**JDBC Select on executeQuery() :**

* We can call the executeQuery() method on statement object to select the data from the database.
* executeQuery() method returns **ResultSet** object.
* The ResultSet object contains rows of table.
* The ResultSet object maintains the cursor, and it is initially positioned at before of the first row.
* To read the data of row by row, we need to move the curson to the next row by calling **next()** method available in ResultSet.
* The next() method returns a boolean value, either true or false.
* When there is a row available in ResultSet object then next() returns true if no more rows then it returns false.
* By default ResultSet object is non-scrollable. That means, a cursor of the ResultSet object can be moved in forward direction only.
* If require, we can convert a ResultSet object into Scrollable type. If the ResultSet object is scrollable then the cursor will be moved in forward and backward direction.
* One statement object can create only one ResultSet object at a time. If another ResultSet object is created by same statement object, then it automatically closes the previous ResultSet object.

ResultSet rs1 = stmt.executeQuery("select \* from emp);

ResultSet rs2 = stmt.executeQuery("select \* from dept);

Here rs1 is closed automatically, when rs2 is created.

**JDBC Select Program Example :**

Jdbc\_Select\_Example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

public class Jdbc\_Select\_Example {

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

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

Connection con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/onlinetutorialspoint", "root", "123456");

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));

}

rs.close();

stmt.close();

con.close();

}

}

To do the insert operation in JDBC, the api given us executeUpdate(String qry) and execute(String qry) methods. We can apply the both executeUpdate() aand execute() methods on statement object.

### execueteUpdate() Example :

executepdate() method returns the integer value. The value represents that the number of rows effected in the database.

int rowsEffected = stmt.executepUpdate("insert command");

### execuete() Example :

execute() method returns boolean value. As we already discussed in the JDBC select example, we can use the execute() method for both select and non-select operations. If the resultant object contains ResultSet then the execute() method returns the true, it returns false if it is an update count or no records found.

boolean isResultSet = stmt.executep("insert command");

## JDBC Insert Program Example :

Jdbc\_InsertionOperation\_Example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.Scanner;

public class Jdbc\_InsertionOperation\_Example {

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

Connection connection = null;

Statement statement = null;

Scanner scanner = null;

try {

int studentNo = 0;

String studentName = null;

String studentAddress = null;

int studentAge = 0;

scanner = new Scanner(System.in);

if (scanner != null) {

System.out.println("Enter Student No");

studentNo = scanner.nextInt();

System.out.println("Enter Student Name");

studentName = scanner.next();

System.out.println("Enter Student Address");

studentAddress = scanner.next();

System.out.println("Enter Student Age");

studentAge = scanner.nextInt();

}

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

connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/onlinetutorialspoint", "root", "systemuser23!");

if (connection != null) {

statement = connection.createStatement();

String insertQuery = "insert into student values('" + studentNo + "','" + studentName +"','"+ studentAddress + "','"+studentAge+"')";

int result = statement.executeUpdate(insertQuery);

if (result == 0) {

System.out.println("Record Inserted Failed");

} else {

System.out.println(result+ " Record(s) Inserted ");

}

}

} catch (ClassNotFoundException cnfe) {

cnfe.printStackTrace();

} catch (SQLException sqe) {

sqe.printStackTrace();

} catch (Exception e) {

e.printStackTrace();

} finally {

try {

if (statement != null) {

statement.close();

}

} catch (SQLException sqe) {

sqe.printStackTrace();

}

try {

if (connection != null) {

connection.close();

}

} catch (SQLException sqe) {

sqe.printStackTrace();

}

}

}

}

By using the any one of those two, we can do the JDBC update program in Java. Here is the difference between the Jdbc executeUpdate() and execute().

### execueteUpdate() Example :

executepdate() method returns the integer value. The value represents that the number of rows effected in the database.

int rowsEffected = stmt.executepUpdate("update command");

### execuete() Example :

execute() method returns boolean value. As we already discussed in the JDBC select example, we can use the execute() method for both select and non-select operations. If the resultant object contains ResultSet then the execute() method returns the true, it returns false if it is an update count or no records found.

boolean isResultSet = stmt.executep("update command");

Here is the simple example Jdbc Update program.

