

# Defence Against SQL Injections

## 1. What is SQL Injection

**SQL Injections** are a type of injection flaw. **Injection flaws** are a security vulnerability that allows the user to gain access to the backend database, shell command, or operating system call if the web app takes user input.

In SQL Injection, hackers append additional information within input boxes and can create, read, update, or delete data within the database. SQL Injection is the most common type of injection attack.

A SQL injection attack consists of insertion or “injection” of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

Let's us have a look at some these attacks and how a vulnerable or not properly designed application gets affected when a user tries to use malicious methods to do things that shouldn't be possible under normal circumstances and intended uses/

### 1. Create a new Database for this demonstration

```
mysql> create database ISA
-> ;
Query OK, 1 row affected (0.13 sec)

mysql> use ISA
Database changed
mysql>
```

### 2. Create a new Table Students, we will use this table for exploitations.

```
mysql> use ISA
Database changed
mysql> CREATE TABLE students( id SERIAL, name VARCHAR(255), marks INT, PRIMARY KEY(id) );
Query OK, 0 rows affected (0.14 sec)

mysql>
```

3. Let us now enter some dummy data.

```
mysql> INSERT INTO students(name, marks) VALUE
-> ('Suyash', 95),
-> ('Ujjwal', 100),
-> ('Rajat', 110),
-> ('Somya', 69),
-> ('Rishabh', 80);
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

4. Finally let's take a look at the created table

```
mysql> select * from Students;
+----+-----+-----+
| id | name  | marks |
+----+-----+-----+
| 1  | Suyash | 95    |
| 2  | Ujjwal | 100   |
| 3  | Rajat  | 110   |
| 4  | Somya  | 69    |
| 5  | Rishabh | 80    |
+----+-----+-----+
5 rows in set (0.01 sec)

mysql>
```

Now we have our Database Ready along with the tables and data that we need to test our application on for any vulnerabilities especially in regard to SQL Injection attack.

It is now time for us to work on our application written in java which uses JDBC to connect to our MySQL database. Below is the code for the same.

```
import java.sql.*;
import java.util.Scanner;

public class SQL_Injection {
    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter id to see marks");
        String id = sc.nextLine();

        try{
            String host = "jdbc:mysql://localhost:3306/ISA";
            String uName = "root";
            String uPass = "password";
            Connection con = DriverManager.getConnection(host, uName, uPass);
            Statement st=con.createStatement();

            String query ="select * from Students where id =" +id+"";
            ResultSet rs=st.executeQuery(query);

            while(rs.next())
            {
                System.out.println();
                System.out.println("id: "+rs.getString(1));
            }
        }
    }
}
```

