**Introduction**

* Introduce yourself
* Explain about your teaching style
* Ask them to be punctual about timings. Tell them If they make use of you they can learn many things like Design Patterns & Java Mail & even some of the Core Java topics etc
* Explain the importance of J2EE

FY2012 ====>  **2, 30, 000 IT jobs** opened (Excluding BPO / IT IS jobs) in India

VTU Intake Capacity ====> **208** colleges affiliated

67, 100 UG Students

12, 666 PG students

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

**79, 766 Total Students**

**70K Pass outs**

**50K Valid Job Aspirants**

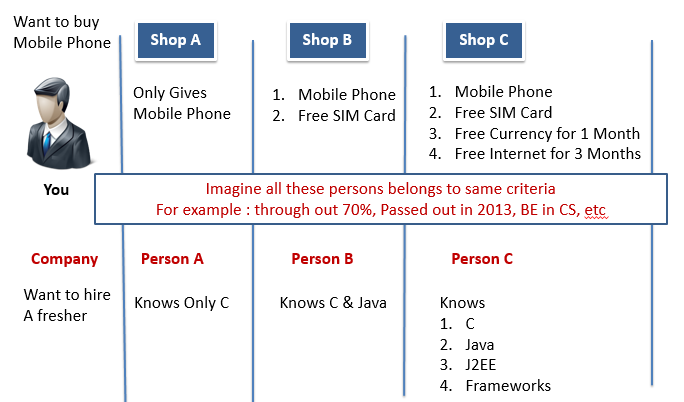
So it means that **per college on an average** **240 Valid Job Aspirants get passed per year. Let make it as 200**

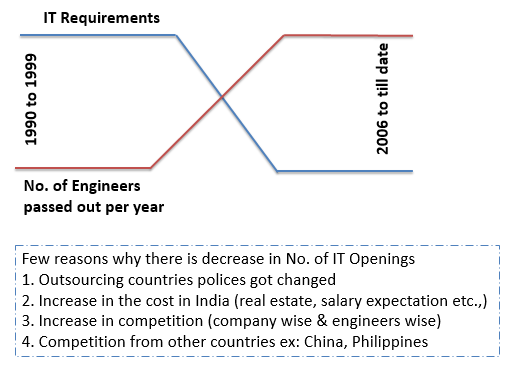
Total Number of Engineering colleges in India ===> 3, 380

So ( 3380 X 200 ) = 6, 76, 000 Valid Job Aspirants get passed per year in India

So even if we leave 1, 76, 000 there are **5, 00, 000** Valid Job Aspirants present across India which is double the number of openings in IT

**Along with Internal Competition, we are also facing the challenges of competitiveness in the globalized world, particularly from countries like China and Philippines.**

****



* Tell them about the 100% - 90% come to only getting the job. But if u know concepts then only u get the job.

|  |  |
| --- | --- |
|  | **Category 1:** These students are Proactive, Don’t miss the class, actively do the assignments & their intension is to get learn the concepts, clear the interview & get Job. **I need not worry about them. Because with right attitude they definitely reach to right level**  **Category 2:** These students join to institute with wrong intension (may be his / her BF / GF would have joined the institute ☺ or Parents would have forced them to join or may for some other reason). **I need not worry about them as well. Because their intension itself is wrong**  **Category 3:** These students join to institute just to get job. They think that if we join to this institute we get the job. **I am worried about this category. Their intension is right but there is approach is wrong** |

Probably someone who had done course here & got job would have told you that,

* **I joined this institute for j2ee course**
* This institute is really good with respect to staff & teachers
* Teachers helped to learn the concepts
* I also worked hard & did all the assignments
* That helped me to clear the interview
* **Hence I got the job**

But your mind catches the two things the first (joined the institute) & the last (got job) but forget the rest (i.e. learn & cleared the interview)

So you think, Join the institute & get a job. But actually it does not going to work like that. We (institute + teachers) helps you to get the necessary knowledge required to clear the interview & provides interview opportunities.

So to get a job you have to clear the interview, to clear the interview you have to learn the concepts & practise them & that helps you to get the job.

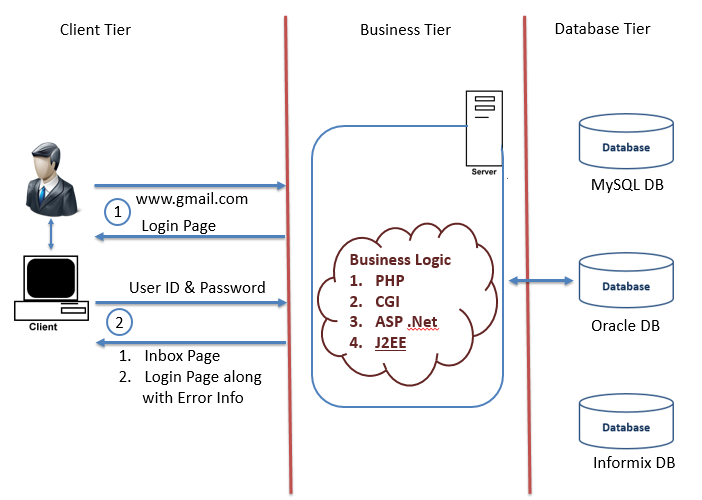
**Please Note: Just by joining to this institute & simply attending the course does not help you at all.**

