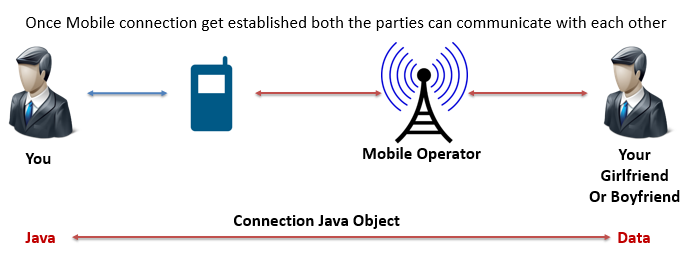
**Interface Connection**

***Explain Connection by using the below picture***



* The Connection Object is an Object representation of the **physical database connection** that can be used by a program to communicate with database.
* Within the scope of a connection, SQL queries are issued and results are returned
* We can have multiple connections to a DB
* Connection Object has methods to deal with Transactions

DatabaseMetaData mtdt = con.getMetaData();

System.*out*.println("URL in use: " + mtdt.getURL());

System.*out*.println("User name: " + mtdt.getUserName());

System.*out*.println("DBMS name: " + mtdt.getDatabaseProductName());

System.*out*.println("DBMS version: " + mtdt.getDatabaseProductVersion());

System.*out*.println("Driver name: " + mtdt.getDriverName());

System.*out*.println("Driver version: " + mtdt.getDriverVersion());

rs = mtdt.getTables( **null**, **null**, **null**, **null** );

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

System.*out*.println("tableName = "+rs.getString(3));

}

rs = mtdt.getColumns(**null**, **null**, "students", **null**);

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

System.*out*.println("columnName = "+rs.getString(4));

}

rs = mtdt.getPrimaryKeys(**null**, **null**, "students");

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

System.*out*.println("PK = "+rs.getString(4));

}

**public** **interface** MyInterface {

**void** myMethod();

}

**public** **class** MyClass1 **implements** MyInterface {

@Override

**public** **void** myMethod() {

System.*out*.println("1111111111");

}

}

**public** **class** MyClass2 **implements** MyInterface {

@Override

**public** **void** myMethod() {

System.*out*.println("222222222");

}

}

**public** **class** UtilityClass {

**public** **static** MyInterface getMyClassInstance(**int** i) {

/\*

\* Some Business Logic

\*/

**if**(i==1) {

**return** **new** MyClass1();

}**else**{

**return** **new** MyClass2();

}

}//End of getMyClassInstance()

}//End of Class

**public** **class** MainMethodClass {

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

MyClass1 class1 = **new** MyClass1();

class1.myMethod();

MyClass2 class2 = **new** MyClass2();

class2.myMethod();

MyInterface myInterface = **null**;

myInterface = **new** MyClass1();

myInterface = **new** MyClass2();

myInterface.myMethod();

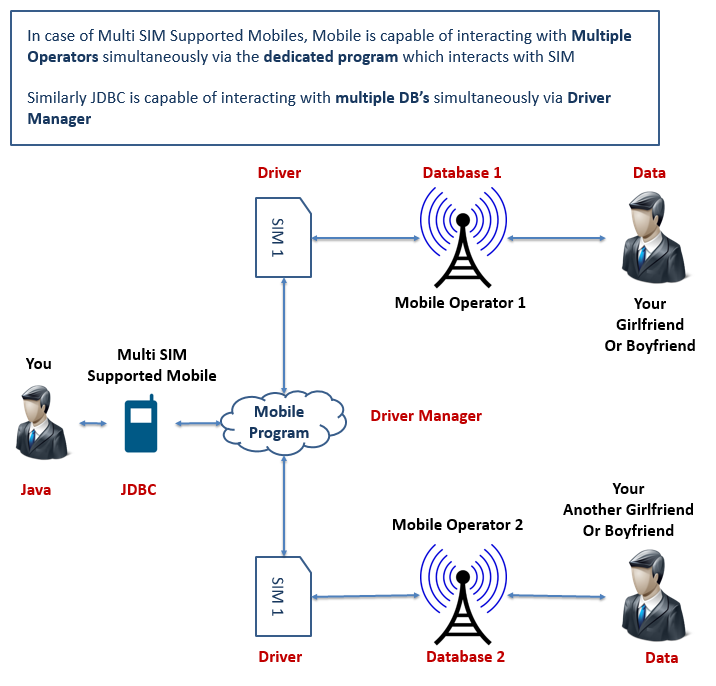
**int** i = 1;

myInterface = UtilityClass.*getMyClassInstance*(i);

myInterface.myMethod();

