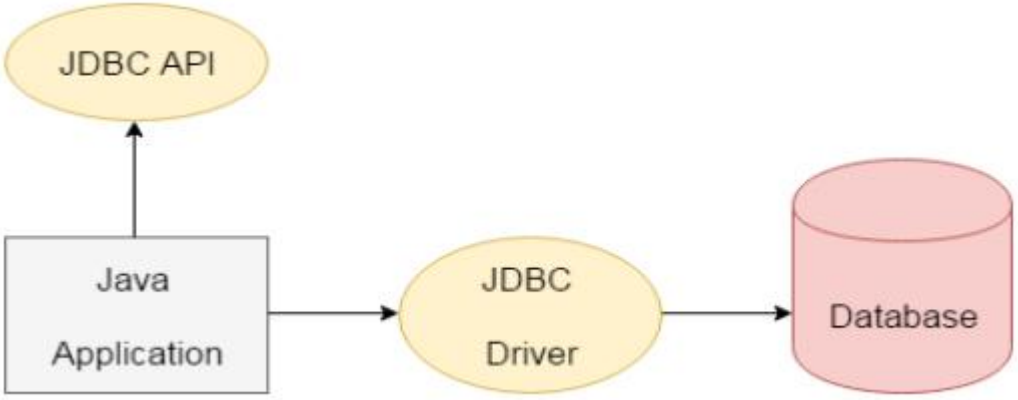
Chapter Seven
JDBC SQL Programming

Overview of JDBC

Java JDBC is a java API to connect and execute query with the database. JDBC API uses jdbc drivers to connect with the database.



Why use JDBC

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).

API (Application programming interface) is a document that contains description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

There are 5 steps to connect any java application with the database in java using JDBC. They are as follows:

- Register the driver class
- Creating connection
- Creating statement
- Executing queries
- Closing connection

JDBC Drivers

JDBC's main interfaces and classes are all available under the java.sql package:

- **DriverManage**r: this class is used to register driver for a specific database type (e.g. MySQL in this tutorial) and to establish a database connection with the server via its getConnection() method.
- **Connection:** this interface represents an established database connection (session) from which we can create statements to execute queries and retrieve results, get metadata about the database, close connection, etc.

Statements and Prepared Statements

Statement and PreparedStatement: these interfaces are used to execute static SQL query and parameterized SQL query, respectively. Statement is the super interface of the PreparedStatement interface. Their commonly used methods are:

- boolean execute(String sql): executes a general SQL statement. It returns true if the query returns a ResultSet, false if the query returns an update count or returns nothing. This method can be used with a Statement only.
- int executeUpdate(String sql): executes an INSERT, UPDATE or DELETE statement and returns an update account indicating number of rows affected (e.g. 1 row inserted, or 2 rows updated, or 0 rows affected).
- ResultSet executeQuery(String sql): executes a SELECT statement and returns a ResultSet object which contains results returned by the query.

A prepared statement is one that contains placeholders (in form question marks ?) for dynamic values will be set at runtime. For example:

SELECT * from Users WHERE user_id=?

Here the value of user_id is parameterized by a question mark and will be set by one of the setXXX() methods from the PreparedStatement interface,

Example: setInt(int index, int value).

o ResultSet: contains table data returned by a SELECT query. Use this object to iterate over rows in the result set using next() method, and get value of a column in the current row using getXXX() methods (e.g. getString(), getInt(), getFloat() and so on). The column value can be retrieved either by index number (1-based) or by column name.

o SQLException: this checked exception is declared to be thrown by all the above methods, so we have to catch this exception explicitly when calling the above classes' methods.

Connecting to a Database: Results sets

Supposing the MySQL database server is listening on the default port 3306 at localhost. The following code snippet connects to the database name SampleDB by the user root and password secret:

```
String dbURL = "jdbc:mysql://localhost:3306/sampledb";
String username = "root";
String password = "secret";
try {
Connection conn = DriverManager.getConnection(dbURL, username,
password);
if (conn != null) {
System.out.println("Connected");
}
} catch (SQLException ex) {
ex.printStackTrace();
}
```

Once the connection was established, we have a Connection object which can be used to create statements in order to execute SQL queries. In the above code, we have to close the connection explicitly after finish working with the database:

1	conn.close();

try-with-resources statement which will close the connection automatically, as shown in the following

```
try (Connection conn = DriverManager.getConnection(dbURL,
username,
password)) {
// code to execute SQL queries goes here...
} catch (SQLException ex) {
ex.printStackTrace();
}
```

Executing INSERT statement

Let's write code to insert a new record into the table Users with following details:

o username: bill
o password: secretpass
o fullname: Bill Gates
o email: bill.gates@microsoft.com

Here's the code snippet:

```
String sql = "INSERT INTO Users (username, password,
fullname,
email) VALUES (?, ?, ?, ?)";
PreparedStatement statement = conn.prepareStatement(sql);
statement.setString(1, "bill");
statement.setString(2, "secretpass");
statement.setString(3, "Bill Gates");
statement.setString(4, "bill.gates@microsoft.com");
int rowsInserted = statement.executeUpdate();
if (rowsInserted > 0) {
System.out.println("A new user was inserted successfully!");
}
```

The PreparedStatement interface provides various setXXX() methods corresponding to each data type, for example:

o setBoolean(int parameterIndex, boolean x)

o setDate(int parameterIndex, Date x)

o setFloat(int parameterIndex, float x)

o …

And so on. Which method to be used is depending on the type of the corresponding column in the database table.

Finally we call the PreparedStatement's executeUpdate() method to execute the INSERT statement. This method returns an update count indicating how many rows in the table were affected by the query, so checking this return value is necessary to ensure the query was executed successfully. In this case, executeUpdate() method should return 1 to indicate one record was inserted.

Executing SELECT statement

The following code queries all records from the Users table and print out details for each record:

```
String sql = "SELECT * FROM Users";
Statement statement = conn.createStatement();
ResultSet result = statement.executeQuery(sql);
int count = 0;
while (result.next()){
String name = result.getString(2);
String pass = result.getString(3);
String fullname = result.getString("fullname");
String email = result.getString("email");
String output = "User #%d: %s - %s - %s - %s";
System.out.println(String.format(output, ++count, name, pass,
fullname, email));}
```

Output:

User #1: bill - secretpass - Bill Gates - bill.gates@microsoft.com