Prerequisite:

* Should have knowledge of Java
* Good to have SQL Knowledge
* Good to have Eclipse IDE Knowledge
* Good to have HTML knowledge

**J2EE Introduction (Explain using the below picture & then explain about J2EE):**

Start explaining about Gmail, Facebook & FlipCart application (how to access these applications, behaviour, functionality, and necessary steps involved, supportive tolls required like, internet, and browser)

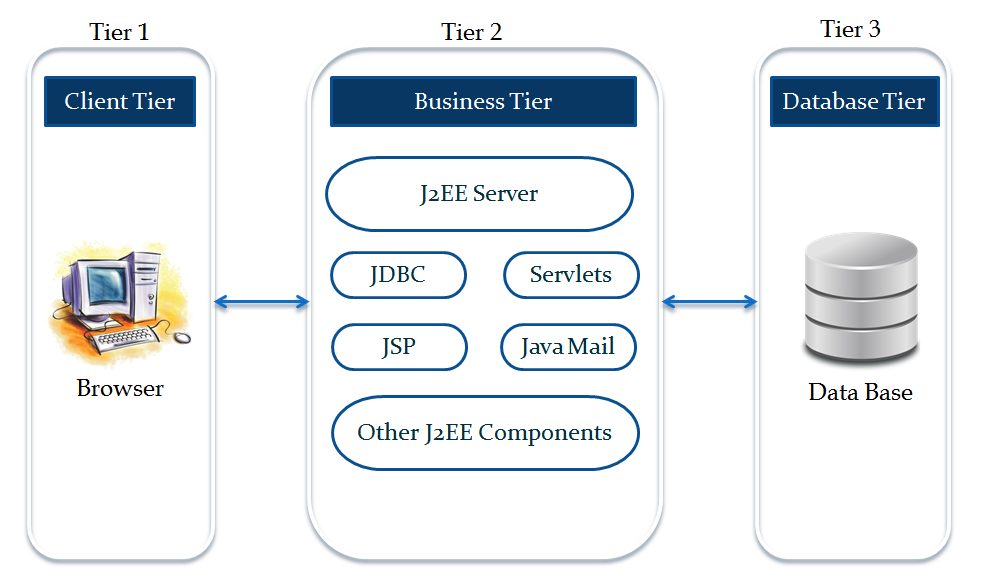
****

**J2EE / JEE**

Java 2 Platform Enterprise Edition (JEE) is a Platform-Independent, Java-Centric environment for developing Enterprise Web Applications. The J2EE platform consists of a set of services, APIs, and protocols that helps us for developing multitier, Web-based enterprise applications.

**Interview Question:** Validate the Statement : JDK is Platform Dependent ? Yes / No

**J2EE 3-Tier Architecture**



**Major Elements of J2EE**

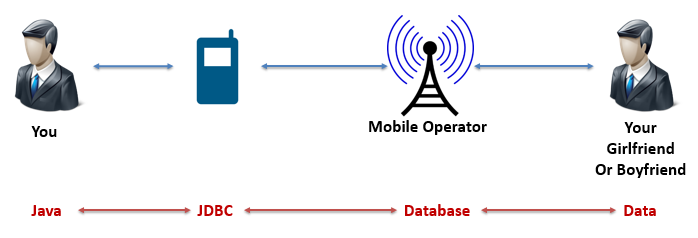
**JDBC :** An API that provides connectivity between databases and the J2EE platform

**Servlets** : A Java program, used to generate dynamic content that executes on a web server

**JSP** : A technology used to return dynamic content to a client, typically a web browser

**Introduction to JDBC**

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

****

* ***Explain about installing the MySQL Server***
* ***Give an overview on the SQL Commands using the below notes***
* ***Explain them about Creating the Database, Table & inserting some data into it***

**JDBC Prerequisites:**

1. Install the DB Server (MySQL)
2. Create a DB (by name COLLEGE / INSTITUTE)
3. Create a table in the above DB (by name STUDENTS)
4. Insert some records in to the above table

**DB Queries:**

**CREATE** DATABASE college;

**CREATE** **TABLE** students

( regno **INT**(50) **NOT** **NULL**,

firstname **VARCHAR**(50),

middlename **VARCHAR**(50) **DEFAULT** ‘Not Avilable’,

lastname **VARCHAR**(50),

**PRIMARY** **KEY** (regno)

);

**INSERT** **INTO** students **VALUES** (1, ‘XYZ’, ‘NA’, ‘ABC’);

**INSERT** **INTO** students (regno, firstname, lastname) **VALUES** (2, ‘Praveen’, ‘D’);

**JDBC**

* Java Data Base Connectivity is a Java API that provides connectivity between database and the J2EE platform.
* If you have a J2EE Web Application & it has a DB, then you have to interact with DB to Read, Update, Delete & Insert data. **In Java, JDBC is the One & Only way to interact with DB**.