}//End of Class

}//End of Interface

****

**Overview of SQL:** SQL commands are divided into 2 categories

1. **Data Definition Language (DDL) commands**. DDL commands are used to create and destroy databases / tables / other database objects such as views and indexes. These commands are primarily used by DB administrators. Common DDL Commands : CREATE TABLE, DROP TABLE, ALTER TABLE
2. **Data Manipulation Language (DML) commands**. DML commands deal with either retrieving or modifying DB data. Common DML Commands (These are referred as CRUD / CURD operations) : SELECT, INSERT, DELETE, UPDATE

At any point of time DB Server gives **ONLY two type of Results**

1. Integer Counts (DDL, Insert, Update, Delete)
2. DB Results (Only Select Statement)
3. **int[] executeBatch() :** Execute Batch groups the related SQL statements into a batch and submit them with one call to the database. If all commands execute successfully, returns an array of update counts.

JDBC statements also supports following methods which are primarily used for Batch processing along with the above executeBatch() method.

1. **void addBatch() :** we add individual Statement or PreparedStatement or CallableStatement to batch using this method
2. **void clearBatch() :**  this method removes all the statements present in the batch. However, you cannot selectively choose which statement to remove from batch.

Statement stmt = con.createStatement();

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

int count = stmt.executeUpdate("insert into branch values(2,'test')");

System.*out*.println("No. Rows Affected are: "+count);

**boolean** isExecuted = stmt.execute("create table branch ( branchid int(50) not null,

branchname varchar(50), primary key (branchid))");

//boolean isExecuted = stmt.execute("insert into branch values(2,'test')");

//boolean isExecuted = stmt.execute("select \* from branch");

**if**(isExecuted)

{

System.*out*.println("Result has ResultSet object");

rs = stmt.getResultSet();

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

System.*out*.println("Reg No.= " + rs.getString("regno")

+" First Name = " + rs.getString("firstname")

+" Middle Name = "+ rs.getString("middlename")

+" Last Name = "+ rs.getString("lastname"));

}

}**else**{

System.*out*.println("Result has NO ResultSet object but has Update Counts or there are

no results (in case of DDL Commands)");

System.*out*.println(stmt.getUpdateCount());

}

PreparedStatement pstmt = con.prepareStatement("select \* from students where regno=?");

pstmt.setInt(1, 1);

ResultSet rs=pstmt.executeQuery();

**public** **class** PreparedStatementSample

{

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

{

Students studentsObj1 = **new** Students(3, "ABC", "XYZ");

Students studentsObj2 = **new** Students(4, "123", "456");

Students studentsObj3 = **new** Students(5, "qwerty", "asdfg");

Students studentsObj4 = **new** Students(6, "zxcv", "zxcv");

ArrayList<Students> dataList = **new** ArrayList<Students>();

dataList.add(studentsObj1);

dataList.add(studentsObj2);

dataList.add(studentsObj3);

dataList.add(studentsObj4);

*processData*(dataList);

}//End of Main Method

**public** **static** **void** processData(ArrayList<Students> dataList)

{

//Initialize the Variables

//1. Load the JDBC Driver

//2. Get the Connection from DB via Driver

//3. Issue SQL Queries to DB through Connection Object

pstmt = con.prepareStatement(

"insert into students(regno,firstname,lastname) values (?, ?, ?) ");

**for**(Students data : dataList)

{

pstmt.setInt(1, data.getRegno());

pstmt.setString(2, data.getFirstname());

pstmt.setString(3, data.getLastname());

//4. Process the ResultSet returned by SQL Statements

**int** count = pstmt.executeUpdate();

System.*out*.println("No. of Rows Affected : "+count);

}

//5. Close all the JDBC Objects

}//End of processData()

}//End of Class

**Here are examples of host database types which Java can convert to with a function.**

|  |  |
| --- | --- |
| **Oracle Datatype** | **setXXX()** |
| CHAR | setString() |
| VARCHAR2 | setString() |
| INTEGER | setInt() |
| FLOAT | setDouble() |
|  |  |

***Explain the PreparedStatemets Performance by using below points***

**How does PreparedStatements increase the performance?**

* When a DB receives a SQL statement, the DB engine first parses the statement and looks for any syntax errors.
* Once the statement is parsed DB engine starts preparing the Execution Plan. i.e. It figure out the most efficient way to execute the SQL statement. This step is **computationally quite expensive**. The database checks what indexes, if any, can help, or whether it should do a full read of all rows in a table. Databases use statistics on the data to figure out what is the best way. Once the Execution Plan is prepared DB engine uses this plan for executing the SQL statement.
* Once the Execution Plan is prepared, DB caches this plan. This cache uses the **whole statement** itself as a key and the Execution Plan as a value corresponding to this key. This allows the DB engine to reuse the Execution Plans.
* If we send the same SQL statement again, then DB engine will reuse the Execution Plan generated for the first statement. This saves lot of server resources along with the time for generating Execution Plan.

