ADVANCED JAVA

BCA-V SEM

NOTES

Prepared By

Prof. Prabhu Kichadi, M.Tech
KLES SSMS BCA College Athani

UNIT – III

CONTENTS

JDBC Architecture: Introduction to JDBC, Java and JDBC, JDBC VS ODBC, JDBC DRIVER MODEL, JDBC Driver Types, Types of Driver Managers, JDBC Connection process, Statement object, PreparedStatement object, operations on Resultset (Read, insert, update and delete), transaction processing, Metadata, ResultsetMetadata, Data types.

JDBC Architecture

Introduction to JDBC - JDBC (Java DataBase Connectivity)

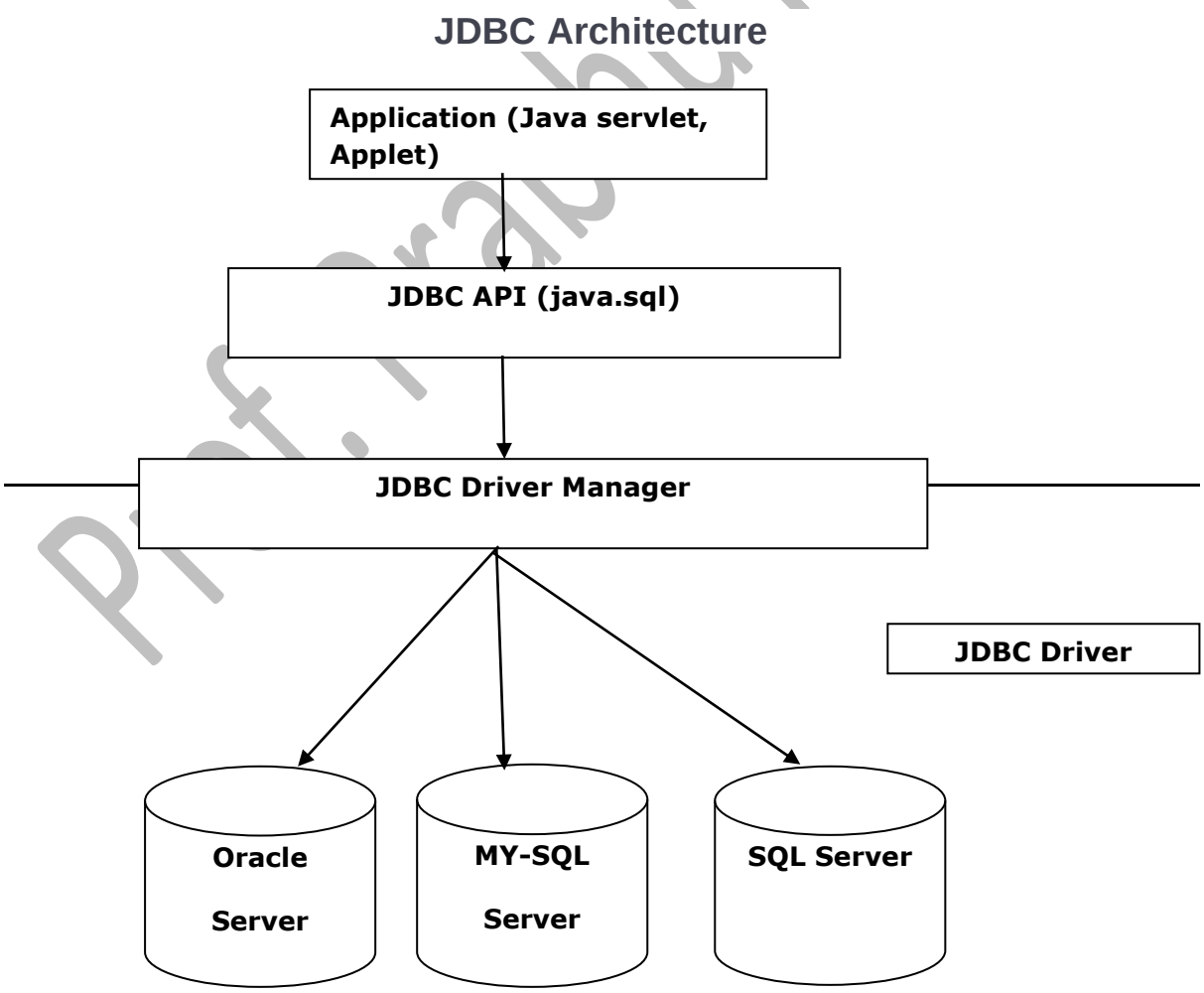
*"JDBC is an API(Application programming interface) which is used in java programming to interact with various databases, with the help of package **java.sql** which contains classes & interfaces".*

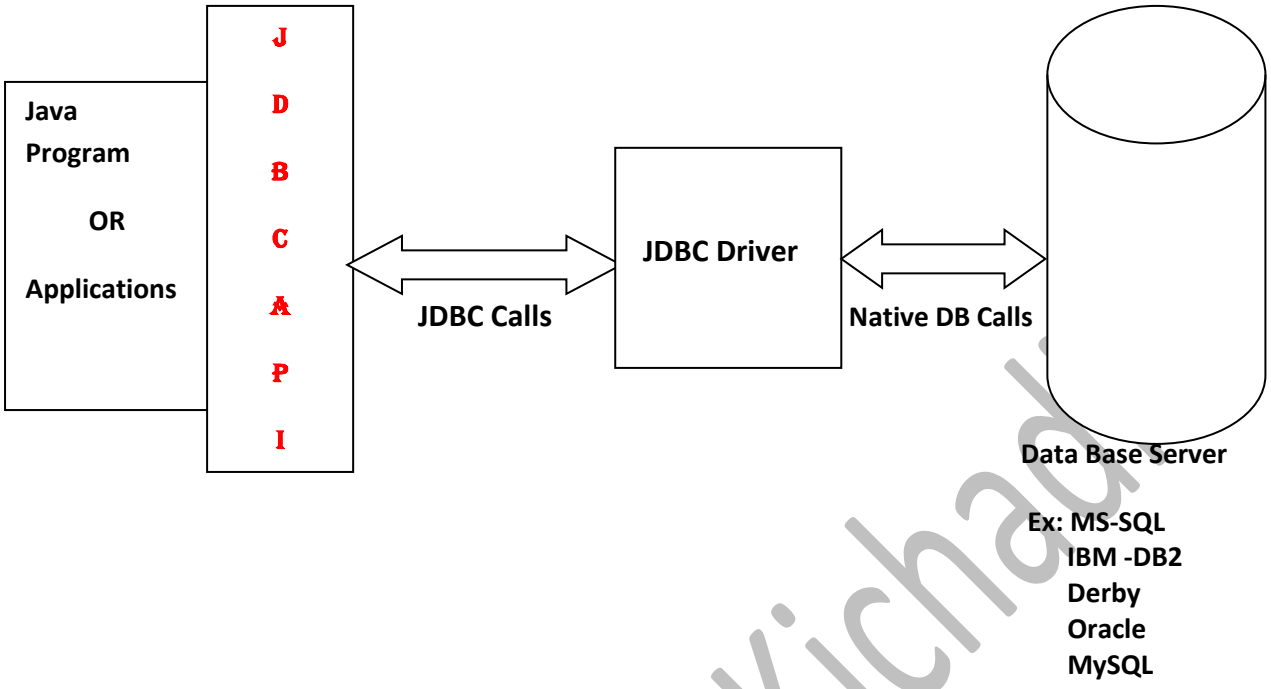
It provides the language with java database connectivity standard. It is used to write programs required to access databases. The enterprise data stored in a relational database (RDB) can be accessed with the help of JDBC APIs.