```

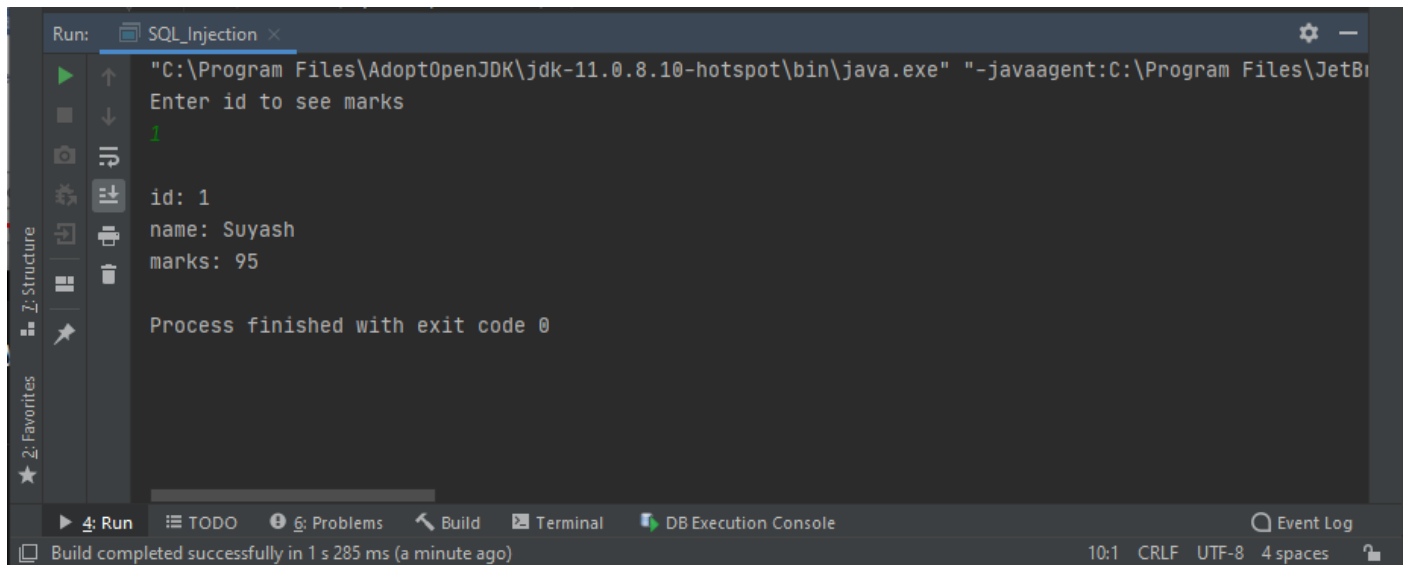
        System.out.println("name: "+rs.getString(2));
        System.out.println("marks: "+rs.getString(3));
    }

} catch (SQLException err) {
    System.out.println(err.getMessage());
}

}
}

```

In the application above our program takes input from the user as **id** from which the user can take a look at his or her marks lets see what we get when we give a normal input of **1** as **\_id**.



```

Run: SQL_Injection x
"C:\Program Files\AdoptOpenJDK\jdk-11.0.8-hotspot\bin\java.exe" "-javaagent:C:\Program Files\JetBr
Enter id to see marks
id: 1
name: Suyash
marks: 95
Process finished with exit code 0