**For example**,

If we sent the database a statement such as "select A,B from XYZ where C = 2", then the computed Execution Plan is cached. If we send the same statement later, the DB will reuse the previous Execution Plan to execute the query & because of this, time take to execute this query & getting results are much faster compared to first time.

However, that the entire statement is the key. If we later sent the statement "select A, B from XYZ where C = 3", it would not find the Execution Plan in cache. This is because the "C = 3" is different from the cached plan "C = 2"

* In case of prepared statements SQL statements will not have the query values instead they will have query parameters (?), which are set at runtime. Hence DB engine will cache the generated Execution Plan by using SQL statements with query parameters (?) as a key. Because of this reason **parameterized Prepared Statements (?)** are performance oriented.

The first command you see is DELIMITER //. This command is not related to the stored procedure. DELIMITER statement is used to change the standard delimiter (semicolon) to another, in this case the delimiter is changed to //, so you can have multiple SQL statements inside stored procedure which can separate by the semicolon. After the END keyword we use delimiter // to show the end of the stored procedure. The last command changes the delimiter back to the standard one (semicolon).

Three types of parameters exist in Stored Procedures: IN, OUT, and INOUT

**IN :** A parameter whose value is set at runtime using setXXX() methods.

**OUT :** A parameter whose value is returned by Stored Procedure.

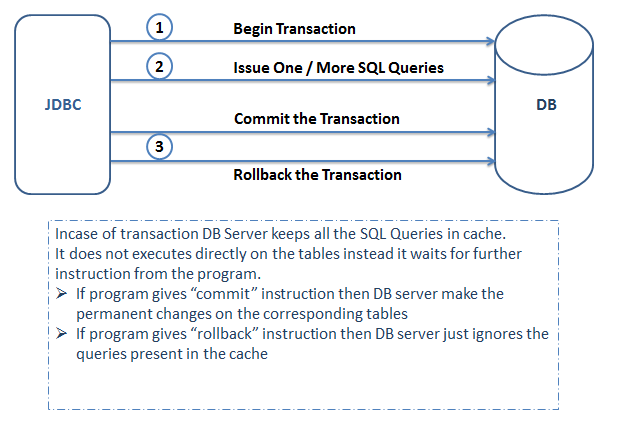
You retrieve values from the OUT parameters using getXXX() methods.

**INOUT :** A parameter that is used for both input and output.

setXXX() methods are used to setting the input value &

getXXX() methods are used to retrieve the values.

* Result Sets are produced by **Select SQL Query** or certain Metadata method calls

****

1. Calling rollback() is mandatory when if it is a pooled connection. It will reset the transactional state of the connection. Not calling rollback() may lead that the next lease of the pooled connection will still have the (succesful) queries of the previous transaction in its memory.

**Java Bean:**

If a Java class has,

* + public default constructor (this allows easy instantiation)
  + private class level properties
  + public getters, setters (this allows easy inspection and updating of an object state)
  + and does not have any other methods
  + Implements Serializable interface (this allows applications to reliably save, store, and restore the objects state in a fashion that is independent of the VM and platform)

then it is called a Java Bean

**Data Transfer Object (DTO) Design Pattern**

Data transfer object (DTO), formerly known as value objects or VO, is a design pattern that helps us the transfer the data between one program to another program in the form of Java Object using Java Bean

**public** **class** JDBCTransactionExample

{

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

{

/\*

\* Capture Data From Screen simulation

\*/

StudentInfoBean studentInfo = **new** StudentInfoBean();

studentInfo.setRegno(16);

studentInfo.setfName("Karishma");

studentInfo.setmName("NA");

studentInfo.setlName("Kapoor");

studentInfo.setGfName("Saif");

studentInfo.setGmName("Ali");

studentInfo.setGlName("Khan");

*insertData*(studentInfo);

}

**public** **static** **void** insertData(StudentInfoBean studnetInfo)

{

Connection con = **null**;

PreparedStatement pstmt1 = **null**;

PreparedStatement pstmt2 = **null**;

**try**

{

//1. Load the Driver

//2. Get the DB Connection via Driver

/\*

\* I. Begin the Transaction

\*/

con.setAutoCommit(**false**);

/\*

\* II. Issue one or more SQL Queries

\*/

//3. Issue SQL Queries

String query1 = "insert into students\_info values(?,?,?,?)";

String query2 = "insert into gaurdian\_info values(?,?,?,?)";

pstmt1 = con.prepareStatement(query1);

pstmt1.setInt(1, studnetInfo.getRegno());

pstmt1.setString(2, studnetInfo.getfName());

pstmt1.setString(3, studnetInfo.getmName());

pstmt1.setString(4, studnetInfo.getlName());

