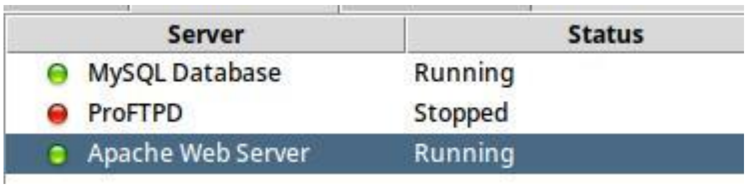
Connection Java to MySQL

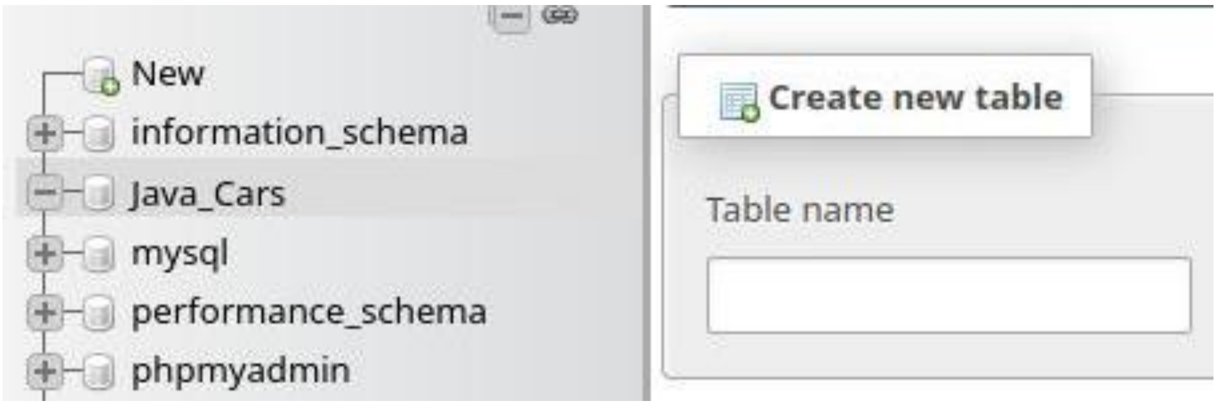
First steps

First, there is no need to say that we need to have an Apache Server and a MySQL client running.



Server	Status
● MySQL Database	Running
● ProFTPD	Stopped
● Apache Web Server	Running

To have a connection from Java to a MySQL we need a database already created, in order to establish a connection directly with the database we need a name, in this case the database is going to be named “Java_Cars”.

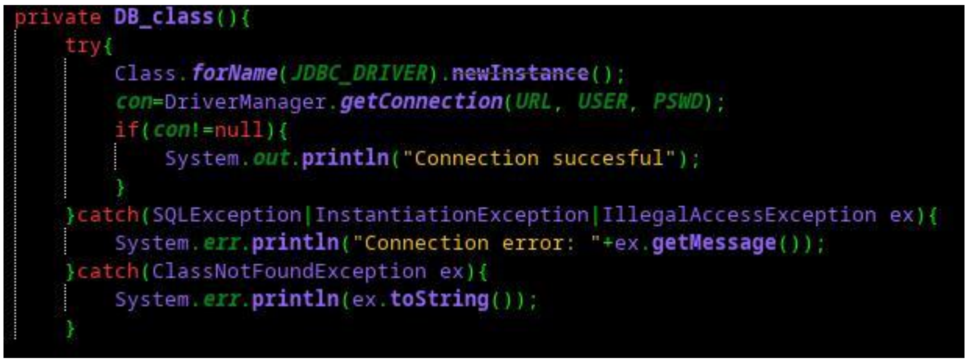


Having the database created we can begin with the Java project.

Database connection

In the constructor we must call for a new instance of the driver, create a connection (con) using the driver and establish that connection through the function .getConnection.

If the con isn't null a message shows to confirm the connection.



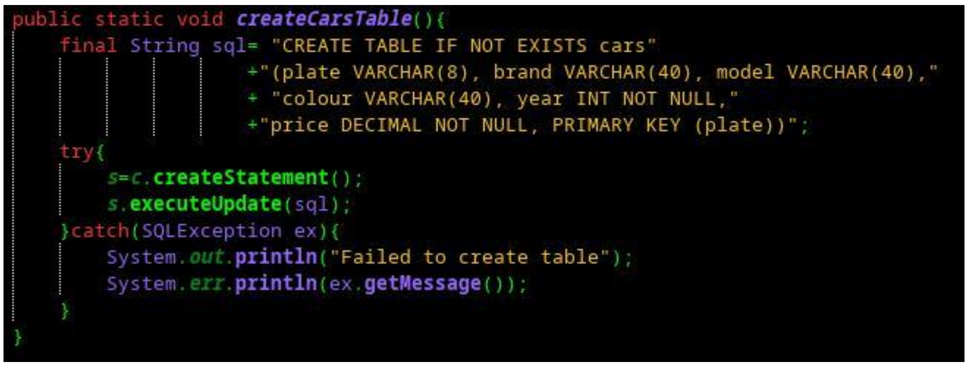
```
private DB_class(){
    try{
        Class.forName(JDBC_DRIVER).newInstance();
        con=DriverManager.getConnection(URL, USER, PSWD);
        if(con!=null){
            System.out.println("Connection succesful");
        }
    }catch(SQLException|InstantiationException|IllegalAccessException ex){
        System.err.println("Connection error: "+ex.getMessage());
    }catch(ClassNotFoundException ex){
        System.err.println(ex.toString());
    }
```

The following line of code is the most important as it is used on every interaction with MySQL, it uses the getInstance method to create the connection and then uses that connection retrieving the variable “con”.

