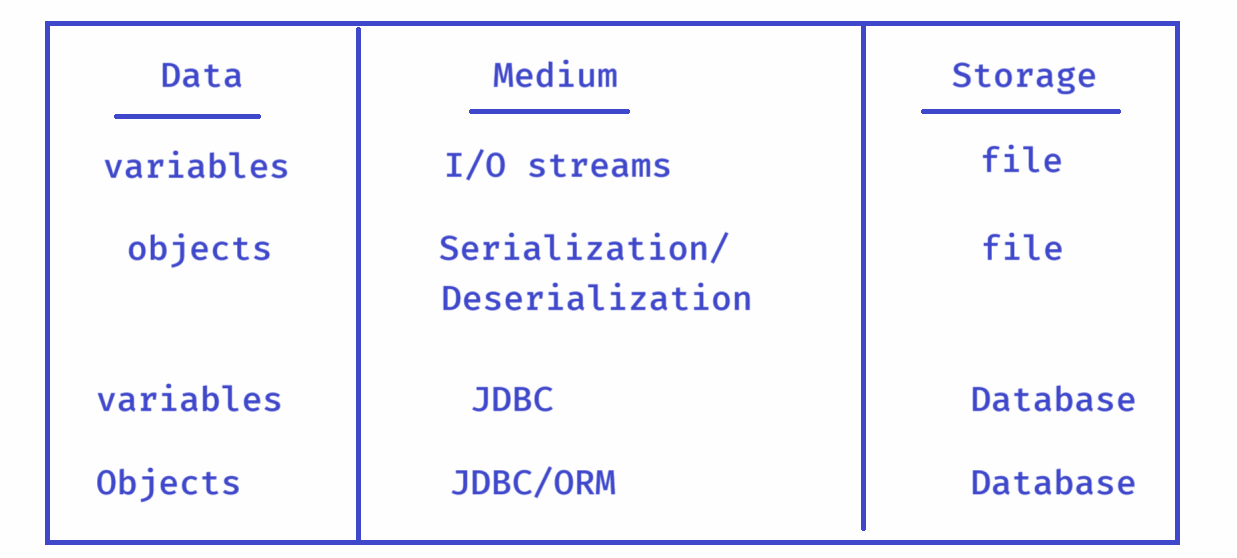
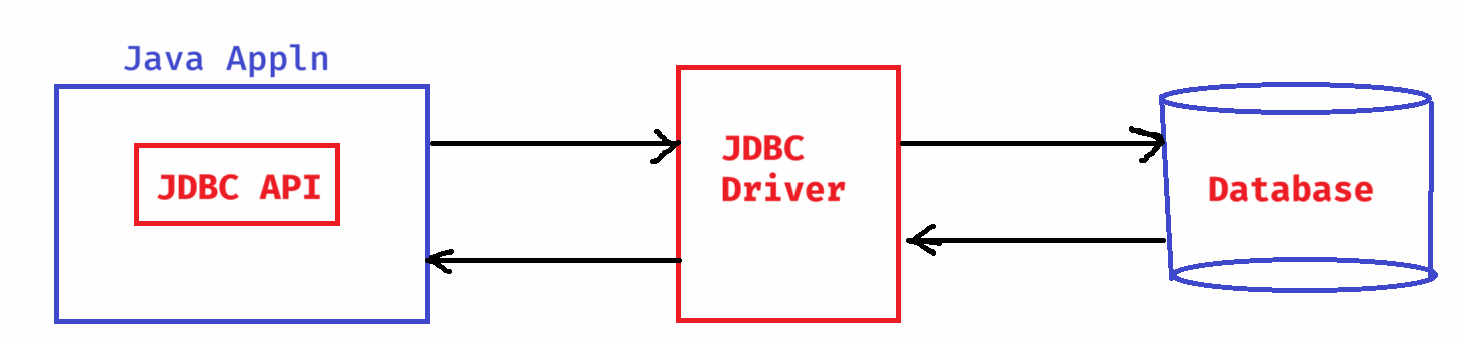
Java Database Connectivity

(JDBC)