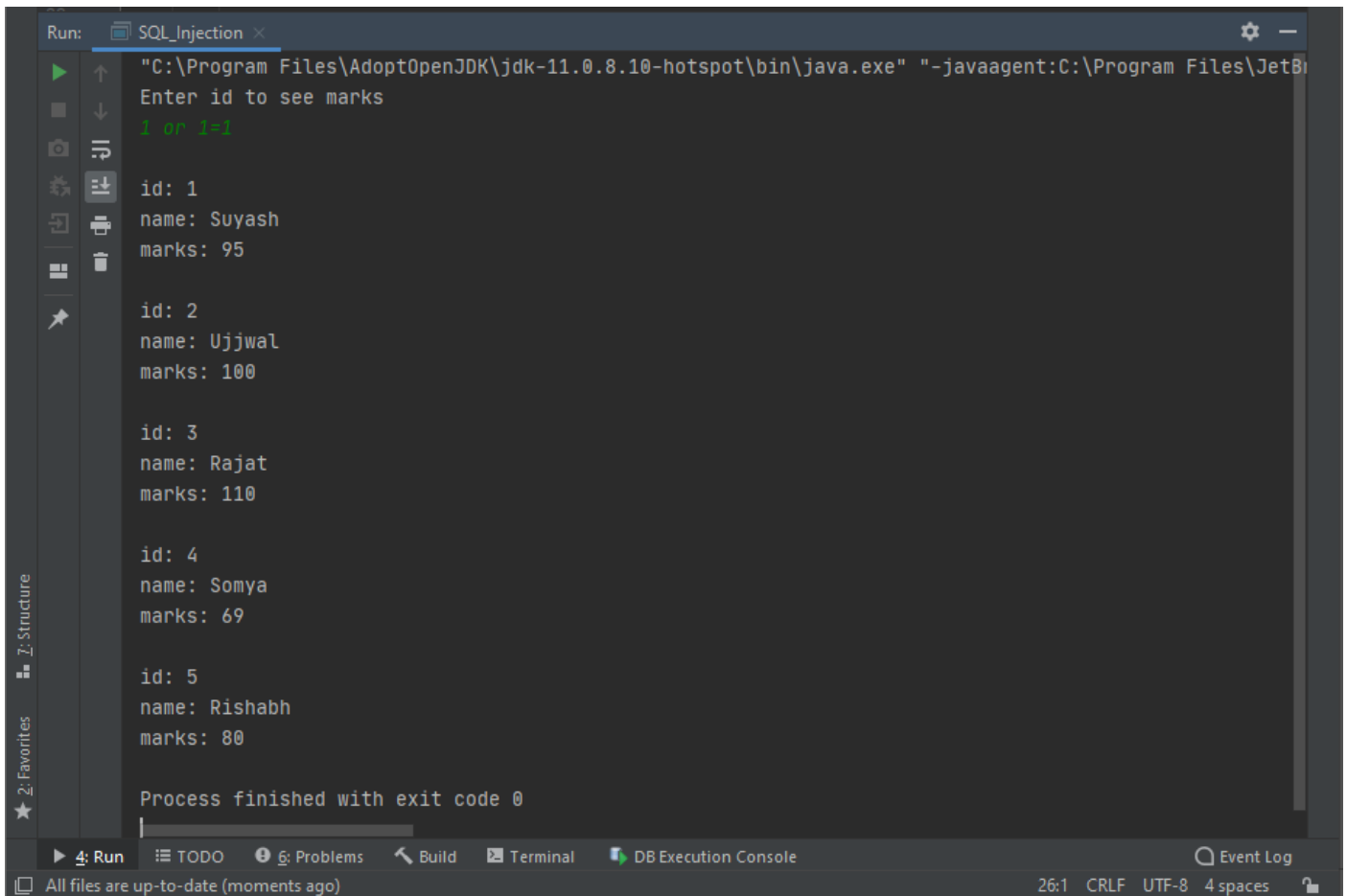
```

As we can see we get normal Output when the user gives a normal input and our program is working perfectly fine as intended.

Now let's see how a hacker or a user with malicious intent might misuse or abuse the application against its intended purpose.

### Common Technique 1 →

The 1=1 method is used to gain unauthorized access to the user's account. The following command can be added to access a user's account using SQL Injection.



```
Run: SQL_Injection x
"C:\Program Files\AdoptOpenJDK\jdk-11.0.8.10-hotspot\bin\java.exe" "-javaagent:C:\Program Files\JetB
Enter id to see marks
1 or 1=1

id: 1
name: Suyash
marks: 95

id: 2
name: Ujjwal
marks: 100

id: 3
name: Rajat
marks: 110

id: 4
name: Somya
marks: 69

id: 5
name: Rishabh
marks: 80

Process finished with exit code 0
```

As we can see if a user tries exploit this vulnerability, he can give optional parameters such as **1=1** which will always be true so if executed there will be no such error as user not found. In this example the user has gained access to all users' information which under normal circumstances he shouldn't have had.

## Common Technique 2 →

**Batched Query**, more problem arises when the malicious user or hacker tries to attempt some very dangerous queries in to send multiple commands to the SQL database. It can be used to delete, modify, or show the database.

First, let us make a few changes to our application, we will modify it so at to set the marks of a student to a set value using his or her id.