**Advantages of JDBC**

Few reasons why we need JDBC are:

1. JDBC is NOT DB Dependent
2. Using JDBC we can interact with Multiple DB simultaneously
3. Since it is built upon Java it eventually inherit the advantages of Java
4. JDBC supports Stored Procedures
5. You can achieve High Performance by using JDBC (using CallableStatements, PreparedStatements, etc)

**Interview Question:** What are the advantages we get with Java / Why Java?

**Necessary Steps to work with JDBC**

1. Load the JDBC **Driver**
2. Get the **DB Connection** via **Driver**
3. Issue **SQL Queries** to via **DB Connection**
4. Process the **Results** returned by **SQL Statements**
5. Close all the **JDBC Objects**

**Sample JDBC Program:**

**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();

}

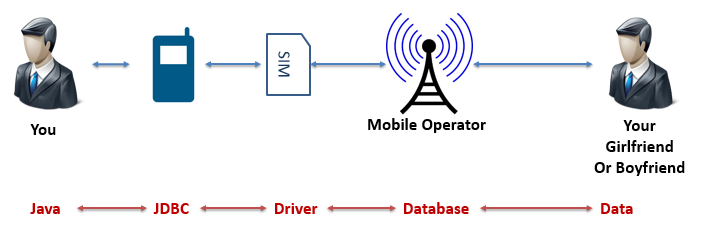
}//End of Try & Catch Block

}//End of Main Method

}//End of Class

**Drivers**

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

****

**Another Example:** Necessary steps to access the Gmail account, 1. Gmail Account 2. Internet Connection

***3.*** ***Browser: Must Require; Additional software component required to access Gmail*** *4.*Access the Mails

Likewise, Drivers are additional software component required by JDBC to interact with DB

* A Driver is an additional software component required by JDBC to interact with a database
* **Drivers are provided by DB Vendor & they are DB specific**
* The Driver helps us to establish DB Connection, transfers the DB query and results between Java program and database

**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

**Steps to Configure the DB Driver**

Step 1 – Configure Project Build Path:

* Right Click on the Project, select Build Path ---> Configure Build Path
* Go to “libraries” tab, Click on the “Add External JARs” button
* select the Driver JAR file, click on “Open” button
* Click on OK button

Step 2 – Load the Driver into Program:

There are two ways to load the Driver,

1. Class.forName("**com.mysql.jdbc.Driver**");
2. Driver driver = **new** com.mysql.jdbc.Driver();

DriverManager.*registerDriver*(driver);

* “Driver Class”, which is an entry point to driver, is the one that implements the java.sql.Driver interface
* The most common approach to register a driver is to use Java's Class.forName() method which dynamically loads the driver's class file into memory & automatically registers it.
* The main advantage with Class.forName is, it can accept the class name as a String argument so that we can pass the driver class name dynamically. But if we create an instance of a driver class using new operator, then driver class name can't be changed dynamically.

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.

***Explain the above topic by using below program***

**Drivers Types**

There are 4 different types of driver

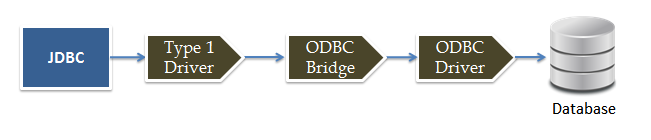
1. Type 1 Driver : JDBC-ODBC Bridge
2. Type 2 Driver : Native-API Driver
3. Type 3 Driver : Network-Protocol Driver
4. Type 4 Driver : Native-Protocol Driver

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

* This driver makes use of ODBC Bridge & ODBC Driver to connect to the database.

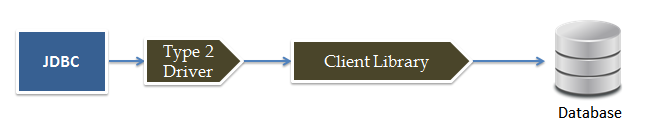
The driver converts JDBC method calls to DB calls using ODBC Driver.