* variables can store the temporarily.
* objects can also store the temporarily.
* files and Databases can store the data perminently.
* To write/read the data from files, we use I/O streams.
* To write/read the objects from files, we use serialization/deserialization
* To write/read the data from Database, we use JDBC technology.
* To write/read the objects from Database, we use JDBC or ORM(Object Relational Mapping) technology.
* Java Programs are developed in Java language and Databases can only understand SQL. So, Java Can’t directly talk to Databases.
* So, JDBC Technology has provided to make Java and Databases to talk to each other.
* JDBC is a specification from Sun Microsystems(now it is Oracle).
* JDBC is called a specification because it does not provided implementation for the interfaces.
* The Database vendors like Oracle, MySQL, PostgreSQL, etc.. will provide the actual implementation for the interfaces to talk with the database.
* These database vendors provided implementations are called as JDBC Drivers.
* These Drivers contains the real code to execute the queries on a database given by Java program and returns the response back to the Java program.



Types of JDBC drivers:

--------------------

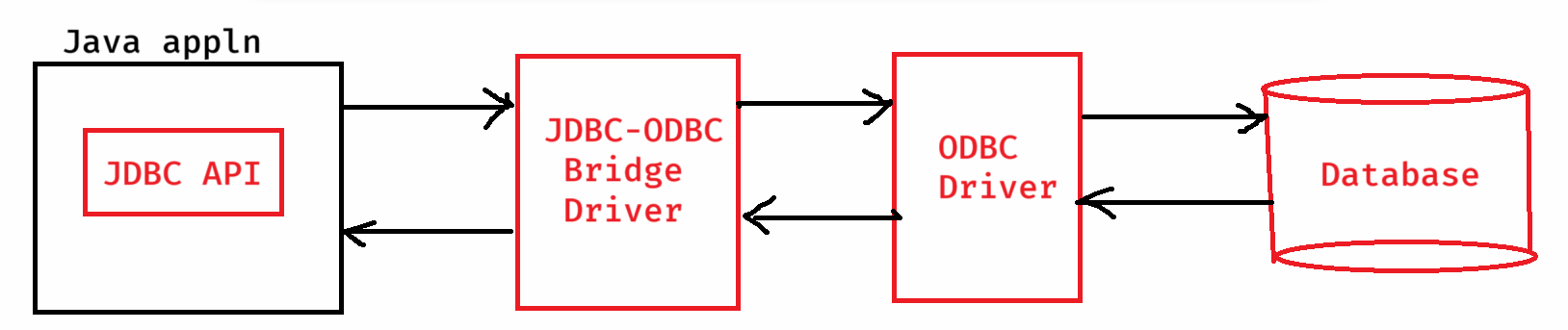
Type-1: JDBC-ODBC Bridge Driver

Type-2: Native API Driver

Type-3: Network Protcol Driver

Type-4: Native Protocol Driver

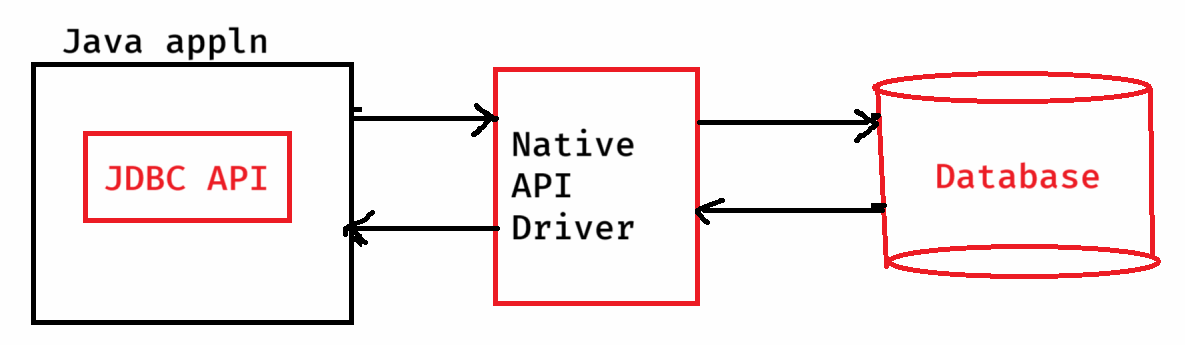
Type-1:



Note: This Type-1 drivers are used for the development purpose.

This Type-1 drivers are dropped from Java8.

