JDBC

J2EE:

- ***J2EE*** *stands for* ***Java 2 Enterprise Edition where 2 refers to the version.***
- *It contains* ***larger set of libraries*** *compared to* ***J2SE*** *using which some* ***additional functionalities can be performed.***

Need for J2EE:

- *It is needed for the* ***Simplification of Web Application Development.***

JAR FILE

- ***JAR means Java Archive (Compress).***
- *It is a file format based on* ***ZIP file format*** *which is used to* ***compress many files into one Single file.***

Contents of Jar File:

- ***.java (optional):*** *Contains* ***source code*** *statement in it.*
- ***.class:*** *Contains* ***byte code*** *statements in it* ***(Machine level language).***
- ***config files:*** *Contains* ***configuration data*** *in it.*
- ***.xml:*** *It is used to* ***configure the resource*** *with the help of* ***User-defined tags or Custom tags.***
- ***.properties:*** *It is used to provide a* ***set of properties*** *in the form of* ***key and value pair*** *where* ***key must be unique.***

Need for Jar File:

- *Jar File is needed to **import the properties based on the requirement.***

❖ *Total inbuilt class in java are **6000+** present in **rt.jar** where rt refers to **run-time.***

Steps to create a Jar File:

- *Right click on project and select **Export option.***
- *Open java folder and **select jar file** and click on **next.***
- *Browse for appropriate location to place the jar file and **name the jar** file and click on **Save and Finish.***

Steps to build a Java path:

- ***There are 2 different ways to build a java path to import the properties from a jar file namely***

a) Externally

b) Internally

(a) Externally:

- *Right click on **project** and select **properties option.***
- *Select **java build path** and click on **libraries tab.***
- *Click on **add external jars** and select a jar file from **the respective path** and click on **open and OK.***

(b) Internally:

- *Right click on **project** and create a new folder with **name lib.***
- *Add the **respective jar** file into the **lib folder.***
- *Right click on **project** and select **properties option.***
- *Select **java build path** and click on **libraries tab.***
- *Click on **add jars** and select jar file from the **lib folder** of respective project and click on **Open and Ok.***

Note:

❖ ***It is always a good practice to import the properties from a jar file Internally.***

Parameter:

- ***It is the one which holds an argument.***

Note:

- ***Whenever the method parameter type is an Interface, then the argument for that method must be any of its Implementation class object.***
- ***Whenever the method parameter type is Super class, then the argument for that method must be any of its sub class object.***
- ***Whenever the method parameter type is an Abstract class, then the argument for that method must be any of its sub class or implementation class object.***

Argument:

- ***Data or value passed to any methods is an argument.***

DESIGN PATTERN

*An optimized solution for commonly re-occurring design problems or design needs is known as a **Design Pattern.***

Eg: MVC Architecture (Model View Controller).

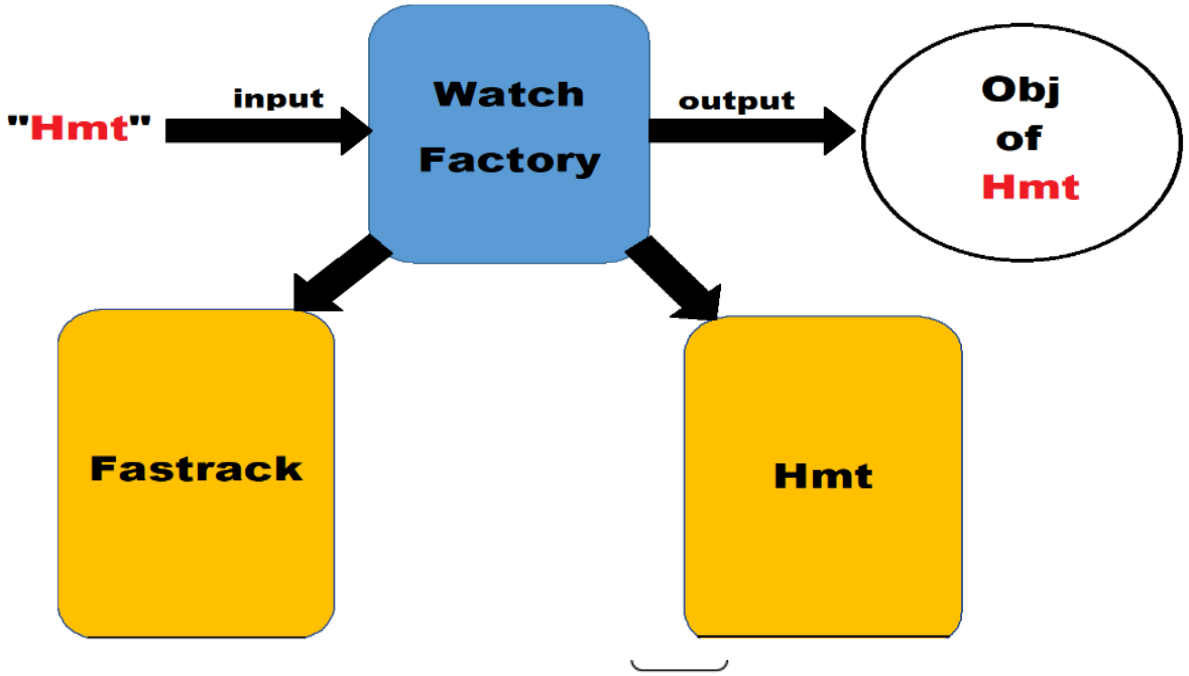
- ***There are 2 different categories of Design pattern present namely***

a) Creational Design Pattern

b) Factory Design Pattern

- ***Creational Design Pattern:*** ***It involves only Object Creation logic.***
- ***Factory Design Pattern:*** ***It is the one which creates or produces "Different Objects of same type".***

- ***Factory generally takes an input of String type.***



```
+ class WatchFactory (Factory Class)
{
        + static watch getWatch(String type) // Factory/Helper method
        {
                if(type.equalsIgnoreCase("Fastrack"))
                {
                    return new Fastrack();
                }
                else if(type.equalsIgnoreCase("Hmt"))
                {
                    return new Hmt();
                }
                else
                {
                    System.out.println("No Watch Found!!!!");
                }
                return null;
        }
}
```

***Factory Design pattern** is always associated with 3 different types of logic namely:*

a) Implementation Logic
b) Object creation Logic
c) Consumer or Utilization Logic

(a) Implementation Logic:** It is the **basic fundamental logic** which contains only **Implementations** according to which an **implementation object has to be created.

(b) Object creation Logic**: It is used to create **implementation object** according to the **implementation logic** by using a **factory or helper method** within the **factory class.

(c) Consumer or Utilization Logic:** It is the **most important logic** which is used to **access the functionalities from the Implementation Objects.

Note:

- ***Without writing a consumer or utilization logic, none of the implementations will work.***

- ***Costly Resources:** Resources which makes use of **system properties** are known as **Costly resources.***

- ***Factory or Helper method:*** *It is used to create* ***Implementation object.***

- ***NullPointerException:*** *Pointing towards an* ***object which is not present*** *throws an exception called* ***NullPointerException.***

ABSTRACTION

Hiding the implementation and providing functionalities to the user with the help of Interface is known as Abstraction.

- ***The output of Abstraction is Loose coupling which can be achieved with the help of Interface.***

- ***Interface:*** ***It is a media to communicate between User and any Device.***

- ***Loose Coupling:*** ***Change in the implementation which does not affect the user is known as Loose Coupling.***

- ***Tight Coupling:*** ***Change in the implementation which affects the user is known as Tight Coupling.***

Factory Code implementing all three different types of logic:

```
package org.btm.lightApp;
public interface ISwitch
{
        void sOn( );
        void sOff( );
}

package org.btm.lightApp;
public class TubeLightImpl implements ISwitch
{ @Override
        public void sOn( )
    {
        System.out.println("TubeLight is On !!");
    }
@Override
public void sOff( )
```

```
    {
        System.out.println("TubeLight is Off !!");
     }
}

package org.btm.lightApp;
public class LedLightImpl implements ISwitch
{ @Override
        public void sOn( )
     {
        System.out.println("LedLight is On!!");
     }
@Override
public void sOff( )
    {
        System.out.println("ledLight is Off!!");
    }
}

package org.btm.lightApp;
public class LightFactory
{
        public static ISwitch getLight(String type)
        {
                if (type.equalsIgnoreCase("ledLight"))
                {
                        return new LedLightImpl();
                }
                else if (type.equalsIgnoreCase("TubeLight"))
                {
                        return new TubeLightImpl();
                }
                else
                {
                        System.err.println("No such light found");
                }
                return null;
        }

}
```