## JDBC Update Program Example :

Jdbc\_Update\_Example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;

public class Jdbc\_Update\_Example {

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

Connection connection = null;

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

connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/onlinetutorialspoint", "root",

"123456");

Statement stmt = connection.createStatement();

int count = stmt

.executeUpdate("update student set sname='rajesh' where sid=102");

System.out.println(count + " Record(s) updated.");

}

}

By using the any one of those two, we can do the jdbc delete program in java. Here is the difference between the Jdbc executeUpdate() and execute().

### execueteUpdate() Example :

executepdate() method returns the integer value. The value represents that the number of rows effected in the database.

int rowsEffected = stmt.executepUpdate("delete command");

### execuete() Example :

execute() method returns boolean value. As we already discussed in the JDBC select example, we can use the execute() method for both select and non-select (insert, update, delete) operations. If the resultant object contains ResultSet then the execute() method returns the true, it returns false if it is an update count or no records found.

boolean isResultSet = stmt.executep("update command");

## JDBC Delete Program Example :

Here is the complete example for JDBC Delete.

Jdbc\_DeleteOperation\_example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;

import java.util.Scanner;

public class Jdbc\_DeleteOperation\_Example {

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

Scanner scanner = new Scanner(System.in);

System.out.println("Enter Student Number to delete the record");

int studentNo = scanner.nextInt();

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

Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/onlinetutorialspoint", "root", "systemuser23!");

con.setAutoCommit(false);

Statement stmt = con.createStatement();

String query = "delete from student where sid='" + studentNo + "'";

int result = stmt.executeUpdate(query);

con.commit();

if (result == 0) {

System.out.println("record not found to delete");

} else {

System.out.println(result+" no.of record(s) found and deleted");

}

stmt.close();

con.close();

}

}

n this tutorial, I am going to show how to prepare the **JDBC connection with properties** using a [*Java properties file*](https://www.onlinetutorialspoint.com/java/properties-class-example-in-java.html).

When we write any JDBC application, we have to specify the specific details regarding driver name, URL, database user and password etc.. If you see in the previous basic [*JDBC Select Program Example*](https://www.onlinetutorialspoint.com/jdbc/jdbc-select-program-example.html)we have created the connection with hard-coded values like below:

Connection con = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/onlinetutorialspoint", "root", "123456");

For any reason If we want to change these details, we must have to change the Java program accordingly, and we have to recompile it. It leads to higher maintenance activity.

To avoid this complications typically we should have to read these details from a resoure bubdle file or a [*properties file*](https://www.onlinetutorialspoint.com/java/properties-class-example-in-java.html). In this example, we are going to show how to read this connection informations from a properties file.

**Learn more**

### JDBC Connection with Properties File :

A resource bundle file or properties file is one which contains the data in the form of (key, value) pair.

Dname=sun.jdbc.odbc.JdbcOdbcDriver

URL=jdbc:mysql://localhost:3306/onlinetutorialspoint

Uname=root

password=123456

### Reading the data from Properties File:

To read the data from the resource bundle file, we have to open the resource bundle file in a reading mode with the help of the FileInputStream class.

FileInputStream fis=new FileInputStream (“connection.prop”);

Since files do not support to read data separately in the form of (key, value), hence, it is recommended to get the data of the file we must create an object of a predefined class called *[java.util.Properties](https://www.onlinetutorialspoint.com/java/properties-class-example-in-java.html)*.

### Complete Example :

connection.prop

Dname=sun.jdbc.odbc.JdbcOdbcDriver

URL=jdbc:mysql://localhost:3306/onlinetutorialspoint

Uname=root

password=123456

**JDBC\_ReadDataFromProps.java**

JDBC\_ReadDataFromProps.java

package com.onlinetutorialspoint.jdbc;

import java.io.FileInputStream;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

import java.util.Properties;

public class JDBC\_ReadDataFromProps {

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

FileInputStream fis=new FileInputStream("connection.prop");

Properties p=new Properties ();

p.load (fis);

String dname= (String) p.get ("Dname");

String url= (String) p.get ("URL");

String username= (String) p.get ("Uname");

String password= (String) p.get ("password");

Class.forName(dname);

Connection con = DriverManager.getConnection(

url, username, password);

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));

}

rs.close();

stmt.close();

con.close();

}

}

**JDBC PreparedStatement :**

Here are the some important points about **JDBC PreparedStatement** object.

