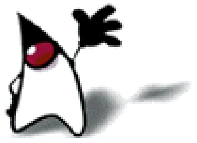
ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

503111

JAVA TECHNOLOGY

CHAPTER 2: JDBC



JDBC –
Java DataBase Connectivity

JDBC API Overview

JDBC is Java API that allows the Java programmers to access database management system from Java code. The JDBC API makes it possible to do three things:

- *Establish a connection with a database or access any tabular data source*
- *Send SQL statements*
- *Process the results*

To connect with individual databases, JDBC requires drivers for each database. The JDBC driver gives out the connection to the database and implements the protocol for transferring the query and result between client and database.

Types of Drivers

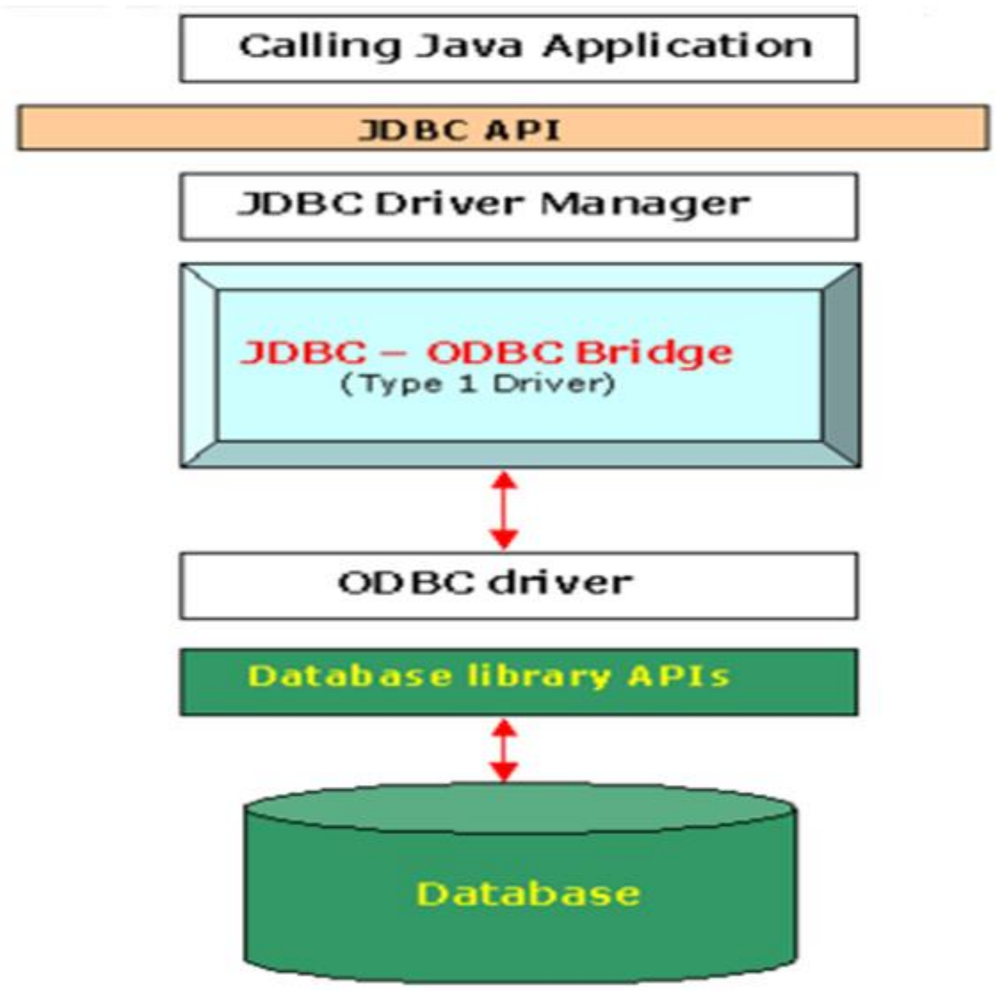
- Type-1 JDBC-ODBC bridge
- Type-2 Native API, part java driver
- Type-3 Pure-java driver for database middleware
- Type-4 Pure-java driver for direct-to-database

Type – 1 Driver

Functions

- Translates query obtained by JDBC into corresponding ODBC query, which is then handled by the ODBC driver.
- Sun provides a JDBC-ODBC Bridge driver. sun.jdbc.odbc.JdbcOdbcDriver. This driver is native code and not Java.
- Client -> JDBC Driver -> ODBC Driver -> Database

Type – 1 Driver



Type – 1 Driver

Advantages

- Easy to connect.