“private static Connection c = DB_class.getInstance().getConnection();”

What this line of code does is create a variable “c” that eases the access to the connection for the methods that will create the database, insert the data and so on.

Create table

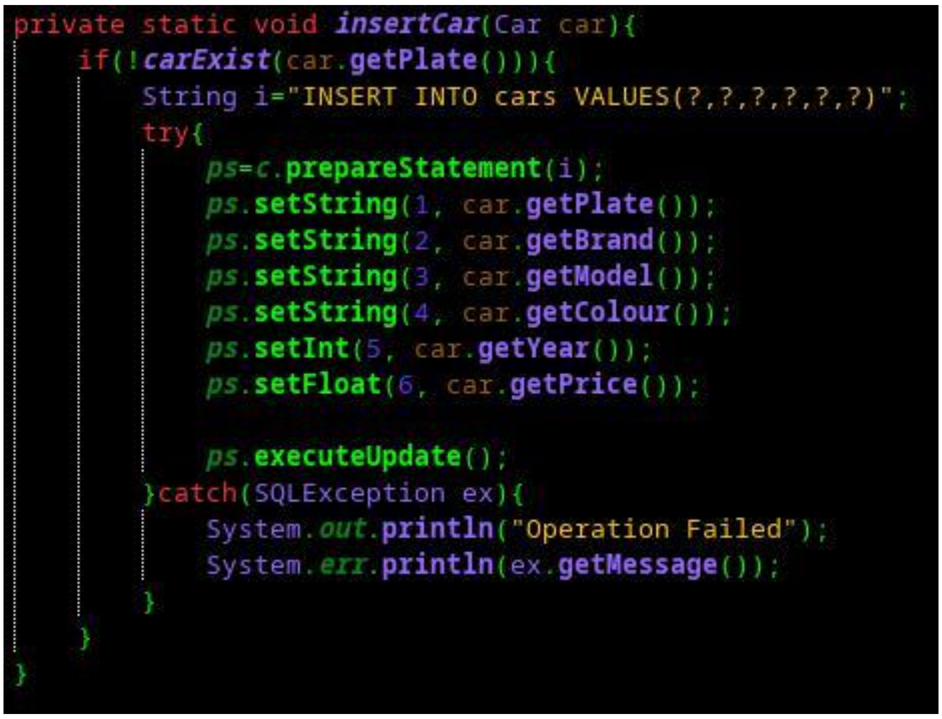


```
public static void createCarsTable(){
    final String sql= "CREATE TABLE IF NOT EXISTS cars"
                +"(plate VARCHAR(8), brand VARCHAR(40), model VARCHAR(40),"
                + "colour VARCHAR(40), year INT NOT NULL,"
                +"price DECIMAL NOT NULL, PRIMARY KEY (plate))";
    try{
        s=c.createStatement();
        s.executeUpdate(sql);
    }catch(SQLException ex){
        System.out.println("Failed to create table");
        System.err.println(ex.getMessage());
    }
}
```

It is easier to write the MySQL command beforehand so is more visible, but the command is no different to the command that we would use on the MySQL shell.

A Statement type variable called “s” is written with the “c” variable using the create statement method that will later use the executeUpdate method with the sql command, pushing this piece of code into the database.

Insert Car

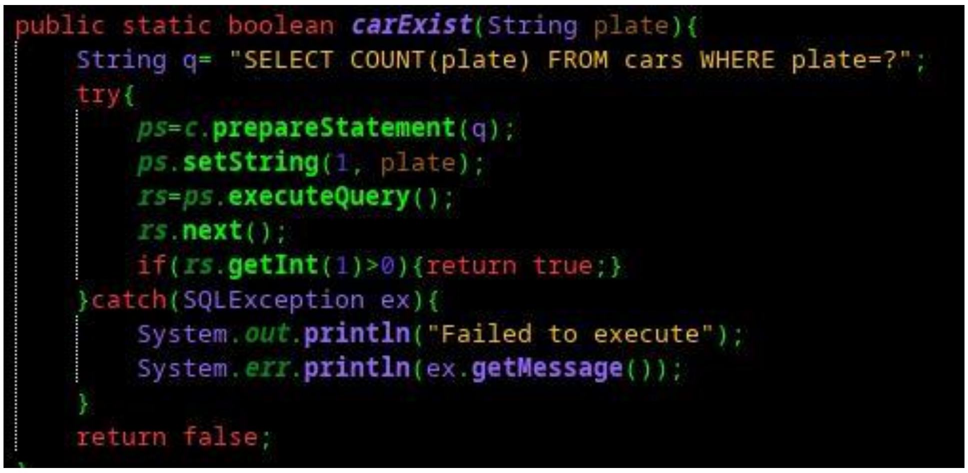


```
private static void insertCar(Car car){
    if(!carExist(car.getPlate())){
        String i="INSERT INTO cars VALUES(?,?,?,?,?,?)";
        try{
            ps=c.prepareStatement(i);
            ps.setString(1, car.getPlate());
            ps.setString(2, car.getBrand());
            ps.setString(3, car.getModel());
            ps.setString(4, car.getColour());
            ps.setInt(5, car.getYear());
            ps.setFloat(6, car.getPrice());

            ps.executeUpdate();
        }catch(SQLException ex){
            System.out.println("Operation Failed");
            System.err.println(ex.getMessage());
        }
    }
}
```