The classes and interfaces of JDBC API allows application to send request made by users to the specified database.

Purpose Of JDBC:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.





JDBC architecture involves Four major components.

1. Application: Any java program or a java application which must use JDBC API to communicate with the database to access data source.

2. The JDBC API: The JDBC API allows Java programs to execute SQL statements and retrieve results.

JDBC API is nothing but java.sql package which contains many interface, impl classes, helper classes, those are

DriverManager (c)
Driver(i)
Connection(i)
Statement(i)
PreparedStatement(i)
CallableStatement(i)
ResultSet(i)
ResultSetMetaData(c)
SQL data

3. DriverManager: It plays an important role in the JDBC architecture. It uses some database-specific drivers to effectively connect enterprise applications to databases.

4. **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.
JDBC Driver is a piece of code which helps to communicate from a java program or application to a database server.
JDBC driver translates JDBC instructions (JDBC Calls) into native database instructions.

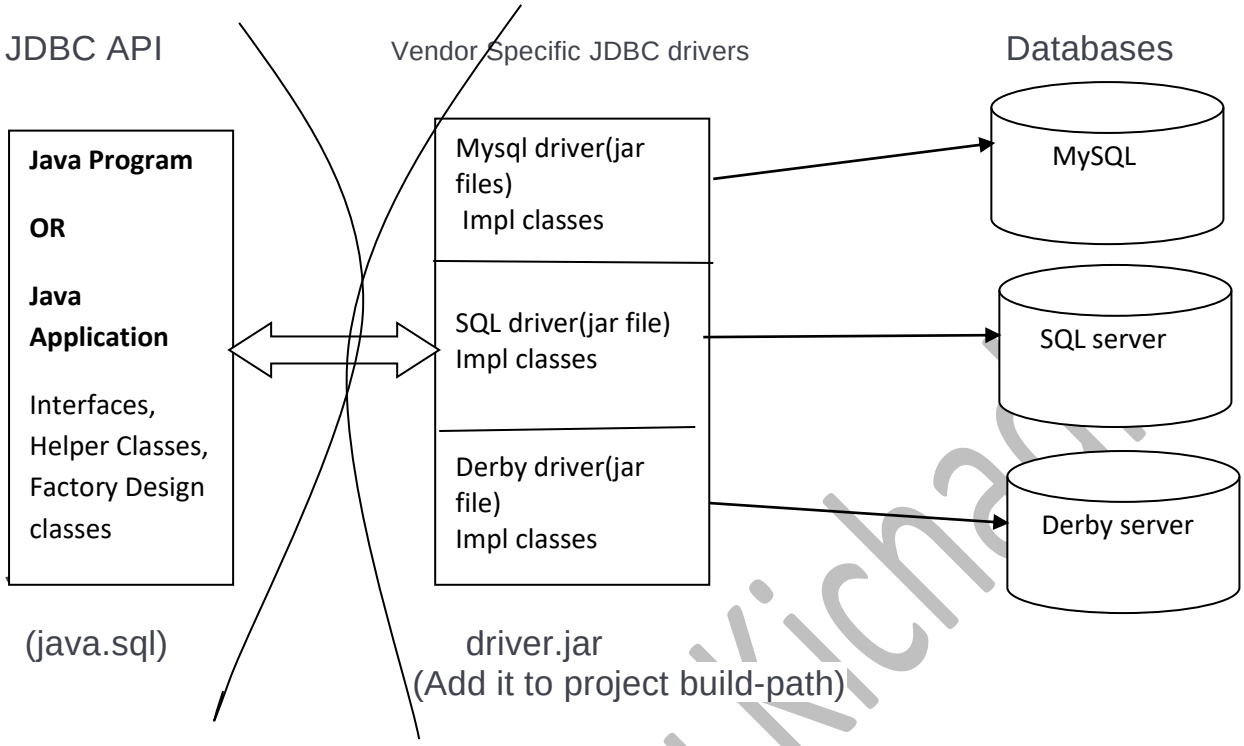
JDBC V/S ODBC:

ODBC is an SQL-based Application Programming Interface (API) created by Microsoft that is used by **Windows software applications** to access databases via SQL.
ODBC stands for Open DataBase Connectivity.

JDBC is an SQL-based API created by Sun Microsystems to enable **Java applications** to use SQL for database access.

ODBC	JDBC
ODBC Stands for Open Database Connectivity.	JDBC Stands for java database connectivity.
Introduced by Microsoft in 1992.	Introduced by SUN Micro Systems in 1997.
We can use ODBC for any language like C,C++,Java etc.	We can use JDBC only for Java languages.
We can choose ODBC only windows platform.	We can Use JDBC in any platform.
Mostly ODBC Driver developed in native languages like C,C++.	JDBC Stands for java database connectivity.
For Java applications it is not recommended to use ODBC because performance will be down due to internal conversion and applications will become platform Dependent.	For Java application it is highly recommended to use JDBC because there we no performance & platform dependent problem.
ODBC is procedural.	JDBC is object oriented.

JDBC API & JDBC Drivers:



JDBC Driver: *JDBC Driver is a software component that enables java application to interact with the database. JDBC driver is an implementation of JDBC API.*

JDBC API drivers contains implementations given by DB providers.
JDBC drivers are given in the form .jar(Java Archive) files, that has to added to project build-path.

JAR : Java Archive: It is a collection of compressed java related files such as .java, .class, .conf.
Ex: mysql.jar -> which contains executable class files related to mysql database.