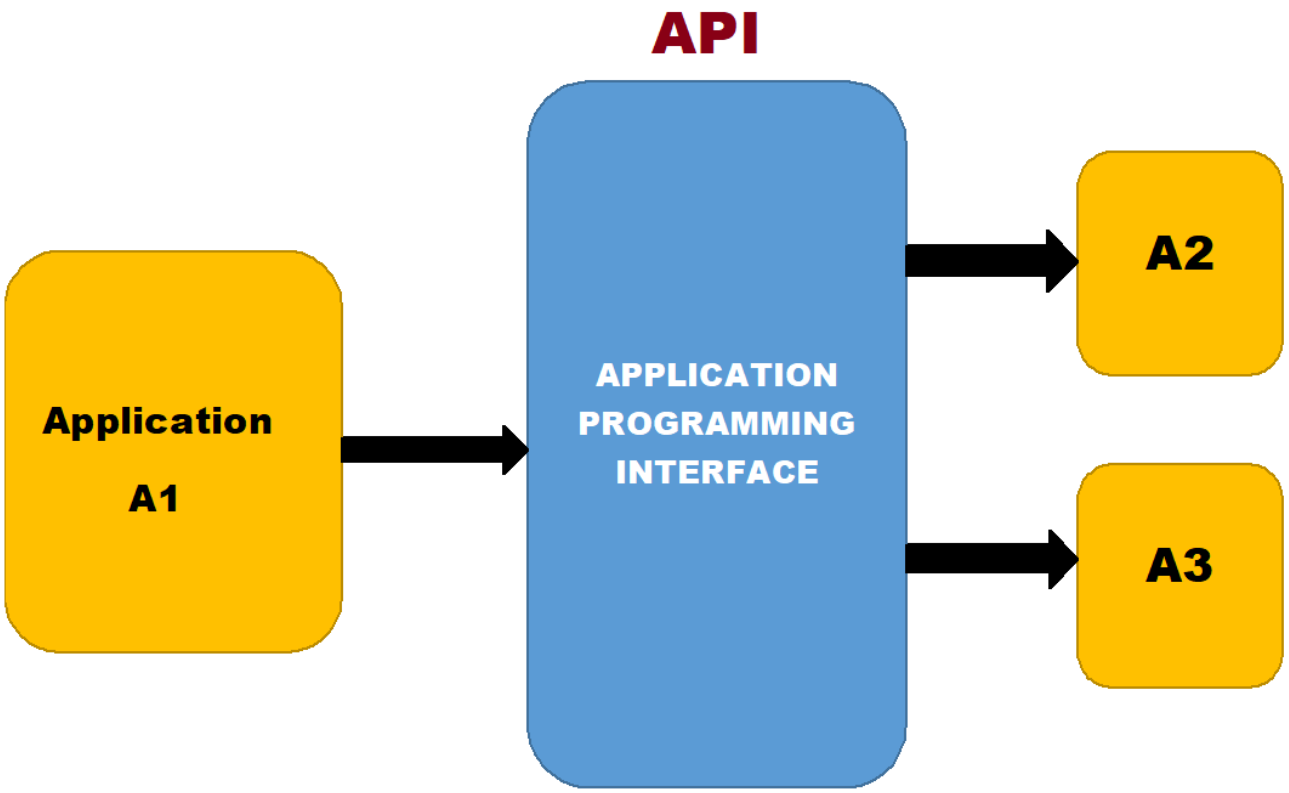
```
package org.btm.lightApp;
import java.util.Scanner;
public class Test
{
	public static void main(String[ ] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the type of Light");
		String type = sc.next();
		sc.close();
		ISwitch is= LightFactory.getLight(type);
		if (is != null)
		{
			is.sOff( );
			is.sOn( );
		}
	}
}
```

Note:

- ***Whenever the User provides implementation, then the same user has to even create an object of implementation.***
- ***Whenever the Vendor provides implementation, then the same Vendor has to even create an object of implementation.***

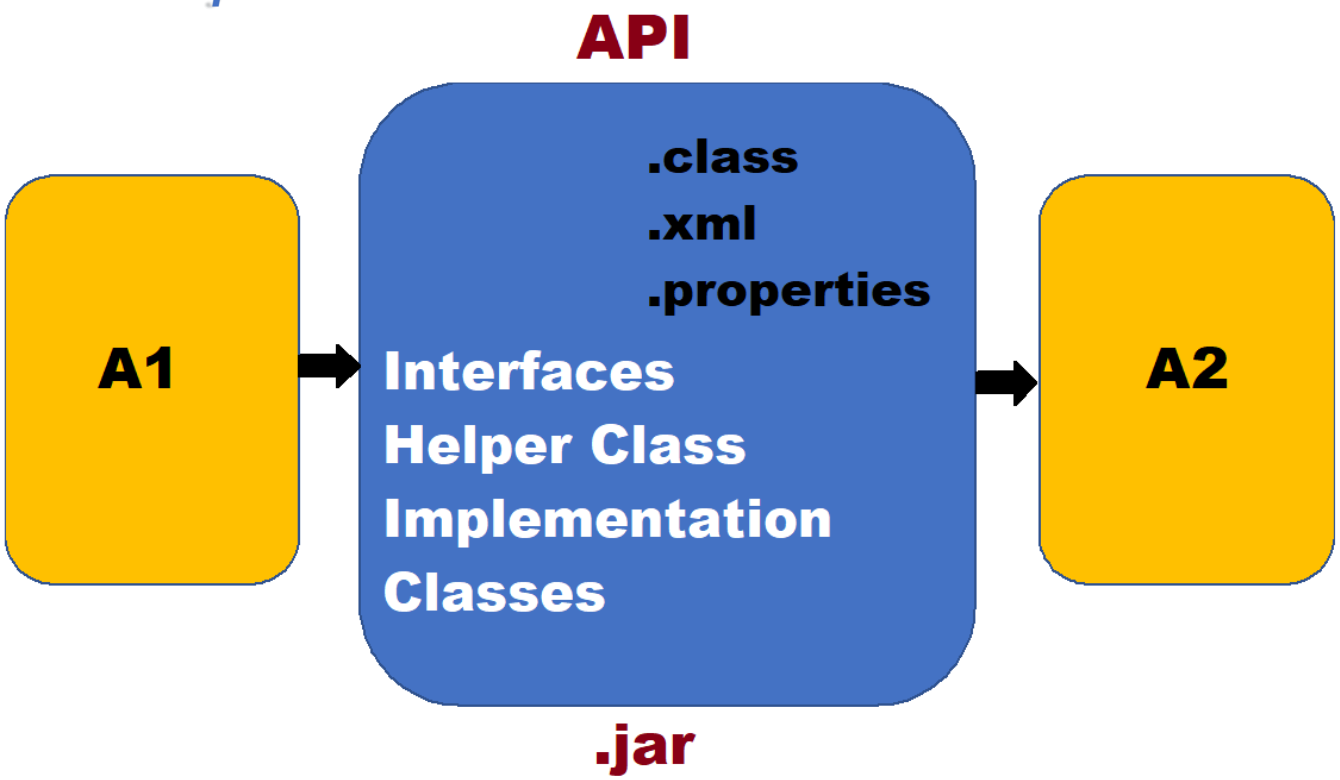
APPLICATION PROGRAMMING INTERFACE(API)

- ***API used for inter-application communication i.e. one application can communicate with other application with the help of API to achieve Loose coupling.***
- *Backbone of API is **Abstraction.***
- *The output of **Abstraction is Loose Coupling** which is achieved with the help of **Interface.***



Example for API's are: Apache POI, Jaxcel, JDBC, Server...etc.

Contents of API:



The contents of API are ***Interfaces, Helper Class and Implementation Classes*** in the form of Jar file.

> ***Note:*** Whenever the API's are given in the form of jar files, then such API's developed using java as programming language.

There are two different forms of API present namely:
(a) 1st form of API (b) 2nd form of API

a) 1st form of API:

- This form of API contains ***interfaces, helper classes and implementation classes*** in the form of jar file.
- This form of API is exposed to the developer or the programmer to write the ***consumer or utilization logic.***

Eg: Apache PoI, Jaxcel, etc

> ***Note:*** A ***consumer or utilization*** logic can always be written only after the implementations are known.

b) 2nd form of API:

JDBC API:

- It was given by ***Sun micro System*** to achieve loose coupling between Java application and Database Server.
- It contains ***Interfaces and Helper Class*** in the form of jar file.
- It is distributed into 2 different packages namely ***java.sql*** ***and*** ***javax.sql***
- The interfaces of ***JDBC API*** are ***Driver, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, MetaData, etc.***
- ***JDBC API*** contains only one helper class in it by name ***DriverManager.***

JDBC DRIVER:

- It is an implementation of ***JDBC API.***
- It contains implementation classes in the form of jar file.
- It is always specific to the particular Database Server or Vendor.
- These are provided by respective Database Server or Vendors.

Eg: Jdbc, Servlets, etc.

> ***Note:***
>
> - For us to communicate with any Database Server and perform any data base operation, we need to have two different components mandatorily namely ***JDBC API and JDBC DRIVER.***
> - In Java JDBC program we always write only the ***Consumer or Utilization logic.***

Advantages of JDBC:

- We can achieve loose coupling between Java Application and Database Server.
- Platform-independent.

