Unit 4: JDBC

- JDBC stands for Java Database Connectivity which is a technology that allows Java applications to interact with relational databases and execute SQL queries against the databases
- JDBC is a java API to connect and execute query with the database.
- JDBC API uses jdbc drivers to connect with the database.
- The JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.

Contd…

- JDBC helps to write Java applications that manage these three programming activities:
  - Connect to a data source, like a database
  - Send queries and update statements to the database
  - Retrieve and process the results received from the database in answer to your query

Contd….

- Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database.
- Java can be used to write different types of executable, such as −
  - Java Applications
  - Java Applets
  - Java Servlets
  - Java ServerPages (JSPs)
  - Enterprise JavaBeans (EJBs).
- All of these different executable are able to use a JDBC driver to access a database, and take advantage of the stored data.

Contd….

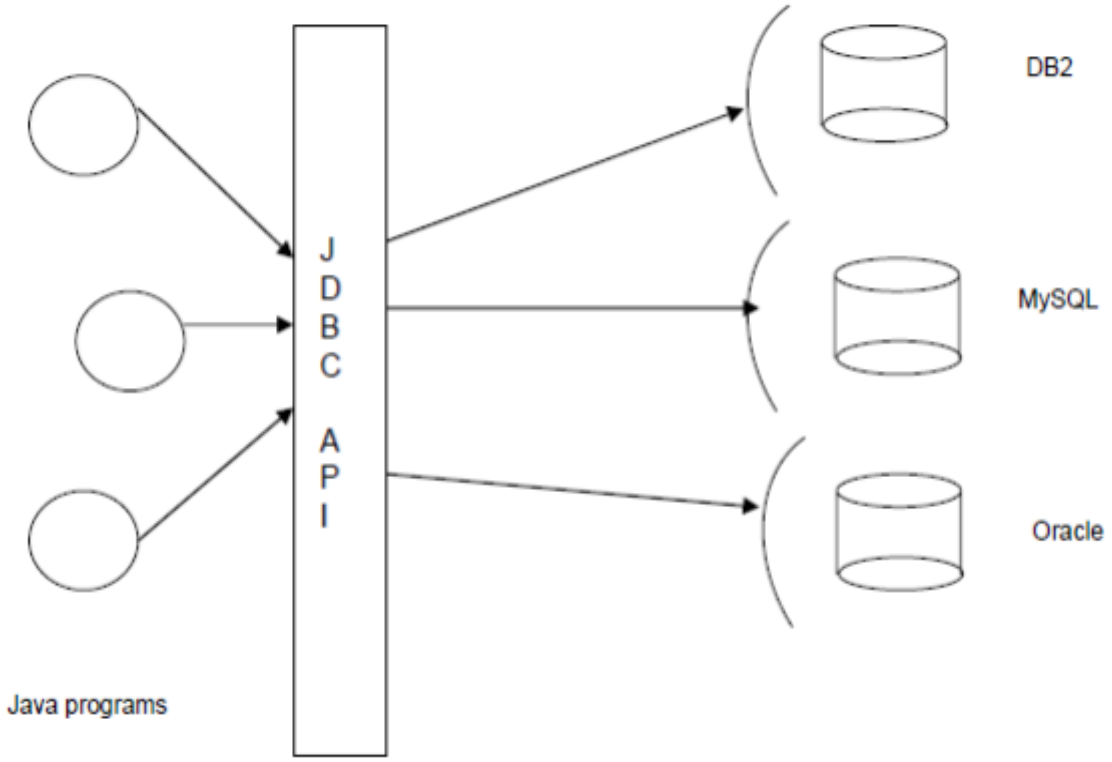


Figure 1 JDBC and Databases

How JDBC talks with Databases?

- Databases are proprietary applications that are created by noted companies like Oracle, Microsoft and many more. These could be developed in any language like C, C++ etc by these vendors.
- Every database will expose something called Driver through with one can interact with database for SQL operations.
- A driver is like a gateway to the database.
- Therefore, for any Java application to work with databases, it must use the driver of that database.

Database Drivers

- A driver is not a hardware device. It's a software program that could be written in any language.
- Every database will have its own driver program.
- Given a driver program of a particular database, the challenge is how to use it to talk with database.
- To understand this, we need to know the different types of drivers.

ODBC

- (Open Database Connectivity) is a standard database access method developed by Microsoft Corporation.
- ODBC makes it possible to access data from any application, regardless of which database management system (DBMS) is handling the data.