Creation of jar file:

Open cmd->cd dir(The folder for which you want to create a jar file)->
jar -cvf filename.jar * . *

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.

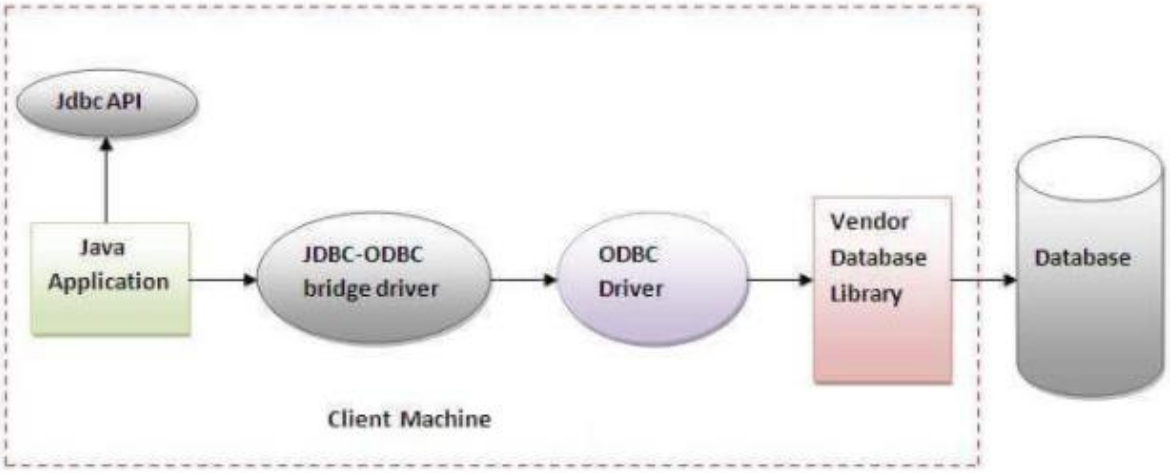


Figure- JDBC-ODBC Bridge Driver

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 (partially java 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.

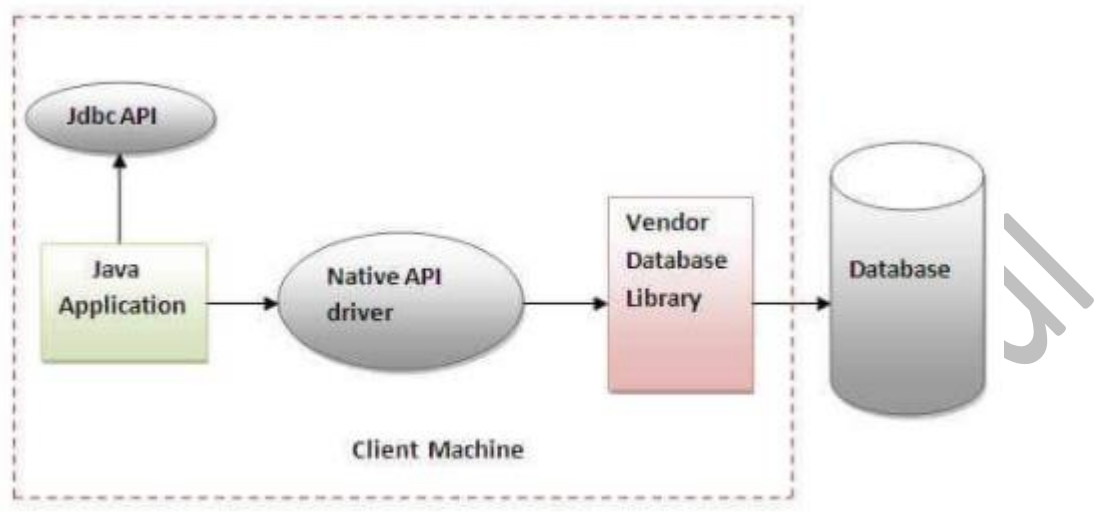


Figure- Native API Driver

Advantage:

- Performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

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

3. Network Protocol driver (fully java 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.

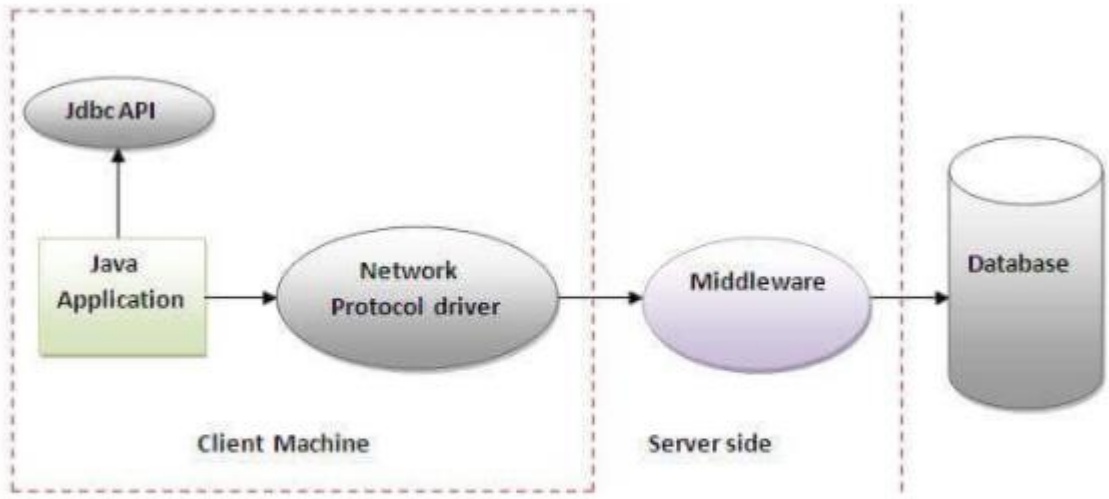


Figure- Network Protocol Driver

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 (fully java 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.

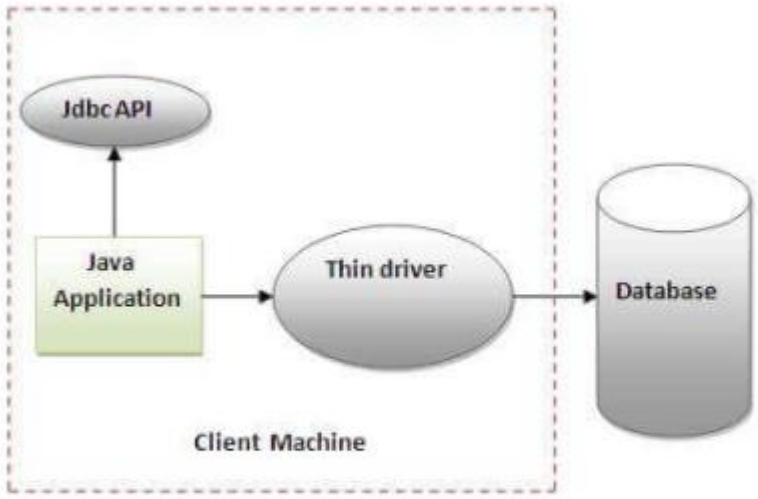


Figure- Thin Driver

Advantage:

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

Disadvantage:

- Drivers depend on the Database.