Type-2:

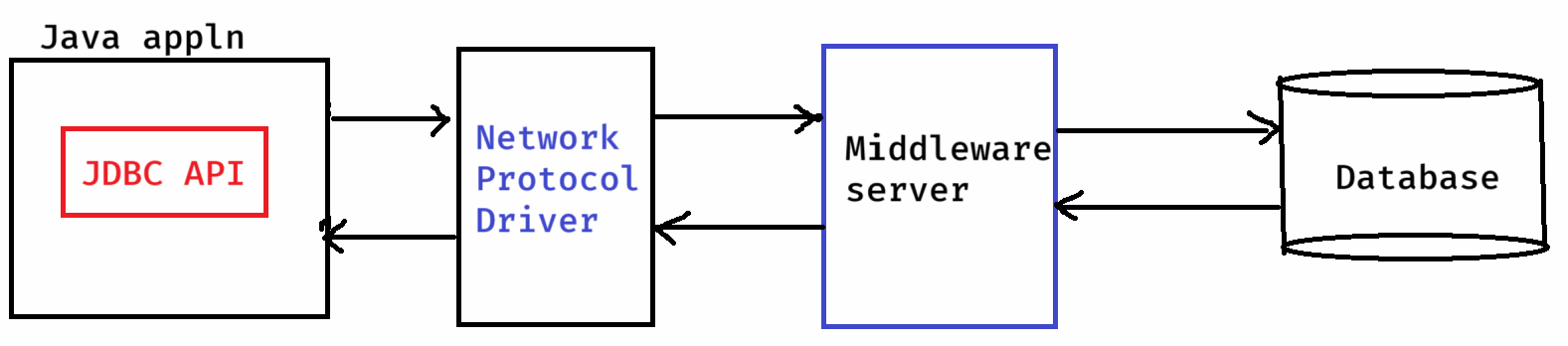


* This Type-2 drivers uses Native API of Databases, to talk to the databases.
* Native API is nothing but, it a library developed in C/C++ languages.
* Whenever you install Database software, automatically Native libraries are also installed into the system.

For example, When you install Oracle Database software, the native libraries are also installed into the system.

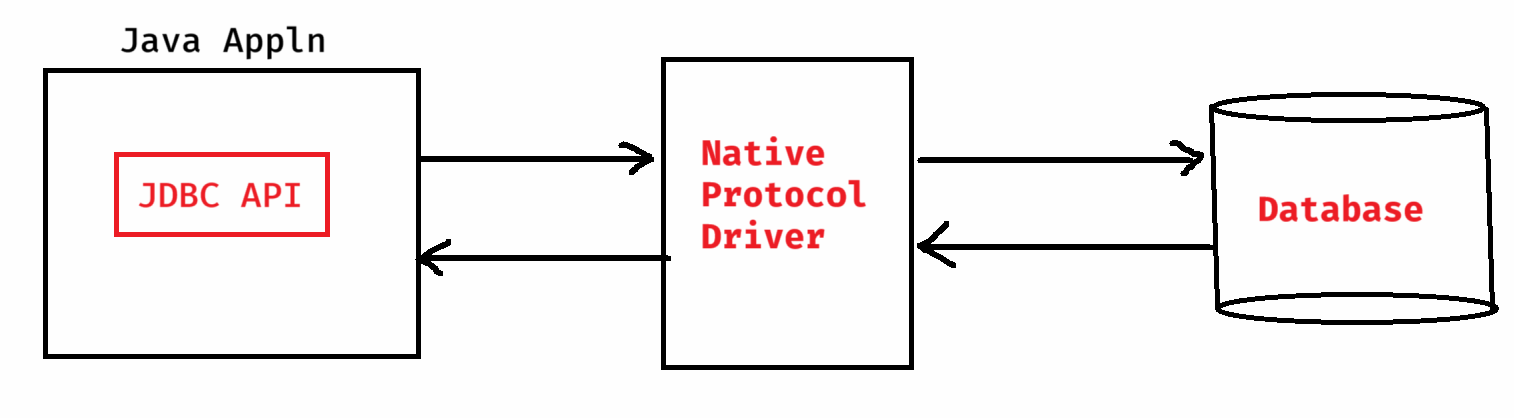
SQL Plus uses these native libraries only internally to connect with the Oracle Database.

Type-3:



* This Type-3 drivers are database indepdenent, because they don’t drirectly connects to the databases.
* The Type-3 drivers connects to the middleware server, that server will handle actual database connectivity.
* The drawback, we also need to install a middleware server in the system.