Port Number: It is the one which helps us to get connected to a ***particular server.***

Different port numbers:

- ***Oracle=1521***
- ***MYSQ=3306***
- ***MS-SQL=1433***
- ***Derby=1527***

Thin-client application:

It is a ***light-weight software*** which is used to communicate with ***a particular Database Server or Database Vendor.***

There are 2 different categories of thin client application present namely

a) ***Mobile client application***

b) ***Database client application***

a) ***Mobile client application:*** Banking software, Facebook, Messenger, etc.

b) ***Database client application:*** Toad, Squirrel, SQL workbench, SQL Developer, SQLyog, SQLultima, etc.

Host: It is a ***Platform*** on which all the ***Applications can be executed.***

There are 2 different types of host present namely

a) ***Local Host*** b) ***Remote Host***

a) ***Local Host:*** In this case, the applications are restricted or limited only to that particular System.

Eg: ***Standalone application.***

b) ***Remote Host:*** In case of remote host, the applications are not limited or restricted toany particular system.

Eg: ***Any Real time application.***

URL (Uniform Resource Locator): It is the path using which we can access the resources uniquely over the network.

The contents of ***URL*** are ***Protocol, Host +Port Number / Domain name, Resource Name, Data (optional).***

- The protocol for web applications are ***http or https.***
- The protocol for JDBC is ***jdbc:sub-protocol .***
- In case of ***URL***, data refers to Key and Value pair which is provided by the user which is Optional.

Standard way of writing JDBC URL:

mainprotocol: subprotocol://Host+Port/dbname(optional)

For **MySql** Database [user name must be ***root*** password ***Any***]

Local Host: *jdbc:mysql://localhost:3306/Studentdb?user=root&password=admin*

jdbc:mysql://localhost:3306?user=root&password=admin

Remote Host: *jdbc:mysql://192.168.10.16:3306/Studentdb?user=root&password=admin*

jdbc:mysql://192.168.10.16:3306?user=root&password=admin

For Oracle Database [user name must be ***Scott*** password must be ***Tiger***]

Local host: *jdbc:oracle://localhost:1521/Studentdb?user=scott&password=tiger*

jdbc:oracle://localhost:1521user=scott&password=tiger

Remote Host: *jdbc:oracle://192.168.10.16:1521/Studentdb?user=scott&password=tiger*

jdbc:oracle://192.168.10.16:1521?user=scott&password=tiger

CLASS LOADING

Loading a ***dot class file*** into the ***JVM memory*** is known as ***Class Loading.***

- A class is generally loaded in 2 different ways namely:
- By calling any of the members of a class which can either be ***a constructor, methods, variables, blocks etc.***
- By using a static method called ***forName( )*** which is present in ***java.lang*** package and whenever we use this method, it throws an exception called ***ClassNotFoundException***.

Syntax:

Class.forName("***Fully Qualified Class Name***");

1st Way:

```
package org.btm.loadApp;
public class Snake
{
	void eat( )
	{
		System.out.println("Eating frog");
	}
}
public class Test
{
	public static void main(String[ ] args)
	{
		Snake s =new Snake();
		s.eat();
	}
}
```

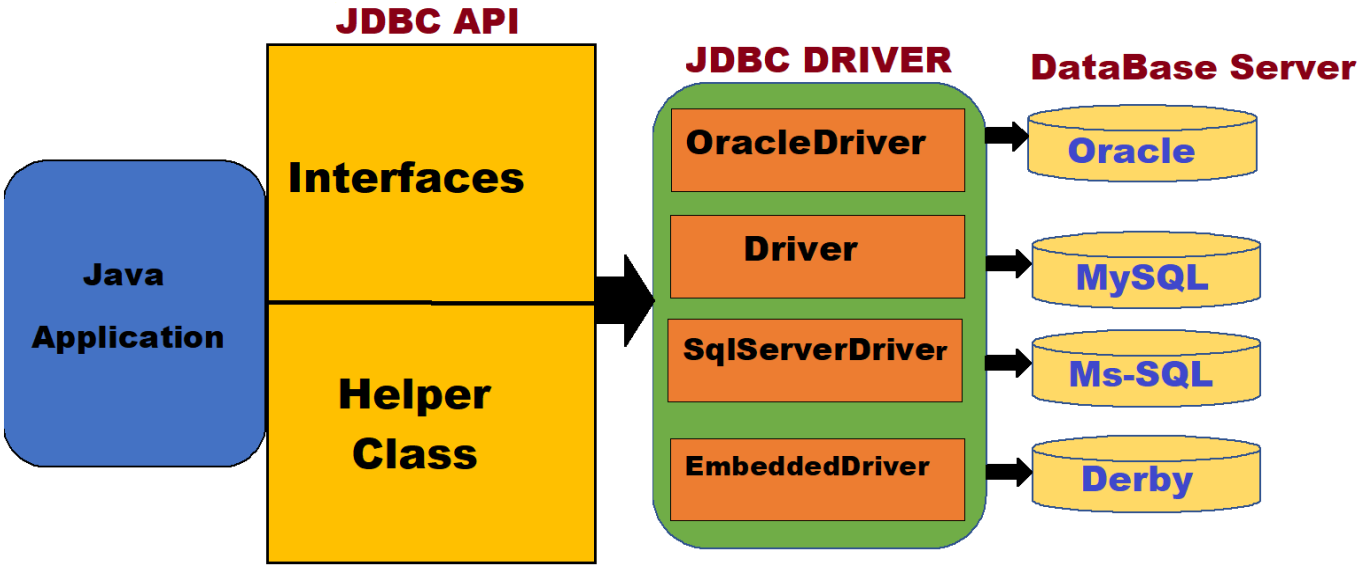
2nd way:

```
package org.btm.loadApp;
public class Snake
{
	void eat( )
	{
		System.out.println("Eating frog");
	}
}
package org.btm.loadApp;
public class Test
{
	public static void main(String[ ] args)
	{
		try
		{
			Class.forName("org.btm.loadApp.Snake");
		}
		catch(ClassNotFoundException e)
		{
			e.printStackTrace();
```

```
            }
        }

}
```

Note: *Driver Classes* are provided by the respective Database Servers or vendors in the form of jar file.

Different Driver classes provided by respective Database Servers or Vendors:



- All the *Driver Classes* which is a part of JDBC Driver are provided by respective Database Servers or vendors in the form of jar file.
- All the *Driver classes* must mandatorily implements *java.sql.Driver* Interface which is a part of *JDBC API.*

Specifications of JDBC:

- All the *Driver classes* must contain one *Static block* in it.
- All the *Driver classes* must mandatorily implement *java.sql.Driver* Interface which is a part of *JDBC API.*
- All the *Driver classes* must mandatorily be registered with *DriverManager* using a static method called *registerDriver( ).*

Definition of JDBC:

(Java Data Base Connectivity) is a specification given in the form of abstraction API to achieve ***loose coupling between Java application and Database server.***

Steps of JDBC:

a) Load and Register the Driver (Driver refers to Driver class provided by respective Database server or vendor).

b) Establish a connection with the Database Server.

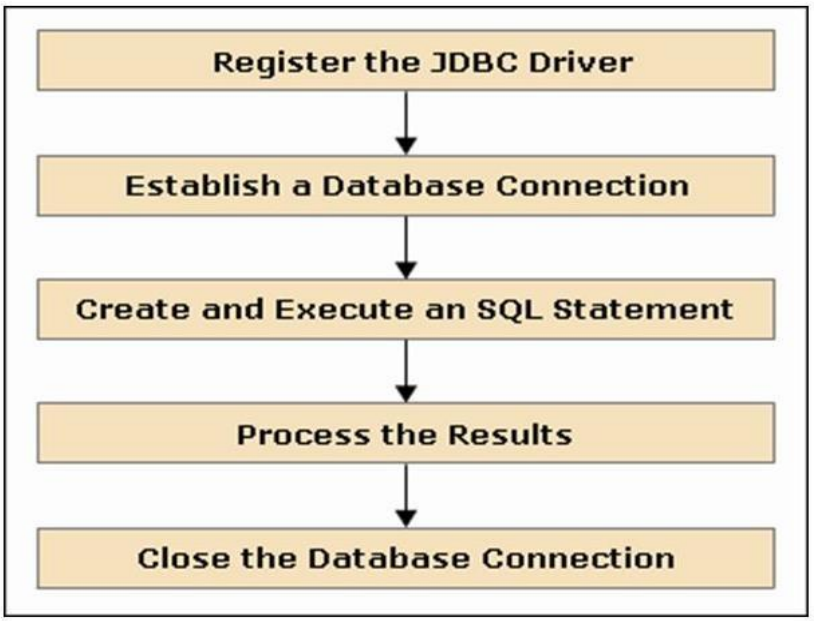
c) Create a Statement or Platform.

d) Execute the Sql Queries or Sql Statement.

e) Process the Resultant data (optional).

f) Close all the Costly resources.

Steps to Develop a JDBC Application



a) Load and Register the Driver:

- In this step, we have to load and register the ***Driver classes*** which is a part of ***JDBC Driver*** which are provided by respective ***Database Server or Vendors.***
- ***Driver classes*** can be loaded and registered in 2 different ways namely:

Manually: In this case an object of the ***Driver class*** is created by user and then registered with ***DriverManager*** using static method called ***registerDriver( )***.