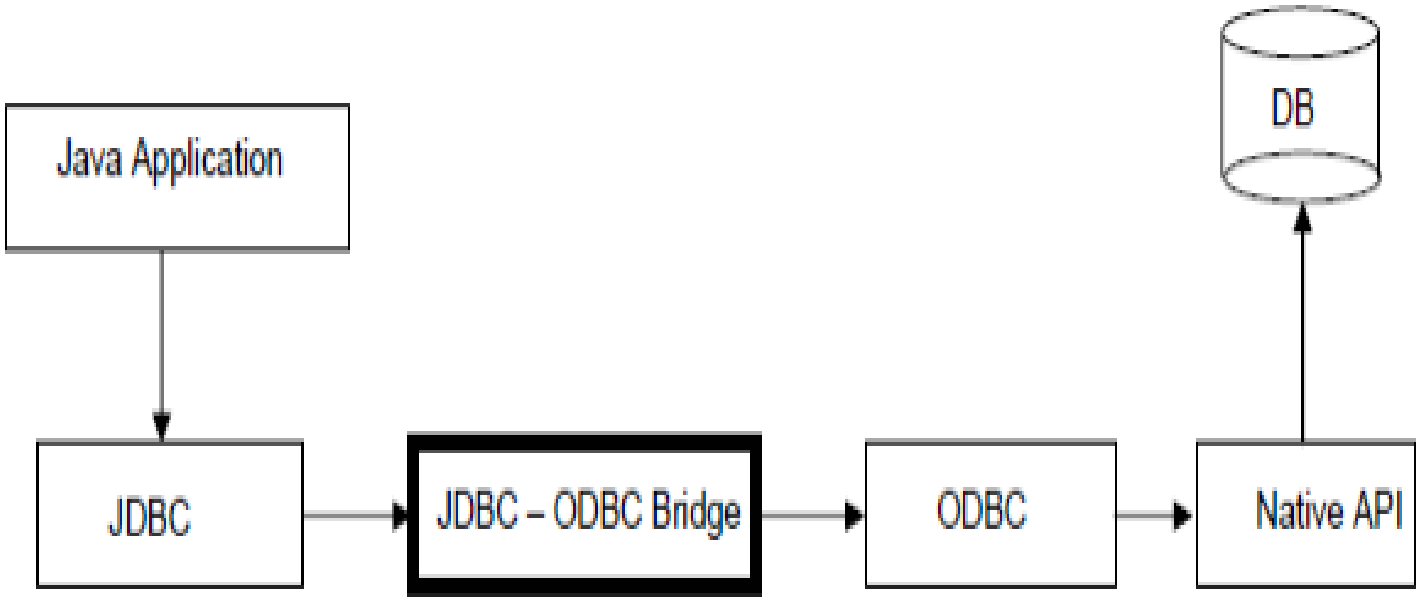
Before JDBC, ODBC API was the database API to connect and execute query with the database.

But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

- *ODBC is hard to learn.*
- *ODBC has a few commands with lots of complex options. The preferred style in the Java programming language is to have simple and intuitive methods, but to have lots of them.*
- *ODBC relies on the use of void* pointers and other C features that are not natural in the Java programming language.*
- *An ODBC-based solution is inherently less safe and harder to deploy than a pure Java solution.*

JDBC Divers Type

- There are basically 4 different types of database drivers as described below:
- **1. JDBC-ODBC Bridge Driver (Type -1 driver)**