**int** count1 = pstmt1.executeUpdate();

**if**(count1 > 0)

{

System.*out*.println("Insert data into Students\_info is

successful. Count = "+count1);

/\*

\* Error Simulation

\*/

//int i = 100/0;

pstmt2 = con.prepareStatement(query2);

pstmt2.setInt(1, studnetInfo.getRegno());

pstmt2.setString(2, studnetInfo.getGfName());

pstmt2.setString(3, studnetInfo.getGmName());

pstmt2.setString(4, studnetInfo.getGlName());

**int** count2 = pstmt2.executeUpdate();

System.*out*.println("Insert data into gaurdian\_info is

successful. Count = "+count2);

}

/\*

\* III. No Error...Comit the Transaction

\*/

con.commit();

} **catch** (Exception e) {

/\*

\* IV. Error Occurred !!! Rollback the transaction

\*/

**try** {

con.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

e.printStackTrace();

} **finally**{

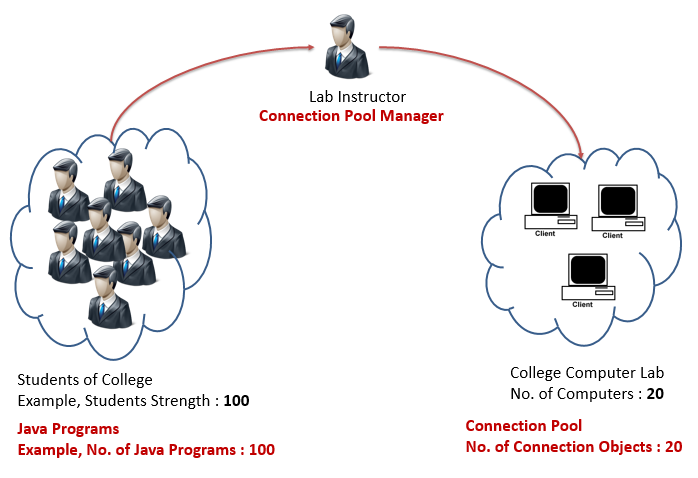
//5. Close All JDBC Objects

}//End of try-catch block

}//End of insertData

}//End of Class

***Explain ConnectionPool by using the below picture***

****

* With Connection Pooling the pool of Connection objects are created only once & whenever java program needs to interact with DB rather than creating Connection object it can make use of the connection object present in the pool. Once the program completes the DB interaction, it can return connection object back to pool

2. At any point of time, connection pool contains number of connections which are “greater than or equal to pool size”

**ConnectionPoolManager Program**

**class** ConnectionPoolManager

{

**private** **final** String DRIVER\_CLASS = "com.mysql.jdbc.Driver";

**private** **final** String DB\_URL = "jdbc:mysql://localhost:3306/vtu";

**private** **final** String DB\_USER\_NAME = "j2ee";

**private** **final** String DB\_PASSWORD = "j2ee";

**private** **final** **int** MAX\_POOL\_SIZE = 5;

**private** Vector<Connection> connectionPool

= **new** Vector<Connection>();

**private** **static** ConnectionPoolManager *instance* = **null**;

**private** ConnectionPoolManager()

**throws** Exception

{

Class.*forName*(DRIVER\_CLASS);

initializeConnectionPool();

}

**public** **static** ConnectionPoolManager getInstance()

**throws** Exception

{

**if**(*instance* == **null**) {

*instance* = **new** ConnectionPoolManager();

}

**return** *instance*;

}

**private** **void** initializeConnectionPool()

**throws** Exception

{

System.*out*.println("Creating the Connection Pool for the First

Time with Pool Size : "+MAX\_POOL\_SIZE);

**for** (**int** i = 0; i < MAX\_POOL\_SIZE; i++) {

connectionPool.addElement(createNewConnectionForPool());

}

System.*out*.println("Successfully Created the Connection Pool");

}

// Creating a connection

**private** Connection createNewConnectionForPool()

**throws** SQLException

{

Connection con = **null**;

con = DriverManager.*getConnection*

(DB\_URL, DB\_USER\_NAME, DB\_PASSWORD);

**if** (con != **null**) {

System.*out*.println("Got the Connection from DB to add it to Pool");

}

**return** con;

}

**public** **synchronized** Connection getConnectionFromPool()

**throws** Exception

{

System.*out*.println(

"Giving the Connection to client Program from Pool");

System.*out*.println("Before giving the Connection to Client

Program, Pool Size is : "+connectionPool.size());

Connection con = **null**;

// Check if there is a connection available.

