

Using JDBC

What is JDBC

JDBC (Java Database Connectivity) is the Java API that manages connecting to a database, issuing queries and commands, and handling result sets obtained from the database.

The description above comes from [this website](#). You have learned “what is API” before ---- they are a set of pre-coded classes and functions that serves a certain purpose. We follow this definition, so JDBC is a set of classes and functions that allows you to use databases. The definition of these classes and functions are mostly in the package “java.sql”, All classes are presented in the link below for an overview:

<https://docs.oracle.com/javase/7/docs/api/index.html?java/sql/package-tree.html>

The classes in “java.sql” provides necessary functions to operate on different database systems, such as preparing queries and processing obtained query results. However, they do not provide the implementation for interacting with databases directly. That’s because there are so many database systems available with their own underlying mechanisms. Luckily, most database systems provide something called “connectors” that enables Java to connect. For our csse-mysql or XAMPP installation, we can use MariaDB connector.


Setting up the connector

Firstly, you need to download this connector. The reason I choose MariaDB connector is because you can use it to connect to both MySQL and MariaDB. To get the connector:

<https://downloads.mariadb.org/connector-java/>



You will get a jar file.

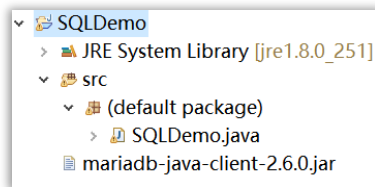
 mariadb-java-client-2.6.0.jar

This jar file contains a few class files, if you are interested, you can open it with 7zip or any other decent file compress program.

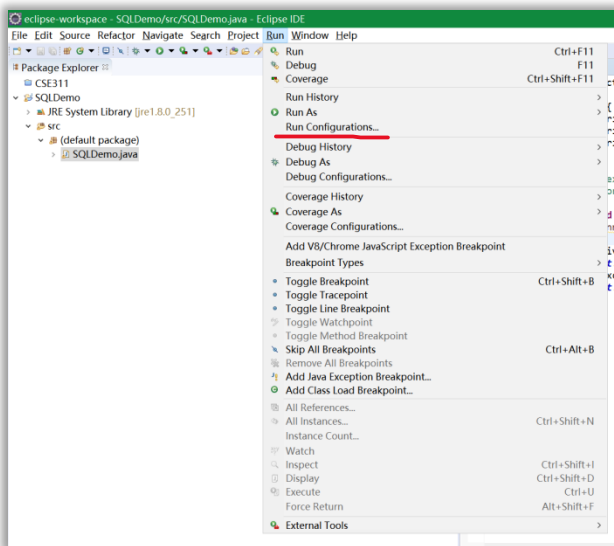
Just downloading this jar file alone is not enough, because Java don't know where to find it. Here, I will show you how, using three integrated development environments (IDEs): "Eclipse", "NetBeans" and "IntelliJ IDEA". If you are using something different, you are on your own.

Setting up connectors in Eclipse:

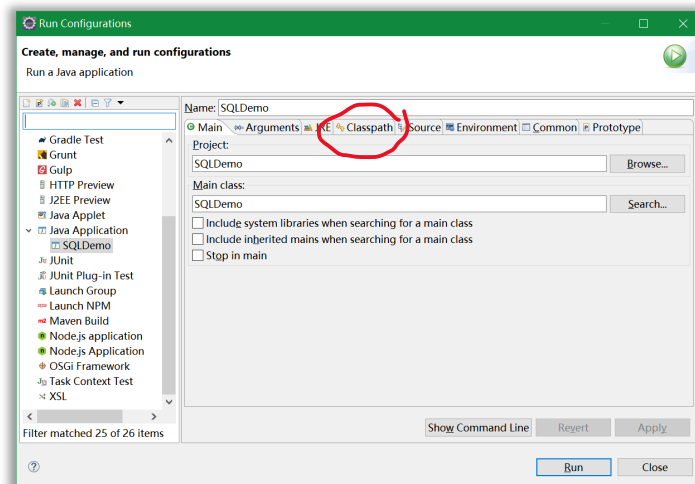
First, you should copy the connector jar file into your project. I just put it under the project root folder:



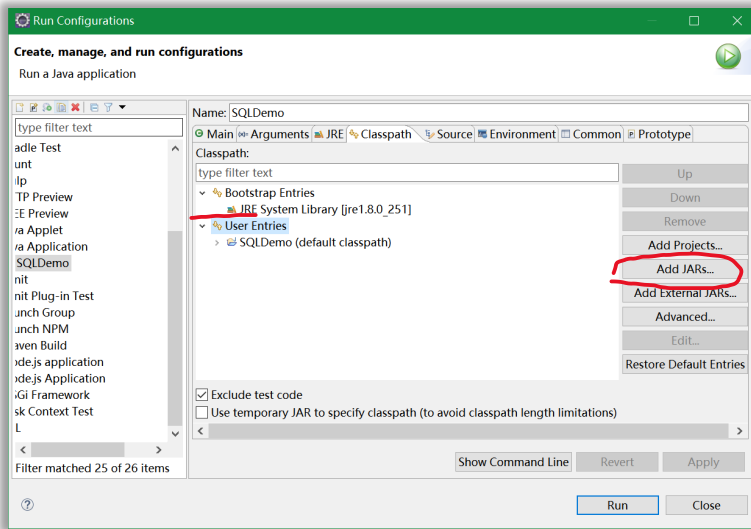
Then click “Run”, and click “Run Configuration”



You will see:



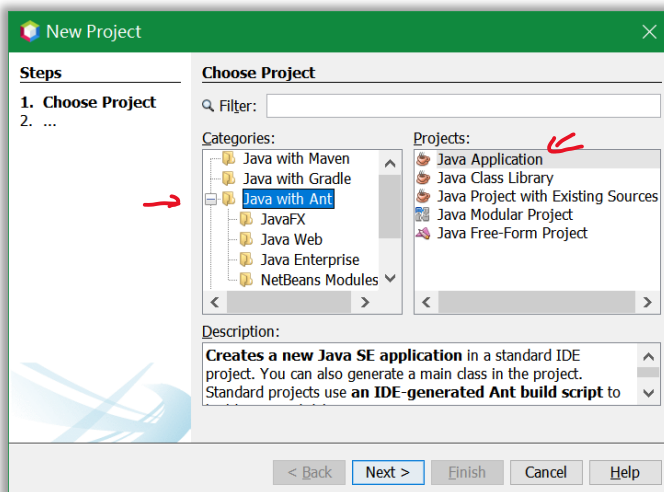
Choose “Classpath”, click “user entries” then click “Add JARs” on the right:



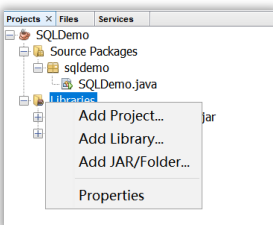
Add your jar file then click “Apply”. Your program will now run with this connector!

Setting up connectors in NetBeans:

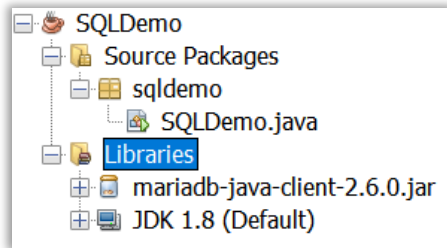
First, **create a new project using “Java with Ant”** option. Don’t use “Java with Maven” or “Java with Gradle” unless you know how to fetch the connector from online repositories.



Once created, right click on “Libraries” under the project:



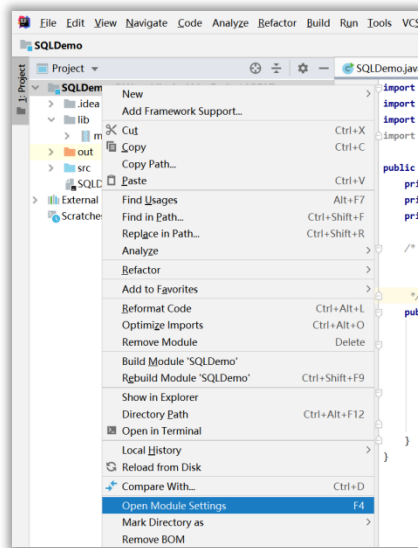
Click “add jar/folder” and find your Jar file.