* In an application if we want to run a same query for multiple times with different parameters (values) then we can go with PreparedStatement.
* If we use the normal Statement, it compiles the sql command for each time before going to execute.
* If the same command is compiled again and again then the performance of an application is going to be decreased.
* In case of PreparedStatement, first a command will be sent to database for compilation, then the compiled code will be stored in PreparedStatement object.
* Now the code can be executed for any number of times by without recompiling the command again and again.
* Another important limitation of the Statement object is, it can only transfer the data of type text format only. Where as PreparedStatement can transfer binary format also.
* We can obtain the PreparedStatement object by calling the prepareStatement() method on connection object.

**JDBC PreparedStatement Syntax :**

PreparedStatement pstmt = connection.prepareStatement("insert into student values(?,?,?)");

In the above example, at first the insert command will be sent to database and compiles the command and then the compiled code will assigned into PreparedStatement object.

In the syntax, we used **“?”** symbol in place of values in insert command. **“?”** symbol is called as index parameter or replace operator or place resolution operator.

We can use only **“?”** symbols in the place of values. No other symbols are allowed here.

* “?” symbol is not allowed for DDL commands.
* “?” symbol is not allowed to replace table names and column names.

**Example :**

select ?,? from emp; illegal

select \* from emp where ? = ?; --illegal

select \* from emp where empId = : ?; --legal

update ? set sal = ? where empId = ?; --illegal

Before we run the compiled code (stored in the PreparedStatement object), we need to set the values to the compiled code by calling **setXxx()** methods.

Here is the complete example for JDBC PreparedStatement.

**JDBC PreparedStatement Example :**

Jdbc\_PreparedStatement\_Example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.sql.SQLException;

import java.util.Scanner;

public class Jdbc\_PreparedStatement\_Example {

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

Connection connection = null;

PreparedStatement pstatement = null;

Scanner scanner = null;

try {

scanner = new Scanner(System.in);

int n = 0;

System.out.println("Enter no. of Students to insert");

n = scanner.nextInt();

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

connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/onlinetutorialspoint", "root", "123456");

if (connection != null)

pstatement = connection.prepareStatement("insert into student values(?,?,?,?)");

if (pstatement != null) {

for (int i = 1; i <= n; i++) {

System.out.println("Enter " + i + " Student Details");

System.out.println("Enter Student No : ");

int studentNo = scanner.nextInt();

System.out.println("Enter Student Name : ");

String studentName = scanner.next();

System.out.println("Enter Student Address : ");

String studentAddress = scanner.next();

System.out.println("Enter Student Age : ");

int studentAge = scanner.nextInt();

pstatement.setInt(1, studentNo);

pstatement.setString(2, studentName);

pstatement.setString(3, studentAddress);

pstatement.setInt(4, studentAge);

int result = pstatement.executeUpdate();

if (result == 0) {

System.out.println("Student details: are not inserted");

} else {

System.out.println(result + " records(s) are inserted");

}

}

}

} catch (ClassNotFoundException cnfe) {

cnfe.printStackTrace();

} catch (SQLException se) {

se.printStackTrace();

} catch (Exception ex) {

ex.printStackTrace();

} finally {

try {

if (pstatement != null) {

pstatement.close();

}

} catch (SQLException se) {

se.printStackTrace();

}

try {

if (connection != null) {

connection.close();

}

} catch (SQLException se) {

se.printStackTrace();

}

}

}

}

**CallableStatement in JDBC :**

* CallableStatement in JDBC is a sub interface of PreparedStatement.
* CallableStatement in JDBC has all the benefits of PreparedStatement and also it has one more additional feature, that is we can **call the procedures or functions** of a database.
* CallableStatement is only for calling a procedure or a function of a database. But it is not for creating a procedure or function.
* CallableStatement has two syntaxes, one is for executing commands and another is for calling the procedure or function.
* We can get the CallableStatement object by calling the prepareCall() method on connection object.