// There are times when all the connections in the pool

// may be used up

**if** (connectionPool.size() > 0) {

con = (Connection) connectionPool.firstElement();

connectionPool.removeElementAt(0);

} **else** {

con = createNewConnectionForPool();

}

System.*out*.println("After giving the Connection to Client Program,

Pool Size is : "+connectionPool.size());

**return** con;

}

**public** **synchronized** **void** returnConnectionToPool(Connection connection)

{

System.*out*.println("Returing the Connection back to Pool");

System.*out*.println("Before returning the Connection to Pool, Pool

Size is : "+connectionPool.size());

connectionPool.addElement(connection);

System.*out*.println("After returning the Connection to Pool, Pool Size is :

"+connectionPool.size());

}

}// End of Class

**Connection Pool Client Program**

**public** **class** ConnectionPoolSample

{

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

{

Connection con1 = **null**;

Statement stmt = **null**;

ResultSet rs = **null**;

Connection con2 = **null**;

Connection con3 = **null**;

ConnectionPoolManager pool = **null**;

**try**

{

System.*out*.println("Getting Connection from Pool");

pool = ConnectionPoolManager.*getInstance*();

con1 = pool.getConnectionFromPool();

System.*out*.println("Got the Connection from Pool");

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

System.*out*.println("Using the Connection Interact with DB");

stmt = con1.createStatement();

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

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

{

System.*out*.println("Reg No.= " + rs.getString("regno")

+" First Name = " + rs.getString("firstname")

+" Middle Name = "+ rs.getString("middlename")

+" Last Name = "+ rs.getString("lastname"));

}

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

System.*out*.println("Testing the Connection Pool");

con2 = pool.getConnectionFromPool();

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

con3 = pool.getConnectionFromPool();

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

}**catch**(Exception e){

e.printStackTrace();

}**finally**{

**try**

{

stmt.close();

rs.close();

System.*out*.println("Returning the First connection to Pool");

pool.returnConnectionToPool(con1);

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

System.*out*.println("Returning the Second connection to Pool");

pool.returnConnectionToPool(con2);

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

System.*out*.println("Returning the Third connection to Pool");

pool.returnConnectionToPool(con3);

} **catch** (SQLException e) {

e.printStackTrace();

}

}//End of Try Catch Block

}//End of Main Method

}//End of Class

**Assignment II:**

Move all the DB interaction methods to one class by name “StudentsDAO”

Make use of connection pool to get the connection & use it for interacting with DB.

**Batch Processing: Example: Brining items from Provision Store One by One / brining all items as a whole**

* Batch Processing allows you to group related SQL statements into a batch and submit them with one call to DB. It is used with Data Manipulation Commands (Update, Insert or Delete)
* When you send several SQL statements to the database at once, it reduces the amount of communication overhead, thereby improving performance
* We can use the DatabaseMetaData.supportsBatchUpdates() method to determine if the target database supports batch update processing (**This point is not Required**)
* Following steps are followed to handle Batch Processing in JDBC

1. Add individual Statement or PreparedStatement or CallableStatement to batch using addBatch() method of JDBC Statement
2. Using executeBatch() method start the execution of batch. The executeBatch() returns an array of integers, and each element of the array represents the count for the respective DML commands
3. For example, if there were 20 operations in the batch, the first 13 succeeded, and the 14th generated an exception, then the update counts array will have 13 elements, containing actual update counts of the successful operations. You can either commit or roll back the successful operations in this situation, as you prefer.
4. Use clearBatch() method to remove all the statements present in the batch. However, you cannot selectively choose which statement to remove from batch.

**Batch Processing Example for PreparedStatement:-**

// Create SQL statement

String SQL = "INSERT INTO Employees (id, first, last, age) VALUES(?, ?, ?, ?)";

// Create PrepareStatement object

PreparedStatemen pstmt = conn.prepareStatement(SQL);

// Set the variables

pstmt.setInt( 1, 400 );

pstmt.setString( 2, "Pappu" );

pstmt.setString( 3, "Singh" );

pstmt.setInt( 4, 33 );

// Add it to the batch

pstmt.addBatch();

// Set the variables

pstmt.setInt( 1, 401 );

pstmt.setString( 2, "Pawan" );

pstmt.setString( 3, "Singh" );

pstmt.setInt( 4, 31 );

// Add it to the batch

pstmt.addBatch();

//Create an int[] to hold returned values

int[] count = stmt.executeBatch()

**Batch Processing Example for Statement:-**

// Create statement object

Statement stmt = conn.createStatement();

// Create SQL statement & Add above SQL statement in the batch.