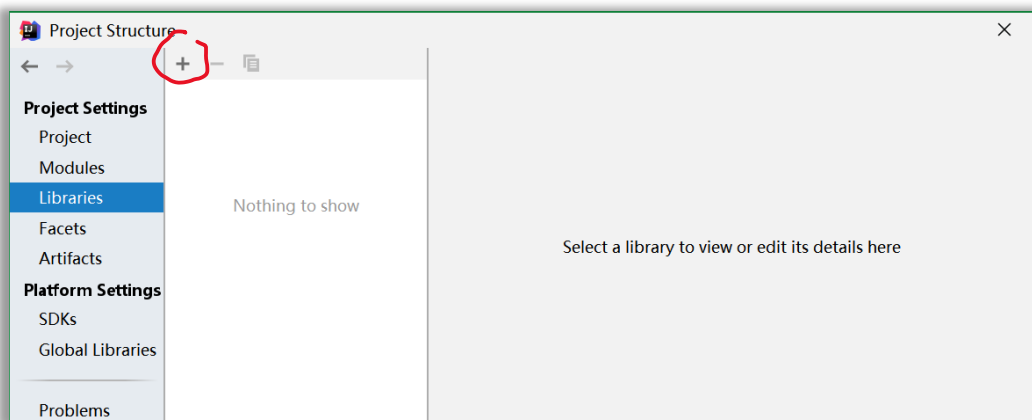
Once done, you are ready to connect!

Setting up connectors in IntelliJ IDEA:

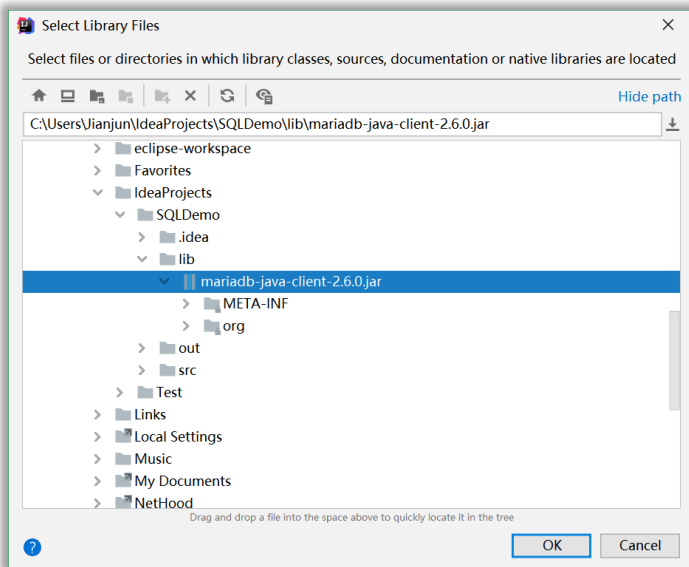
Right click on your project and select “Open Module Settings”:



Under Project Settings, choose Libraries, then press +, and choose “java”:



Locate your jar file and press ok:



Once done, you are ready.

Two more technical approaches to set up connectors (optional reading)

All of the above IDEs use one of the following two methods to tell Java where to find the connector:

1. Setting the `CLASSPATH` environment variable.
2. Explicitly include “-classpath” when running your program.

Both methods are a bit technical but quite important if you want to get related jobs.

Setting up `CLASSPATH` env

There is a detailed tutorial for setting up all kinds of environment variables in Windows, Linux and MacOS. The link is presented below:

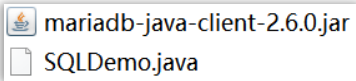
https://www.ntu.edu.sg/home/ehchua/programming/howto/Environment_Variables.html

Check this link if you are interested.

Using -classpath

Recall that you can use “java classname” to run your java programs, as long as the classname is a java class that contains “public static void main()”. When you use “java” command, the java will try to find the existence of such class within its class paths. The class path is just a list of directories where java classes are (supposed to be) stored.

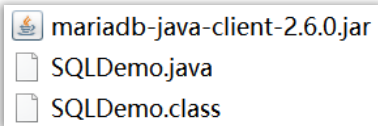
For example, in my D:\ drive, I have two files:



The “SQLDemo.java” is my source code for connecting MySQL. I compiled it using “javac”:

```
javac SQLDemo.java
```

Now my D:\ has three files:



The “SQLDemo.class” I got is the class file that can be executed by java. To run this program:

```
PS D:\> java -classpath "D:\mariadb-java-client-2.6.0.jar;D:" SQLDemo  
Connected to database
```

The “-classpath” includes two positions: the jar file “D:\mariadb-java-client-2.6.0.jar” and drive “D:”, where the SQLDemo.class is located at. You need “D:”, otherwise, SQLDemo won’t run as Java won’t be able to find it.