Syntax:

Driver d =new Driver();

DriverManager.registerDriver(d);

> ***Note:***
> - This is not a good practice since it causes tight coupling between java application and Database Server.

b) By using a static method called ***forName( )*** , we can load and register the ***Driver classes*** provided by respective ***Database Servers or Vendors.***

syntax:

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

```
package org.btm.jdbcApp;
public class JdbcDemo
{
	public static void main(String[ ] args)
	{
		try
		{
			Class.forName("com.mysql.jdbc.Driver");
			System.out.println("Driver class loaded and registered");
		}
		catch(ClassNotFoundException e)
		{
			e.printStackTrace();
		}
	}
}
```

Costly Resources:

- Resources which make use of System properties in the form of Stream are known as ***Costly Resources.***
- It is always a good practice to close all the costly resources since it decreases the performance of an application.
- All the ***Costly resources*** must be closed within the finally block using an if condition to avoid ***NullPointerException.***

Note:

- All the ***Interfaces of JDBC API*** are considered to be costly resources of ***JDBC.***
- ***Interfaces of JDBC API*** are ***Driver, Connection, Statement, PreparedStatement, CallableStatement, ResultSet, etc.***

b) Establish a Connection between Java Application and Database Server:

- In this step, we have to establish a connection between the Java application and the Data base Server by using ***getConnection( ).***
- ***getConnection( )*** is static factory/helper method which is used to create and return an implementation object of Connection Interface ***based on URL.***
- Hence, the return type for the ***getConnection( )*** method is Connection interface.
- There are 3 diff overloaded variants of ***getConnection( )*** method present namely
  getConnection(***String URL***)
  getConnection(***String URL, Properties info***)
  getConnection(***String URL, String User, String password***)
- Whenever we use either of the overloaded variants of ***getConnection( )*** it throws a checked exception called ***SQLException.***

Syntax:

java.sql.Connection con=DriverManager.getConnection(“URL”);

java.sql.Connection:

- It is an interface which is a part of ***JDBC API*** and the implementations are provided by respective Database Servers or Vendors as a part of ***JDBC Driver***.
- It is always a good practice to close ***Connection*** Interface since it is considered to be a ***Costly Resources*** which decreases the performance of the application.

java.sql.DriverManger:

- It is a Helper class which is a part of ***JDBC API*** which contains two important static methods in its namely ***registerDriver( ) and getConnection( ).***

c) Create a Statement or Platform:

- We need to create a Statement or Platform to ***execute the Sql queries or Sql Statements.***
- A Platform can either be created by using ***Statement or PreparedStatement or CallableStatement*** Interface which are the part of ***JDBC API.***

java.sql.Statement:

- It is the ***Interface*** which is the part of ***JDBC API*** and the implementations are provided by respective Database Servers or Vendors as a part of ***JDBC Driver.***
- ***createStatement( )*** is a factory method which is used to create and return implementation object of ***Statement Interface.***
- Hence, the return type for ***createStatement( )*** is ***Statement Interface.***

Syntax:

java.sql.Statement stmt=con.createStatement( );

```
package org.btm.jdbcApp;
import java.sql.*;
public class JdbcDemo
{
        public static void main(String[ ] args)
        { Connection con = null;
        Statement stmt = null;
        try {
                Class.forName("com.mysql.jdbc.Driver");
                System.out.println("Driver class loaded and register ");
                con=DriverManager.getConnection("jdbc:mysql://localhost:3306?user
                                                        =root&password=tiger");
                System.out.println("Connection Established");
                stmt = con.createStatement( );
                System.out.println("Platform created");
        }
        catch (ClassNotFoundException | SQLException e)
```

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

d) Execute the Sql queries or Sql Statements:

- To execute the different types of Sql Statement or Queries, there are three different methods present namely ***execute( ), executeUpdate( ), executeQuery( ).***
- All these methods are declared in ***Statement Interface*** but it can be used either with ***Statement or PreparedStatement or CallableStatement Interface*** which is a part of ***JDBC API.***

execute( ):

- It is a ***Generic method*** since it is used to ***execute any type of Sql queries or statements.***
- Hence, return type of ***execute( )*** is ***boolean.***
- ***execute( )*** returns a ***boolean true*** value in case of ***DQL*** and ***boolean false*** in case of ***DML or DDL.***

+boolean execute(“Generic SQL Query”);

Syntax:

stmt.execute("Generic SQL Queries");

> ***Note:***
> The outcome of ***DML*** is ***0 to n Integer value*** which gives the total no of records affected in the data base server.

executeUpdate( ):

- This method is a Specialized method which is used to execute only DML Queries.
- Hence, the return type for ***executeUpdate( )*** is ***Integer.***
- Whenever we try to execute a ***DDL or DQL*** query using this method, it throws an exception called ***SQLException.***

+ int executeUpdate("Only DML")

Syntax: **stmt.execute("DML");**

Code to Insert Single Record into the Database Server using Statement Interface:

```
package org.btm.jdbcApp;
import java.sql.*;
public class JdbcDemo
{
	public static void main(String[ ] args)
	{
		Connection con = null;
		Statement stmt = null;
		String qry= "insert into btm.student values(1, 'Arvind', 56.77)";
		try {
			Class.forName("com.mysql.jdbc.Driver");
			System.out.println("Driver class loaded and register ");
			con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
                                                    user=root&password=tiger");
			System.out.println("Connection Established");
			stmt = con.createStatement();
			System.out.println("Platform created");
			stmt.executeUpdate(qry);
			System.out.println("data Inserted");
		}
```

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

            }

      }

}
```

Code to Update a Record in the Database Server using Statement Interface:

```
package org.btm.jdbcApp;
import java.sql.*;
public class JdbcDemo
{
      public static void main(String[ ] args)
      {
            Connection con = null;
            Statement stmt = null;
            String qry= " update btm.student set name= 'Raj' where id=1 ";
```

```
            try {
                Class.forName("com.mysql.jdbc.Driver");
                System.out.println("Driver class loaded and register ");
                con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
                                                    user=root&password=tiger");
                System.out.println("Connection Established");
                stmt = con.createStatement();
                System.out.println("Platform created");
                stmt.executeUpdate(qry);
                System.out.println("Data Updated");
            }
            catch (ClassNotFoundException | SQLException e)
            {
                e.printStackTrace();
            }
            finally {
                if (stmt!= null) {
                    try {
                        stmt.close();
                    } catch (SQLException e)
                    {
                        e.printStackTrace();
                    }
                }
                if(con!=null)
                {
                    try {
                        con.close();
                    } catch (SQLException e)
                    {
                        e.printStackTrace();
                    }
                }

            }

        }

}
```

Code to Delete a Record from the Database Server using Statement Interface:

```
package org.btm.jdbcApp;
import java.sql.*;
public class JdbcDemo
{
	public static void main(String[ ] args)
	{
		Connection con = null;
		Statement stmt = null;
		String qry= " delete from btm.student where id=3";
		try {
			Class.forName("com.mysql.jdbc.Driver");
			System.out.println("Driver class loaded and register ");
			con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
                                                    user=root&password=tiger");
			System.out.println("Connection Established");
			stmt = con.createStatement();
			System.out.println("Platform created");
			stmt.executeUpdate(qry);
			System.out.println("Data Deleted");
		}
		catch (ClassNotFoundException | SQLException e)
		{
			e.printStackTrace();
		}
		finally {
			if (stmt!= null) {
				try {
					stmt.close();
				} catch (SQLException e)
				{
					e.printStackTrace();
				}
			}
			if(con!=null)
			{
				try {
					con.close();
				} catch (SQLException e)
```

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

        }

    }

}
```

Code to Insert Multiple Record into the Database Server using Statement Interface:

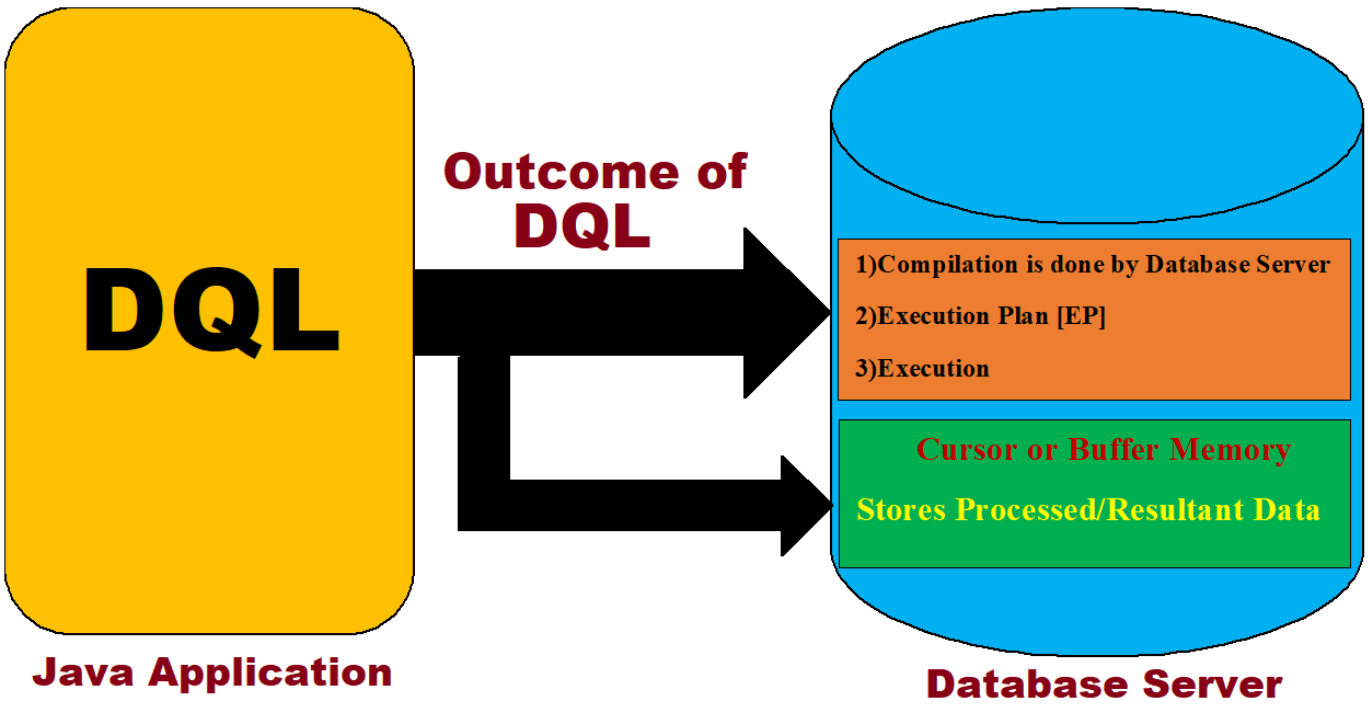
```
package org.btm.jdbcApp;
import java.sql.*;
public class JdbcDemo
{
    public static void main(String[ ] args)
    {
        Connection con = null;
        Statement stmt = null;

        String qry1= "insert into btm.student values(1, 'Arvind', 86.23)";
        String qry2= "insert into btm.student values(2, 'Raju', 76.67)";
        String qry3= "insert into btm.student values(3, 'Rahul', 84.17)";

        try {
            Class.forName("com.mysql.jdbc.Driver");
            System.out.println("Driver class loaded and register ");
            con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
                                                    user=root&password=tiger");
            System.out.println("Connection Established");
            stmt = con.createStatement();
            System.out.println("Platform created");
            stmt.executeUpdate(qry1);
            stmt.executeUpdate(qry2);
            stmt.executeUpdate(qry3);
            System.out.println("Data Inserted");
        }
```

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

            }

      }

}
```

Note:

- Since ***Compilation takes place each time along with Execution*** whenever we execute query using Statement interface, ***Performance of an application decreases***.
- Hence, it is a good practice to make use of ***PreparedStatement*** Interface to deal with multiple records.

Fetching the Data from Database Server:



- Whenever we execute a ***DQL query or DQL Statement*** we may get a result which is referred as ***Processed or Resultant data***
- The ***Processed or Resultant data*** is stored in the ***Cursor or Buffer memory***, which can be fetched by using ***ResultSet*** interface which is a part of ***JDBC API.***
- The Structure of Cursor or Buffer memory may differ from the Structure of Database.

Note:
- ***Column name*** is also known as ***column label.***
- ***Column number*** is also known as ***column index.***

java.sql.ResultSet:

- It is an ***Interface*** which is a part of ***JDBC API*** and the implementations are provided by respective Database Servers or vendors as a part of ***JDBC Driver.***
- A set of methods of ***ResultSet*** interface are used to fetch the processed or resultant data from the cursor or buffer memory which are known as ***getXXX( ).***
- There are two different overloaded variants of ***getXXX( )*** present namely:

+ XXX getXXX(int columnnumber**)**

+ XXX getXXX(String Columnname**)**

If datatype is integer:

+int getInt(int columnNumber)

(or)

+int getInt(String columnname)

If datatype is String:

+String getString(int columnNumber)

(or)

+String getString(String columnname)

If datatype is double:

+double getDouble(int columnNumber)

(or)

+ double getDouble(String columnname)

Note:

- By default, the ***ResultSet*** interface does not point to any record in the Cursor or Buffer memory.

next( ):

- It is used to check whether the ***next record*** is present in the cursor or buffer memory and returns a ***boolean*** value called ***true or false*** but not the record.
- It can be used whenever there are ***minimum no of records*** present in the cursor or buffer memory.
  + boolean next( )

absolute( ):

- It is used to check whether a ***particular record*** present is present in the cursor or buffer memory based on the parameter called ***int rownumber*** and returns a ***boolean*** value called ***true or false*** but not the record.
- It can be used whenever there are ***n number of records*** present in the cursor or buffer memory.

+ **boolean** *absolute*(int rownumber)

executeQuery( ):

- It is a ***Specialized method*** which is used to execute only ***DQL Queries.***
- The outcome of ***DQL*** is Processed or Resultant data which is stored in the cursor or buffer memory which can be fetched with the help of ***ResultSet Interface*** which is a part of ***JDBC API.***
- Hence, the return type for ***executeQuery( )*** is ***ResultSet Interface.***
- whenever we try to execute a ***DML or DDL*** query using this method, it throws an exception called ***SQLException.***

  + **ResultSet** *executeQuery*("Only DQL")

Creating an implementation object of ResultSet interface:

- There are 2 different ways to create an implementation object of ***ResultSet Interface*** which are as follows:

1st way:

```
boolean val= stmt.execute("DQL Query");
if(val)
{
   java.sql.ResultSet rs =stmt.getResultSet( );
}
```

2nd way:

```
ResultSet rs =stmt.executeQuery("DQL") ;
```

PlaceHolder

- It is a parameter which holds ***Dynamic values*** at the run time by the ***User.***
- In case of ***JDBC***, place holder is represented as ***?***

Declaration of Placeholder in the Query:

```
String inqry="insert into btm.student values(?,?,?)";
String upqry ="update btm.student set name=? where id=?";
 String delqry="delete from btm.student where id=?";
String selqry="select * from btm.student where id =?";
```

Rules to set the data for a PlaceHolder:

- There are 3 different rules to set the data for a ***PlaceHolder*** namely:

a) We have to set the data for a ***PlaceHolder before the execution.***

b) The number of data must exactly match the number of ***PlaceHolder***.

c) We have to set the data for a ***PlaceHolder*** by using ***setXXX( ).***

+void *setXXX*(int placeholdernumber/placeholderindex ,*XXX data*)

Eg: **setInt(1,34);**

setString(2,"Rakesh");

setDouble(3,55.55);

Note:

- By default, the return type for ***getXXX( )*** method is ***respective data-types***, where as the return type for ***setXXX( )*** method is ***void.***

java.sql.PreparedStatement:

- It is an interface which is the part of ***JDBC API***, and its implementation are provided by respective Database Servers or Vendors as a part of ***JDBC Driver.***
- ***PreparedStatement*** interface is ***sub-interface of Statement interface.***
- It supports the concept of ***PlaceHolder*** to take dynamic values at the run time by the user
- It supports the concept of ***compile once and execute many times*** (***Execution plan*).**
- In case of ***PreparedStatement***, the ***query*** is passed at the time of implementation object creation of ***PreparedStatement*** interface but ***not in the execute( ).***

- Hence, ***PreparedStatement*** are also known as ***Pre-compiled Statement.***

Syntax:

java.sql.*PreparedStatement* pstmt=con.*prepareStatement*(Query);

- ***prepareStatement( )*** is factory or helper method which creates and returns implementation object of ***PreparedStatement interface.***
- Hence, the return type for ***prepareStatement( )*** is ***PreparedStatement interface.***

> ***Note:***
>
> - Hence, ***PreparedStatement*** is faster in performance compared to ***Statement Interface.***

Code to Insert multiple data into the Database Server using PreparedStatement interface with PlaceHolder:

```
package org.btm.jdbcApp;
import java.sql.*;
public class InsertMultiple
{
	public static void main(String[ ] args)
	{
		Connection con = null;
		PreparedStatement pstmt = null;
		String inqry="insert into btm.student values(?,?,?)";
		try {
			Class.forName("com.mysql.jdbc.Driver");
			System.out.println("Driver class loaded and register ");
			con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
			user=root&password=tiger");
			System.out.println("Connection Established");
			pstmt = con.prepareStatement(inqry);
			System.out.println("Platform created");
		// SET THE VALUE PLACE HOLDER BEFORE EXECUTION //
			pstmt.setInt(1, 1);
			pstmt.setString(2, "Dhanush");
			pstmt.setDouble(3, 75.00);
			pstmt.executeUpdate();

			pstmt.setInt(1, 2);
			pstmt.setString(2, "Kalyan");
			pstmt.setDouble(3, 70.00);
			pstmt.executeUpdate();
```