Disadvantages

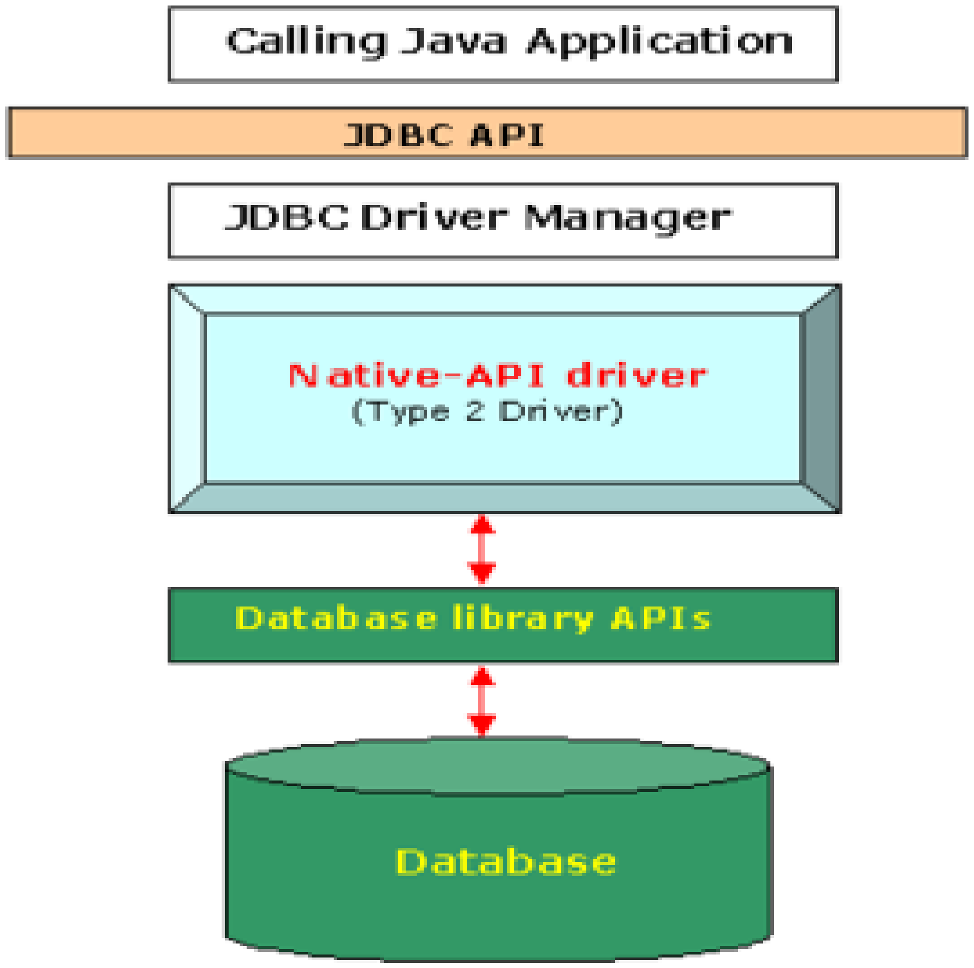
- Performance overhead since the calls have to go through the JDBC overhead bridge to the ODBC driver, then to the native db connectivity interface.
- For prototyping only and not for production
- The ODBC driver needs to be installed on the client machine.
- Compared to other driver types it's slow.
- This driver depends on the ODBC Drivers , and therefore , java applications also become indirectly dependent on ODBC drivers.

Type – 2(Native API) Drivers

Function

- The JDBC type 2 driver is a database driver implementation that uses the client-side libraries of the database. The driver converts JDBC method calls into native calls of the database API
- The type 2 driver is not written entirely in Java as it interfaces with non-Java code that makes the final database callSs.
- Client→ JDBC API→ Database specific Native APIs

Type – 2(Native API) Drivers



Type – 2(Native API) Drivers

Advantages:

- This Driver helps in accessing the data faster as compared to Type-1 drivers

Disadvantages:

- The vendor client library needs to be installed on the client machine since the conversion from JDBC call to database specific native call is done on the client machine.
- Database specific native functions are executed on the client machine and any bug in this driver can crash the JVM.
- This driver is platform dependent .