**CallableStatement for Executing command :**

CallableStatement cstmt = connection.prepareCall("sql command");

**CallableStatement for calling procedures :**

CallableStatement cstmt = connection.prepareCall("{call procedure(args)}");

**CallableStatement for calling functions :**

CallableStatement cstmt = connection.prepareCall("{?=call function(args)}");

**CallableStatement in JDBC Procedure Example :**

**Creating Procedure in MySql :**

mysql> delimiter $

mysql> create procedure square(IN a int, OUT b int)

-> begin

-> set b=a\*a;

-> end;

-> $

**Example:**

JDBC\_Procedures\_Example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.CallableStatement;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Types;

public class JDBC\_Procedures\_Example {

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

Connection connection = null;

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

connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/onlinetutorialspoint", "root",

"123456");

CallableStatement cStmt = connection.prepareCall("{call square(?,?)}");

cStmt.setInt(1, 20);

cStmt.registerOutParameter(2, Types.INTEGER);

cStmt.execute();

System.out.println("The Square is : " + cStmt.getInt(2));

cStmt.close();

}

}

In the above code, we register the out parameter to CallableStatement by using the registerOutParameter(). It is mandatory because if we do not register the **“out”** parameter, then by default CallableStatement will take all the parameters as **“in”** parameters. So to inform CallableStatement that it is an out parameter we need to  register it.

While registering the out parameter, we need to tell the CallableStatement about the parameter type also. CallableStatement can understand only JDBC Types, but not java or database types. In JDBC [*Types*](https://docs.oracle.com/javase/6/docs/api/java/sql/Types.html) class contains all the JDBC data types. Types is a class in java.sql package.

**Output :**

Terminal

The Square is : 400

**CallableStatement in JDBC Function Example :**

**Creating Function in MySQL :**

mysql> DELIMITER $

mysql> CREATE FUNCTION mul(a int, b int) RETURNS INT

-> BEGIN

-> DECLARE c INT;

-> SET c = a\*b;

-> RETURN c;

-> END;

-> $

**Example:**

JDBC\_Function\_Example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.CallableStatement;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Types;

class JDBC\_Function\_Example {

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

Connection connection = null;

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

connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/onlinetutorialspoint", "root",

"123456");

CallableStatement cStmt = connection.prepareCall("{?=call mul(?,?)}");

cStmt.registerOutParameter(1, Types.INTEGER);

cStmt.setInt(2, 20);

cStmt.setInt(3, 60);

cStmt.execute();

System.out.println("The Multiplication is : " + cStmt.getInt(1));

cStmt.close();

}

}

**Output:**

Terminal

The Multiplication is : 1200

Here is an example for how to **read an image in JDBC**. In the previous example, we insert an image using JDBC. Now in this example, we are going to read that image from the database. If you are not coming through the previous example, it is recommended to read once [*insert an image using JDBC*](https://www.onlinetutorialspoint.com/jdbc/insert-an-image-using-jdbc-in-mysql-db-example.html).

As we already discussed in the previous tutorial, reading an image is nothing but reading binary data. For reading operations in JDBC, we can use the executeQuery() method on a statement.

executeQuery() method returns the ResultSet object. It contains the data rows. Usually, we can read the data from ResultSet using getXXX() methods.

But here, in our example, we are not reading ordinary data. We are going to retrieve an image i.e binary data. To read the binary objects, ResultSet given us **getBinaryStream(String col)** method.

To call the getBinaryStream() method, we need to pass the column name of BOLB type in database, in our case it “photo”.

InputStream is = rs.getBinaryStream("photo");

Here is the complete example for read an image in jdbc.

## Read an Image in JDBC Example :

JDBC\_Getting\_LargeObjects\_Example.java

public class JDBC\_Getting\_LargeObjects\_Example {

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

Connection connection = null;

Statement stmt = null;

try {

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

connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/jdbc", "root", "123456");

stmt = connection.createStatement();

ResultSet rs = stmt.executeQuery("select \* from employee");

if (rs.next()) {

InputStream is = rs.getBinaryStream("photo");

FileOutputStream fos = new FileOutputStream(

"C:/Users/cgoka/Desktop/sample\_NEW.jpg");

int bytesRead = 0;

byte[] buffer = new byte[4096];

while ((bytesRead = is.read(buffer)) != -1) {

fos.write(buffer, 0, bytesRead);

}

is.close();

fos.close();

}

connection.close();

System.out.println("Image created");

} catch (Exception e) {

e.printStackTrace();

}

}

}

In this tutorial, we are going to learn about how to **insert an image using JDBC** in MySQL database. As we already discussed in the previous example on PreparedStatement in JDBC, one of the main advantages of PreparedStatement is, it can handle the large objects (images, videos and etc..).