```
import java.sql.*;
import java.util.Scanner;

public class SQL_Injection {
    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter id to update marks");
        String id = sc.nextLine();

        try{
            String host = "jdbc:mysql://localhost:3306/ISA?allowMultiQueries=true";
            String uName = "root";
            String uPass = "password";
            Connection con = DriverManager.getConnection(host, uName, uPass);
```

```

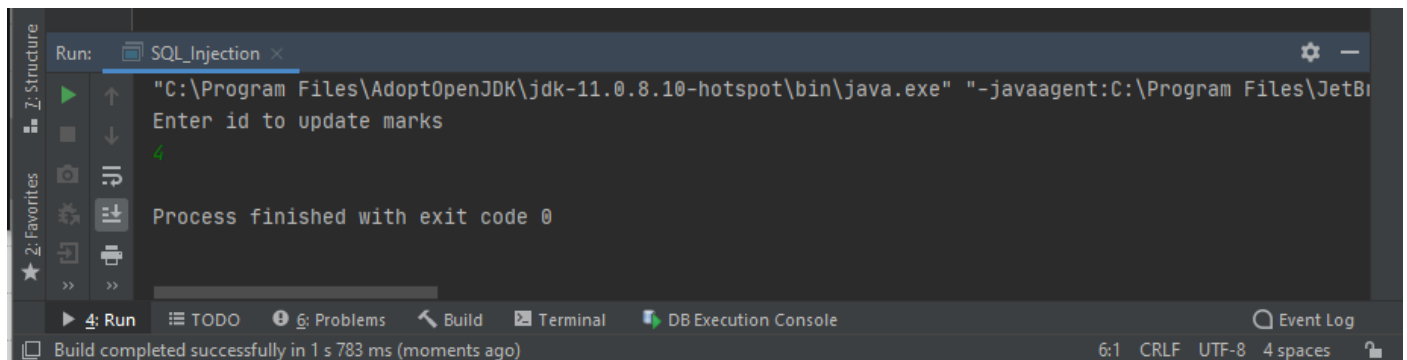
Statement st=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);

String query ="UPDATE Students set marks=100 WHERE id =" +id+"";
st.execute(query);

} catch (SQLException err) {
    System.out.println(err.getMessage());
}
}
}

```

Let us take a look how this might take place, under normal circumstances



We gave the id of 4 so in our database the marks should have been updated. Let us go over to our database and take a look at the changes,

```

mysql> select * from Students;
+----+-----+-----+
| id | name  | marks |
+----+-----+-----+
| 1  | Suyash | 95    |
| 2  | Ujjwal | 100   |
| 3  | Rajat  | 110   |
| 4  | Somya  | 69    |
| 5  | Rishabh | 80    |
+----+-----+-----+
5 rows in set (0.00 sec)

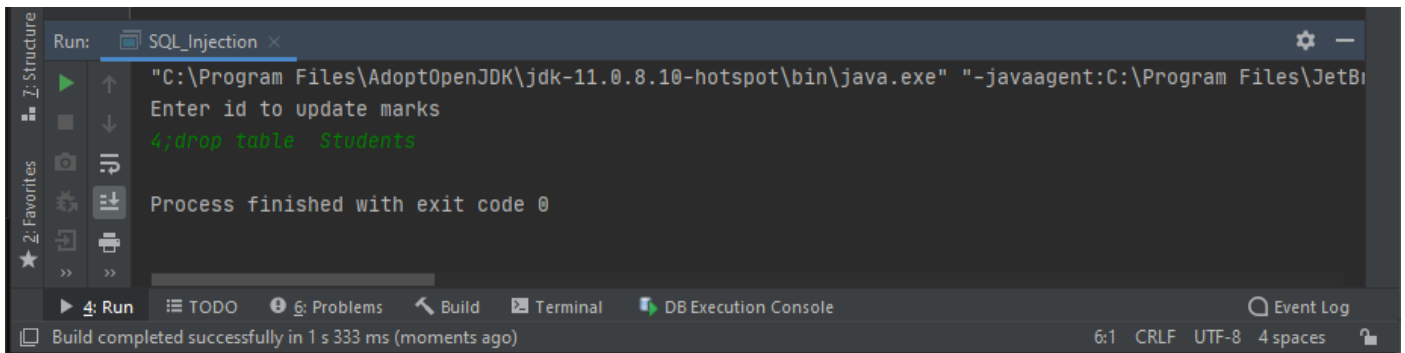
mysql> select * from Students;
+----+-----+-----+
| id | name  | marks |
+----+-----+-----+
| 1  | Suyash | 95    |
| 2  | Ujjwal | 100   |
| 3  | Rajat  | 110   |
| 4  | Somya  | 100   |
| 5  | Rishabh | 80    |
+----+-----+-----+
5 rows in set (0.00 sec)