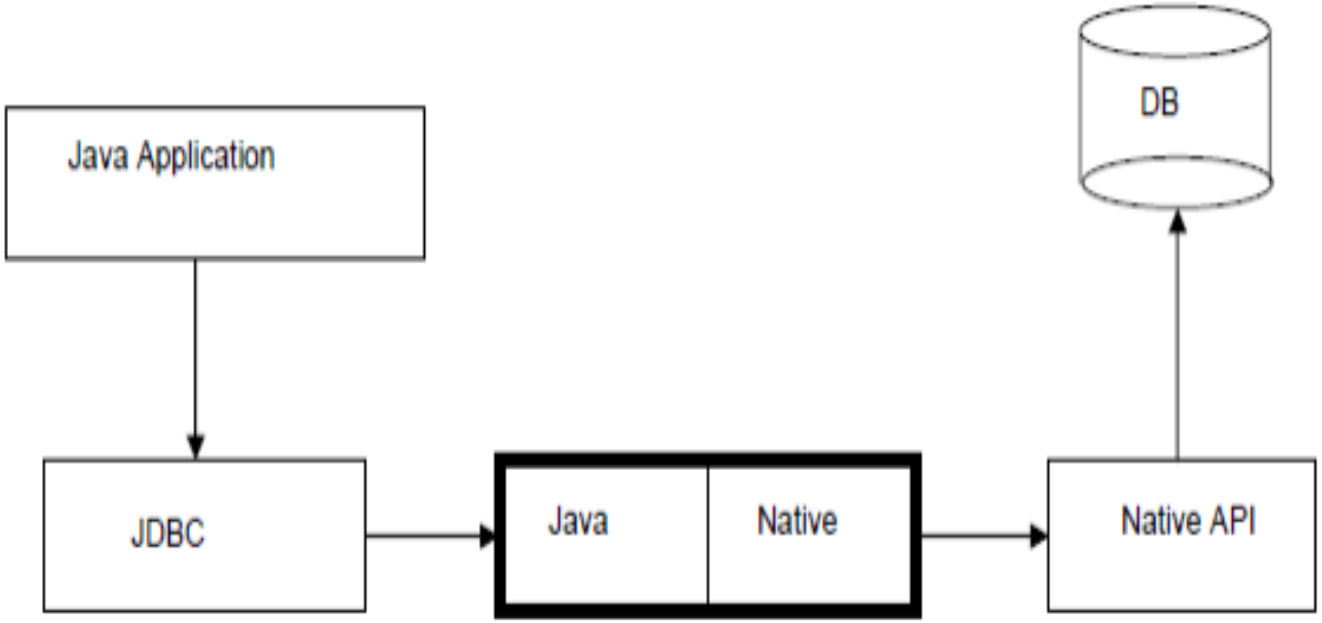


Contd….

This is also called as **Type-1 driver**

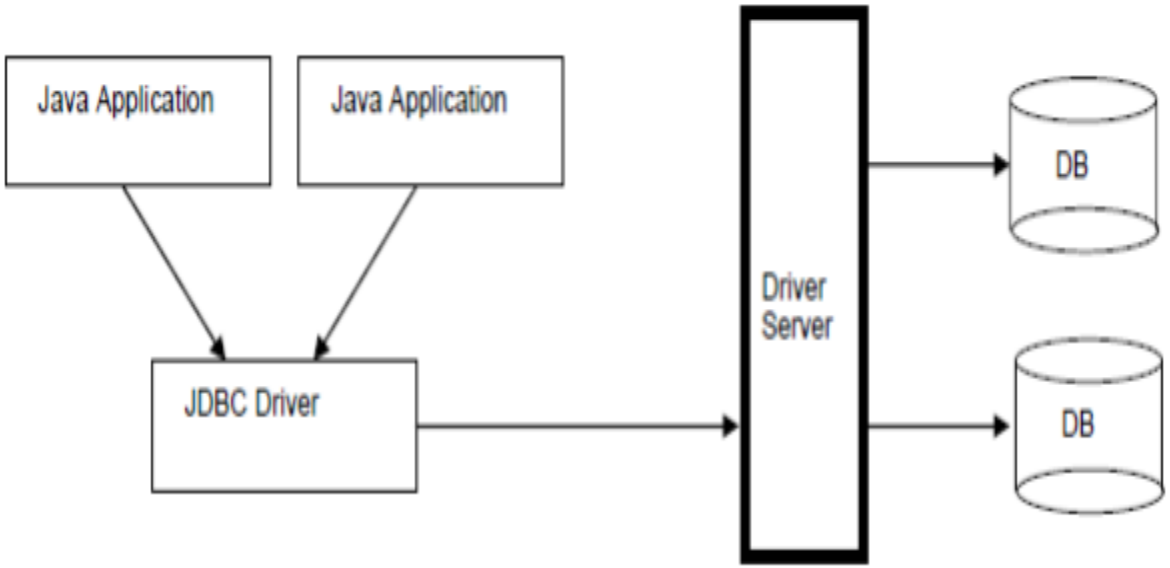
- As you can notice from the above figure, this driver translates the JDBC calls to ODBC calls in the ODBC layer.
- The ODBC calls are then translated to the native database API calls.
- Because of the multiple layers of indirection, the performance of the application using this driver suffers seriously.
- This type of driver is normally used for training purposes and never used in commercial applications.
- The only good thing with this driver is that you can access any database with it.
- In simple words, with this driver for JDBC to talk with vendor specific native API, following translations will be made JDBC - > ODBC -> Native .It’s a 2 step process.
- The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

Partly Java and Partly Native Driver (Type -2 driver)