Credentials for JDBC Connection:

Database server: MySQL

Mysql driver jar file: mysql-connector-java-8.0.23.zip-> mysql-connector-java-8.0.23.jar

MySQL driver class: com.mysql.jdbc.Driver

URL of MySQL server: jdbc:mysql://localhost:3306/dbname

User: root

Password: tiger

Database server: SQL Server

Mysql driver jar file:

MySQL driver class:

URL of MySQL server:

User:

Password:

JDBC Connection Process OR JDBC Steps

1. Import **JDBC** package.
2. Load and register the **JDBC** driver class.
3. Open a **Connection** to the database.
4. Create a **Statement** object to perform a query.
5. Execute the Statement object with SQL queries and return a query **ResultSet**.
6. **Process** the **ResultSet.**
7. Close the **ResultSet, Statement** object & **Connection** object.

1. Import JDBC API Packages & JDBC Driver Packages.

- java program must contain import statement, to access **java.sql** package classes.

```
import java.sql. *;
```

2. Load and register the JDBC driver class.

There are two ways we can load & register driver class.

1. using registerDriver (Driver dr) method:

- By using static method **registerDriver (Driver dr)** of **DriverManager** class, we can load & register Driver class into JRE memory.

```
import java.sql.*;
Java.sql.Driver dr = new com.mysql.jdbc.Driver();
try
{
        DriverManager.registerDriver(dr);
}
catch(SQLException e)
{
        e.printStackTrace();
}
```

2. using Class.forName(String classname)

By using static method **forName (String driver)** of **java. lang. Class** class, we can load & register Driver class into JRE memory.

```
import java.sql.*;
String driver = “com.myql.jdbc.Driver”;

try
{
        Class.forName(driver);
}
catch(ClassNotFoundException e)
{
        e.printStackTrace();
}
```

3. Open a Connection to the database.

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

1. **public static Connection getConnection(String url) throws SQLException**

2. public static Connection getConnection(String url, String user, String password) throws SQLException

We should get first URL, username, password, Database name of the database server, these three parameters are strings must be passed to getConnection( ) method. IT returns Connection object.

Ex:

```
import java.sql.*;

String url = "jdbc:mysql://localhost:3306/db_kle";
String user = "root";
String pass = "admin";

Connection con = null;

con = DriverManager.getConnection(url, user, pass);
```

Note: If URL, user & passwords are accurate then Connection object will be returned, else it will throw SQLException.

4. Create a Statement object to perform a query.

The **createStatement()** method of **Connection** interface is used to create statement. The object of statement is responsible to execute queries with the database. It must be called through the same Connection object.

Method Syntax:

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

Example:

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

5 & 6. Execute the Statement object with SQL queries and return a query ResultSet.

Some of the Statement object methods which are used for CRUD operations to database.

1) public ResultSet executeQuery(String sqlQry): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sqlQry): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sqlQry): is used to execute queries that may return multiple results.

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

Ex:

```
import java.sql.*;
Statement stmt;
ResultSet rs;
String insQry = "insert into tab_student(id, name) values(11,'Ravi')";
stmt = con.createStatement();
stmt.execute(insQry);
rs = smt.executeQuery("select * from tab_student");
//Process rs which contains retrieved data from database
rs = stmt.executeQuery("select * from tab_student");

        while(rs.next())
        {
                System.out.println("ID : "+rs.getInt(1)+" Name :
"+rs.getString(2));
```

```
}
```

7. Close the resources ResultSet, Statement & Connection

- After work done, we need to close costly resources. Using close( ) method.

```
rs.close();
stmt.close();
con.close();
```

Statement Interface object

The **Statement interface** provides methods to execute queries with the database. The statement interface is a factory of ResultSet i.e., it provides factory method to get the object of **ResultSet**.

- First we need to create an object of Statement, by using below method of **Connection** interface.
- **public Statement createStatement( );**

Some of the **Statement** object methods which are used for CRUD operations to database.

1) public ResultSet executeQuery(String sqlQry): is used to execute SELECT query. It returns the object of ResultSet.

2) public int executeUpdate(String sqlQry): is used to execute specified query, it may be create, drop, insert, update, delete etc.

3) public boolean execute(String sqlQry): is used to execute queries that may return multiple results.

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

1. Queries: SELECT Command: It is used to fetch the database table records to java program or an application.

Statement Method used for SELECT Command:

public ResultSet executeQuery(String selQry);

- We need to pass a SELECT Query as string to this method, It returns ResultSet Interface object, which is used for retrieving data from database table.
- ResultSet object contains the data fetched from SELECT query.

Example on Statement & ResultSet

```
import java.sql.*;
public class MyStatement
{
	public static void main(String[] args)
	{
		String dr = "com.mysql.jdbc.Driver";
		String url = "jdbc:mysql://localhost:3306/db_kle";
		String user = "root"; String pass = "admin";
		Connection con = null;;
		Statement stmt = null;
		String selQry = "select * from tab_student";
		ResultSet rs = null;;

		try
		{
			Class.forName(dr);
			System.out.println("Driver Loaded");
			con = DriverManager.getConnection(url, user, pass);
			System.out.println("Connection established");
			stmt = con.createStatement();
			System.out.println("Statement Created");
			rs = stmt.executeQuery(selQry);
			System.out.println("Select Query is exeuted");
			while(rs.next())
			{
				System.out.println("ID: "+rs.getInt(1)+" Name : “
							+rs.getString(2));
			}
		}
		catch (ClassNotFoundException | SQLException e)
		{
			e.printStackTrace();
		}
		finally
		{
			try
			{
				rs.close();
				stmt.close();
				con.close();
			}
			catch (SQLException e)
			{
				e.printStackTrace();
			}
		}
	}
```

}

Output:

```
Driver Loaded
Connection established
Statement Created
Select Query is exeuted
ID: 11 Name : Ravi
ID: 12 Name : raju
ID: 13 Name : rohit
```

2. Queries: INSERT, UPDATE, DELETE Command: It is used to Modify the table records, DML queries.

Statement Method used for these Commands:

> **public int executeUpdate(String sqlQry):** is used to execute specified query, it may be create, drop, insert, update, delete etc.