String SQL = "INSERT INTO Employees (id, first, last, age) VALUES(200,'Zia', 'Ali', 30)";

stmt.addBatch(SQL);

String SQL = "INSERT INTO Employees (id, first, last, age) VALUES(201,'Raj', 'Kumar', 35)";

stmt.addBatch(SQL);

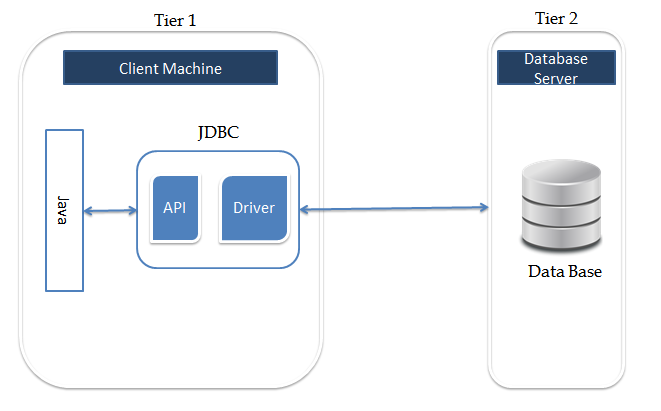
// Create an int[] to hold returned values

int[] count = stmt.executeBatch();

**Batch Processing Example for CallableStatement:-**

*Same as PreparedStatement*

**Revisit J2EE 2-Tier Architecture**

****

**In any given area on particular road there are N number of houses exists & we use house number to identify a particular house in that road. Similarly Port number is the unique number that identifies the specific software program on the server hardware**

If the driver is loaded using Class.forName("Driver Class") & When the method getConnection() is called, the DriverManager will attempt to locate a suitable driver from amongst those loaded at initialization and those loaded explicitly using the same classloader as the current program / application. Class.forname method simply loads the class using the JVM's bootstrap class loader. If this class is a jdbc driver class then the driver upon loading registers itself using DriverManager.registerDriver method. When DriverManager.getConnection is called then the driver class corresponding to the connection URL is called. This is all managed by the DriverManager.

**Interview Question:** What are different ways to create the Objects in Java ?

Have a look at the Cloning Concept

**public** **class** MyClass

{

**public** **int** myInt;

**public** String myString;

/\*

\* Generate Getters & Setters

\*/

}//End of MyClass

**public** **class** DifferentWaysToCreateObject

{

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

{

/\*

\* First Way : Using new Operator

\*/

Bean bean = **new** Bean();

System.*out*.println("First Object - Int Value : "+bean.getMyInt());

System.*out*.println("First Object - String Value : "+bean.getMyString());

System.*out*.println("Setting the values");

bean.setMyInt(200);

bean.setMyString("Hello");

System.*out*.println("First Object - Int Value : "+bean.getMyInt());

System.*out*.println("First Object - String Value : "+bean.getMyString());

/\*

\* Using Class.forName()

\*/

**try**

{

/\*

\* Returns the Class object associated with the class or interface

\* with the given string name. A call to forName("X") causes the

\* class named X to be initialized.

\*/

Class beanClass = Class.*forName*("com.jspiders.myjava.beans.Bean");

/\*

\* Creates a new instance of the class represented by this Class

\* object. The class is instantiated as if by a new expression with

\* an empty argument list. The class is initialized if it has not

\* already been initialized. Returns: a newly allocated instance of

\* the class represented by this object.

\*/

Bean bean2 = (Bean) beanClass.newInstance();

System.*out*.println("Second Object-Int Value : "+bean2.getMyInt());

System.*out*.println("Second Object-String Value : "+bean2.getMyString());

System.*out*.println("Setting the values");

bean2.setMyInt(200);

bean2.setMyString("Hello");

System.*out*.println("Second Object - Int Value : "+bean2.getMyInt());

System.*out*.println("Second Object-String Value: "+bean2.getMyString());

} **catch** (Exception e) {

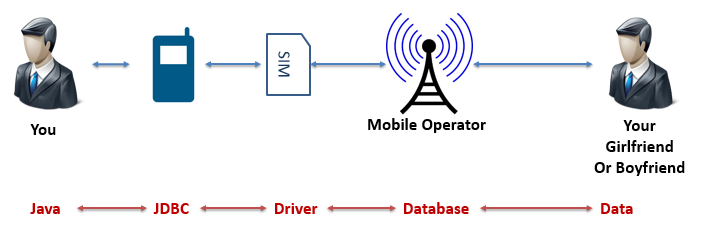
e.printStackTrace();

}

}//End of Main

}//End of Class

***Explain JDBC by using the below picture***

****

**MyFirstJDBCProgram.java :**

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

{

Connection con = **null**;

Statement stmt = **null**;

ResultSet rs = **null**;

**try**

{

//1. Load the JDBC Driver

String driverClass = "com.mysql.jdbc.Driver";

Class.forName(driverClass);

//2. Get the Connection from DB via Driver

String connectionURL = "jdbc:mysql://localhost:3306/vtu?user=j2ee&password=j2ee";

con = DriverManager.*getConnection* (connectionURL);

//3. Issue SQL Queries to DB through Connection Object

stmt = con.createStatement();