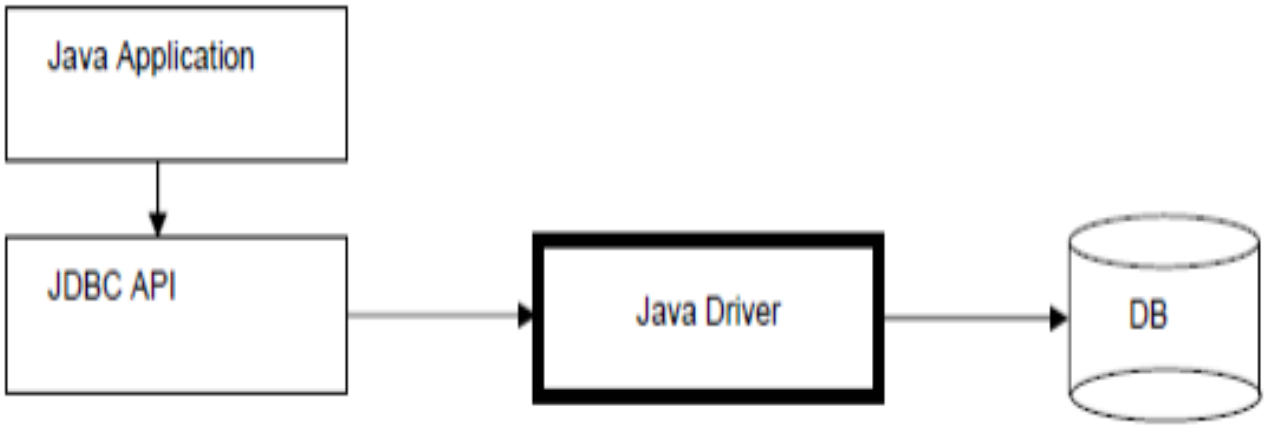
- This is also called as **Type-2 driver**
- As the name suggests, this driver is partly built using Java and partly with vendor specific API
- With this driver, the JDBC calls are translated to vendor specific native calls in just one step. It completely eliminates the ODBC layer. Because of this ODBC layer elimination, the applications using this driver perform better.
- The Oracle Call Interface (OCI) driver is an example of a **Type 2 driver**

- 3. **Intermediate Database access Driver Server (Type -3 Driver)**



- This is also called as **Type-3 driver**
- If you look at the Type-2 driver configuration, we only access one database at a time. However there will be situations where multiple Java programs need to access to multiple databases. This is where Type-3 driver is used.
- It acts as a middleware for the applications to access several databases. The Type-3 configuration internally may use Type-2 configuration to connect to database
- IDS JDBC Driver is an example

- 4. **Pure Java Drivers (Type -4 Drivers)**



- These are called as **Type-4 drivers.**
- Because of the widespread usage of Java, to facilitate easy and faster access to databases from Java applications, database vendors built the driver itself using Java like good friends helping each other. This is really cool as it completely eliminates the translation process.
- JDBC is Java based technology, and with the driver also written in Java, it can directly invoke the driver program as if it were any other Java program.
- This is the driver that all the J2EE applications use to interact with databases.
- Because this is a Java to Java interaction, this driver offers the best performance.
- MySQL's Connector/J driver is a **Type 4 driver**.

Which Driver should be Used?

- If you are accessing one type of database, such as Oracle,MYSQL, Sybase, or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

Driver JAR Files

- You need to obtain the JAR file in which the driver for your database is located (e.g. for Derby database, you need the file derbyclient.jar)
- Include the driver JAR file on the class path
- Launch programs from the command line with java –classpath .;driverJar ProgramName

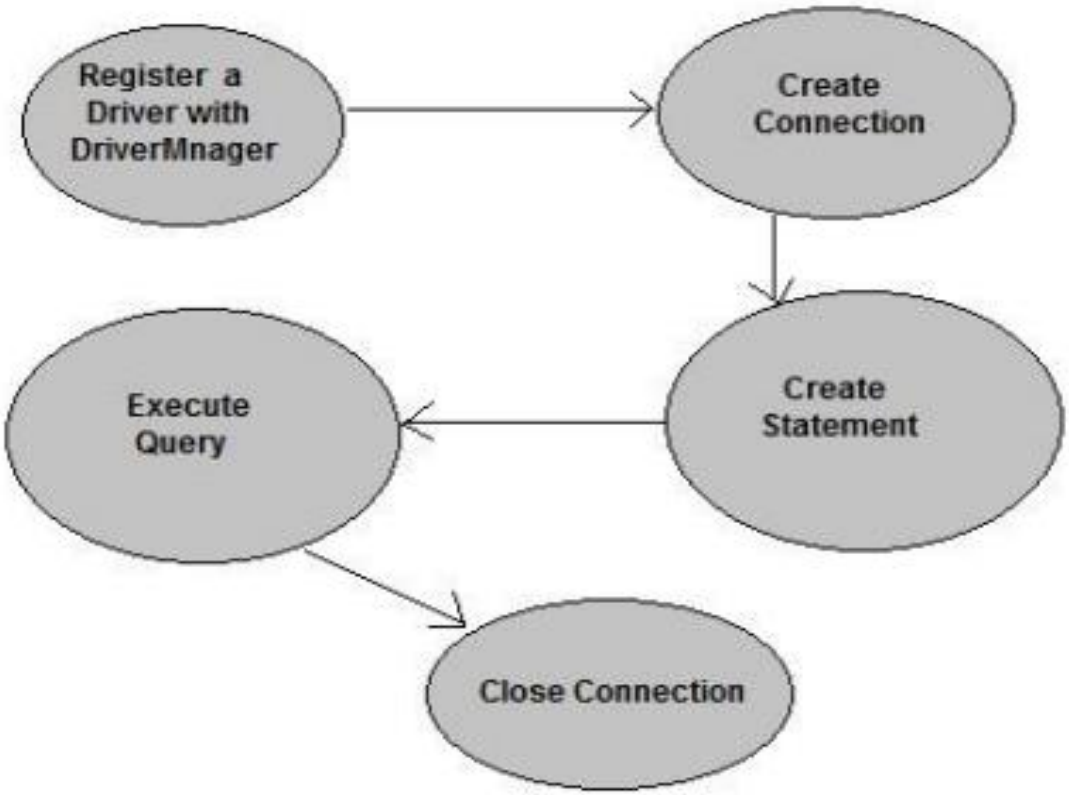
- JDBC API JDBC technology is nothing but an API.
- It is a set of classes and interfaces that our Java programs use to execute the SQL queries against the database.
- The fundamental idea behind JDBC is that, Java programs use JDBC API, and JDBC API will in turn use the driver to work with the database.
- Therefore, to work with MySQL database, we need to configure the JDBC with all the MySQL information. This information is nothing but:
  - *Name of the MySQL driver class*
  - *URL of the database schema.*
- In simple words, for a Java program to work with the database, we first need to configure JDBC with the above database info and then make the Java program use the JDBC.