Type-4:



* Native protocol means, a specific lanaugage that a database can directly understand.
* This type-4 drivers speaks to the databases in their own language. So they connect Java Application with a database fastly.
* This Type-4 drivers are pure java drivers. So, these are platform independent drivers.
* The drawback of this Type-4 drivers is, they are database dependent drivers.
* It means, we need a separate driver for each database.
* Suppose, if you change the database then you also need to change the Jdbc driver.
* This Type-4 drivers are only used in the real-time applications.(production).

Steps to Write a JDBC(Java) Program:

step-1: Load the JDBC Driver

syntax:

Class.forName(“fully qualified driver classname”);

ex:

Class.forName(“oracle.jdbc.driver.OracleDriver”);

ex:

Class.forName(“com.mysql.cj.jdbc.Driver”);

Note: From Java8 version, loading the JDBC driver is optional in stand-alone applications.

step-2: Open/obtain a connection with a database

syntax:

Connection conn = DriverManager.getConnection(databaseUrl, username, password);

ex:

String databaseUrl = “jdbc:oracle:thin:@localhost:1521:xe”;

String username = “system”;

String password = “tiger”;

Connection conn = DriverManager.getConnection(databaseUrl, username, password);

Step-3: create a Statement object

syntax:

Statement stmt = conn.createStatement();

Step-4: execute the SQL queries.

executeUpdate() : To execute Non-select

operations(write operations)

executeQuery() : To execute select operation(read operation)

ex:

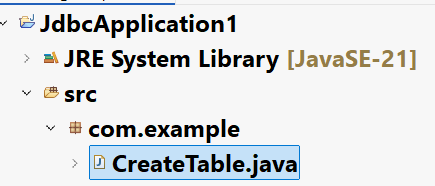
int count = stmt.executeUpdate(“UPDATE EMP SET SAL = 5000 WHERE DEPTNO = 20”);

ResultSet rs = stmt.executeQuery(“SELECT \* FROM EMP”);

Step-5: process the result

Step-6: close the Database connection.

conn.close();