Example on Statement with DML:
1. INSERT Command

```
import java.sql.*;
public class MyStatement1
{
	public static void main(String[] args)
	{
		String dr = "com.mysql.jdbc.Driver";
		String url = "jdbc:mysql://localhost:3306/db_kle";
		String user = "root";
		String pass = "admin";
		Connection con = null;;
		Statement stmt = null;
		String insQry = "insert into tab_student(id,name) values(101, 'Mohan')";
		try
		{
			Class.forName(dr);
			System.out.println("Driver Loaded");
			con = DriverManager.getConnection(url, user, pass);
			System.out.println("Connection established");
			stmt = con.createStatement();
			System.out.println("Statement Created");
			stmt.executeUpdate(insQry);
```

```
                System.out.println("INSERT Query is exeuted");

            }

            catch (ClassNotFoundException | SQLException e)
            {
                e.printStackTrace();
            }
            finally
            {
                try
                {
                    stmt.close();
                    con.close();
                }
                catch (SQLException e)
                {
                    e.printStackTrace();
                }
            }
        }
}
```

Output:

```
Driver Loaded
Connection established
Statement Created
INSERT Query is exeuted
```

Database Table: tab_student

```
mysql> select * from tab_student;
+-----+-------+
| ID  | NAME  |
+-----+-------+
| 11  | Ravi  |
| 12  | raju  |
| 13  | rohit |
| 100 | Reeta |
| 101 | Mohan |
+-----+-------+
```

Example on Statement with DML:
2. UPDATE Command

```
import java.sql.*;

public class MyStatement2
{
	public static void main(String[] args)
	{
		String dr = "com.mysql.jdbc.Driver";
		String url = "jdbc:mysql://localhost:3306/db_kle";
		String user = "root";
		String pass = "admin";
		Connection con = null;;
		Statement stmt = null;
		String upQry = "update tab_student set name = 'RAGHAV' where id =
101";

		try
		{
			Class.forName(dr);
			System.out.println("Driver Loaded");
			con = DriverManager.getConnection(url, user, pass);
			System.out.println("Connection established");
			stmt = con.createStatement();
			System.out.println("Statement Created");
			stmt.executeUpdate(upQry);
			System.out.println("UPDATE Query is exeuted");

		}
		catch (ClassNotFoundException | SQLException e)
		{
			e.printStackTrace();
		}
		finally
		{
			try
			{
				stmt.close();
				con.close();
			}
			catch (SQLException e)
			{
				e.printStackTrace();
			}
		}
	}
```

}

Output:

```
Driver Loaded
Connection established
Statement Created
UPDATE Query is exeuted
```

Before Update:

```
mysql> select * from tab_student;
+-----+--------+
| ID  | NAME   |
+-----+--------+
| 11  | Ravi   |
| 12  | raju   |
| 13  | rohit  |
| 100 | Reeta  |
| 101 | Mohan  |
+-----+--------+
```

After Update:

```
mysql> select * from tab_student;
+-----+--------+
| ID  | NAME   |
+-----+--------+
| 11  | Ravi   |
| 12  | raju   |
| 13  | rohit  |
| 100 | Reeta  |
| 101 | RAGHAV |
+-----+--------+
```

Example on Statement with DML:
3. DELETE Command

```
import java.sql.*;

public class MyStatement3
{
	public static void main(String[] args)
	{
		String dr = "com.mysql.jdbc.Driver";
		String url = "jdbc:mysql://localhost:3306/db_kle";
		String user = "root";
		String pass = "admin";
		Connection con = null;;
		Statement stmt = null;
		String delQry = "delete from tab_student where id = 101";


		try
		{
			Class.forName(dr);
			System.out.println("Driver Loaded");
			con = DriverManager.getConnection(url, user, pass);
			System.out.println("Connection established");
			stmt = con.createStatement();
			System.out.println("Statement Created");
			stmt.executeUpdate(delQry);
			System.out.println("DELETE Query is exeuted");

		}

		catch (ClassNotFoundException | SQLException e)
		{
			e.printStackTrace();
		}
		finally
		{
			try
			{
				stmt.close();
				con.close();
			}
			catch (SQLException e)
			{
				e.printStackTrace();
			}
		}
	}
```

}

Output:

```
Driver Loaded
Connection established
Statement Created
DELETE Query is exeuted
```

Before DELETE:

```
mysql> select * from tab_student;
+-----+--------+
| ID  | NAME   |
+-----+--------+
| 11  | Ravi   |
| 12  | raju   |
| 13  | rohit  |
| 100 | Reeta  |
| 101 | RAGHAV |
+-----+--------+
```

After DELETE:

```
mysql> select * from tab_student;
+-----+-------+
| ID  | NAME  |
+-----+-------+
| 11  | Ravi  |
| 12  | raju  |
| 13  | rohit |
| 100 | Reeta |
+-----+-------+
```

PreparedStatement object

- PreparedStatement interface object is used to compile queries once, setting parameters for placeholders (?) using setter methods and executes queries.
- It makes applications faster because queries are compiling only once.

Create PreparedStatement Object:

- Using below method we can create an instance of the PreparedStatement interface.
- Method is present in Connection interface.

```
public PreparedStatement prepareStatement(String query)
        throws SQLException{ }
```

Ex:

```
Connection con = null;
PreparedStatement pstmt = con.prepareStatement();
```

Queries format for PreparedStatement object:

- Queries must contain a place holder (?) parameter, these parameters values must be set with the help of following methods of PreparedStatement object.