For this example, we are going to use MySql database for inserting an image. In Mysql, if we want to store the binary data, we have to define the column with **BLOB** (Binary Large Object) type.

**Insert an Image using JDBC :**

Some important points about to handle the large objects in JDBC.

* If want to insert an image using JDBC into database, or read an image from database then we need to use PreparedStatement of JDBC
* In database, the image will not be stored directly. The bytes of an image (binary data) will be stored.
* To store the image in data base, we should declare the column type as [*BLOB*](https://dev.mysql.com/doc/refman/5.7/en/blob.html).
* BLOB type of column can store the data up-to a maximum of 4 GB.
* To set the binary data (bytes) of an image to sql command, we need to use the setBinaryStream() method on PrepareStatement object.

Here is the complete example to insert an image using JDBC into database.

**Insert an Image using JDBC Example :**

**Create a table like below :**

CREATE TABLE employee (

id int(10) DEFAULT NULL,

name varchar(100) DEFAULT NULL,

salary varchar(100) DEFAULT NULL,

photo blob

);

**Example:**

JDBC\_LargeObjects\_Example.java

import java.io.File;

import java.io.FileInputStream;

import java.io.InputStream;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

public class JDBC\_LargeObjects\_Example {

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

// TODO Auto-generated method stub

Connection connection = null;

PreparedStatement pstatement = null;

Scanner scanner = null;

String img = "C:/Users/cgoka/Desktop/sample.jpg";

try {

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

connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/jdbc", "root", "123456");

pstatement = connection

.prepareStatement("insert into employee values(?,?,?,?)");

pstatement.setInt(1, 101);

pstatement.setString(2, "Chandra Shekhar");

pstatement.setString(3, "3000");

File file = new File(img);

FileInputStream stream = new FileInputStream(file);

pstatement.setBinaryStream(4, stream);

pstatement.executeUpdate();

System.out.println("Image Inserted");

} catch (Exception e) {

e.printStackTrace();

}

}

}

On the above example, we are using the FileInputStream to read the image from the local computer. And passing the FileInputStream to **PrepareStatement** object using **setBinaryStream()** method.

In this tutorial, we are going to learn about **ResultSetMetaData in JDBC**. The typical definition of a MetaData is data about the data. Here ResultSetMetaData means, the data about the ResultSet is called ResultSetMetaData.

Here we listed some of the important points about **ResultSetMetaData**in JDBC.

**ResultSetMetaData in JDBC :**

* ResultSetMetaData is an interface, which is coming from java.sql package.
* If we want to [*read the data from database*](https://www.onlinetutorialspoint.com/jdbc/jdbc-select-program-example.html) we need to use executeQuery() method. Obviously executeQuery() method returns ResultSet object, which consists the data rows.
* If we want to read the data from ResultSet object, then we should have the knowledge about the data stored in ResultSet.
* Because based on the type of data only, we need to use appropriate getXxx() method like below:

ResultSet rs = stmt.executeQuery("select \* from student");

int id = rs.getInt(1);

String name = rs.getString(2);

* If we don’t know the exact data about the ResultSet, what type of get???() method we use?
* No problem, here is the solution. ResultSetMetaData is actually comes to help us regarding this.
* ResultSetMetaData provides the data about the ResultSet object.
* It will provide all necessary information about the data available in ResultSet.
* We can get the ResultSetMetaData object by calling **getMetadata()** method on ResultSet object.

ResultSet rs = stmt.executeQuery("select \* from student");

ResultSetMetaData resultSetMetaData = rs.getMetaData();

**ResultSetMetaData in JDBC Example :**

Here is the complete example for ResultSetMetaData in JDBC.

Jdbc\_ResultSetMetaData\_Example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.\*;

import java.util.Scanner;

import com.mysql.jdbc.DatabaseMetaData;

public class Jdbc\_ResultSetMetaData\_Example {

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

Connection connection = null;

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

connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/onlinetutorialspoint", "root", "123456");

Statement stmt = connection.createStatement();

String sqlQuery = "select \* from student";

ResultSet rs = stmt.executeQuery(sqlQuery);

ResultSetMetaData resultSetMetaData = rs.getMetaData();

int columnCount = resultSetMetaData.getColumnCount();

for (int i = 1; i <= columnCount; ++i) {

System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

System.out.print("Column Name : " + resultSetMetaData.getColumnLabel(i) + " \n");

