JDBC - STATEMENTS, PREPAREDSTATEMENT AND CALLABLESTATEMENT

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.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

Following table provides a summary of each interface's purpose to understand how do you decide which interface to use?

Interfaces	Recommended Use
Statement	Use 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 when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use when you want to access database stored procedures. The CallableStatement interface can also accept runtime input parameters.

The Statement Objects:

Creating Statement Object:

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example:

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

Once you've created a Statement object, you can then use it to execute a SQL statement with one of its three execute methods.

- **boolean execute(String SQL)** : Returns a boolean value of true if a ResultSet object can be retrieved; otherwise, it returns false. Use this method to execute SQL DDL statements or when you need to use truly dynamic SQL.

- **int executeUpdate(String SQL)** : Returns the numbers of rows affected by the execution of the SQL statement. Use this method to execute SQL statements for which you expect to get a number of rows affected - for example, an INSERT, UPDATE, or DELETE statement.

- **ResultSet executeQuery(String SQL)** : Returns a ResultSet object. Use this method when you expect to get a result set, as you would with a SELECT statement.

Closing Statement Obeject:

Just as you close a Connection object to save database resources, for the same reason you should also close the Statement object.

A simple call to the close() method will do the job. If you close the Connection object first it will close the Statement object as well. However, you should always explicitly close the Statement object to ensure proper cleanup.

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   stmt.close();
}
```

For a better understanding, I would suggest to study [Statement - Example Code](#).

The PreparedStatement Objects:

The *PreparedStatement* interface extends the Statement interface which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object:

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   . . .
}
```

All parameters in JDBC are represented by the **?** symbol, which is known as the parameter marker. You must supply values for every parameter before executing the SQL statement.

The **setXXX()** methods bind values to the parameters, where **XXX** represents the Java data type of the value you wish to bind to the input parameter. If you forget to supply the values, you will receive an SQLException.

Each parameter marker is referred to by its ordinal position. The first marker represents position 1, the next position 2, and so forth. This method differs from that of Java array indices, which start at 0.

All of the **Statement object's** methods for interacting with the database (a) execute(), (b) executeQuery(), and (c) executeUpdate() also work with the PreparedStatement object. However, the methods are modified to use SQL statements that can take input the parameters.

Closing PreparedStatement Obeject:

Just as you close a Statement object, for the same reason you should also close the PreparedStatement object.

A simple call to the close() method will do the job. If you close the Connection object first it will close the PreparedStatement object as well. However, you should always explicitly close the PreparedStatement object to ensure proper cleanup.

```
PreparedStatement pstmt = null;
try {
```

```
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   pstmt.close();
}
```

For a better understanding, I would suggest to study Prepare - Example Code.

The CallableStatement Objects:

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object:

Suppose, you need to execute the following Oracle stored procedure:

```
CREATE OR REPLACE PROCEDURE getEmpName
   (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS
BEGIN
   SELECT first INTO EMP_FIRST
   FROM Employees
   WHERE ID = EMP_ID;
END;
```

NOTE: Above stored procedure has been written for Oracle, but we are working with MySQL database so let us write same stored procedure for MySQL as follows to create it in EMP database:

```
DELIMITER $$

DROP PROCEDURE IF EXISTS `EMP`.`getEmpName` $$
CREATE PROCEDURE `EMP`.`getEmpName`
   (IN EMP_ID INT, OUT EMP_FIRST VARCHAR(255))
BEGIN
   SELECT first INTO EMP_FIRST
   FROM Employees
   WHERE ID = EMP_ID;
END $$

DELIMITER ;
```

Three types of parameters exist: IN, OUT, and INOUT. The PreparedStatement object only uses the IN parameter. The CallableStatement object can use all three.

Here are the definitions of each:

Parameter	Description
IN	A parameter whose value is unknown when the SQL statement is created. You bind values to IN parameters with the setXXX() methods.
OUT	A parameter whose value is supplied by the SQL statement it returns. You retrieve values from the OUT parameters with the getXXX() methods.
INOUT	A parameter that provides both input and output values. You bind variables with the setXXX() methods and retrieve values with the getXXX() methods.

The following code snippet shows how to employ the **Connection.prepareCall()** method to instantiate a **CallableStatement** object based on the preceding stored procedure:

```
CallableStatement cstmt = null;
try {
    String SQL = "{call getEmpName (?, ?)}";
    cstmt = conn.prepareCall (SQL);
    . . .
}
catch (SQLException e) {
    . . .
}
finally {
    . . .
}
```

The String variable SQL represents the stored procedure, with parameter placeholders.

Using CallableStatement objects is much like using PreparedStatement objects. You must bind values to all parameters before executing the statement, or you will receive an SQLException.

If you have IN parameters, just follow the same rules and techniques that apply to a PreparedStatement object; use the setXXX() method that corresponds to the Java data type you are binding.

When you use OUT and INOUT parameters you must employ an additional CallableStatement method, registerOutParameter(). The registerOutParameter() method binds the JDBC data type to the data type the stored procedure is expected to return.

Once you call your stored procedure, you retrieve the value from the OUT parameter with the appropriate getXXX() method. This method casts the retrieved value of SQL type to a Java data type.

Closing CallableStatement Obeject:

Just as you close other Statement object, for the same reason you should also close the CallableStatement object.

A simple call to the close() method will do the job. If you close the Connection object first it will close the CallableStatement object as well. However, you should always explicitly close the CallableStatement object to ensure proper cleanup.

```
CallableStatement cstmt = null;
try {
    String SQL = "{call getEmpName (?, ?)}";
    cstmt = conn.prepareCall (SQL);
    . . .
}
catch (SQLException e) {
    . . .
}
finally {
    cstmt.close();
}
```

For a better understanding, I would suggest to study Callable - Example Code.