* 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



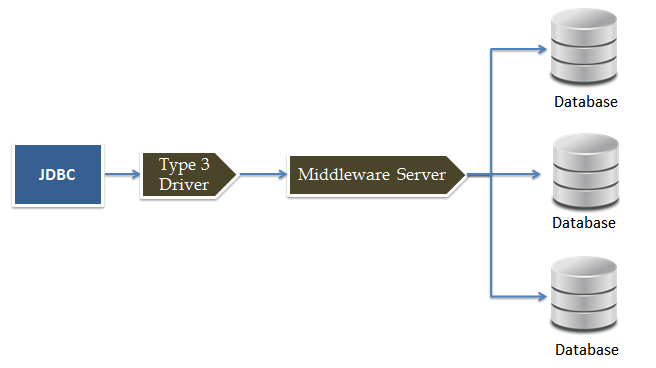
**Type 2 Driver: Native-API Driver**

* 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.

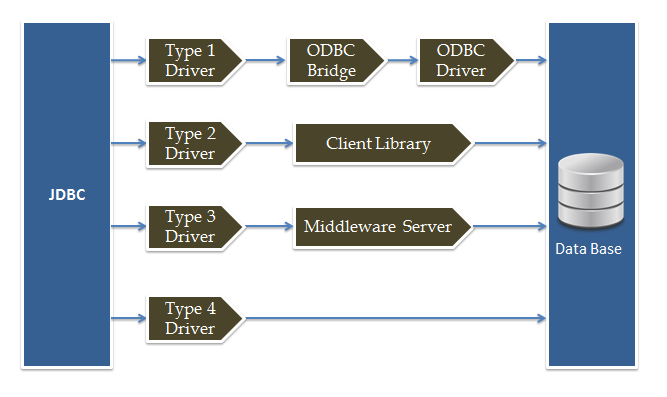


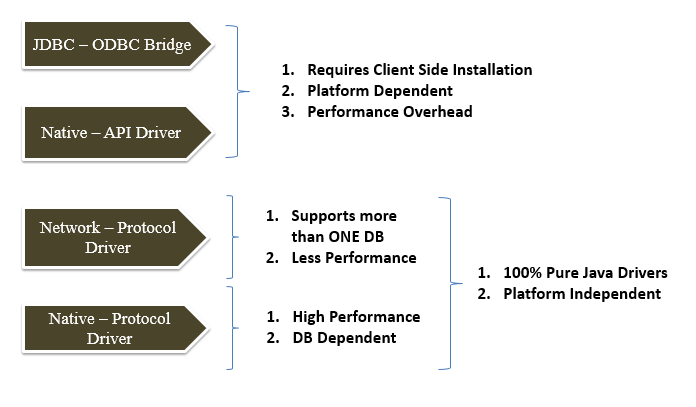
**Type 4 Driver: Native-Protocol Driver**

* 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:**



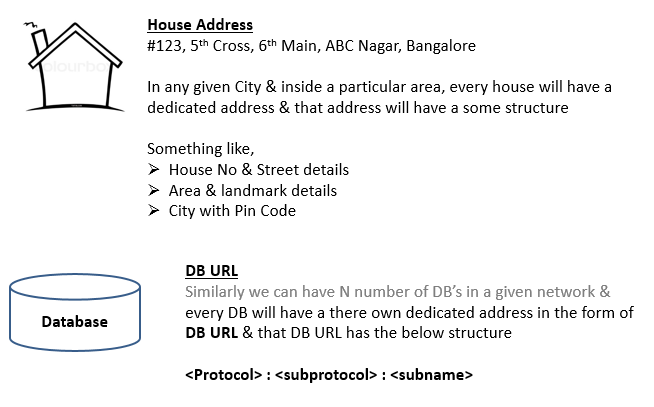


**Questions:-**

* Let's say your application design is a two-tier architecture where the database client operates as a standard Java application. What driver type are best suited to this type of application design?
* Which driver type is best suited to a single-database environment?

**DB URL (Uniform Resource Locators)**

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

****

* DB URL, uniquely identifies the database in network (internet / intranet)
* It is a String consists of 3 components separated by colons