- Few class and interfaces of JDBC API are as follows:

Class/Interface	Description
`DriverManager`	This classes is used for managing the Drivers
`Connection`	It's an interface that represents the connection to the database
`Statement`	Used to execute static SQL statements
`PreparedStatement`	Used to execute dynamic SQL statements
`CallableStatement`	Used to execute the database stored procedures
`ResultSet`	Interface that represents the database results
`ResultSetMetaData`	Used to know the information about a table
`DatabaseMetadata`	Used to know the information about the database
`SQLException`	The checked exception that all the database classes will throw.

- JDBC programming is the simplest of all.
- There is a standard process we follow to execute the SQL queries. Following lists the basic steps involved in any JDBC program.

1. Import necessary packages
2. Load and Register the Driver
3. Establish the connection to the database
4. Create the statements(using Statement/ PreparedStatement/ CallableStatement)
5. Execute the statements
6. Process the results
7. Close the statements
8. Close the connection.


Register a Driver with DriverMnager
Create Connection
Create Statement
Execute Query
Close Connection

1. The **forName()** method of Class class is used to register the driver class.

- This method is used to dynamically load the driver class.
- Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

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

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

- Syntax of getConnection() method

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

2) public static Connection getConnection(String url,String name,String passwrd) throws SQLException

Example:

Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/world","root","Prithvi123#"**);**

3. The **createStatement()** method of Connection interface is used to create statement.

The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

public Statement createStatement()throws SQLException

Example to create the statement object ***Statement stmt=con.createStatement();***

4. The **executeQuery()** method of Statement interface is used to execute queries to the database.

- This method returns the object of **ResultSet** that can be used to get all the records of a table.
- Syntax of **executeQuery()** method

public ResultSet executeQuery(String sql)throws SQLException

- Example to execute query ResultSet

```
rs=stmt.executeQuery("select * from emp");
    while(rs.next())
    {
            System.out.println(rs.getInt(1)+"  "+rs.getString(2));
    }
```

- By closing connection object **statement** and **ResultSet** will be closed automatically.
- The **close()** method of Connection interface is used to close the connection.
- Syntax of **close()** method

public void close()throws SQLException

- **Example to close connection**

con.close();

Database URLs

- Database URL Formulation After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection()** method.
- For easy reference, let us list the three overloaded **DriverManager.getConnection()** methods −

- ***getConnection(String url)***
- ***getConnection(String url, Properties prop)***
- ***getConnection(String url, String user, String password)***

Here each form requires a database URL. A database URL is an address that points to your database. Formulating a database URL is where most of the problems associated with establishing a connection occurs.

- Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	jdbc:mysql://hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	jdbc:db2:hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	jdbc:sybase:Tds:hostname: port Number/databaseName

For MySQL Connection:

- **Driver class:** The driver class for the mysql database is com.mysql.jdbc.Driver.
- **Connection URL**: The connection URL for the mysql database is jdbc:mysql://localhost:3306/mydb where
  - jdbc is the API,
  - mysql is the database,
  - localhost is the server name on which mysql is running, we may also use IP address,
  - 3306 is the port number and
  - mydb is the database name..
- **Username:** The default username for the mysql database is root.
- **Password:** Password is given by the user at the time of installing the mysql database.

1. Reading Data:

- **Executing SQL statements**

For SELECT queries, use

*ResultSet rs = statement.executeQuery("SELECT * FROM Books")*

- Process **ResultSet object using**

```
while (rs.next())
{
    // look at a row of the result set
}
```

- *Inside each row, get field value using*

```
String isbn = rs.getString(1); // using column index
double price = rs.getDouble("Price"); // using column name
```

Commonly used methods of Statement interface:

public ResultSet executeQuery(String sql): is used to execute SELECT query. It returns the object of ResultSet.
public int executeUpdate(String sql): is used to execute specified query, it may be create, drop, insert, update, delete etc.
public boolean execute(String sql): is used to execute queries that may return multiple results.
public int[] executeBatch(): is used to execute batch of commands.