CreateTable.java

**package com.example;**

**import java.sql.Connection;**

**import java.sql.DriverManager;**

**import java.sql.SQLException;**

**import java.sql.Statement;**

**public class CreateTable {**

**public static void main(String[] args) throws ClassNotFoundException, SQLException {**

**//step-1: Load the JDBC driver**

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

**System.*out*.println("JDBC Driver is loaded");**

**//step-2: open a connection with a Database**

**Connection conn = DriverManager.*getConnection*( "jdbc:oracle:thin:@localhost:1521:xe", "system", "tiger");**

**System.*out*.println("Connection is created");**

**//step-3: create a Statement object**

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

**System.*out*.println("Statement object is created");**

**//step-4: execute the SQL query**

**String query = "CREATE TABLE CUSTOMER ("**

**+ "ID NUMBER(9) PRIMARY KEY,"**

**+ "NAME VARCHAR2(20),"**

**+ "EMAIL VARCHAR2(40),"**

**+ "PHONE NUMBER(10)"**

**+ ")";**

**stmt.executeUpdate(query);**

**//step-5: process the result**

**System.*out*.println("Table is created");**

**//step-6: close the connection**

**stmt.close();**

**conn.close();**

**System.*out*.println("Connection is closed");**

**}**

**}**

InsertRecords.java

**package com.example;**

**import java.sql.Connection;**

**import java.sql.DriverManager;**

**import java.sql.Statement;**

**public class InsertRecords {**

**public static void main(String[] args) {**

***insert*(101, "JOHN", "john@gmail.com", 8998778663L);**

***insert*(102, "JAMES", "james@gmail.com", 9988099776L);**

***insert*(103, "JEFFRY", "jef@yahoo.com", 7990667722L);**

***insert*(104, "JACK", "jack@gmail.com", 6304458922L);**

***insert*(105, "JORDAN", "jordan@outlook.com", 7784432908L);**

**}**

**private static void insert(int id, String name, String email, long phone) {**

**try {**

**Connection conn = DriverManager.*getConnection*( "jdbc:oracle:thin:@localhost:1521:xe", "system", "tiger");**

**System.*out*.println("Connection is created");**

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

**System.*out*.println("Statement object is created");**

**String query = "INSERT INTO CUSTOMER VALUES("**

**+ id + ", '" + name + "', '" +**

**email + "', " + phone + ")";**

**stmt.executeUpdate(query);**

**System.*out*.println("Record inserted");**

**stmt.close();**

**conn.close();**

**System.*out*.println("connection is closed");**

**}**

**catch(Exception ex) {**

**ex.printStackTrace();**

**}**

**}**

**}**

FetchRecords.java

**package com.example;**

**import java.sql.Connection;**

**import java.sql.DriverManager;**

**import java.sql.ResultSet;**

**import java.sql.SQLException;**

**import java.sql.Statement;**

**public class FetchRecords {**

**public static void main(String[] args) {**

***retrieveRecords*("CUSTOMER");**

**}**

**private static void retrieveRecords(String table) {**

**Connection conn = null;**

**Statement stmt = null;**

**try {**

**String databaseUrl = "jdbc:oracle:thin:@localhost:1521:xe";**

**String username = "system";**

**String password = "tiger";**

**conn = DriverManager.*getConnection*(databaseUrl, username, password);**

**System.*out*.println("\u001B[31m" + "Connection is created");**

**stmt = conn.createStatement();**

**System.*out*.println("\u001B[34m" + "Statement object is created");**

**ResultSet rs = stmt.executeQuery("SELECT \* FROM " + table);**

**System.*out*.println("\u001B[32m" + " Records of the TABLE : " + table);**

**while(rs.next()) {**

**System.*out*.println(rs.getString(1) + " " + rs.getString(2) + " " + rs.getString(3) + " " + rs.getString(4));**

**}**

**rs.close();**

**}**

**catch(Exception ex) {**

**ex.printStackTrace();**

**}**

**finally {**

**try {**

**if( stmt != null)**

**stmt.close();**

**if( conn != null)**

**conn.close();**

**System.*out*.println("\u001B[35m" + "Connection is closed");**

**}**

**catch(SQLException ex) {**

**System.*out*.println(ex);**

**}**

**}**

**}**

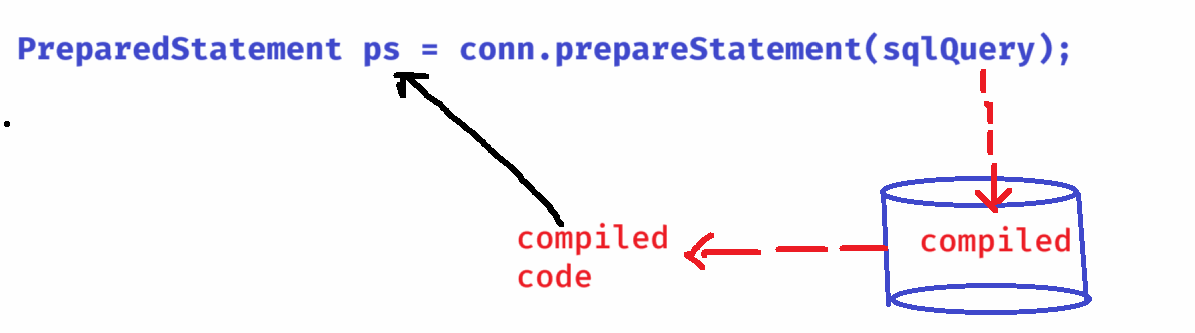
**}**

PreparedStatement interface:

* PreparedStatement is a sub interface of Statement interface.
* PreparedStatement will be used for the below reasons.

1. To execute the same query for multiple times, without re-compilation.
2. To send/receive the images from the Database.

* While creating a PreparedStatement object only, the sql query will be sent to the database for pre-compilation.
* The database will compile the query and that compiled code will be stored back into PreparedStatement object.



* While creating the PreparedStatement object, the sqlQuery will be sent to the database, only for compilation. So, it should not contain values.
* In place of values, we should put positional parameters. The symbol is ?.

ex:

String sqlQuery= “INSERT INTO STUDENT VALUES(?,?,?,?)”;

PreparedStatement ps = conn.prepareStatement(sqlQuery);

* Working with PreparedStatement involves 3 steps.

1. preparing the query.
2. parameter settings.
3. executing the query.

