3, ...);
```

```
SQL> insert into emp values (1,'akshay','somwanshi');
```

Step 5: Compile and Run our Program

Example

```
import java.sql.*;
public class OracleCon{
public static void main(String args[])
{
try
{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");

//step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","123");

//step3 create the statement object
Statement stmt=con.createStatement();

//step4 execute query
ResultSet rs=stmt.executeQuery("select * from emp");
while(rs.next())
System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));

//step5 close the connection object
con.close();

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

Example : Create Table

```
import java.sql.*;
public class CreateTable{
public static void main(String args[])
{
try
{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");

//step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","123");

//step3 create the statement object
Statement stmt=con.createStatement();

//step4 execute query
stmt.execute("create table student (rollno number(5),name varchar(10),marks number(10))");
System.out.println("Table Created Successfully!!!");

//step5 close the connection object
con.close();

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

Example : Insert Row into Table

```
import java.sql.*;
public class InsertRow{
public static void main(String args[])
{
try
{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");

//step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","123");

//step3 create the statement object
Statement stmt=con.createStatement();

//step4 execute query
stmt.executeUpdate("insert into student values (1,'Akshay',98)");
stmt.executeUpdate("insert into student values (2,'Amol',98)");
stmt.executeUpdate("insert into student values (3,'Shree',98)");

System.out.println(" Rows Inserted Successfully");

//step5 close the connection object
con.close();

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

Example : UpdateRow from Table

```
import java.sql.*;
public class UpdateRow{
public static void main(String args[])
{
try
{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");

//step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","123");

//step3 create the statement object
Statement stmt=con.createStatement();

//step4 execute query
int x=stmt.executeUpdate("update student set marks=10,name='Akshay' where rollno=1");
System.out.println(" No of Rows Updated into table:"+x);

//step5 close the connection object
con.close();

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

Example : Delete Row from Table

```
import java.sql.*;
public class DeleteRow{
public static void main(String args[])
{
try
{
//step1 load the driver class
Class.forName("oracle.jdbc.driver.OracleDriver");

//step2 create  the connection object
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","123");

//step3 create the statement object
Statement stmt=con.createStatement();

//step4 execute query
int x=stmt.executeUpdate("delete from student where name='Akshay'");
System.out.println(" No of Rows Updated into table:"+x);

//step5 close the connection object
con.close();

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

.

PreparedStatement

A **PreparedStatement** is a pre-compiled SQL statement. Prepared Statement objects have some useful additional features than Statement objects. Instead of hard coding queries, PreparedStatement object provides a feature to execute a parameterized query.

Advantages of PreparedStatement

- When PreparedStatement is created, the SQL query is passed as a parameter. This Prepared Statement contains a pre-compiled SQL query, so when the PreparedStatement is executed, DBMS can just run the query instead of first compiling it.
- We can use the same PreparedStatement and supply with different parameters at the time of execution.

Steps to use PreparedStatement

1. Create Connection to Database

Connection myCon = DriverManager.getConnection(path,username,password)

2. Prepare Statement

Instead of hardcoding queries like,
select * from students where age>10 and name ='Arrow'

Set parameter placeholders(use question mark for placeholders) like,
select * from students where age> ? and name = ?

PreparedStatement myStmt = myCon.prepareStatement(select * from students where age> ? and name = ?);

3. Set parameter values for type and position

myStmt.setInt(1,20);
myStmt.setString(2,"Amol");

4. Execute the Query

ResultSet myRs= myStmt.executeQuery();

Methods of PreparedStatement:

- **setInt(int, int):** This method can be used to set integer value at the given parameter index.
- **setString(int, string):** This method can be used to set string value at the given parameter index.
- **setFloat(int, float):** This method can be used to set float value at the given parameter index.
- **setDouble(int, double):** This method can be used to set a double value at the given parameter index.
- **executeUpdate():** This method can be used to create, drop, insert, update, delete etc. It returns int type.
- **executeQuery():** It returns an instance of ResultSet when a select query is executed.

Example of Prepared Statement:-

Insert Values into table

```
import java.util.*;
import java.sql.*;
class Program4
{
	public static void main(String args[])
	{
		try
		{
		Class.forName("oracle.jdbc.driver.OracleDriver");
	Connection con =
	DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","123");

		Prepared**d**Statement pstmt=con.prepareStatement("insert into student values(?,?,?)");
		Scanner sc=new Scanner(System.in);
		System.out.println("Enter Roll No of a student ");
		int roll = sc.nextInt();
		System.out.println("Enter Student Name:");
		String name =sc.next();
		System.out.println("Enter Student Marks:");
		int marks=sc.nextInt();
		pstmt.setInt(1,roll);
		pstmt.setString(2,name);
		pstmt.setInt(3,marks);
		int x=pstmt.executeUpdate();
		System.out.println(" No of Rows Inserted into table:"+x);
		con.close();
		}
		catch (Exception e)
		{
			System.out.println(e);
		}
	}
}
```

Deleting row from table:

```
import java.util.*;
import java.sql.*;
class Program4
{
	public static void main(String args[])
	{
		try
		{
	Class.forName("oracle.jdbc.driver.OracleDriver");
	Connection con =
	DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","123");
	PreparedStatement pstmt=con.prepareStatement("delete from student where rollno=?");
	Scanner sc=new Scanner(System.in);
	System.out.println("Enter Roll No for Deletion:");
	int roll=sc.nextInt();
	pstmt.setInt(1,roll);
	int x=pstmt.executeUpdate();
	System.out.println(" No of Rows Deleted into table:"+x);
		con.close();
		}

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

Inserting Rows into Table untill User types ‘n’