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

//4. Process the Result

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

{

System.*out*.println("Reg No.= " + rs.getString("regno")

+" First Name = " + rs.getString("firstname")

+" Middle Name = "+ rs.getString("middlename")

+" Last Name = "+ rs.getString("lastname"));

}

}**catch**(Exception e){

e.printStackTrace();

}**finally** {

//5. Close all the JDBC Objects

**try** {

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

} **catch** (SQLException e) {

e.printStackTrace();

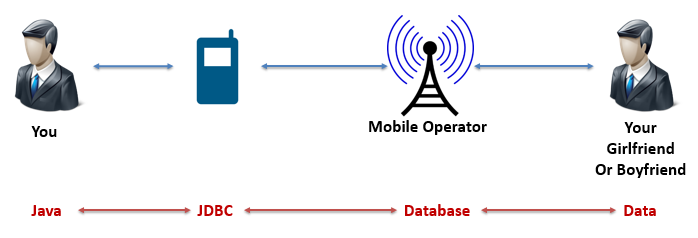
}

}//End of Try & Catch Block

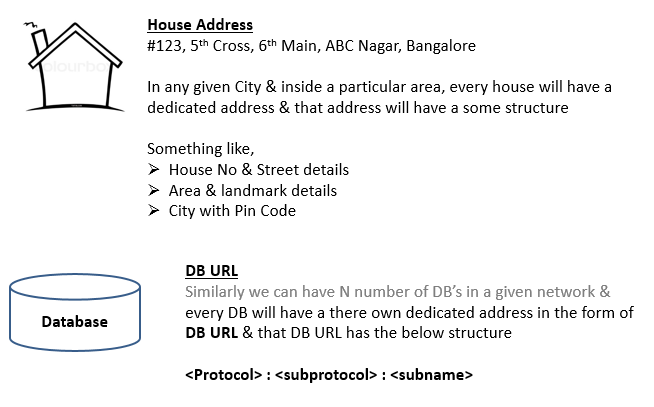
}//End of Main Method

}//End of Class

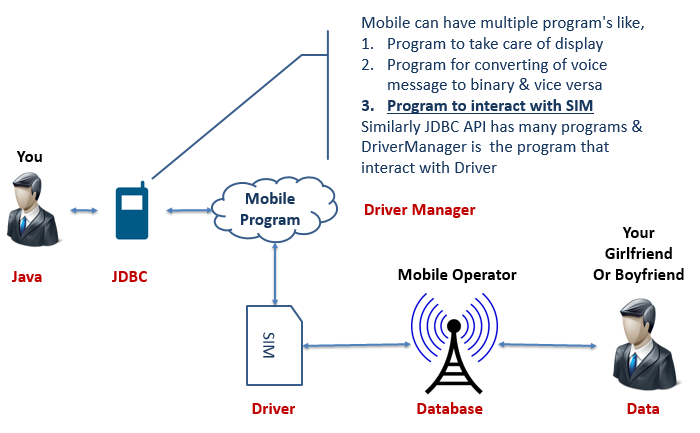
***Explain JDBC by using the below picture***

****

***Explain URL by using the below picture***

****

***Explain DriverManager by using the below picture***

****

1. String query = "{ CALL GetAllStudents() }";

cstmt = con.prepareCall(query);

rs = cstmt.executeQuery();

2. **int** regNo = Integer.*parseInt*(args[0]);

String query = "{ CALL GetStudentInfo(?) }";

cstmt = con.prepareCall(query);

cstmt.setInt(1, regNo);

rs = cstmt.executeQuery();

3. **int** regNo = Integer.*parseInt*(args[0]);

String firstNm = args[1];

String middleNm = args[2];

String lastNm = args[3];

String query = "{ CALL StudentsUpsert(?, ?, ?, ?) }";

cstmt = con.prepareCall(query);

cstmt.setInt(1, regNo);

cstmt.setString(2, firstNm);

cstmt.setString(3, middleNm);

cstmt.setString(4, lastNm);

rs = cstmt.executeQuery();

**JDBC Statements Summarized:**

**1. Statement**

Statement stmt = con.createStatement();

ResultSet rs = stmt.executeQuery("<Select Static SQL Query>");

int count = stmt.executeUpdate("<DDL, Delete, Insert, Update Static SQL

Query>");

boolean isDbResults = stmt.execute(“<DDL, DML Static SQL Query >”);

**2. PreparedStatement**

PreparedStatement pstmt= con.prepareStatement("<Query with Question Mark>");

pstmt.setXXX(<Question Mark Position>, <Dynamic Value>);

//For Select SQL Query

ResultSet rs = pstmt.executeQuery();

//DDL, Delete, Insert, Update SQL Query

int count = pstmt.executeUpdate();

//Any SQL Query (DDL / DML)

boolean isDbResults = pstmt.execute();