Connecting to the database

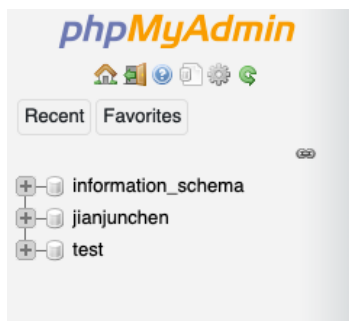
The example below will allow you to connect to a MySQL server. The `Connection` object allows you to execute queries later.

```
public static void main(String[] args) {  
    Connection conn = null;  
    try {  
        conn = DriverManager.getConnection(csseSQLURL);  
        System.out.println("Connected to database");  
    } catch (SQLException e) {  
        System.out.println(e.getMessage());  
    }  
}
```

The parameter “url” can be a string like:

```
"jdbc:mysql://localhost:3306/cse103?user=root"
```

This URL connects your program to the MySQL server that is running on your own computer¹. It will try to connect to a **database schema** called “cse103” under the user account “root”. In case you don’t know, the root account is the default MySQL account of your XAMPP installation. **If you haven’t create any schemas in your database, you need to create it before referring to it in you URL.**



The MySQL server offered by our CSSE department has many users and each user account has a password. As a result, we need a different URL string, like this:

```
"jdbc:mysql://csse-mysql:3306/jianjunchen?user=JianjunChen&password=secret"
```

Or this (full address):

```
"jdbc:mysql://csse-mysql.xjtlu.edu.cn:3306/jianjunchen?user=JianjunChen&password=secret"
```

¹ Tested with XAMPP, but it should work for the official MySQL, too

Remember to catch the exception thrown by the method “`DriverManager.getConnection()`”, This happens when your program cannot access the database. Use “`e.getMessage()`” to see what actually caused the connection failure.

In general, the format of this URL string follows this pattern:

The database type, we can use MySQL even if it is actually MariaDB

Can be a host name like:
csse-mysql.xjtlu.edu.cn
or an IP address like:
192.168.1.1

```
jdbc:mysql://[host][,failoverhost...]  
[:port]/[database]  
[?propertyName1]=propertyValue1  
[&propertyName2]=propertyValue2]...
```

For example:

?user=SillyEgg&password=pfft&connectTimeout=50

“?” Signifies the start of properties, “&” is the separator of these individual properties. Check here for the full list of properties supported:

<https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-reference-configuration-properties.html>

For this module, you only need “user” and “password”

The full example so far:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.util.Properties;

public class SQLDemo {
    private static String localURL =
        "jdbc:mysql://localhost:3306/cse103?user=root";
    private static String csseURL = "jdbc:mysql://csse-
mysql:3306/jianjunchen?user=JianjunChen&password=123";
    private static String csseAltURL = "jdbc:mysql://csse-
mysql.xjtlu.edu.cn:3306/jianjunchen?user=JianjunChen&password=123";

    public static void main(String[] args) {
        Connection conn = null;
        try {
            conn = DriverManager.getConnection(csseAltURL);
            System.out.println("Connected to database");
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Executing SQL queries

To execute any SQL queries, you need an object of “`java.sql.Statement`”. The `Statement` itself is only an interface, but the class “`java.sql.Connection`” provides an implementation of it and you can get an object using “`Connection.createStatement()`”:

```
Statement statement = null;
try {
    statement = conn.createStatement();
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```

Once we get an object of it, we can start executing queries. SQL queries can be classified into two types:

1. One type returns no tables after execution: CREATE TABLE, INSERT, UPDATE.
2. The other type returns results: SELECT-related.

To apply any queries, you need to call “`Statement.execute()`”. The detailed description about this function is available at <https://docs.oracle.com/javase/8/docs/api/>. Again, learning from manuals is a skill that you must master. Below is a screenshot of the manual:

execute

```
boolean execute(String sql)
    throws SQLException
```

Executes the given SQL statement, which may return multiple results. In some (uncommon) situations, a single SQL statement may return multiple result sets and/or update counts. Normally you can ignore this unless you are (1) executing a stored procedure that you know may return multiple results or (2) you are dynamically executing an unknown SQL string.

The `execute` method executes an SQL statement and indicates the form of the first result. You must then use the methods `getResultSet` or `getUpdateCount` to retrieve the result, and `getMoreResults` to move to any subsequent result(s).

Note: This method cannot be called on a `PreparedStatement` or `CallableStatement`.

Parameters:

`sql` - any SQL statement

Returns:

true if the first result is a `ResultSet` object; false if it is an update count or there are no results

Throws:

`SQLException` - if a database access error occurs, this method is called on a closed `Statement`, the method is called on a `PreparedStatement` or `CallableStatement`

`SQLTimeoutException` - when the driver has determined that the timeout value that was specified by the `setQueryTimeout` method has been exceeded and has at least attempted to cancel the currently running `Statement`

See Also:

`getResultSet()`, `getUpdateCount()`, `getMoreResults()`

The function returns true if the query returns any results (such as SELECT statements). It returns false if no results are available. The code example below creates "myTable" and inserts 5 rows into this table.