```
                pstmt.setInt(1, 3);
                pstmt.setString(2, "Adam");
                pstmt.setDouble(3, 80.00);
                pstmt.executeUpdate();
                 System.out.println("Data Inserted Successfuly");
             } catch (ClassNotFoundException | SQLException e)
            {
                e.printStackTrace();
            }
            finally
            {
                if (pstmt!= null)
                {
                    try
                    {
                        pstmt.close();
                    }
                    catch (SQLException e)
                    {
                        e.printStackTrace();
                    }
                }
                if(con!=null)
                {
                    try {
                        con.close();
                    }
                    catch (SQLException e)
                    {
                        e.printStackTrace();
                    }
                }
                System.out.println("All Costly Resources Closed");
            }
        }
}
```

Code to Fetch all Records from the cursor or buffer memory by using ResultSet Interface:

```
package org.btm.jdbcApp;
import java.sql.*;
public class FetchData
{
	public static void main(String[ ] args)
	{
		Connection con = null;
		PreparedStatement pstmt = null;
		ResultSet rs = null;
		String qry = "select * from btm.student ";

		try {
			Class.forName("com.mysql.jdbc.Driver");
			System.out.println("Driver class loaded and register ");
			con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
			user=root&password=tiger");
			System.out.println("Connection Established");
			pstmt = con.prepareStatement(qry);
			System.out.println("Platform created");
			rs = pstmt.executeQuery();
			while (rs.next())
			{
				int id = rs.getInt("id");
				String name = rs.getString(2);
				double per = rs.getDouble(3);
				System.out.println(id + " " + name + " " + per);
			}
		} catch (ClassNotFoundException | SQLException e)
		{
			e.printStackTrace();
		}
		Finally
		{
			if (rs != null) {
				try {
					rs.close();
				}
				catch (SQLException e)
				{
					e.printStackTrace();
				}
			}
			if (pstmt != null)
```

```
                {
                        try {
                                pstmt.close();
                        }
                        catch (SQLException e)
                        {
                                e.printStackTrace();
                        }
                }
                if (con != null) {
                        try {
                                con.close();
                        }
                        catch (SQLException e)
                        {
                                e.printStackTrace();
                        }
                }
                System.out.println("All Costly Resources Closed");
        }

} }
```

Code to Fetch a Record from the cursor or buffer memory by using ResultSet Interface:

```
package org.btm.jdbcApp;
import java.sql.*;
public class FetchData
{
        public static void main(String[ ] args)
        {
                Connection con = null;
                PreparedStatement pstmt = null;
                ResultSet rs = null;
                String qry = "select * from btm.student ";

                try {
                        Class.forName("com.mysql.jdbc.Driver");
                        System.out.println("Driver class loaded and register ");
                        con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
                        user=root&password=tiger");
                        System.out.println("Connection Established");
                        pstmt = con.prepareStatement(qry);
                        System.out.println("Platform created");
                        rs = pstmt.executeQuery();
```

```
                if (rs.next())
                {
                        int id = rs.getInt("id");
                        String name = rs.getString(2);
                        double per = rs.getDouble(3);
                        System.out.println(id + " " + name + " " + per);
                }
            } catch (ClassNotFoundException | SQLException e)
            {
                e.printStackTrace();
            } finally {
                if (rs != null) {
                    try {
                        rs.close();
                    }
                    catch (SQLException e)
                    {
                        e.printStackTrace();
                    }
                }
                if (pstmt != null)
                {
                    try {
                        pstmt.close();
                    }
                    catch (SQLException e)
                    {
                        e.printStackTrace();
                    }
                }
                if (con != null) {
                    try {
                        con.close();
                    }
                    catch (SQLException e)
                    {
                        e.printStackTrace();
                    }
                }
                System.out.println("All Costly Resources Closed");
            }

        }
}
```

Code to Fetch a Particular Record from the cursor or buffer memory where id=?

```
package org.btm.jdbcApp;
import java.sql.*;
import java.util.Scanner;
public class FetchDataOnId
{
	public static void main(String[ ] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter Id??");
		int id=sc.nextInt();
		sc.close();
		Connection con = null;
		PreparedStatement pstmt = null;
		ResultSet rs = null;
		String qry = "select * from btm.student where id=?";

		try {
			Class.forName("com.mysql.jdbc.Driver");
			System.out.println("Driver class loaded and register ");
			Con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
			user=root&password=tiger");
			System.out.println("Connection Established");
			pstmt = con.prepareStatement(qry);
			System.out.println("Platform created");
	// SET THE VALUE FOR PLACEHOLDER BEFORE EXECUTION //
			pstmt.setInt(1,id);
			rs = pstmt.executeQuery();
			if (rs.next())
			{
				String name = rs.getString(2);
				double per = rs.getDouble(3);
				System.out.println(name +"  "+ per);
			}
			else
			{
				System.err.println("No Data found for Id " +id);
			}
			catch (ClassNotFoundException|SQLException e)
```

```
                {
                        e.printStackTrace();
                }
        }
        finally {
                if (rs != null)
                {
                        try {
                                rs.close();
                        }
                        catch (SQLException e)
                        {
                                e.printStackTrace();
                        }
                }
                if (pstmt != null) {

                        try {
                                pstmt.close();
                        }
                        catch (SQLException e)
                        {
                                e.printStackTrace();
                        }
                }
                if (con != null) {
                        try {
                                con.close();
                        }
                        catch (SQLException e)
                        {
                                e.printStackTrace();
                        }
                }
                System.out.println("Closed All Costly Resources");
        }

    }
}
```

Code to Fetch a Particular Record from the cursor or buffer memory where name=?

```
package org.btm.jdbcApp;
import java.sql.*;
import java.util.Scanner;
public class FetchDataOnName
{
	public static void main(String[ ] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter Name??");
		String name=sc.next();
		sc.close();
		Connection con = null;
		PreparedStatement pstmt = null;
		ResultSet rs = null;
		String qry = "select * from btm.student where name=?";

		try {
			Class.forName("com.mysql.jdbc.Driver");
			System.out.println("Driver class loaded and register ");
			Con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
			user=root&password=tiger");
			System.out.println("Connection Established");
			pstmt = con.prepareStatement(qry);
			System.out.println("Platform created");
	// SET THE VALUE FOR PLACEHOLDER BEFORE EXECUTION //
			pstmt.setInt(1,id);
			rs=pstmt.executeQuery();
			if (rs.next())
			{
				int id= rs.getInt(1);
				double per=rs.getDouble(3);
				System.out.println(id +"  "+ per);
			}
			else
			{
				System.err.println("No Data found for Name " +name);
			}
```

```
            catch (ClassNotFoundException | SQLException e)
            {
                e.printStackTrace();
            }
        }
        finally
        {
            if (rs != null)
            {
                try {
                    rs.close();
                }
                catch (SQLException e)
                {
                    e.printStackTrace();
                }
            }
            if (pstmt != null) {

                try {
                    pstmt.close();
                }
                catch (SQLException e)
                {
                    e.printStackTrace();
                }
            }
            if (con != null) {
                try {
                    con.close();
                }
                catch (SQLException e)
                {
                    e.printStackTrace();
                }
            }
            System.out.println("Closed All Costly Resources");
        }

    }
}
```

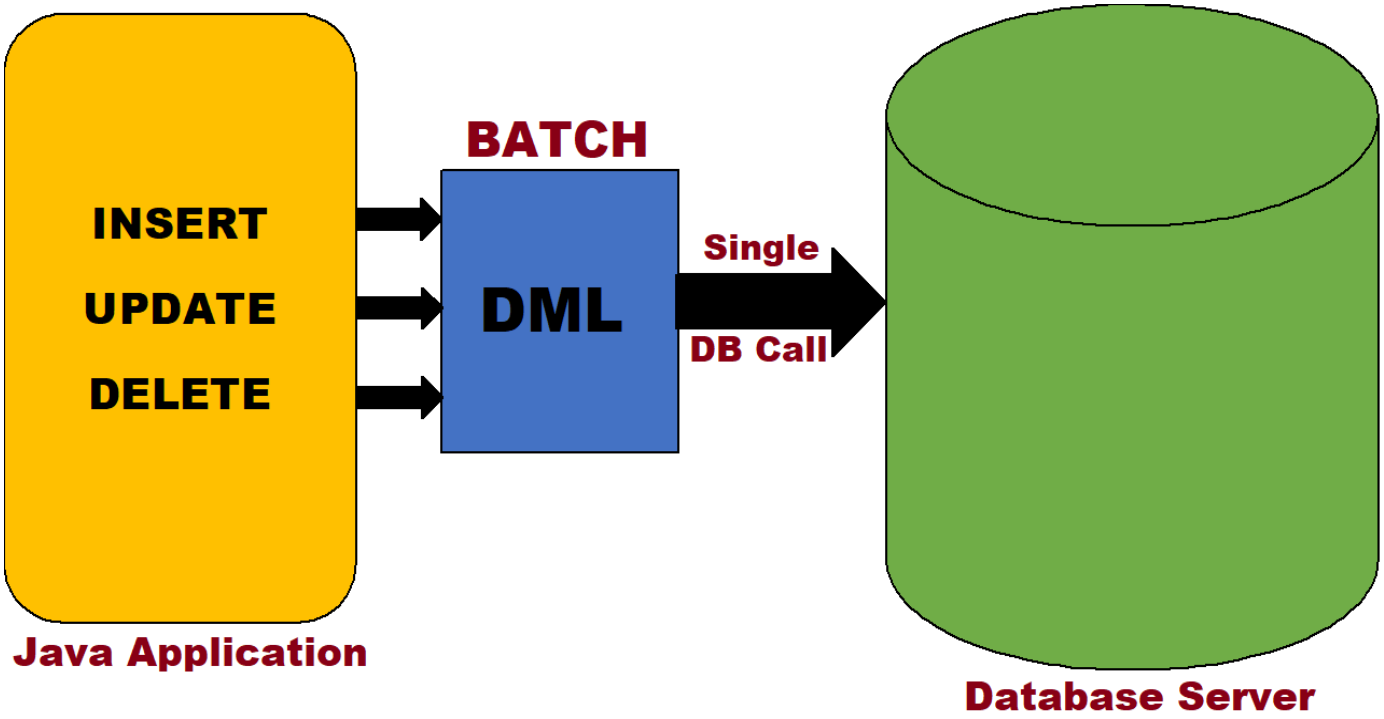
Code for Login Validation using standard steps of JDBC:

```
package org.btm.jdbcApp;
import java.sql.*;
import java.util.Scanner;
public class LoginValidation
{
	public static void main(String[ ] args)
	{
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter Name??");
		String name=sc.next();
		System.out.println("Enter Password??");
		String password=sc.next();
		sc.close();
		Connection con = null;
		PreparedStatement pstmt = null;
		ResultSet rs = null;
		 String qry = "select username from btm.user where name=? and
                                                                password=?";
		try
		{
			Class.forName("com.mysql.jdbc.Driver");
			System.out.println("Driver class loaded and register ");
			con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
			user=root&password=tiger");
			System.out.println("Connection Established");
			pstmt = con.prepareStatement(qry);
			System.out.println("Platform created");
		// SET THE VALUE PLACE HOLDER BEFORE EXECUTION //
			pstmt.setString(1,name);
			pstmt.setString(2,password);
			rs=pstmt.executeQuery();
			if (rs.next())
			{
				String username=rs.getString(1);
				System.out.println("Welcome "+username);
			}
			else
			{
				System.err.println("Invalid name or password");
			}
```

```
        }
        catch (ClassNotFoundException | SQLException e)
        {
                e.printStackTrace();
        }
        finally
        {
                if (rs!= null) {
                        try {
                                rs.close();
                        }
                        catch (SQLException e)
                        {
                                e.printStackTrace();
                        }
                }
                if (pstmt!= null)
                { try
                {
                        pstmt.close();
                }
                catch (SQLException e)
                {
                        e.printStackTrace();
                }
                }
                if (con!= null) {
                        try {
                                con.close();
                            }
                        catch (SQLException e)
                        {
                                e.printStackTrace();
                        }
                }
                System.out.println("All Costly Resources are Closed");
        }
}
```

BATCH UPDATE

- Each and every individual DB call which is made from java application to the Database Server is considered to be a ***Costly Operation.***
- The more no of DB call reduces the performance of an application. Hence, it is better to use ***Batch update.***



Definition for Batch: Batch is collection of only ***DML Queries.***

Need for Batch: Batch update is needed to make one single DB call and affect multiple records of multiple tables ***at once.***

Advantage of Batch

- It ***greatly*** improves the performance of an application.

Process of Batch

- It is applicable only for ***DML query.***
- In case of ***Batch Update***, after adding all the ***DML queries*** into the batch, the entire batch will be executed only once by making ***one single DB call.***
- It can be used both with ***Statement*** as well as ***PreparedStatement*** interface.
- There are two different methods assosiated with Batch namely:
  a) addBatch( ) b) executeBatch( )

a) addBatch( ): It is used to add all the ***DML query into the Batch.***

b) executeBatch( ): It is used to execute all ***DML queries*** present in the Batch only once by ***making one single DB call.***

> ❖ ***executeBatch( )*** returns an ***Integer array.***

- The ***size*** of this array represent the total number of ***DML queries added into the Batch.***
- ***Sequence*** of the Integer array is same as the ***sequence in which the DML queries are added into the batch.***

Code for Batch with Statement interface:

```
package org.btm.jdbcApp;
import java.sql.*;
public class BatchStmt
{
	public static void main(String[ ] args)
	{
	Connection con = null;
	Statement stmt = null;
	String inqry =" insert into btm.user values( 1 , ' Adam ' , 65.99 ) ";
	String delqry =" delete from btm.student where perc=55.55 ";
	String upqry =" update btm.student set perc=89.99 where id=4 ";
	try {
		Class.forName("com.mysql.jdbc.Driver");
		System.out.println("Driver class loaded and register ");
		con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
		user=root&password=root");
		System.out.println("Connection Established");
		stmt = con.createStatement();
		System.out.println("Platform created");

		// ADD DML QUERIES INTO BATCH //

		stmt.addBatch(inqry);
		stmt.addBatch(delqry);
		stmt.addBatch(upqry);

		// EXECUTE BATCH //
		int arr[ ] = stmt.executeBatch( );
		for (int i : arr)
```

```
                {
                        System.out.print(i);
                 }
        }
        catch (ClassNotFoundException | SQLException e)
        {
                e.printStackTrace();
        }
        finally
        {
                if (stmt != null)
                {
                        try {
                                stmt.close();
                        } catch (SQLException e)
                        {
                                e.printStackTrace();
                        }
                }
                if (con != null)
                {

                        try {
                                con.close();
                        } catch (SQLException e)
                        {
                                e.printStackTrace();
                        }
                }
                System.out.println("Closed All Costly Resources");
        }
    }
}
```

Code for Batch with PreparedStatement Interface:

```
package org.btm.jdbcApp;
import java.sql.*;
public class BatchPstmt
{
        public static void main(String[ ] args)
        {
          Connection con = null;
          PreparedStatement pstmt=null;
          PreparedStatement pstmt1=null;
          String inqry = "insert into btm.user values(42,'Ayush',6544)";
```

```
 String delqry = "delete from btm.student where perc=55.55";
try {
        Class.forName("com.mysql.jdbc.Driver");
        System.out.println("Driver class loaded and register ");
        con=DriverManager.getConnection("jdbc:mysql://localhost:3306?
                user=root&password=root");
        System.out.println("Connection Established");
        pstmt = con.prepareStatement(inqry);
        System.out.println("Platform created");
        // ADD DML QUERY INTO BATCH //
        pstmt.addBatch( );
        int arr[ ] = pstmt.executeBatch();
        for (int i : arr)
        {
                System.out.print(i);
        }
        pstmt1 = con.prepareStatement(delqry);
        System.out.println("Platform  created");
        pstmt1.addBatch();
        int arr1[ ] = pstmt1.executeBatch( );
        for (int j : arr1)
        {
                System.out.print(j);
        }

}
catch (ClassNotFoundException | SQLException e)
{
        e.printStackTrace( );
}
finally
{
        if (pstmt1 != null)
        {
                try
                {
                        pstmt1.close( );
                }
                catch (SQLException e)
                {
                        e.printStackTrace( );
                }
        }
        if (pstmt != null)
        {
                try
                {
```