In this method we can see the usage of the method prepareStatement of the library SQL.PreparedStatement, what this does is letting us write an uncomplete SQL command that we later can write with set*[DataType]*() function using an index for the gaps in the command and the data that will fill the uncompleted fields.

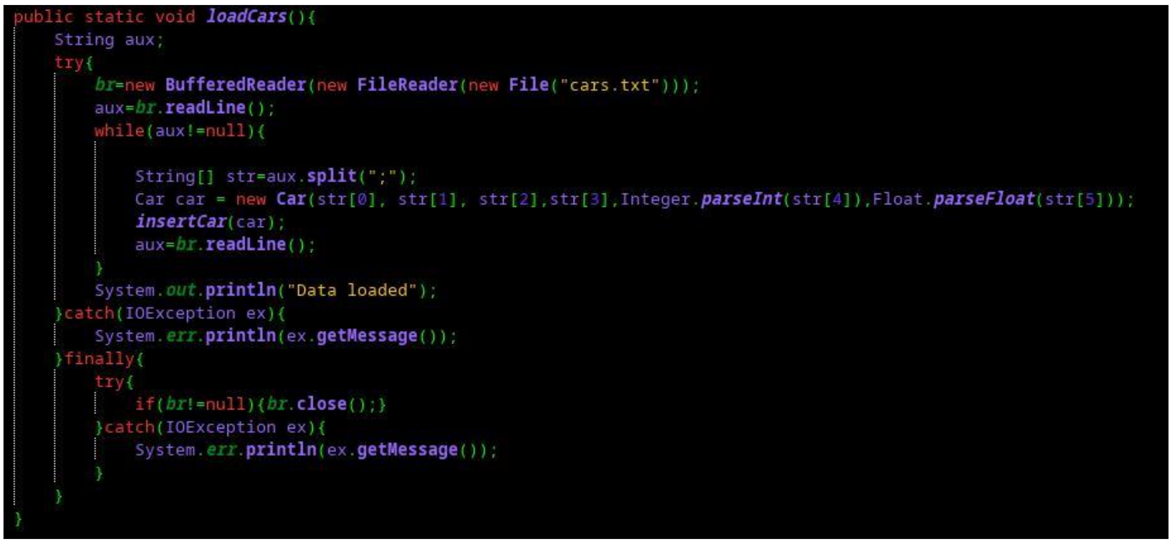
Does the car exist?



```java
public static boolean carExist(String plate){
    String q= "SELECT COUNT(plate) FROM cars WHERE plate=?";
    try{
        ps=c.prepareStatement(q);
        ps.setString(1, plate);
        rs=ps.executeQuery();
        rs.next();
        if(rs.getInt(1)>0){return true;}
    }catch(SQLException ex){
        System.out.println("Failed to execute");
        System.err.println(ex.getMessage());
    }
    return false;
```

Uses prepare statement to make a query searching for an instance of a car with the given plate, if there is at least one it returns a true boolean value.

Load cars



```java
public static void loadCars(){
    String aux;
    try{
        br=new BufferedReader(new FileReader(new File("cars.txt")));
        aux=br.readLine();
        while(aux!=null){

            String[] str=aux.split(";");
            Car car = new Car(str[0], str[1], str[2],str[3],Integer.parseInt(str[4]),Float.parseFloat(str[5]));
            insertCar(car);
            aux=br.readLine();
        }
        System.out.println("Data loaded");
    }catch(IOException ex){
        System.err.println(ex.getMessage());
    }finally{
        try{
            if(br!=null){br.close();}
        }catch(IOException ex){
            System.err.println(ex.getMessage());
        }
    }
}
```

This method utilizes BufferedReader, FileReader and File libraries in order to read a file with the information that we will import into the MySQL database. We must create a variable (“br”) to use these libraries at the same time, so in the constructor we must use them sequentially being the bottom-line File, so we can view or navigate the file system, then FileReader that takes the path and reads the file.

To advance inside the text file we must use the BufferedReader method readLine() and that string must be stored on an auxiliary variable so we can use it in a string split method that will store our 6 fields in an array, this array will be used to create an instance of Car that will later be used as a parameter of the previously explained method “insertCars”. After the insertion a new line of the buffer is written into the auxiliary variable and the process repeats itself until the BufferedReader doesn't find any next line on the text file.