Reading Data Example

```
import java.sql.*;
public class App {
   public static void main(String[] args) throws Exception {
      try
      {
        Class.forName("com.mysql.cj.jdbc.Driver");
        Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/world","root","Prithvi123#");
       Statement stmt=con.createStatement();
       ResultSet rs=stmt.executeQuery("select name from country where population<10000");
       while(rs.next())
       {
           System.out.println(rs.getString(1));
       }
      stmt.close();
Con.close();
       }
      catch(Exception e)
      {
         System.out.println(e);
      }
   }
}
```

Contd….

- This is the most important and widely performed operation. In order to retrieve the data, we use the following method on the statement object:

public ResultSet executeQuery (String sql)

- The above method takes the query and returns a **ResultSet object that contains all the records returned by the database. We should then iterate over it and display the data. To retrieve the data from the ResultSet, there are several get methods available based on the type of the data.**
- We use the **executeQuery() method to execute the SELECT query. This method will return a ResultSet object as shown below:**

ResultSet rs = stmt.executeQuery(myquery);

Contd….

- A resultset object can be imagined like a table of records with a pointer at the beginning of the table as shown below:
- To read the records, we need to move the pointer to the record using the **next() method until the pointer reaches the end of the records. This is what the while loop does. Once we are in the loop, we need to read the record that the pointer points using the get methods.**
- The get methods can either take the column number starting from 1 or the column name itself.
- In our example, the column numbers have been used.
- Following are the most commonly used get methods:

  public String getString() //Used for VARCHAR columns

  public int getInt() //Used for NUMERIC columns

  public Date getDate() //Used for DATE columns.

Commonly used methods of ResultSet interface

`public boolean next()`	is used to move the cursor to the one row next from the current position.
`public boolean previous()`	is used to move the cursor to the one row previous from the current position.
`public boolean first()`	is used to move the cursor to the first row in result set object.
`public boolean last()`	is used to move the cursor to the last row in result set object.
`public boolean absolute(int row)`	is used to move the cursor to the specified row number in the ResultSet object.
`public boolean relative(int row)`	is used to move the cursor to the relative row number in the ResultSet object, it may be positive or negative.
`public int getInt(int columnIndex)`	is used to return the data of specified column index of the current row as int.
`public int getInt(String columnName)`	is used to return the data of specified column name of the current row as int.
`public String getString(int columnIndex)`	is used to return the data of specified column index of the current row as String.
`public String getString(String columnName)`	is used to return the data of specified column name of the current row as String.

Writing Data

- **Note:**If the query is SELECT, use the **executeQuery() method. For all other queries (CREATE, INSERT, DELETE, UPDATE etc) use the executeUpdate() method.**

Writing Data Example

```
import java.sql.*;
public class App {
    public static void main(String[] args) throws Exception {
        try
        {
          Class.forName("com.mysql.cj.jdbc.Driver");
          Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/student","root","Prithvi123#");
         Statement stmt=con.createStatement();
         int re=stmt.executeUpdate("insert into user(name,email,phone) values('shyam','xyz@gmail.com',123456)");
           System.out.println(re +"row(s) inserted");
           stmt.close();
           con.close();

         }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

Task1:

- **Write a java program that allows a user to insert values to a table of particular database (Suppose database is in MySql server). The program should take the values to insert from console.**

Task2:

- **WAP using JDBC to display the records from a table of given database (Suppose database is in MySql server). Assume the following table : result(roll_no , course_name ,marks_obtained)**
- **The program should read the roll number value from console and display the corresponding record.**

PreparedStatement

- Once a connection is obtained we can interact with the database.
- The JDBC **Statement, CallableStatement, and PreparedStatement interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.**

Contd….

Interfaces	Recommended Use
Statement	Use this for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The **Statement** interface cannot accept parameters.
PreparedStatement	Use this when you plan to use the SQL statements many times. The **PreparedStatement** interface accepts input parameters at runtime.
CallableStatement	Use this when you want to access the database stored procedures. The **CallableStatement** interface can also accept runtime input parameters.

- The **PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.**
- If you want to execute a **Statement object many times, it usually reduces execution time to use a PreparedStatement object instead.**
- We should use **PreparedStatement when we plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.**
- The main feature of a **PreparedStatement object is that, unlike a Statement object, it is given a SQL statement when it is created.**
- The usage of **PreparedStatement is pretty simple. It involves three simple steps as listed below:**
    - 1. Create a PreparedStatement
    - 2. Populate the statement
    - 3. Execute the prepared statement.

- The basic idea with **PreparedStatement is using the '?' symbol in the query where ever the data is to be substituted and then when the data is available, replace the symbols with the data.**
- **Step1:**

➢ **To create a prepared statement we use the following method on the Connection** object as shown below:

PreparedStament prepareStatement (String sql);

- The SQL query to be passed to the above method will look as shown below:

String myQuery = “INSERT INTO students VALUES (?, ?, ?, ?, ?)”;

- Since we need to insert data into five columns, we used five question marks.
- If you notice the query, there are no single quotes like we had in earlier examples.
- Once we have the above SQL query , we create the prepared statement as shown below:

PreparedStatement stmt = con.prepareStatement (myQuery);

- **Step 2: Now that we created a prepared statement, its time to populate it with the data.**
- **To do this, we use the setXXX() which methods bind values to the parameters, where XXX represents the Java data type of the value you wish to bind to the input parameterr.**
- **If you forget to supply the values, you will receive an SQLException.**
- Example:

stmt.setInt(1,101);//1 specifies the first parameter in the query

stmt.setString(2,"Ratan");

The important methods of `PreparedStatement` interface are given below:

Method	Description
public void setInt(int paramIndex, int value)	sets the integer value to the given parameter index.
public void setString(int paramIndex, String value)	sets the String value to the given parameter index.
public void setFloat(int paramIndex, float value)	sets the float value to the given parameter index.
public void setDouble(int paramIndex, double value)	sets the double value to the given parameter index.
public int executeUpdate()	executes the query. It is used for create, drop, insert, update, delete etc.
public ResultSet executeQuery()	executes the select query. It returns an instance of ResultSet.

Example: Prepared Statement