* we should call setXxx() methods for parameter setting.

ex:

ps.setInt(1, 101);

ps.setString(2, “John”);

ps.setInt(3, 690);

ps.setString(4, “A+”);

ps.executeUpdate();

========================================

example code:

package com.example;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.PreparedStatement;

import java.util.Scanner;

public class InsertRecords {

public static void main(String[] args) {

createRecords();

}

private static void createRecords() {

Connection conn = null;

PreparedStatement pstmt = null;

try {

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

conn = DriverManager.getConnection("jdbc:mysql://@localhost:3306/test", "root", "root" );

String query = "INSERT INTO STUDENT VALUES(?, ?, ?, ?)";

//preparing the query

pstmt = conn.prepareStatement(query);

Scanner scanner = new Scanner(System.in);

for(int i = 1; i <= 5; i++) {

System.out.println("Enter student id");

int sid = scanner.nextInt();

scanner.nextLine();

System.out.println("Enter student name");

String sname = scanner.nextLine();

System.out.println("Enter marks");

int marks = scanner.nextInt();

scanner.nextLine();

System.out.println("Enter the grade");

String grade = scanner.nextLine();

//parameter setting

pstmt.setInt(1, sid);

pstmt.setString(2, sname);

pstmt.setInt(3, marks);

pstmt.setString(4, grade);

//execute the query

pstmt.executeUpdate();

System.out.println("Recored inserted...........");

System.out.println("\*+\*\*".repeat(20));

}

} catch(Exception ex) {

ex.printStackTrace();

}

finally {

try {

if(pstmt != null)

pstmt.close();

if(conn != null)

conn.close();

} catch(Exception ex) {

ex.printStackTrace();

}

}

}

}

**CREATE TABLE ITEM(**

**ITEMID INT PRIMARY KEY,**

**ITEMNAME VARCHAR(30),**

**IMAGE BLOB**

**);**

InsertImage.java

**package com.example;**

**import java.io.FileInputStream;**

**import java.sql.Connection;**

**import java.sql.DriverManager;**

**import java.sql.PreparedStatement;**

**public class InsertImage {**

**public static void main(String[] args) {**

***insertImage*();**

**}**

**private static void insertImage() {**

**try ( Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test", "root","root");**

**PreparedStatement pstmt = conn.prepareStatement("INSERT INTO ITEM VALUES(?, ?, ?)"))**

**{**

**pstmt.setInt(1, 102);**

**pstmt.setString(2, "HP Laptop");**

**pstmt.setBinaryStream(3, new FileInputStream("C:\\Users\\WINDOWS\\Downloads\\hplaptop.jfif"));**

**pstmt.executeUpdate();**

**System.*out*.println("Record inserted...");**

**}**

**catch(Exception ex) {**

**ex.printStackTrace();**

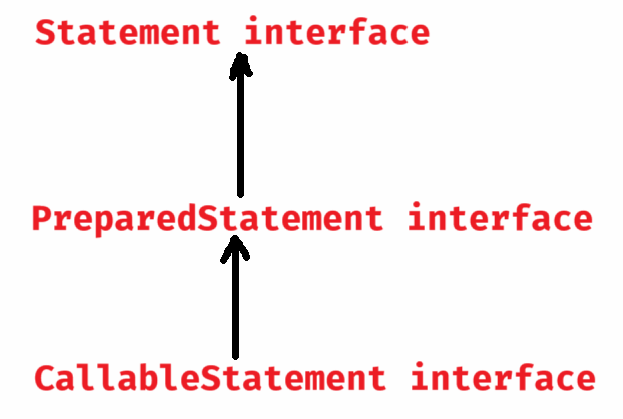
**}**

**}**

**}**

CallableStatement interface:

* CallableStatement can be used to call the procedures or the functions of the database.
* Like creating methods in a Java class, we can create procedures/functions in the datatbase to define some logic.
* If a procedure/function is created in a database, then it will be resusable across multiple Java applicatons also.
* While designing the database, if the requirement is to perform an action, not to perform any calculations and then return the value, then procedure will be created.
* If the requirement is to perform the calcualations and return the value, then function will be created.
* For example, if the task is like delete inactive users from the database from a long time then procedure will be created.
* For example, if the task is like calculate the experience of an employee based on hiredate and return the value, then function will be created.
* Sometimes, we create procedures to return multiple values, because a function can return a single value.



ex:

**CallableStatement cs = conn.prepareCall(“{call getEmpBonus(?)}”);**

**//example on CallableStatement**

**package com.example;**

**import java.sql.CallableStatement;**

**import java.sql.Connection;**

**import java.sql.DriverManager;**

**import java.sql.Types;**

**public class EmpBonus {**

**public static void main(String[] args) {**

***getBonus*(7708);**

***getBonus*(7065);**

**}**

**private static void getBonus(int empno) {**

**// TODO Auto-generated method stub**

**try ( Connection conn = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/test", "root","root");**

**CallableStatement cs = conn.prepareCall("{? = call getEmpBonus(?)}"))**

**{**

**cs.registerOutParameter(1, Types.*DECIMAL*); //OUT**

**cs.setInt(2, empno); //IN**

**cs.execute();**

**//retrieve the value**

**double bonus = cs.getDouble(1);**

**System.*out*.println("Empno : "+ empno + " , Bonus : " + bonus);**

**} catch(Exception ex) {**

**ex.printStackTrace();**

**}**

**}**

**}**

**ResultSetMetaData interface:**

* **When you read/select the data from a database, In JDBC, we use ResultSet object to store the data.**
* **If you are selecting the data from table, where you have no idea about that table structure then it would be difficult to read the data from the result set object.**
* **So, you can use ResultSetMetaData object, to find the meta data of the result set, like columns count, columns names, data types, columns size, etc..**

**ResultSetMetaData metadata = rs.getMetaData();**

* **you can call the below methods.**

1. **getColumnCount() : returns the no of column in result set.**
2. **getColumnTypeName(index): returns the data type of this col.**
3. **getColumnDisplaySize(index): returns the size of the col.**
4. **getColumnName(index): returns the column name.**

**MetaDataTest.java**

**package com.example;**

**import java.sql.Connection;**

**import java.sql.DriverManager;**

**import java.sql.PreparedStatement;**

**import java.sql.ResultSet;**

**import java.sql.ResultSetMetaData;**

**public class MetaDataTest {**

**public static void main(String[] args) {**

***fetchMetaData*();**

**}**

**private static void fetchMetaData() {**

**Connection conn = null;**

**PreparedStatement pstmt = null;**

**try {**

**Class.*forName*("com.mysql.cj.jdbc.Driver");**

**conn = DriverManager.*getConnection*("jdbc:mysql://@localhost:3306/test", "root", "root" );**

**String query = "SELECT \* FROM ORDERS";**

**//preparing the query**

**pstmt = conn.prepareStatement(query);**

**ResultSet rs = pstmt.executeQuery();**

**ResultSetMetaData rsmd = rs.getMetaData();**

**for(int i = 1; i <= rsmd.getColumnCount(); i++) {**

**System.*out*.println("Column index : " + i);**

**System.*out*.println("Column Name : " + rsmd.getColumnName(i));**

**System.*out*.println("Column Data type : " + rsmd.getColumnTypeName(i));**

**System.*out*.println("Column Size : " + rsmd.getColumnDisplaySize(i));**

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

**}**

**rs.close();**

**} catch(Exception ex) {**

**ex.printStackTrace();**

**}**

**finally {**

**try {**

**if (pstmt != null)**

**pstmt.close();**

**if (conn != null)**

**conn.close();**

**} catch(Exception ex) {**

**ex.printStackTrace();**

**}**

**}**

**}**

**}**

**JDBC Transaction:**

* **A transaction is a group of or a sequence of database operations that are executed as a single unit of work.**
* **A transaction can be ended as successful or failed.**
* **If all the operations of the unit are executed without any errors then transaction will be successful.**
* **If any one of the operations of the unit are failed, then the transaction will be failed.**
* **The program code has to commit the successful transactions and has to rollback the failure transactions.**
* **For example, in an online purchase, the sequence of operations are,**
* **1. deduct the product from the inventory**
* **2. payment**
* **3. generate the order**
* **If all the operations are executed, then transaction is success. If any operation fails, then transaction is rolled back.**
* **Transactions are 2 types.**
* **1. local transaction**
* **2. distributed transaction**
* **In a local transaction, all the operations of the unit are executed on a single database.**
* **In a distributed transaction, the operations of the unit are executed on more than one database.**
* **JDBC API can provide support for only Local transactions.**
* **For Distributed transactions, you have to spring framework.**
* **The methods of JDBC Transaction are,**
* **1.setAutoCommit(false)**
* **2. commit()**
* **3. rollback()**