**<Protocol> : <Subprotocol> : <Subname>**

**1. Protocol: -** In case of JDBC, it is always “**jdbc”**

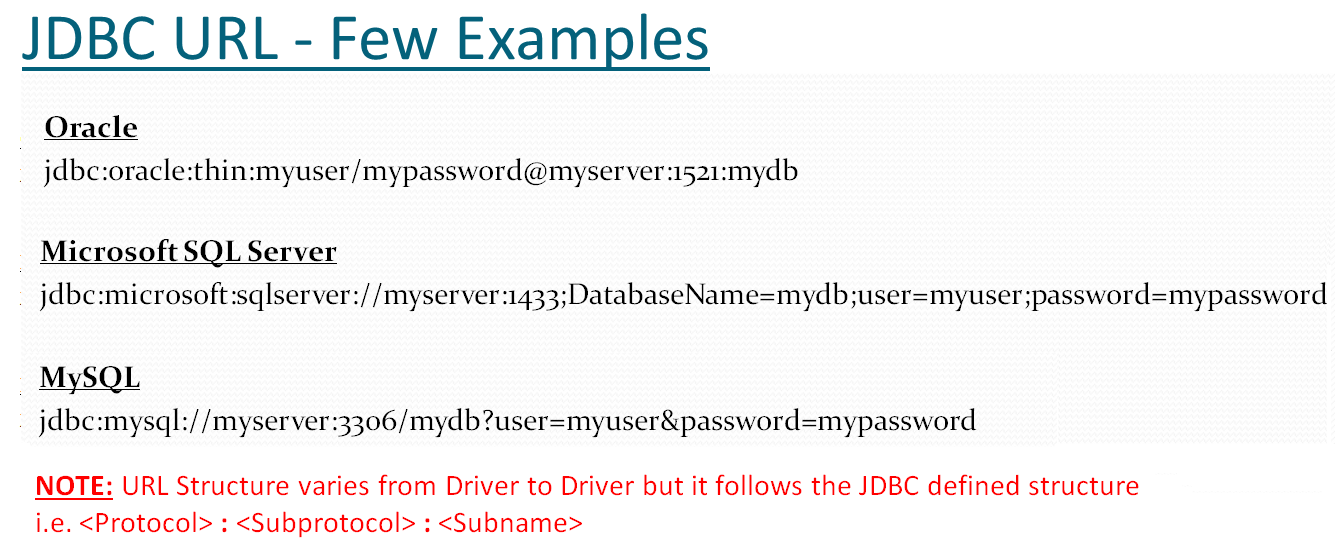
i.e. jdbc:<subprotocol>:<subname>

**2. Subprotocol: -** It is the name of the database connectivity mechanism. If the mechanism for connecting to DB is “JDBC-ODBC Bridge” then, subprotocol must be odbc.

i.e. jdbc:odbc:<subname>

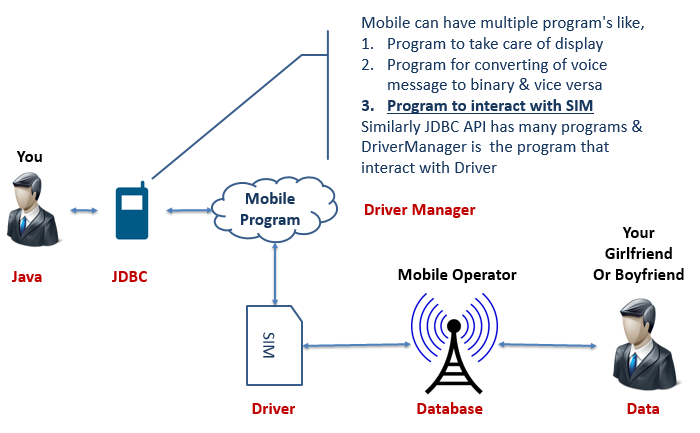
**3. Subname: -** It identifies the data base in a network. It is also called as “Data Source Name (DSN)”. It contains,

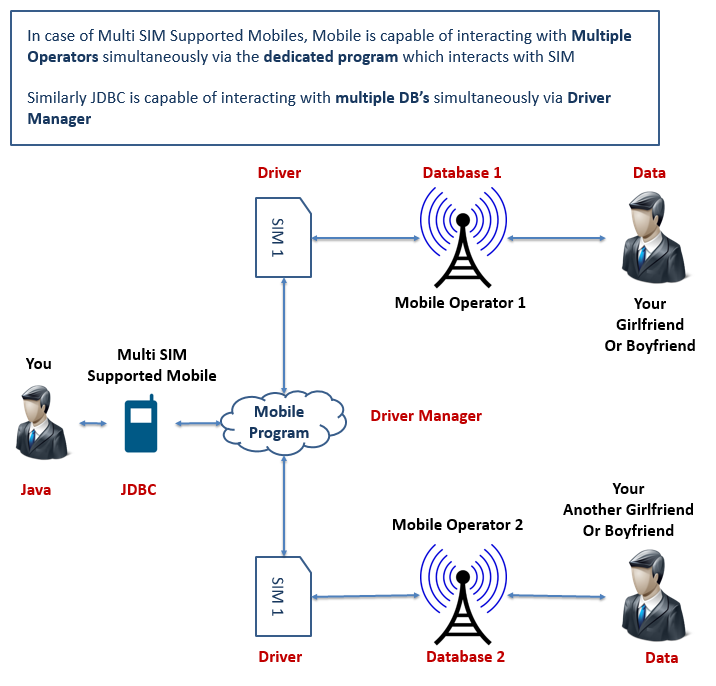
1. Host Name (machine / computer on which DB Server is installed)
2. Port on which DB Server is installed
3. DB name
4. User ID & Password to connect to DB