```
                pstmt.close( );
            }
            catch (SQLException e)
            {
                e.printStackTrace( );
            }
        }
        if (con != null)
        {
            try
            {
                con.close( );
            }
            catch (SQLException e)
            {
                e.printStackTrace( );
            }
        }
        System.out.println("All Costly Resources are Closed");
    }
  }
}
```

- Whenever we use ***Batch with Statement Interface***, then ***one single batch*** can contain ***all types of DML Queries in it.***
- Whenever we use ***Batch with PreparedStatement*** Interface, then ***one single batch*** can contain ***only one DML Query in it.***

❖ Hence, ***Batch with Statement*** Interface is ***Faster in Performance.***

JDBC Transaction

Exchange of data or information between two media is known as Transaction.

- By default, the ***AutoCommit*** mode is set to ***boolean true*** value because of which all the data are automatically saved into the Database Server whenever we perform any Database operation.
- We can explicitly ***disable AutoCommit*** mode by using ***setAutoCommit( )*** and passing a boolean false argument.

Syntax: **+void *setAutoCommit( )***

Eg: **con.setAutoCommit**(false);

Note:

- We have to disable the ***AutoCommit*** mode before we ***begin the transaction but after establishing a connection with Database Server.***

- Once the ***AutoCommit*** mode is disabled we have to explicitly save the data into the Database Server by using ***commit( )*** at the ***end of the transaction.***

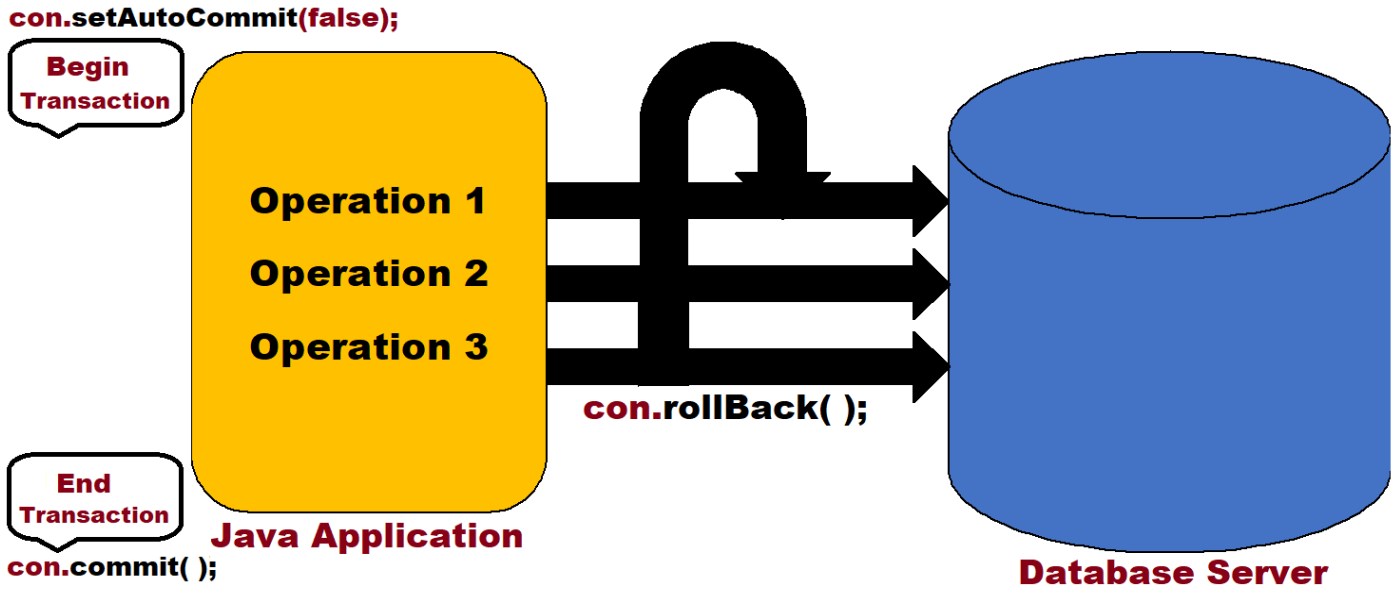
Syntax: **+void *commit( )***

Eg: **con.commit( );**

- If any one of the Database operation fails, then the ***rollback*** operation is called which is used to rollback ***Entire executed Database Operation and transaction starts from beginning.***

Syntax: **+void rollBack( )**

Eg: **con.rollBack( );**



Definition for JDBC Transaction:

- ***JDBC Transaction*** is considered to be a ***Single Business Unit*** Which may have ***multiple SQL Statements which must be executed.***

Need for JDBC Transaction:

- ***JDBC Transaction*** is needed to maintain ***Data consistency in the Database Server.***

Advantage of JDBC Transaction:

- It is used to achieve ***ACID properties or rules*** where ***A refers to Atomicity, C for Consistency, I for Isolation and D for Durability.***

Atomicity: Atomicity means ***Do Everything or Do Nothing.***

- ***Do Everything*** refers to ***Complete or Successful transaction*** where if all the Database operation are successfully executed, ***then the data's are saved into the Database Server leading to Data Consistency.***
- ***Do Nothing*** refers to ***Incomplete or Unsuccessful transaction*** where if any one of Database operation fails, then the ***rollBack operation*** is called which is used to roll back the ***Entire executed Database operation and transaction starts from beginning without saving any data into the Database Server due to Data Inconsistency.***

Code for JDBC Transaction:

```
package org.btm.jdbcApp;
import java.sql.*;
import java.util.Scanner;
public class Transaction
{
	public static void main(String[ ] args)
	{
		Connection con = null;
		PreparedStatement pstmt = null;
		PreparedStatement pstmt1 = null;
		String inqry1= "insert into btm.student1 values(?,?,?,?)";
		String inqry2 = "insert into btm.student2 values(?,?,?)";
		Scanner sc =new Scanner(System.in);
		System.out.println("Enter Id?");
		int id =sc.nextInt( );
		System.out.println("Enter Name?");
		String name=sc.next( );
		System.out.println("Enter Dept?");
		String dept=sc.next( );
```

```
System.out.println("Enter Perc?");
double perc=sc.nextDouble( );
System.out.println("Enter Place?");
String place=sc.next( );
sc.close( );
try
{
        Class.forName("com.mysql.jdbc.Driver");
        con =DriverManager.getConnection("jdbc:mysql://localhost:3306?
        user=root&password=tiger");
        // Disable AutoCommit( ) //
        con.setAutoCommit(false);
        pstmt=con.prepareStatement(inqry1);
        pstmt.setInt(1, id);
        pstmt.setString(2, name);
        pstmt.setString(3, dept);
        pstmt.setDouble(4,perc);
        pstmt.executeUpdate( );
        System.out.println("Student Educational Details Executed");

        pstmt1=con.prepareStatement(inqry2);
        pstmt1.setInt(1, id);
        pstmt1.setString(2, name);
        pstmt1.setString(3, place);
        pstmt1.executeUpdate( );
        System.out.println("Student Personal Details Executed");
        con.commit( );

}
catch (ClassNotFoundException | SQLException e)
{
        try
        {
                con.rollback( );
                System.out.println("Operations Rolled Back");
        }
        catch (SQLException e1)
        {
                e1.printStackTrace( );
        }
        e.printStackTrace( );
}
finally
{
        if (pstmt1!= null)
        {
                try
```

```
                    {
                        pstmt.close( );
                    }
                    catch (SQLException e)
                    {
                        e.printStackTrace( );
                    }
                }
                if (pstmt!= null)
                {
                    try
                    {
                        pstmt.close( );
                    }
                    catch (SQLException e)
                    {
                        e.printStackTrace( );
                    }
                }
                if (con!= null)
                {
                    try
                    {
                        con.close( );
                    }
                    catch (SQLException e)
                    {
                        e.printStackTrace( );
                    }
                }
            System.out.println("All Costly Resources are Closed");
            }
    }
}
```

Savepoint:

Syntax:

java.sql.Savepoint sp=con.setSavepoint( );

- ***setSavepoint( )*** is a ***factory or helper method*** which is used to ***create and return implementation object of Savepoint interface.***
- Hence, the return type of ***setSavepoint( )*** is ***Savepoint interface.***

Code for JDBC Transaction using Savepoint Interface:

```
package org.btm.jdbcApp;
import java.sql.*;
import java.util.Scanner;
public class Transaction
{
	public static void main(String[ ] args)
	{
		Connection con = null;
		PreparedStatement pstmt = null;
		PreparedStatement pstmt1 = null;
		Savepoint sp= null;
		String inqry1= "insert into btm.student1 values(?,?,?,?)";
		String inqry2 = "insert into btm.student2 values(?,?,?)";
		Scanner sc =new Scanner(System.in);
		System.out.println("Enter Id?");
		int id =sc.nextInt( );
		System.out.println("Enter Name?");
		String name=sc.next( );
		System.out.println("Enter Dept?");
		String dept=sc.next( );
		System.out.println("Enter Perc?");
		double perc=sc.nextDouble( );
		System.out.println("Enter Place?");
		String place=sc.next( );
		sc.close( );
		try
		{
			Class.forName("com.mysql.jdbc.Driver");
			con =DriverManager.getConnection("jdbc:mysql://localhost:3306?
			user=root&password=tiger");
			// Disable AutoCommit( ) //
			con.setAutoCommit(false);
			pstmt=con.prepareStatement(inqry1);
			pstmt.setInt(1, id);
			pstmt.setString(2, name);
			pstmt.setString(3, dept);
			pstmt.setDouble(4,perc);
			pstmt.executeUpdate( );
			System.out.println("Student Educational Details Executed")

			sp=con.setSavepoint( );

			pstmt1=con.prepareStatement(inqry2);
			pstmt1.setInt(1, id);
			pstmt1.setString(2, name);
```

```
            pstmt1.setString(3, place);
            pstmt1.executeUpdate( );
            System.out.println("Student Personal Details Executed");
            con.commit( );

        }
        catch (ClassNotFoundException | SQLException e)
        {
            try
            {
                con.rollback(sp);
                con.commit( );
                System.out.println("Operations Rolled Back");
            }
            catch (SQLException e1)
            {
                e1.printStackTrace( );
            }
            e.printStackTrace( );
        }
        finally
        {
            if (pstmt1!= null)
            {
                try
                {
                    pstmt.close( );
                }
                catch (SQLException e)
                {
                    e.printStackTrace( );
                }
            }
            if (pstmt!= null)
            {
                try
                {
                    pstmt.close( );
                }
                catch (SQLException e)
                {
                    e.printStackTrace( );
                }
            }
            if (con!= null)
            {
                try
```

```
                        {
                                con.close( );
                        }
                        catch (SQLException e)
                        {
                                e.printStackTrace( );
                        }
                }
        System.out.println("All Costly Resources are Closed");
        }
    }
}
```