Ex: **insert into tab_student(id, name) values(?, ?);**
Update tab_student set name = ? where id = ? ;

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create,

	drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Example on PreparedStatement:

```
import java.sql.*;

public class MyPreparedStatement
{
	public static void main(String[] args)
	{
		String dr = "com.mysql.jdbc.Driver";
		String url = "jdbc:mysql://localhost:3306/db_kle";
		String user = "root";
		String pass = "admin";
		Connection con = null;;
		PreparedStatement pstmt = null;
		String insQry = "insert into tab_student(id,name) values(?, ?)";

		try
		{
			Class.forName(dr);
			System.out.println("Driver Loaded");
			con = DriverManager.getConnection(url, user, pass);
			System.out.println("Connection established");
			pstmt = con.prepareStatement(insQry);
			System.out.println("Prepared Statement Created with Query is
compiled");

			pstmt.setInt(1, 201);
			pstmt.setString(2, "Rakesh");
			System.out.println("Query is set with parameters");
			pstmt.executeUpdate();
			System.out.println("Query with parameters are executed");

			pstmt.setInt(1, 202);
			pstmt.setString(2, "Rakesh");
			System.out.println("Query is set with parameters");
			pstmt.executeUpdate();
			System.out.println("Query with parameters are executed");
```

```
            }

            catch (ClassNotFoundException | SQLException e)
            {
                  e.printStackTrace();
            }
            finally
            {
                  try
                  {
                        pstmt.close();
                        con.close();
                  }
                  catch (SQLException e)
                  {
                        e.printStackTrace();
                  }
            }

      }
}
```

Output:

```
Driver Loaded
Connection established
Prepared Statement Created with Query is compiled
Query is set with parameters
Query with parameters are executed
Query is set with parameters
Query with parameters are executed
```

After Insert:

```
mysql> select * from tab_student;
+-----+--------+
| ID  | NAME   |
+-----+--------+
| 11  | Ravi   |
| 12  | raju   |
| 13  | rohit  |
| 100 | Reeta  |
| 201 | Rakesh |
| 202 | Rakesh |
+-----+--------+
```

ResultSet & Operations on ResultSet

java.sql.ResultSet interface is an object that holds result set of a database select query.

After executing SELECT query from Statement object, it returns data set in the form of row & column. Which is stored in a ResultSet object.

A ResultSet object maintains a cursor that points to the current row in the result set. Initially, cursor points to before the first row. If we call next() method it will move to next record.

Creation of ResultSet Object:

```
ResultSet rs = stmt.executeQuery(String selQry);
```

Note: By default, ResultSet object can be moved forward only and it is not updatable.

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:

```
Statement stmt = null;

stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
```

JDBC provides the following connection methods to create statements with desired ResultSet –

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- **prepareCall(String sql, int RSType, int RSConcurrency);**

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

At the time of Statement object creation in createStatement() method we have two int parameters,
public Statement createStatement(int Type, int Concurrency);

Scrollable ResultSet:

Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);

Type of ResultSet

1. ResultSet.TYPE_FORWARD_ONLY - The cursor can only move forward in the result set.

2. ResultSet.TYPE_SCROLL_INSENSITIVE - The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.

3. ResultSet.TYPE_SCROLL_SENSITIVE. - The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occurs after the result set was created.

Concurrency of ResultSet

1. ResultSet.CONCUR_READ_ONLY - Creates a read-only result set. This is the default

2. ResultSet.CONCUR_UPDATABLE - Creates an updateable result set.

The methods of the ResultSet interface can be broken down into three categories –

- **Navigational methods**: Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

Navigational methods of ResultSet:

1	**public void beforeFirst() throws SQLException** Moves the cursor just before the first row.
2	**public void afterLast() throws SQLException** Moves the cursor just after the last row.

3	**public boolean first() throws SQLException** Moves the cursor to the first row.
4	**public void last() throws SQLException** Moves the cursor to the last row.
5	**public boolean absolute(int row) throws SQLException** Moves the cursor to the specified row.
6	**public boolean relative(int row) throws SQLException** Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
7	**public boolean previous() throws SQLException** Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8	**public boolean next() throws SQLException** Moves the cursor to the next row. This method returns false if there are no more rows in the result set.
9	**public int getRow() throws SQLException** Returns the row number that the cursor is pointing to.
10	**public void moveToInsertRow() throws SQLException** Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	**public void moveToCurrentRow() throws SQLException** Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

Get methods: The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions −

- One that takes in a column name.

- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet –

S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName.
2	**public int getInt(int columnIndex) throws SQLException** Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.
3	**public int getFloat(int columnIndex) throws SQLException**
4	**public int getString(int columnIndex) throws SQLException**

Update methods: The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type −

- One that takes in a column name.
- One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods –

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of s.
2	**public void updateString(String columnName, String s) throws**

	SQLException Similar to the previous method, except that the column is specified by its name instead of its index.

There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

S.N.	Methods & Description
1	**public void updateRow()** Updates the current row by updating the corresponding row in the database.
2	**public void deleteRow()** Deletes the current row from the database
3	**public void refreshRow()** Refreshes the data in the result set to reflect any recent changes in the database.
4	**public void cancelRowUpdates()** Cancels any updates made on the current row.
5	**public void insertRow()** Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

Example on ResultSet – Updatable & Scrollable.

```
import java.sql.*;

public class MyResultSet
{
	public static void main(String[] args)
	{
		String dr = "com.mysql.jdbc.Driver";
		String url = "jdbc:mysql://localhost:3306/db_kle";
		String user = "root";
		String pass = "admin";
		Connection con = null;
		Statement stmt = null;
		ResultSet rs = null;

		String selQry = "select * from tab_student";

		try
		{
			Class.forName(dr);
			System.out.println("Driver Loaded");
			con = DriverManager.getConnection(url, user, pass);
			System.out.println("Connection established");
			stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);
			System.out.println("Created Scrollable ResultSet..");

			rs = stmt.executeQuery(selQry);
			System.out.println("ResultSet Object Created With Scrollable &
Updatable");

			System.out.println("\nRecords Before Scroll....");
			while(rs.next())
			{
				System.out.println("Id : "+rs.getInt(1)+" Name :
"+rs.getString(2));
			}

			rs.first();
			System.out.println("First Record : ");
			System.out.println("Id : "+rs.getInt(1)+" Name :
"+rs.getString(2));

			rs.last();
			System.out.println("Last Record : ");
```

```
                    System.out.println("Id : "+rs.getInt(1)+" Name :
"+rs.getString(2));

                    rs.absolute(3);
                    System.out.println("Third Record : ");
                    System.out.println("Id : "+rs.getInt(1)+" Name :
"+rs.getString(2));

                    System.out.println("Moves cursor before first record");
                    rs.beforeFirst();

                    rs.absolute(2);
                    rs.updateString(2, "Rajesh");
                    rs.updateRow();
                    System.out.println("\nRecord Updated...");


            }
            catch(ClassNotFoundException | SQLException e)
            {
                    e.printStackTrace();
            }
            finally
            {
                    if(con!=null && stmt!=null & rs!=null)
                    {
                            try
                            {
                                    con.close();
                                    stmt.close();
                                    rs.close();
                                    System.out.println("Resource Cleaned");
                            }
                            catch (SQLException e)
                            {
                                    e.printStackTrace();
                            }
                    }
            }
    }

}
```