```
import java.sql.*;
public class PSExample1 {
final String DRIVER_NAME = "com.mysql.jdbc.Driver";
final String DB_URL ="jdbc:mysql://localhost:3306/java2lab";
final String DB_USERNAME = "root";
final String DB_PASSWORD = "";
void connectMeAndFireQuery(){
try{
Class.forName(DRIVER_NAME);
Connection con;
con = DriverManager.getConnection(DB_URL, DB_USERNAME, DB_PASSWORD);
String myQuery = "INSERT INTO students VALUES(?,?,?,?,?)";
PreparedStatement pstmt;
pstmt = con.prepareStatement(myQuery);
pstmt.setInt(1, 101);
pstmt.setString(2,"Ratan");
pstmt.setString(3,"Yadav");
pstmt.setLong(4,984100000);
pstmt.setString(5, "Saptari");
int n;
n=pstmt.executeUpdate();
System.out.println("Thank you "+ n+ " record is added!");
```

```
pstmt.close();
con.close();

} catch (Exception e){
System.out.println(e);
}
}
public static void main(String[] args) {
PSExample1 pse1 =new PSExample1();
pse1.connectMeAndFireQuery();
}
}
```

Example of **PreparedStatement** interface that retrieve the records of a table

```
package jdbc_practice;

import java.sql.*;

public class PSExample2 {

     final String DRIVER_NAME = "com.mysql.jdbc.Driver";

     final String DB_URL
="jdbc:mysql://localhost:3306/java2lab";

     final String DB_USERNAME = "root";

     final String DB_PASSWORD = "";

     void  connectMeAndFireQuery(){

        try{

        Class.forName(DRIVER_NAME);

        Connection con;

        con = DriverManager.getConnection(DB_URL, DB_USERNAME,
DB_PASSWORD);

        String myQuery = "SELECT * FROM result";

        PreparedStatement pstmt;

        pstmt = con.prepareStatement(myQuery);

        ResultSet rs;
```

```
        rs=pstmt.executeQuery();
        while(rs.next()){
        int rn = rs.getInt(1);
        String course = rs.getString(2);
        float marks= rs.getFloat(3);
        System.out.println(
                    "Roll No: "+rn+"\t "
                    +"Course Name: "+ course+ "\t"
                    +"Marks: "+marks
                    );
        }
        pstmt.close();
        con.close();
        } catch (Exception e){
            System.out.println(e);
        }
}
```

```
    public static void main(String[] args) {
        PSExample2 pse2 =new PSExample2();
         pse2.connectMeAndFireQuery();
    }
}
```

Example of **PreparedStatement** interface that deletes the record

```
package jdbc_practice;

import java.sql.*;

import java.util.Scanner;

public class PSExample3 {

     final String DRIVER_NAME = "com.mysql.jdbc.Driver";

     final String DB_URL
="jdbc:mysql://localhost:3306/java2lab";

     final String DB_USERNAME = "root";

     final String DB_PASSWORD = "";

     int rn;

     void getRoll(){

         Scanner sc = new Scanner(System.in);

         System.out.println("Whose data is to be deleted? Enter
Roll No: ");

         rn = sc.nextInt();

