JDBC

* JDBC stands for Java Database Connectivity.
* JDBC is a Java API (java.sql.\*).
* JDBC along with database drivers can be used to write java applications that can access database and execute queries on it.
* JDBC provides PARTIAL Database Independence through interfaces written in java.sql package.
* JDBC API supports 2 tier and 3 tier architectures, but in general a JDBC architecture consists of 2 layers: JDBC API and JDBC Driver Manager.
* JDBC API: Java application to JDBC manager communication.
* JDBC Driver API: JDBC driver manager to database driver communication.

I.

Java Application <--> JDBC API <--> Java Driver Manager <--> DB Driver (DB specific) <--> DB Server.

II.

“java.sql” Interface hierarchy:

Statement Interface -> Prepared Statement -> Callable Statement

III.

What makes JDBC Database Independent?

JDBC API provides many interfaces whose implementation classes are present in JDBC Drivers, which are DB specific. So we just have to change the db driver according to the database being used.

IV.

Fixed Connection:

Connection con = DriverManager.getConnection(url, username, password);

DriverManager keeps track of drivers that are available and handles establishing a connection between database and appropriate driver.

Pooled Connection:

Connection con = DataSource.getConnection(url, username, password);

V.

The above method call returns a new connection every time. A database server can provide a limited number of connection at a time. Hence it is important to make out connection provider as a singleton design pattern.

Ex.

Connection con;

If(con == null){

con = DriverManager.getConnection(url, username, password);

}

return con;

VI.

JDBC supports 4 types of drivers:

1. JDBC-ODBC bridge
2. Native API Driver
3. Network Protocol Driver
4. Thin Driver = We use this type of driver.

We will be using Type4 drivers through out the course.

JDBC API

Steps to establish connection to database in Java.



OR

More clean way to establish a singleton instance of MySql connection use below code:



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.

**1.The Statement Object –** Used for general purpose access to DB. The Statement Interface cannot accept parameters.

This interface has 3 methods,

boolean *execute()* = returns true, if ResultSet is obtained. Used with DDL statements.

ResultSet *executeQuery()* = returns a result set

int *executeUpdate()* = returns number of rows affected by DML statement.

Syntax:   Connection con = ConnectionProvider.createConnection();

Statement statement = con.createStatement();

             String sql = "select \* from people";

ResultSet result = statement.executeQuery(sql);

while (result.next()) {

                 System.out.println("Name: " + result.getString("name"));

                 System.out.println("Age:" + result.getInt("age"));

            }

result.close();

statement.close();

**2.The Prepared Statement –** Use this when we plan to use the SQL statement numerous times. It accepts parameters at runtime. This interface also has the same 3 methods as that of above, but without any parameters.

Syntax: String query = "insert into people(name, age) values(?,?)";

PreparedStatement ps = con.prepareStatement(query);

ps.setString(1, name);

ps.setInt(2, age);

ps.executeUpdate();

ps.close();

**3.The Callable Statement –** Use this when we have to access the SQL stored procedures. It accepts parameters at runtime. This interface has just one method: .execute()

Syntax: Statement s = con.createStatement();

CallableStatement c = con.prepareCall("{call peopleinfo(?,?)}");

             cs.setString(1, "Bob");

             cs.setInt(2, 64);

             cs.execute();

             ResultSet result = s.executeQuery("select \* from people");

while (result.next()) {

               System.out.println("Name : " + result.getString(1));

               System.out.println("Age : " + result.getInt(2));

            }

**Result Set**

The SQL statements that read data from a database query, return the data in a result set.

Navigating a result state:

1. first()
2. last()
3. next()
4. previous()
5. absolute(int row\_no)

Viewing a Result Set: There are 8 methods to get data from Result Set, one for each data type as well as for some common types like String and Object.

Example: getInt(columnName/columnIndex), getString(), getDouble(), etc.

Native database types and their respective JDBC database types:

char/varchar/varchar2 String

number (n) int/long

number(n,m) float/double

date java.sql.Date

time java.sql.Time

timestamp java.sql.TimeStamp

clob java.sql.Clob

**SQL Injection**

**What is SQL Injection? How does it work?**

SQL injection usually occurs when you ask a user for input, like their username/userid, and instead of a name/id, the user gives you an SQL statement that you will **unknowingly** run on your database.

**Example1:** SQL Injection Based on “1=1” is Always True.

Look at the following example which creates a SELECT statement by adding a variable (txtUserId) to a select string. The variable is fetched from user input (getRequestString):

txtUserId = getRequestString("UserId");  
txtSQL = "SELECT \* FROM Users WHERE UserId = " + txtUserId;

If there is nothing to prevent a user from entering "wrong" input, the user can enter some "smart" input like this: “UserId: 105 OR 1=1”. Then, the SQL statement will look like this:

SELECT \* FROM Users WHERE UserId = 105 OR 1=1;

The SQL above is valid and will return ALL rows from the "Users" table, since **OR 1=1** is always TRUE. Does the example above look dangerous? What if the "Users" table contains names and passwords?

**Example2:** SQL Based Injection on “=” is Always True

User login on a web site example:

uName = getRequestString("username");  
uPass = getRequestString("userpassword");  
sql = 'SELECT \* FROM Users WHERE Name ="' + uName + '" AND Pass ="' + uPass + '"'

A hacker might get access to user names and passwords in a database by simply inserting " OR ""=" into the user name or password text box.

SELECT \* FROM Users WHERE Name ="" or ""="" AND Pass ="" or ""=""

The SQL above is valid and will return all rows from the "Users" table, since **OR ""=""** is always TRUE.

**Example3:** SQL injection based on Batched SQL Statements.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

txtUserId = getRequestString("UserId");  
txtSQL = "SELECT \* FROM Users WHERE UserId = " + txtUserId;

Injected User id: 105; DROP TABLE Suppliers;

The resulting SQL statement would look like this:

SELECT \* FROM Users WHERE UserId = 105; DROP TABLE Suppliers;

**SQL Injection Prevention:** Use SQL Parameters

Most instances of SQL injection can be prevented by using parameterized queries (also known as prepared statements) instead of string concatenation within the query.

The following code is vulnerable to SQL injection because the user input is concatenated directly into the query:

String query = "SELECT \* FROM products WHERE category = '"+ input + "'";

Statement statement = connection.createStatement();

ResultSet resultSet = statement.executeQuery(query);

This code can be easily rewritten in a way that prevents the user input from interfering with the query structure:

PreparedStatement statement = connection.prepareStatement("SELECT \* FROM products WHERE category = ?");

statement.setString(1, input);

ResultSet resultSet = statement.executeQuery();

For a parameterized query to be effective in preventing SQL injection, the string that is used in the query must always be a hard-coded constant, and must never contain any variable data from any origin.