System.out.print("Column Type : " + resultSetMetaData.getColumnType(i) + " \n");

System.out.print("Column Class Name : " + resultSetMetaData.getColumnClassName(i) + " \n");

System.out.print("Column Type Name :" + resultSetMetaData.getColumnTypeName(i) + " \n");

System.out.println("Database Name : " + resultSetMetaData.getCatalogName(i));

}

rs.close();

stmt.close();

connection.close();

}

}

In this tutorial, we are going to learn about DatabaseMetaData in JDBC with example. In the previous tutorial, we have discussed about what is *[ResultSetMetaData with example](https://www.onlinetutorialspoint.com/jdbc/resultsetmetadata-in-jdbc-example.html)*. DatabaseMetaData is some thing similar like ResultSetMetaData.

**DtabaseMetaData in JDBC :**

Here we listed some of the important points about DatabaseMetaData in JDBC.

* DatabaseMetaData is an interface, which is coming from java.sql package.
* We can use the DatabaseMetaData, when we want to know about database capabilities.
* Typically the DatabaseMetaData helps while creating the database tools.
* We can get the capabilities of a database by calling the **getMetaData()** on connection object.

java.sql.DatabaseMetaData dbMetaData = connection.getMetaData();

Here is the simple example for DatabaseMetaData in JDBC.

**DatabaseMetaData in JDBC Example:**

Jdbc\_MetaData\_Example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

import java.sql.Statement;

import java.util.Scanner;

import com.mysql.jdbc.DatabaseMetaData;

public class Jdbc\_MetaData\_Example {

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

Connection connection = null;

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

connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/onlinetutorialspoint", "root",

"123456");

java.sql.DatabaseMetaData dbMetaData = connection.getMetaData();

System.out.println("Database Name= "

+ dbMetaData.getDatabaseProductName());

System.out.println("Database version= "

+ dbMetaData.getDatabaseProductVersion());

System.out.println("Database Driver version= "

+ dbMetaData.getDriverVersion());

System.out.println("Sql keywords = " + dbMetaData.getSQLKeywords());

System.out.println("numeric functions= "

+ dbMetaData.getNumericFunctions());

System.out.println("String functions= "

+ dbMetaData.getStringFunctions());

System.out.println("Search String Escapre= "

+ dbMetaData.getSearchStringEscape());

System.out.println("Supported storage procedures= "

+ dbMetaData.supportsStoredProcedures());

System.out.println("getMaxRowSize=" + dbMetaData.getMaxRowSize());

System.out.println("getMaxStatement length= "

+ dbMetaData.getMaxStatementLength());

System.out.println("get max tables in a select query= "

+ dbMetaData.getMaxTablesInSelect());

System.out.println("get Max Length of Table Name= "

+ dbMetaData.getMaxTableNameLength());

System.out.println("jdbc api version is= "

+ dbMetaData.getJDBCMinorVersion());

}

}

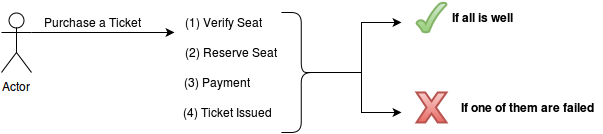
In this tutorials, we are going to learn abut **Transaction Management in JDBC.**A transaction represents a group of operations, used to perform a task.

A transaction is a set of commands, it will take our database from one consistent state to another consistent state.

**Transaction Management in JDBC :**

Here are the most important points about Transaction Management in JDBC.

* A transaction means, it is a group of operations used to perform a task.
* A transaction can reach either success state or failure state.
* If all operations are completed successfully then the transaction becomes success.
* If any one of the operation fail then all remaining operations will be cancelled and finally transaction will reach to fail state.

*[](https://www.onlinetutorialspoint.com/wp-content/uploads/2016/06/Transaction-Management-in-JDBC-min.png)*

**Types of Transactions :**

The basic transactions are of two types.

* Local Transactions
* Global / Distributed Transactions

**Local Transactions :**

If all the operations are executed on one/same database, then it is called as local transaction.

**Global / Distributed Transaction :**

If the operations are executed on more than one database then it is called as global transactions.

**Example :**If we transfer the money from *account1*to *account2*of same bank, then it is called as local transaction. If we transfer the money from *account1*to *account2*of different banks, then it is called as global or distributed transaction.