Output:

```
Driver Loaded
Connection established
Created Scrollable ResultSet..
ResultSet Object Created With Scrollable & Updatable

Records Before Scroll....
Id : 11 Name : Ravi
Id : 100 Name : Akash
Id : 201 Name : Rakesh
Id : 202 Name : Rakesh
First Record :
Id : 11 Name : Ravi
Last Record :
Id : 202 Name : Rakesh
Third Record :
Id : 201 Name : Rakesh
Moves cursor before first record

Record Updated...
Resource Cleaned
```

Before Update:

```
mysql> select * from tab_student;
+-----+--------+
| ID  | NAME   |
+-----+--------+
| 11  | Ravi   |
| 100 | Akash  |
| 201 | Rakesh |
| 202 | Rakesh |
+-----+--------+
4 rows in set (0.00 sec)
```

After Update:

```
mysql> select * from tab_student;
+-----+--------+
| ID  | NAME   |
+-----+--------+
| 11  | Ravi   |
| 100 | Rajesh |
| 201 | Rakesh |
| 202 | Rakesh |
+-----+--------+
```

Batch Processing Statement:

Instead of executing a single query, we can execute a batch (group) of queries.
It makes the better performance.

The java.sql.Statement and java.sql.PreparedStatement interfaces provide methods for batch processing.

addBatch(String qry) of Statement, PreparedStatement & CallableStatement interfaces provide this method.

Add as many as related SQL statements you like into batch using **addBatch()** method on created statement object.

Execute all the SQL statements using **executeBatch()** method on created statement object.

Ex: Statement stmt = con.createStatement();
stmt.addBatch(qry1);
stmt.addBatch(qry1);
stmt.addBatch(qry1);

int a[ ] = stmt.executeBatch();

Example on Batch Processing using Statement.

```
import java.sql.*;

public class MyBatch
{
    public static void main(String[] args)
    {
        String dr = "com.mysql.jdbc.Driver";
        String url = "jdbc:mysql://localhost:3306/db_kle";
        String user = "root";
        String pass = "admin";
        Connection con = null;
        Statement stmt = null;

        String insQry1 = "insert into tab_student(id,name) value(801, 'Gayle')";
        String insQry2 = "insert into tab_student(id,name) value(802, 'KL')";
        String insQry3 = "insert into tab_student(id,name) value(803, 'Manan')";
        try
        {
```

```
                Class.forName(dr);
                System.out.println("Driver Loaded...");
                con =DriverManager.getConnection(url, user, pass);
                System.out.println("Connection to database...");
                stmt = con.createStatement();
                System.out.println("Statement Created...");

                //Adding SQL Queries into Batch
                stmt.addBatch(insQry1);
                stmt.addBatch(insQry2);
                stmt.addBatch(insQry3);

                //Execute Batch at a time
                stmt.executeBatch();
                System.out.println("Executing multiple queries...");


        }
        catch (ClassNotFoundException | SQLException e)
        {
                e.printStackTrace();
        }
        finally
        {
                if(con!=null && stmt!=null)
                {
                        try
                        {
                                stmt.close();
                                con.close();
                        }
                        catch (SQLException e)
                        {

                                e.printStackTrace();
                        }
                }
        }
    }
}
```

Output:

```
Driver Loaded...
Connection to database...
Statement Created...
Executing multiple queries...
```

ResultsetMetadata class

It is a class present in java.sql package.

Metadata: The data about data is called metadata.
It also called catalogue; it provides more information about data.

Ex:

DB Table contains 'n' number of records.
Metadata about table is table name, no of columns, name of the columns, datatype of columns, database name. etc.

*The **ResultSetMetaData** provides information about the obtained **ResultSet** object like, the number of columns, names of the columns, datatypes of the columns, name of the table etc.*

Creation of ResultSetMetaData Object:

```
ResultSet rs = stmt.executeQuery(selQry);
ResultSetMetaData rsm = rs.getMetaData();
```

Following are some methods of **ResultSetMetaData** class.

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.
public String getTableName(int index)throws SQLException	it returns the table name for the specified column index.

Example on ResultSetMetaData:

Getting MetaData about below table:

```
mysql> select * from tab_student;
+-----+--------+
| ID  | NAME   |
+-----+--------+
| 11  | Ravi   |
| 100 | Akash  |
| 201 | Rakesh |
| 202 | Rakesh |
+-----+--------+
4 rows in set (0.00 sec)
```

```
import java.sql.*;

public class MyResultSetMetaData
{
	public static void main(String[] args)
	{
		String dr = "com.mysql.jdbc.Driver";
		String url = "jdbc:mysql://localhost:3306/db_kle";
		String user = "root";
		String pass = "admin";
		Connection con = null;
		Statement stmt = null;
		ResultSet rs = null;

		String selQry = "select * from tab_student";

		try
		{
			Class.forName(dr);
			System.out.println("Driver Loaded");
			con = DriverManager.getConnection(url, user, pass);
			System.out.println("Connection established");
			stmt =  con.createStatement();
			System.out.println("Created ResultSet..");
			rs = stmt.executeQuery(selQry);
			System.out.println("ResultSet Object created");
			ResultSetMetaData rsm = rs.getMetaData();
			System.out.println("Name of the Table :
"+rsm.getTableName(2));
			System.out.println("Number of columns :
"+rsm.getColumnCount());
```

```
                    System.out.println("Name of the column1 :
"+rsm.getColumnName(1));
                    System.out.println("Name of the column2 :
"+rsm.getColumnName(2));
                    System.out.println("Name of the column1 Label :
"+rsm.getColumnLabel(1));
                    System.out.println("Column1 type  :
"+rsm.getColumnTypeName(1));
                }
                catch(ClassNotFoundException | SQLException e)
                {
                        e.printStackTrace();
                }
                finally
                {
                        if(con!=null && stmt!=null & rs!=null)
                        {
                                try
                                {
                                        con.close();
                                        stmt.close();
                                        rs.close();
                                        System.out.println("Resource Cleaned");
                                }
                                catch (SQLException e)
                                {
                                        e.printStackTrace();
                                }
                        }
                }
        }
}
```

Output

```
Driver Loaded
Connection established
Created ResultSet..
ResultSet Object created
Name of the Table : tab_student
Number of columns : 2
Name of the column1 : ID
Name of the column2 : NAME
Name of the column1 Label : ID
Column1 type  : INT
Resource Cleaned
```

Transaction Management in JDBC

Transaction: "Transaction is a set of related SQL statements/activities/task happens all at a time or none".

It treats a single SQL statement or a group of SQL statements as one logical unit, and if any statement fails, the whole transaction fails.

commit() : This used to permanently saves the transaction data to database. Cannot be revert back.