****

**DriverManager**

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

****

****

**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();

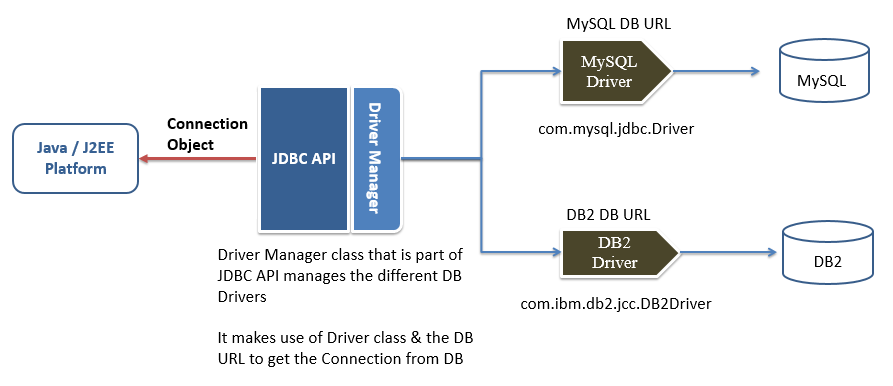
**int** i = 1;

myInterface = UtilityClass.*getMyClassInstance*(i);

myInterface.myMethod();

}//End of Class

}//End of Interface



* As the name implies, it manages the drivers. Since JDBC is not DB dependent & it can interact with multiple DB’s simultaneously, using suitable DB driver & its corresponding URL, DriverManager will get the DB connection from one / more DB’s
* DriverManager class has static getConnection() method which attempts to establish a connection to the given database URL. The DriverManager attempts to select an appropriate driver from the set of registered JDBC drivers.
* DriverManager has 3 overloaded getConnection() methods

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

**throws** SQLException

String dbUrl = "jdbc:mysql://localhost:3308/college?user=j2ee&password=j2ee";

Connection con = DriverManager.*getConnection*(dbUrl);

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

String dbUrl = "jdbc:mysql://localhost:3308/college";

String userId = "j2ee";

String password = "j2ee";

Connection con = DriverManager.*getConnection*(dbUrl, userId, password);

1. **public** **static** Connection getConnection(String url, java.util.Properties prop) **throws** SQLException

String dbUrl = "jdbc:mysql://localhost:3308/college";

String fileAndPathNm = "D:\\j2ee\\db.properties";

FileInputStream stream = **new** FileInputStream(fileAndPathNm);

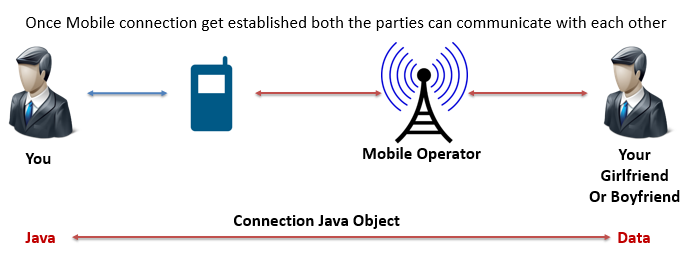
Properties prop = **new** Properties();

prop.load(stream);

Connection con = DriverManager.*getConnection*(dbUrl, prop);

**Connection**

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



* The Connection is a Java Object **that represents physical database connection** that can be used by a program to communicate with database
* Within the scope of a connection, SQL statements are executed and results are returned
* You can have multiple connections to a DB
* It also provides information about Database, tables & fields
* It 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));

}

**JDBC Statements**

**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)

* JDBC Statements send SQL queries and receive data from database.
* JDBC Statements also has methods that help bridge data type differences between Java and DB data types.
* There are 3 different types of JDBC Statements

1. Statement
2. PreparedStatement
3. CallableStatement

* Once you've created a Statement object (any of the above type), you can then use it to execute a SQL statement with one of the below execute methods.

1. **ResultSet executeQuery() :**
   * This method is **used to execute Select SQL query**.
   * This method returns a **DB results in the form of ResultSet object**.
2. **int executeUpdate() :**
   * This method is **used to execute the Insert, Update, or Delete SQL queries.**
   * This method returns the **numbers of rows affected count in the form of integer**.
3. **boolean execute() :**

* This method is used to execute **both DDL & DML SQL queries**
* This method returns true if the result is of type ResultSet object & returns false if it is an update count
* If we use this method then we **must** use the methods getResultSet() or getUpdateCount() to retrieve the result.

1. **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**

Statements are used to execute **Static SQL Queries** (SQL Queries with already set “where conditions”).

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());

}

**Assignment 1:** Write a JDBC Program which accepts any given SQL query (via command line argument) & print the result on console

**PreparedStatement**

* PreparedStatements are used to execute Dynamic SQL queries (SQL Statements where “where conditions” are set at runtime)
* They are also used in a situation where same SQL queries is executed many times
* PreparedStatements are also known as “Precompiled Statements” & they **MUST** be used with query parameters ( ? )
* PreparedStatements increases performance of the application

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.