```
import java.util.*;
import java.sql.*;
class Program6
{
	public static void main(String args[])
	{
		try
		{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","123");

PreparedStatement pstmt=con.prepareStatement("insert into students values(?,?,?)");
		Scanner sc=new Scanner(System.in);
		do
		{
		System.out.println("Enter Roll No of a student ");
		int roll = sc.nextInt();
		System.out.println("Enter Student Name:");
		String name =sc.next();
		System.out.println("Enter Student Marks:");
		int marks=sc.nextInt();
		pstmt.setInt(1,roll);
		pstmt.setString(2,name);
		pstmt.setInt(3,marks);
		int x=pstmt.executeUpdate();
		System.out.println(" No of Rows Inserted into table:"+x);

		System.out.println("Do you want to continue: y/n");
		String s=sc.next();
		if(s.startsWith("n"))
		{
		break;
		}
		}
		while(true);
		con.close();
		}
		catch (Exception e)
		{
			System.out.println(e);
		}
	}
}
```

Java ResultSetMetaData Interface

The metadata means data about data i.e. we can get further information from the data.
If you have to get metadata of a table like total number of column, column name, column type etc. , ResultSetMetaData interface is useful because it provides methods to get metadata from the ResultSet object.

Commonly used methods of ResultSetMetaData interface

Method	Description
public int getColumnCount()throws SQLException	it returns the total number of columns in the ResultSet object.
public String getColumnName(int index)throws SQLException	it returns the column name of the specified column index.
public String getColumnTypeName(int index)throws SQLException	it returns the column type name for the specified index.

Example:

```
import java.sql.*;
class Program5{
public static void main(String args[]){
try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection(
"jdbc:oracle:thin:@localhost:1521:xe","system","123");

PreparedStatement ps=con.prepareStatement("select * from students");
ResultSet rs=ps.executeQuery();
ResultSetMetaData rsmd=rs.getMetaData();

System.out.println("Total columns: "+rsmd.getColumnCount());
System.out.println("Column Name of 1st column: "+rsmd.getColumnName(1));
System.out.println("Column Type Name of 1st column: "+rsmd.getColumnTypeName(1));
System.out.println("Column Name of 2nd column: "+rsmd.getColumnName(2));
System.out.println("Column Type Name of 2nd column: "+rsmd.getColumnTypeName(2));
System.out.println("Column Name of 3rd column: "+rsmd.getColumnName(3));
System.out.println("Column Type Name of 3rd column: "+rsmd.getColumnTypeName(3));
con.close();
}catch(Exception e){ System.out.println(e);}
}
}
```

Output:

```
Total columns: 3
Column Name of 1st column: ROLL
```

Column Type Name of 1st column: NUMBER
Column Name of 1st column: NAME
Column Type Name of 1st column: VARCHAR2
Column Name of 1st column: MARKS
Column Type Name of 1st column: NUMBER

CallableStatement

The **CallableStatement** interface provides methods to execute the stored procedures.

Creating a CallableStatement

You can create an object of the **CallableStatement** (interface) using the **prepareCall()** method of the **Connection** interface. This method accepts a string variable representing a query to call the stored procedure and returns a **CallableStatement** object.

A Callable statement can have input parameters, output parameters or both. To pass input parameters to the procedure call you can use place holder and set values to these using the setter methods (setInt(), setString(), setFloat()) provided by the CallableStatement interface.

Suppose you have a procedure name myProcedure in the database you can prepare a callable statement as:

```
//Preparing a CallableStatement
CallableStatement cstmt = con.prepareCall("{call myProcedure(?, ?, ?)}");
```

Setting values to the input parameters

You can set values to the input parameters of the procedure call using the setter methods.

These accepts two arguments, one is an integer value representing the placement index of the input parameter and, the other is a int or, String or, float etc... representing the value you need to pass as input parameter to the procedure.

Note: Instead of index you can also pass the name of the parameter in String format.

```
cstmt.setString(1, "Raghav");
cstmt.setInt(2, 3000);
cstmt.setString(3, "Hyderabad");
```

Executing the Callable Statement

Once you have created the CallableStatement object you can execute it using one of the **execute()** method.

```
cstmt.execute();
```

Full example to call the stored procedure using JDBC

To call the stored procedure, you need to create it in the database. Here, we are assuming that stored procedure looks like this.

```
create or replace procedure "MyProcedure" (rollno in number,  name in varchar, marks in number)  is
begin
insert into stud values(rollno,name,marks);
end;
/
```

The table structure is given below:

```
create table stud (id number(10), name varchar2(200), marks number);
```

Example

In this example, we are going to call the stored procedure MyProcedure that receives rollno, name and marks as the parameter and inserts it into the table students. Note that you need to create the students table as well to run this application.

```
import java.sql.*;
public class Proc {
public static void main(String[] args) throws Exception{

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

CallableStatement stmt=con.prepareCall("{call MyProcedure(?,?,?)}");
stmt.setInt(1,15);
stmt.setString(2,"Amit");
stmt.setInt(3,87);
stmt.execute();
System.out.println("success");
}
}
```