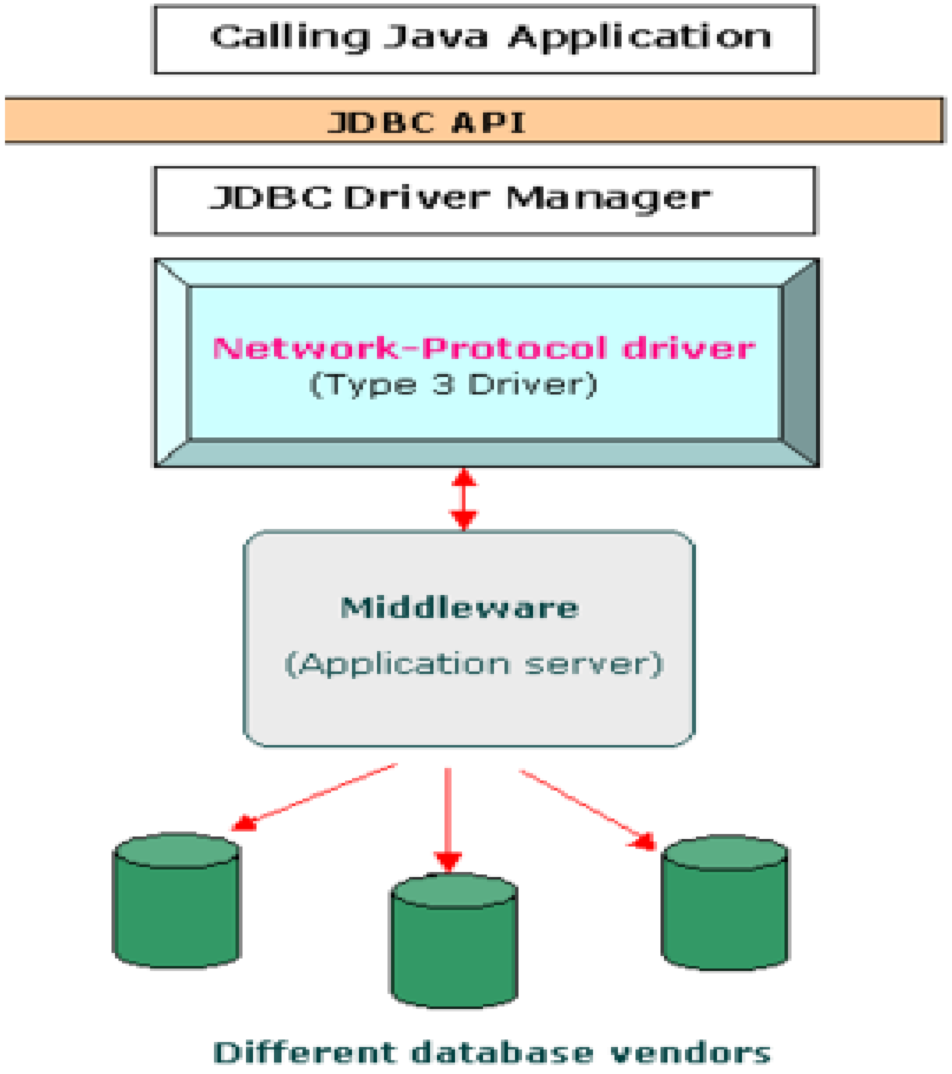
Type – 3(Pure Java) Drivers

Also Named as Network Protocol Drivers

Functions

- Follows a three tier communication approach.
- Can interface to multiple databases - Not vendor specific.
- The JDBC Client driver written in java, communicates with a middleware-net-server using a database independent protocol, and then this net server translates this request into database commands for that database.
- Thus the client driver to middleware communication is database independent.
- Client -> JDBC Driver -> Network-protocol driver -> Middleware-Net Server -> Any Database,...

Type – 3(Pure Java) Drivers



Type – 3(Pure Java) Drivers

Advantages:

- Type-3 drivers are Pure Java Drivers so Platform independent.
- We can switch over from one database to another without changing the client-side driver classes, by just changing the configuration of the middleware.
- Easy deployment

Disadvantage:

- Requires database-specific coding to be done in the middle tier

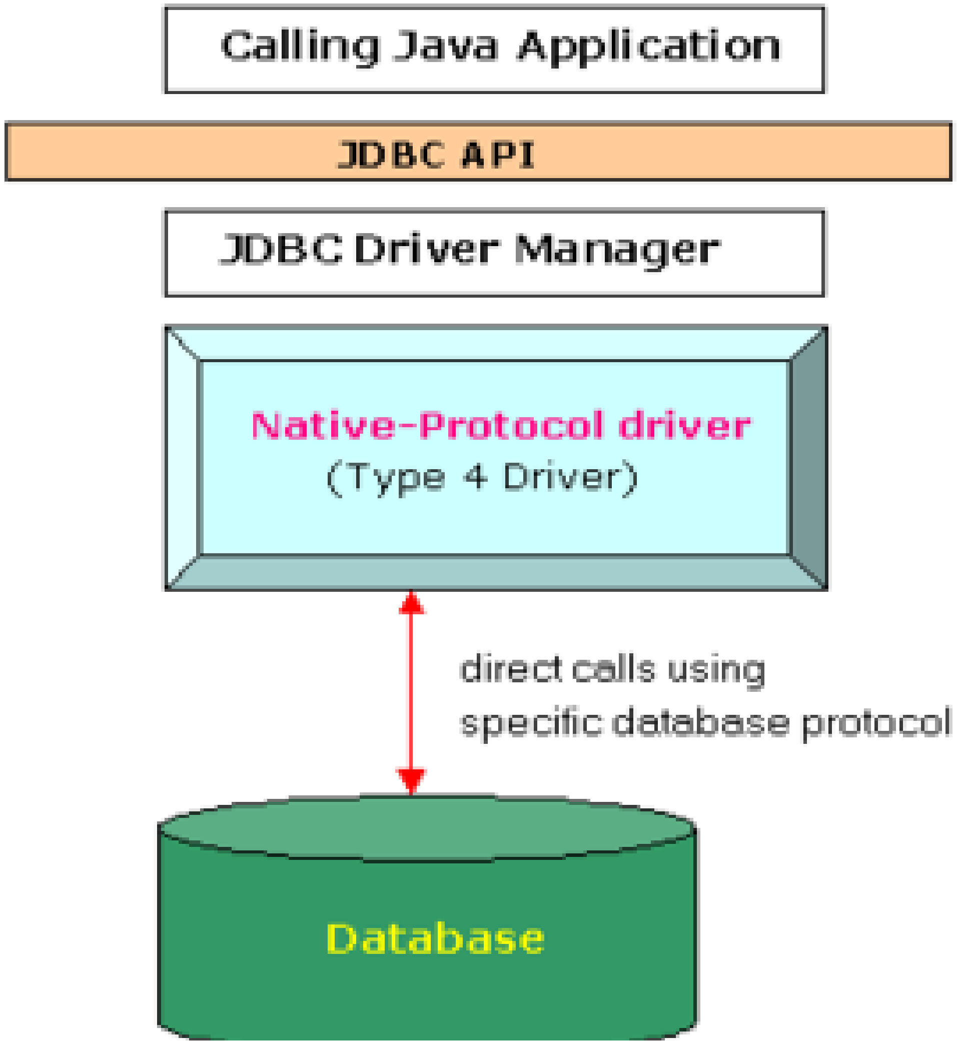
Type – 4(Pure Java) Drivers

Also Named as Native Protocol Drivers

Function:

- Type 4 drivers, coded entirely in Java, communicate directly with a vendor's database, usually through socket connections. No translation or middleware layers are required, improving performance.
- The driver converts JDBC calls into the vendor-specific database protocol so that client applications can communicate directly with the database server.
- Completely implemented in Java to achieve platform independence.
- Client -> Native-protocol JDBC Driver -> database server