Using Connection object, we can do,

con.commit();

rollback(): A rollback operation undoes all the changes done by the current transaction. is used to cancel the previous queries/ or revert back to initial state of a transaction.

Using Connection object, we can do,

con.rollback();

Note: In JDBC all transactions are by default autoCommit(), hence if you want to manage transactions you need to turn-off autoCommit mode.

By using below method:

con.setAutoCommit(false);

Example on Transaction Management in JDBC:

Scenario: We need to insert two records for two different related tables, our transaction includes both INSERT queries should be executed & ***committed*** OR if any one Query fails, then it should be ***rollback***.

```
import java.sql.*;

public class MyTransaction
{

	public static void main(String[] args) throws SQLException
	{
		String dr = "com.mysql.jdbc.Driver";
		String url = "jdbc:mysql://localhost:3306/db_kle";
		String user = "root";
		String pass = "admin";
```

```
		Connection con = null;
		Statement stmt = null;
		ResultSet rs = null;
		boolean res1=true,res2=true;

		String insStud = "insert into tab_student(id, name) values(557,'Azhar')";
		String insEmp = "insert into tab_emp(empid, empname) values(12,
'Chahal')";

		try
		{
			Class.forName(dr);
			System.out.println("Driver Loading...");
			con = DriverManager.getConnection(url, user, pass);
			System.out.println("Connecting to database...");
			con.setAutoCommit(false);
			System.out.println("Disabled Auto Commit...");
			stmt =  con.createStatement();
			System.out.println("Created Statement...");

			res1 = stmt.execute(insStud);
			res2 = stmt.execute(insEmp);

			System.out.println("Both Records Inserted...");

		}
		catch(ClassNotFoundException | SQLException e)
		{
			System.out.println("Exception occured due to Insertion Failure");
		}
		finally
		{
			if(res1 == false && res2 == false)
			{
				con.commit();
				System.out.println("Commited sucess....");
			}
			else
			{
				con.rollback();
				System.out.println("rollbacked...");
			}

		}

	}


}
```

Output1:

```
Driver Loading...
Connecting to database...
Disabled Auto Commit...
Created Statement...
Both Records Inserted...
Commited sucess....
```

Output2:

```
Driver Loading...
Connecting to database...
Disabled Auto Commit...
Created Statement...
Exception occured due to Insertion Failure
rollbacked...
```

JDBC Data Types

- The JDBC driver converts the Java data type to the appropriate JDBC type before sending it to the database. It uses a default mapping for most data types.

For example, a Java int is converted to an SQL INTEGER. Default mappings were created to provide consistency between drivers.

The following table summarizes the default JDBC data type that the Java data type is converted to when you call the setXXX() method of the PreparedStatement or CallableStatement object or the ResultSet.updateXXX() method.

SQL	JDBC/Java	setXXX	updateXXX
VARCHAR	java.lang.String	setString	updateString
CHAR	java.lang.String	setString	updateString
LONGVARCHAR	java.lang.String	setString	updateString
BIT	boolean	setBoolean	updateBoolean
NUMERIC	java.math.BigDecimal	setBigDecimal	updateBigDecimal
TINYINT	byte	setByte	updateByte
SMALLINT	short	setShort	updateShort
INTEGER	int	setInt	updateInt
BIGINT	long	setLong	updateLong
REAL	float	setFloat	updateFloat
FLOAT	float	setFloat	updateFloat
DOUBLE	double	setDouble	updateDouble
VARBINARY	byte[ ]	setBytes	updateBytes
BINARY	byte[ ]	setBytes	updateBytes
DATE	java.sql.Date	setDate	updateDate
TIME	java.sql.Time	setTime	updateTime
TIMESTAMP	java.sql.Timestamp	setTimestamp	updateTimestamp
CLOB	java.sql.Clob	setClob	updateClob
BLOB	java.sql.Blob	setBlob	updateBlob
ARRAY	java.sql.Array	setARRAY	updateARRAY
REF	java.sql.Ref	SetRef	updateRef
STRUCT	java.sql.Struct	SetStruct	updateStruct

JDBC Exceptions

- Exception handling allows you to handle exceptional conditions such as program-defined errors in a controlled fashion.
- JDBC Exception handling is very similar to Java Exception handling but for JDBC, the most common exception is SQLException.
- There are 3 kinds of exceptions that are thrown by JDBC methods:
  1) **SQLException**
  2) **SQLWarning**
  3) **DataTruncation Exception**

1) SQLException

- SQLException commonly reflect a SQL syntax error in the query and thrown by many of methods contained in the java.sql package. This exception is most commonly caused by connectivity issues with the database.
- A SQLException can occur both in the driver and the database. When such an exception occurs, an object of type SQLException will be passed to the catch clause.
- The passed SQLException object has the following methods available for retrieving additional information about the exception:

Method	Description
getErrorCode( )	Gets the error number associated with the exception.
getMessage( )	Gets the JDBC driver's error message for an error handled by the driver or gets the Oracle error number and message for a database error.
getSQLState( )	Gets the XOPEN SQLstate string. For a JDBC driver error, no useful information is returned from this method. For a database error, the five-digit XOPEN SQLstate code is returned. This method can return null.

2) SQLWarning

The SQLWarning throws warnings received by the connection from the DBMS. The getWarning( ) method of Connection object retrieves the warning and getNextWarning( )method of the Connection object retrieves subsequent warnings.

3) DataTruncation Exception

A DataTruncation exception is thrown whenever data is lost due to truncation of the data value.

UNIT-III

Submission Due Date: 10/01/2023

2 Marks

1. Define JDBC driver.
2. What is JDBC? Mention the package name for JDBC.
3. List the classes and interfaces of java.sql package.
4. Differentiate JDBC & ODBC.
5. List any 4 JDBC driver types.
6. What is ResultSet and ResultSetMetaData?
7. Differentiate JDBC & ODBC.
8. List the methods of Statement and PreparedStatement object.
9. List the methods of Connection interface.

5/10 Marks

1. Explain JDBC Architecture.
2. Explain steps for JDBC.
3. Explain the following,
   - PreparedStatement
   - ResultsetMetadata
   - DatabaseMetadata
   - Save Point
4. Explain PreparedStatement object with an example.
5. Explain Statement object with an example.
6. Explain JDBC Process.
7. Explain JDBC driver types.
8. Differentiate JDBC & ODBC.
9. Explain JAR file.
10. What is JDBC Exception? Explain types.
11. What is JDBC Transaction? Explain how to handle JDBC transaction.
12. What is ResultSetMetaData? Explain.
13. Write a java program to demonstrate JDBC.
14. What is ScrollableResultSet? Explain.
15. What are JDBC Data types.