**Assignment 2:**

Write a program to operate on students\_info table by reading the data from the text file. Text file should contain the data in the below format

**Reg No<single\_space>First Name<single\_space>Middle Name<single\_space>Last Name<single\_space>**

**CallableStatements:**

* CallableStatements are used execute **Stored Procedures**
* A stored procedure is a group of SQL statements that perform particular task. It can be invoked by the same way as we call a function or method. As its name implies, it is stored in the database.

**Stored Procedure 1:-**

**DELIMITER** $$

**CREATE** **PROCEDURE** GetAllStudents()

**BEGIN**

**SELECT** \* **FROM** students;

**END** $$

**DELIMITER** ;

**call** GetAllStudents();

**Stored Procedure 2:-**

**DELIMITER** $$

**CREATE** **PROCEDURE**

GetStudentInfo(**IN** inputRegNo **INT**)

**BEGIN**

**SELECT** \* **FROM** students

**WHERE** regno = inputRegNo;

**END** $$

**DELIMITER** ;

**call** GetStudentInfo(1);

**Stored Procedure 3:-**

**DELIMITER** $$

**CREATE** **PROCEDURE** StudentsUpsert

( **in** in\_regno **int**,

**in** in\_fname **varchar**(50),

**in** in\_mname **varchar**(50),

**in** in\_lname **varchar**(50)

)

**BEGIN**

**DECLARE** regnoCount **int**;

**SELECT** **COUNT**(regno) **INTO** regnoCount

**FROM** students

**WHERE** regno = in\_regno;

**IF** regnoCount > 0 **THEN**

**UPDATE** students

**SET** firstname=in\_fname, middlename=in\_mname, lastname=in\_lname

**WHERE** regno = in\_regno;

**ELSE**

**INSERT** **INTO** students

**VALUES** (in\_regno, in\_fname, in\_mname, in\_lname);

**END** **IF**;

**END**$$

**DELIMITER** ;

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).

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();

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.

**Assignment 3:-**

Write a program to operate on “students\_info” table.

Note: make use Stored Procedure which takes regno, first name, middle name, last name & operation name as input arguments. Based on the operation name, SP should operate on “students\_info” table.

**JDBC Statements Summarized:**

1. Statement

Statement stmt = con.createStatement();

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

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

Query>");

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

2. PreparedStatement

PreparedStatement pstmt

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

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

ResultSet rs = pstmt.executeQuery(); //Select SQL Query

int count = pstmt.executeUpdate(); // DDL, Delete, Insert, Update SQL

Query

boolean isDbResults = pstmt.execute(); //Any SQL Query

3. CallableStatement

**Processing the Result**

* SQL Queries to DB there are two kinds of results expected out of DB server

1. No. of Rows affected count (Insert / Update / Delete / DDL SQL Commands)

2. DB Results Data in tabular format (Select SQL Commands)

* In JDBC we can use Integer variable to hold the No. of Rows affected count & ResultSet Object to hold the DB Results

**ResultSet**

* It is an ordered set of rows produced by the Select SQL Query
* Result Sets are produced by **Select SQL Query** or certain Metadata method calls
* Once the Result set is produced, data from result set can be extracted as follows

1. Move to desired row by calling necessary JDBC methods

Ex : next(), last(), etc

1. Retrieve the desired column values using

getXXX(<Position of Column in SQL Query>) OR

getXXX(<column\_name>) where XXX = JDBC data type

**Why we need to Close Necessary JDBC Objects:**

* Connections, Statements, and ResultSets make use of OS resources such as memory. In case of Connections, further DB server resources are consumed.
* It is vital to close any JDBC object as soon as it has played its part; garbage collection should not be relied upon.
* Forgetting to close any JDBC objects properly may result in suspicious errors and misbehavior.

**Transactions:- Example:** **ATM Withdrawal Scenario; different steps involved**

* A transaction is a set of one or more statements that are executed as a unit. So either all of the statements are executed successfully or none of the statements are executed.
* Transactions **helps us to maintain the data consistency**.
* Following steps are followed to handle transaction in JDBC

1. Begin the transaction Disabling Auto-Commit Mode ( con.setAutoCommit(**false**) )
2. Execute one or more SQL Commands (generally Update, Insert or Delete SQL Commonds)
3. If no errors occur then commit the transaction ( con.commit() )
4. If errors occur then rollback the transaction ( con.rollback() )

* Note :

1. Whenever there is a scenario to interact with multiple tables (to Insert, Update or Delete data) then make use of transactions
2. If there is more than one catch block, then we have to have the rollback statement in all catch blocks.