Type – 4(Pure Java) Drivers



Type – 4(Pure Java) Drivers

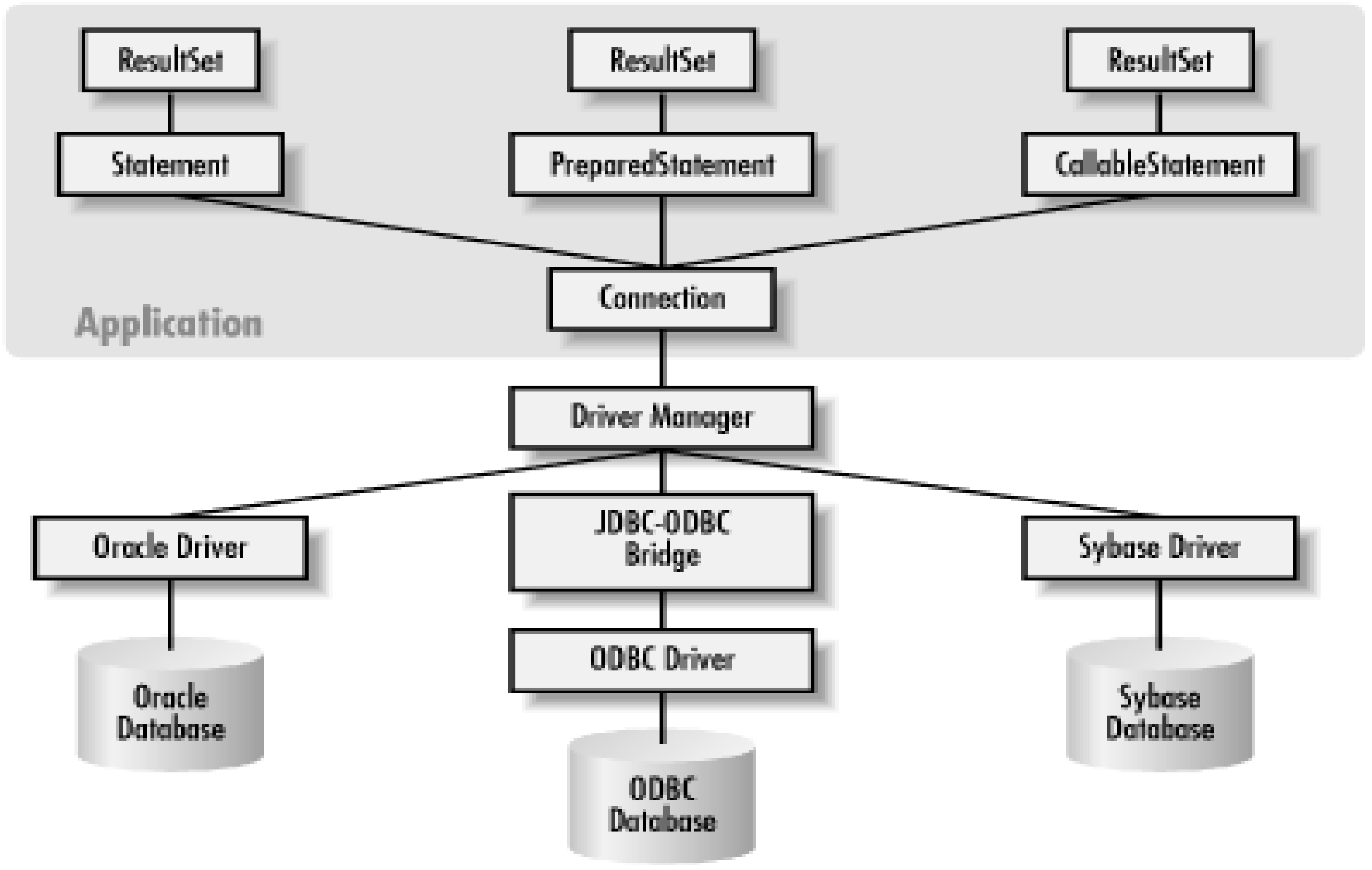
Advantage:

- These drivers don't translate the requests into an intermediary format (such as ODBC), nor do they need a middleware layer to service requests. This can enhance performance considerably.
- The JVM can manage all aspects of the application-to-database connection; this can facilitate debugging.

Disadvantage:

- Drivers are database dependent.

JDBC Architecture



Basic steps to use a database in Java

- 1.Establish a **connection**
- 2.Create JDBC **Statements**
- 3.Execute **SQL** Statements
- 4.GET **ResultSet**
- 5.**Close** connections

1. Establish a connection

- **Import JDBC Packages** **import java.sql.*;**
- **Register the Drivers:**
  - You must register the your driver in your program before you use it.
  - IT is the process by which the Oracle driver's class file is loaded into memory so it can be utilized as an implementation of the JDBC interfaces.
  - Two approaches are used to register the database
    - Class.forName("oracle.jdbc.driver.OracleDriver");
    - DriverManager.registerDriver()

Approach1- Class.forName("oracle.jdbc.driver.OracleDriver");

- The most common approach to register a driver is to use Java's **Class.forName()** method
- This method is used to dynamically load the driver's class file into memory, which automatically registers it.

```
try { Class.forName("oracle.jdbc.driver.OracleDriver"); }
    catch(ClassNotFoundException ex)
{ System.out.println("Error: unable to load driver class!");
}
```