JDBC can supports only local transactions. For distributed transactions, we must use either EJB technology or Spring Framework.

**Transaction Management in JDBC Example :**

We can get the Transaction support in JDBC from Connection interface. The Connection interface given 3 methods to perform Transaction Management in JDBC.

* setAutoCommit()
* commit()
* rollback()

**Transaction setAutoCommit() :**

Before going to begin the operations, first we need to disable the auto commit mode. This can be done by calling setAutoCommit(false).

By default, all operations done from the java program are going to execute permanently in database. Once the permanent execution happened in database, we can’t revert back them (Transaction Management is not possible).

**Transaction commit() :**

If all operations are executed successfully, then we commit a transaction manually by calling the **commit()** method.

**Transaction rollback() :**

If any one of the operation failed, then we cancel the transaction by calling **rollback()** method.

connection.setAutoCommit(false);

try{

----------

----------

connection.commit();

}catch(Exception e){

connection.rollback();

}

**Complete Example :**

Jdbc\_TransactionManagement\_Example .java

package com.onlinetutorialspoint.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;

public class Jdbc\_TransactionManagement\_Example {

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

Connection connection = null;

Statement statement = null;

try {

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

connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/onlinetutorialspoint", "root",

"123456");

connection.setAutoCommit(false);

statement = connection.createStatement();

statement

.executeUpdate("insert person values ('5001','Hyderabad','Chandra Shekhar')");

statement

.executeUpdate("insert person values ('5002','Banglore','Ram')");

connection.commit();

System.out.println("Transaction is commited.");

statement.close();

connection.close();

} catch (Exception e) {

e.printStackTrace();

connection.rollback();

}

}

}

**Output:**

Transaction is committed.

If we update with the statements,

statement.executeUpdate("insert person values ('5003','Hyderabad','Chandra Shekhar')");

statement.executeUpdate("insert person values ('5001','Banglore','Ram')");

We can get the Exception like below and the transaction will be roll backed.

com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Duplicate entry '5001' for key 'PRIMARY'

Transaction is rollbacked !

Happy Learning 🙂

## Batch Processing in JDBC :

Here are the most important points about Batch Processing in JDBC.

* If there are multiple sql operations in a jdbc program then one by one operation will be executed on database sequentially.
* If the commands are executed sequentially (one by one) then the number of trips (rounds) between an application to database will be increased.
* If number of trips are increased then the performance of an application will be decreased.
* In order to improve the performance by reducing the number of trips, we will use **Batch Processing in JDBC**.
* In Batch Processing, the SQL operations will be constructed as a batch and then the batch will be send to database in a single trip.

## How to Prepare a Batch :

To perform the batch processing in JDBC, the Statement interface provided two methods.

* addBatch()
* executeBatch()

#### addBatch() :

addBatch() method is used to construct a batch. Constructing a batch means, storing the SQL commands in a buffer, maintained by Statement object.

**executeBatch() :**

To execute the batch, we will use the executeBacth() method. When **executeBatch()**called, then the commands will be transferred at a time as a batch from buffer to database.

While executing the batch in database, in the middle of command fails then all remaining all commands will be cancelled and finally *[BatchUpdateException](https://docs.oracle.com/javase/7/docs/api/java/sql/BatchUpdateException.html)* will be thrown. So that it is recommended to execute the batch with Transaction Management.

## Batch Processing in JDBC Example :

Here is the complete example for Batch Processing in JDBC.

Jdbc\_BatchProcess\_Example.java

package com.onlinetutorialspoint.jdbc;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;

public class Jdbc\_BatchProcess\_Example {

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

Connection connection = null;

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

connection = DriverManager.getConnection(

"jdbc:mysql://localhost:3306/onlinetutorialspoint", "root",

"systemuser23!");

Statement stmt = connection.createStatement();

stmt.addBatch("update student set sname='Rajesh' where sid=101 ");

stmt.addBatch("insert into student values(30,'Chandra Shekhar','Banglore','30')");

stmt.addBatch("delete from student where sid=3001");

int[] result = stmt.executeBatch();

int sum = 0;

for (int i = 0; i > result.length; i++) {

sum = sum + result[i];

}

System.out.println(sum + " records are effected ");

stmt.close();

connection.close();

}

}