```
Statement statement = null;
try {
    statement = conn.createStatement();
    statement.execute("CREATE TABLE myTable (id INT PRIMARY KEY, col1 VARCHAR(55));");
    System.out.println("Table hath been created.");
    for (int i = 0; i < 5; i++) {
        statement.execute("INSERT INTO myTable VALUES (" + i + ", " + (i + 20) + ")");
    }
    System.out.println("Rows are added!");
} catch (SQLException e) {
    System.out.println(e.getMessage());
}
```

After executing this piece of code, you can see the results in the phpMyAdmin:

+ Options					
				id	col1
<input type="checkbox"/>				0	20
<input type="checkbox"/>				1	21
<input type="checkbox"/>				2	22
<input type="checkbox"/>				3	23
<input type="checkbox"/>				4	24

This example only involves CREATE TABLE and INSERT, you won't need to check their results. If the execution fails, it gives you exceptions. For example, if you run the above program twice, the database will refuse to create another instance of "myTable" because it already exists. You will get the following message:

```
Run: SQLDemo x
"C:\Program Files\Java\jdk1.8.0_201\bin\java.exe" ...
Connected to database
(conn=106) Table 'mytable' already exists
Process finished with exit code 0
```

Working on query results

The SELECT statement returns results after successful execution. Every time the "execute()" function fetches some result, the result will be temporarily stored inside the Statement object until the next query has been executed. To get this result, you need to call getResultSet() to obtain the result.

getResultSet

```
ResultSet getResultSet()
    throws SQLException
```

Retrieves the current result as a ResultSet object. This method should be called only once per result.

Returns:

the current result as a ResultSet object or null if the result is an update count or there are no more results

Throws:

SQLException - if a database access error occurs or this method is called on a closed Statement

See Also:

execute(java.lang.String)

This function returns an object of `ResultSet`. "`java.sql.ResultSet`" is another interface defined in Java.

IMPORTANT: The concept of the `ResultSet` is like follows:

1. After executing a `SELECT` statement, the java program will get many rows from the database. These rows will be contained within the `ResultSet` object. **The row number starts from 1.**
2. There is a hidden cursor inside the `ResultSet` that, initially, points to the empty space **above** the first row of these rows, which is indicated by a row number of 0.
3. You can then use relocation functions to move this cursor to any position you would like.

I highly recommend you to check the API document of "`java.sql.ResultSet`" when going through the following sections.

Navigating through result sets

By default, `ResultSet` object is not updatable and has a cursor that moves forward only. You will need the `next()` function to iterate through results.

`ResultSet.next()`: moves this cursor to the next position, it returns true if the new current row is valid, false if there are no more rows. It is useful for looping through the whole result set:

```
statement.execute("SELECT * FROM myTable;");
ResultSet rs = statement.getResultSet();
while (rs.next()) {
    System.out.println("The id is: " + rs.getInt("id"));
}
```

In the example above, "`rs.next()`" shifts the cursor to the next valid row. Then, calling "`getInt(String columnName)`" will obtain the integer stored at the column "`id`" at that row.

If you want to be able to go back and forth in the result set, you need to specify this when creating the statement object:

```
statement = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

Once this has been done, you can safely use the following functions:

`ResultSet.absolute(int n)`: Moves the cursor to the `n`th row.

`ResultSet.last()`, `ResultSet.first()`: Their names explain a lot .

...

Getting results from the ResultSet

We used “`ResultSet.getInt(String columnName)`” in the example above, but you have a long list of other options. They are described in the API document:

<https://docs.oracle.com/javase/8/docs/api/index.html?java/sql/ResultSet.html>

For example, “`getDate()`” gets the date from the current row, calling it will get you a “`java.sql.Date`” object. The object has a function called “`toLocalDate()`”, which returns an object of “`java.time.LocalDate`” (You learned `LocalDate` in the last semester in CSE105, on week 10).

About this tutorial

This tutorial is based on the information I gathered from the official tutorial and some other websites. I tried to make it as simple as possible without losing necessary technical details. If you think it is not enough, you can check the full tutorial here:

<https://docs.oracle.com/javase/tutorial/jdbc/basics/index.html>

`ResultSet` has many additional functionalities that can be useful. For example, you can update the tuples in relations directly by modifying the data in `ResultSet`. But this functionality can be replaced by SQL queries. You can decide whether to use it or not, but you need to learn it by yourself. Seeking knowledge by yourself is an important skill, but if you tried to learn it and got stuck, you can always email me.