     }
```

```
    void  connectMeAndFireQuery(){
        try{
        Class.forName(DRIVER_NAME);
        Connection con;
        con = DriverManager.getConnection(DB_URL, DB_USERNAME,
DB_PASSWORD);
        String myQuery = "DELETE FROM students WHERE roll_no =?
";
        PreparedStatement pstmt;
        pstmt = con.prepareStatement(myQuery);
        pstmt.setInt(1, rn);
        int n = pstmt.executeUpdate();
```

```
            System.out.println("The record is deleted!");
            pstmt.close();
            con.close();
            } catch (Exception e){
                System.out.println(e);
            }
        }
        public static void main(String[] args) {
            PSExample3 pse3 =new PSExample3();
            pse3.getRoll();
            pse3.connectMeAndFireQuery();
        }
}
```

Tasks

Task3:

Write a java program using PreparedStatement that allows a user to insert values to a table of particular database (Suppose database is in MySql server). The program should take the values to insert from console as long as user want to add new record.

Task4:

WAP using PreparedStatement to display the records from a table of given database (Suppose database is in MySql server). Assume the following table :

```
salary(emp_id , emp_name ,emp_salary)
```

The program should read the employee id value from console and display the corresponding record.

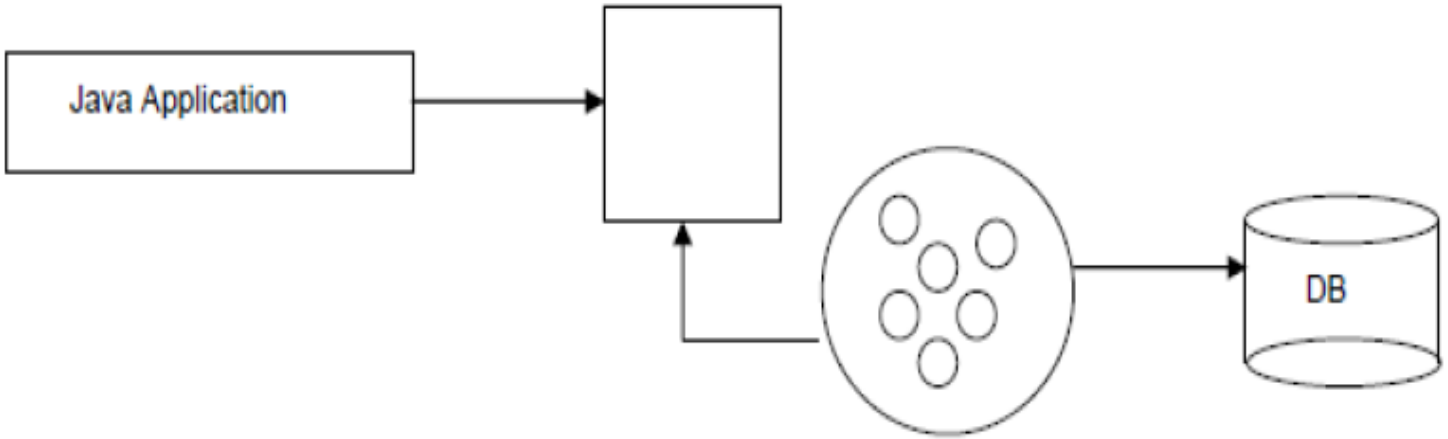
Connection Pooling

- Establishing a database connection is a very resource-intensive process and involves a lot of overhead.
- Connection pooling is a mechanism to create and maintain a collection of JDBC connection objects.
- The primary objective of maintaining the pool of connection object is to leverage re-usability.
- A new connection object is created only when there are no connection objects available to reuse.
- This technique can improve overall performance of the application
- Connection pooling mechanism manage the connections effectively and efficiently by optimizing the usage of database connections
- The connection pooling program comes along when we download the drivers for a particular database

The way the connection pooling works is,

1. It initializes a pool of connections with the database.
2. When a connection is requested by the application, it returns a connection from the pool.
3. The application after using the connection returns it back to the pool.

Connection Pool Program



Connection Pool

- Connection pooling will be implemented in such a way that all the details are hidden behind the scenes.
- The only change we as application developers need to make is the way we retrieve the connection object.
- Once we get a connection object, the rest of the program will be the same.
- Connection pooling is usually used with large scale enterprise applications where thousands of processes need to access the database at the same time.
- In such situations the application server will be configured to use the connection pooling.

Why should we close the database connection?

- Usually, every database has limited number of connections.
- Let’s say a database supports only 10 connections, and if your program doesn't close the connection every time it ran, the 11th time you run, database will refuse to give you a connection.
- If you don't close a connection after using it, from database point of view somebody is still using the connection, and it will not reuse this connection until the connection is closed.
- You need to shut down the database and restart it to release all the connections which is not good.
- So, always make sure you close the database connection.