Approach (II) - DriverManager.registerDriver():

- The second approach you can use to register a driver is to use the static **DriverManager.registerDriver()** method.

```
try {
Driver myDriver = new oracle.jdbc.driver.OracleDriver();
DriverManager.registerDriver( myDriver );
} catch(ClassNotFoundException ex)
{ System.out.println("Error: unable to load driver class!");  }
```

2. Create JDBC statement(s)

- **Make the connection**
  - Connection con = DriverManager.getConnection("jdbc:oracle:thin:@oracle-prod:1521:OPROD", username, passwd);
    - What do you think this statement does?
    - Establishes connection to database by obtaining a *Connection* object

- Statement stmt = con.createStatement() ;
- Creates a Statement object for sending SQL statements to the database

Prepared Statements

- **Prepared statements** are pre-compiled SQL statements. Precompiled SQL is useful if the same SQL is to be executed repeatedly, for example, in a loop.
- The syntax is straightforward: just insert question marks for any parameters that you'll be substituting later on before you send the SQL to the database.

Syantax:

- PreparedStatement pstmt = con.prepareStatement("update Orders set pname = ? where Prod_Id = ?");
  pstmt.setString(1, "Bob"); pstmt.setInt(2, 100);
  pstmt.executeUpdate();

- **Prepared Statements are also useful in preventing SQL injection Hacking.**

Callable Statements

- Callable statements are used to execute database stored Procedures.

Syantax.

```
CallableStatement cstmt = conn.prepareCall("{call getEmpName (?,?)}");
cstmt.setInt(1, 111111111);
cstmt.registerOutParameter(2, java.sql.Types.VARCHAR);
cstmt.execute();
String empName = cstmt.getString(2);
System.out.println(empName);
```

Callable Statements- Example

```
import java.sql.*
Import javax.servlet.*;
Import javax.servlet.http.*;
public class Sample extends HttpServlet
    {
    public  void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException
    {
    try
    {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    }
```

Callable Statements- Example

```
catch(ClassNotFoundException ex)
    { ex.printStackTrace(); }
    Connection con=null;
    CallableStatement cstmt=null;
    try
    {
    con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:
    1521:XE"
    "system" "password");
    cstmt=con.prepareCall("{call p1}"); //called the procedure
```

Callable Statements- Example

```
//how to create procedure had writen in bellow
    cstmt.executeUpdate();
    System.out.println("done");
    }
    catch(SQLException ex) {ex.printStackTrace(); }

    if(cstmt!=null) //close the callablestatement
    {
    cstmt.close();
    cstmt=null;
    }
```

Callable Statements- Example

- //create the procedure and execute in sql command prompt

```
CREATE PROCEDURE P1
   AS
   BEGIN
   INSERT INTO PERSON VALUES(1 'SUBAS');
   INSERT INTO PERSON VALUES(2 'VIJAY');
   INSERT INTO PERSON VALUES(3 'CNU');
   END;
   }
```

Executing SQL Statements

- String createLehigh = "Create table Lehigh " + "(SSN Integer not null, Name VARCHAR(32), " + "Marks Integer)";
  stmt.**executeUpdate**(createLehigh);
  //What does this statement do?

- String insertLehigh = "Insert into Lehigh values“ + "(123456789,abc,100)";
  stmt.**executeUpdate**(insertLehigh);

Get ResultSet

```
String queryLehigh = "select * from Lehigh";

ResultSet rs = Stmt.executeQuery(queryLehigh);
//What does this statement do?

while (rs.next()) {
   int ssn = rs.getInt("SSN");
   String name = rs.getString("NAME");
   int marks = rs.getInt("MARKS");
}
```

Metadata in JDBC

- JDBC ***getMetaData()*** is the collective data structure, data type and properties of a table.

```
ResultSet rs = null;
ResultSetMetaData metaData = rs.getMetaData();
 int rowCount = metaData.getColumnCount();
System.out.println("Table Name : " + metaData.getTableName(2));
 for (int i = 0; i < rowCount; i++) {

                    System.out.print(metaData.getColumnName(i + 1) + "  \t");

                    System.out.print(metaData.getColumnDisplaySize(i + 1) + "\t");

                    System.out.println(metaData.getColumnTypeName(i + 1));
```

```
                }
```

Close connection

- At the end of your JDBC program, it is required explicitly close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

- stmt.close();
- con.close();