**3. CallableStatement**

//S.P without any input arguments

String query = “{ call S.P.Name() }”;

//S.P with hard coded input arguments

String query = “{ call S.P.Name(10, ‘ABC’) }”;

//S.P with dynamic input arguments

String query = “{ call S.P.Name(?) }”;

//S.P with combination of hard coded input arguments & dynamic input arguments

String query = “{ call S.P.Name(?, ‘ABC’) }”;

CallableStatement cstmt = con.prepareCall()

//If Query contains Question Marks

cstmt.setXXX(<Question Mark Position>, <Dynamic Value>);

//If S.P Returns ONLY DB Results

ResultSet rs = cstmt.executeQuery();

//If S.P Returns ONLY Update Counts

int count = cstmt.executeUpdate();

//If S.P Returns either DB Results or Update Counts based on the logic it has

boolean isDbResults = cstmt.execute();

**Assignment III:**

* Write a JDBC Program which takes the register number & password information via command line arguments & update “students\_otherinfo” table with the input password information against the incoming regno

Note:

1. Create a separate method which has the following structure & this method should have DB interaction logic (step 1 to 5) & based on the result it should return true or false

public static boolean changePassword(int regno, String password)

1. Invoke this method in the main method by passing the command line arguments values (after validation) & based on the boolean value print the corresponding message in the console

The ResultSet.getXXX methods are the only way to retrieve data from a ResultSet object, which means that you have to make a method call for each column of a row.

Q: How do you register a driver?

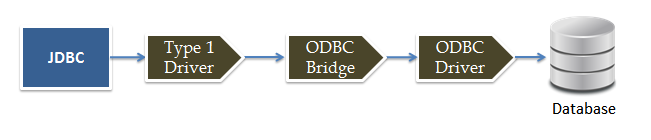
A: There are 2 approaches for registering the Driver:

Class.forName(): This method dynamically loads the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

DriverManager.registerDriver(): This static method is used in case you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

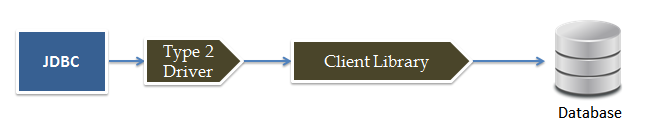
**Type 1 Driver: JDBC-ODBC (Open Data Base Connectivity) Bridge**

* This driver makes use of ODBC Bridge & ODBC Driver to connect to the database.
* The ODBC Bridge & ODBC Driver needs to be installed on the client machine hence, this driver is **platform-dependent**
* **We cannot achieve high performance using this driver**. Since the calls have to go through the JDBC to the ODBC driver & then to DB



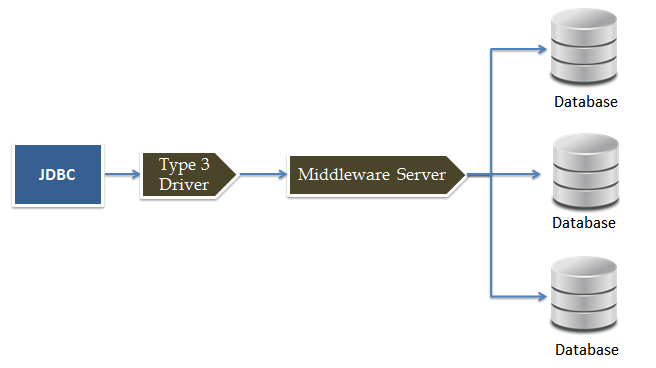
**Type 2 Driver: Native-API Driver**

* This driver makes use of client-side libraries provided by DB Vendor to connect to the database.
* The client-side libraries needs to be installed on the client machine hence, this driver is also **platform-dependent**
* **We cannot achieve high performance using this driver**. Also not all databases provide client side libraries

****

**Type 3 Driver: Network-Protocol Driver**

* It’s a **Pure Java Driver** makes use of a Middleware Server to interact with database.
* This driver is platform-independent
* This driver can be used for interacting with multiple databases but it depends on the number of databases the Middleware Server has been configured to support.
* **We cannot achieve performance using this driver**. Since JDBC calls have to go through Middleware server.



**Type 4 Driver: Native-Protocol Driver**

* This driver also known as the “**Direct to Database Pure Java Driver**” which directly interacts with DB
* This driver is platform-independent
* **We can achieve highest performance using this driver**. Since there is no intermediate layer between Driver & DB.
* The main disadvantage of this Driver is, it is DB dependent
* This driver provides better performance than the type 1 and type 2 drivers as it does not have the overhead of conversion of calls into ODBC or database API calls. Unlike the type 3 drivers, it does not need associated software to work.

****

**JDBC Driver’s summarized:**