**public** **class** TransactionSample

{

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

{

Connection con=**null**;

PreparedStatement pstmt = **null**;

ResultSet rs=**null**;

**try**

{

//1. Load the JDBC Driver

//2. Get the Connection from DB via Driver

/\*

\* I. Begin the Transaction

\*/

con.setAutoCommit(**false**);

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

pstmt = con.prepareStatement

("insert into students(regno,firstname,lastname)

values (?, ?, ?) ");

Students studentsObj1 = **new** Students(1, "ABC1", "XYZ1");

Students studentsObj2 = **new** Students(2, "1231", "4561");

Students studentsObj3 = **new** Students(3, "qwerty1", "asdfg1");

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

dataList.add(studentsObj1);

dataList.add(studentsObj2);

dataList.add(studentsObj3);

Students data = **null**;

**int** count = 0;

/\*

\* II. Execute one or more SQL Commands

\*/

**for**(**int** i=0; i<=dataList.size(); i++)

{

**if**(i==2)

{

**throw** **new** Exception("Error Simulation ...");

}

data = dataList.get(i);

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

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

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

//4. Process the ResultSet returned by SQL Statements

count = pstmt.executeUpdate();

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

}

/\*

\* III. If no errors occur then commit the transaction

\*/

System.*out*.println("No Error Occurred.

Hence Committing the Transaction");

con.commit();

}**catch**(Exception e){

/\*

\* IV. If errors occur then rollback the transaction

\*/

System.*out*.println("Error Occurred.

Rollback the Transaction.");

**try** {

con.rollback();

} **catch** (SQLException e1) {

e1.printStackTrace();

}

System.*out*.println("Stack trace is : "+e.getMessage());

}**finally** {

//5. Close all the JDBC Objects

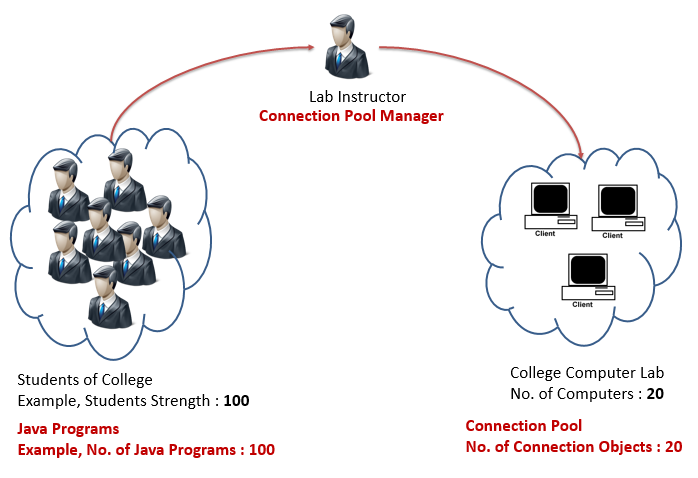
}//End of Try & Catch Block

}//End of Main Method

}//End of Class

**Connection Pooling**

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

****

* A connection pool is a **cache of database connection objects**. The objects represent physical database connections that can be used by a program to interact with database.
* Connection pools promote the reuse of connection objects and reduce the number of times that connection objects are created. **Connection pools significantly improve performance for database-intensive applications** because creating connection objects is costly both in terms of time and resources. Tasks such as network communication, reading connection strings, authentication, and memory allocation all contribute to the amount of time and resources it takes to create a connection object.
* With Connection Pooling the pool of connections are created at the time of application startup / when first DB connection request is made. Hence the application waits less time to get the connection.
* Connection pools often provide properties that are used to optimize a pool’s performance. The properties control the behavior of Connection Pool. It contains information such as the minimum and maximum number of connections allowed in the pool, user id & password to get connection etc.
* Note :

1. Connection Pool make use of Object Pool Design Pattern & Singleton Design Pattern

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

**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);

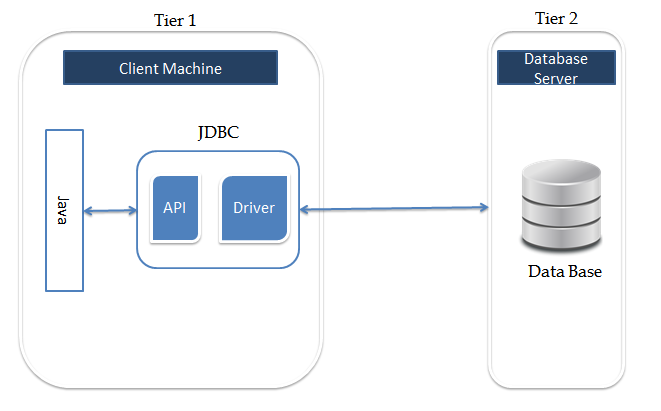
// Create an int[] to hold returned values

int[] count = stmt.executeBatch();

**Batch Processing Example for CallableStatement:-**

*Same as PreparedStatement*

**Revisit J2EE 2-Tier Architecture**

****