mysql>

```

As we can see the values for id 4 has been changed from 69 to 100. The program ran successfully as intended.

Let us now see how a hacker or malicious user might take advantage of this.



Our application found nothing suspicious and simply executed the above statement without a worry in the world. But to see what really happened lets us take a look at our database

```
mysql> select * from Students;
+----+-----+-----+
| id | name   | marks |
+----+-----+-----+
| 1  | Suyash | 95    |
| 2  | Ujjwal | 100   |
| 3  | Rajat  | 110   |
| 4  | Somya  | 100   |
| 5  | Rishabh | 80    |
+----+-----+-----+
5 rows in set (0.00 sec)

mysql> select * from Students;
ERROR 1146 (42S02): Table 'isa.students' doesn't exist
mysql>
```

As we can see on trying to view our table, we get an error that the table doesn't exist. This means that the hacker was successful in his deeds and now our entire table has deleted. Let's Restore our Table.

```
mysql> select * from Students;
ERROR 1146 (42S02): Table 'isa.students' doesn't exist
mysql> CREATE TABLE students( id SERIAL, name VARCHAR(255), marks INT, PRIMARY KEY(id) );
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO students(name, marks) VALUE ('Suyash', 95), ('Ujjwal', 100), ('Rajat', 110), ('Somya', 69), ('Rishabh', 90);
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM students;
+----+-----+-----+
| id | name   | marks |
+----+-----+-----+
| 1  | Suyash | 95    |
| 2  | Ujjwal | 100   |
| 3  | Rajat  | 110   |
| 4  | Somya  | 69    |
| 5  | Rishabh | 90    |
+----+-----+-----+
5 rows in set (0.00 sec)

mysql>
```

## 2. Preventions →

### 1. Enforcing Least Privilege

Avoiding administrative privileges. Don't connect your application to the database using an account with root access. This should be done only if absolutely needed since the attackers could gain access to the whole system. Even the non-administrative accounts server could place risk on an application, even more so if the database server is used by multiple applications and databases.

In our application we have used the root account hence we automatically have all privileges

```
String uName = "root";
String uPass = "password";
Connection con = DriverManager.getConnection(host, uName, uPass);
```

If we had used another account with lesser privileges dropping a table wouldn't have been possible. Also, we should avoid using parameters such as support for execution of multiple queries unless very necessary. In our example we used **allowMultipleQueries=true** this allowed the hacker to execute more than one statement. Same for our result set which was set to support multiple results using the scrolling functionality and concurrent update.

```
String host = "jdbc:mysql://localhost:3306/ISA?allowMultiQueries=true";
String uName = "root";
String uPass = "password";
Connection con = DriverManager.getConnection(host, uName, uPass);
Statement st=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);

String query ="UPDATE Students set marks=100 WHERE id ="+id+"";
st.execute(query);
```

In addition we used **st.execute** which lets us execute multiple queries at once had we gone for a safer approach, we would have encountered an error which is far better than our table being deleted. Following is an example of how we might have prevented this to occur

```
import java.sql.*;
import java.util.Scanner;

public class SQL_Injection {
    public static void main(String[] args){

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter id to update marks");
        String id = sc.nextLine();

        try{
            String host = "jdbc:mysql://localhost:3306/ISA";
            String uName = "root";
            String uPass = "password";

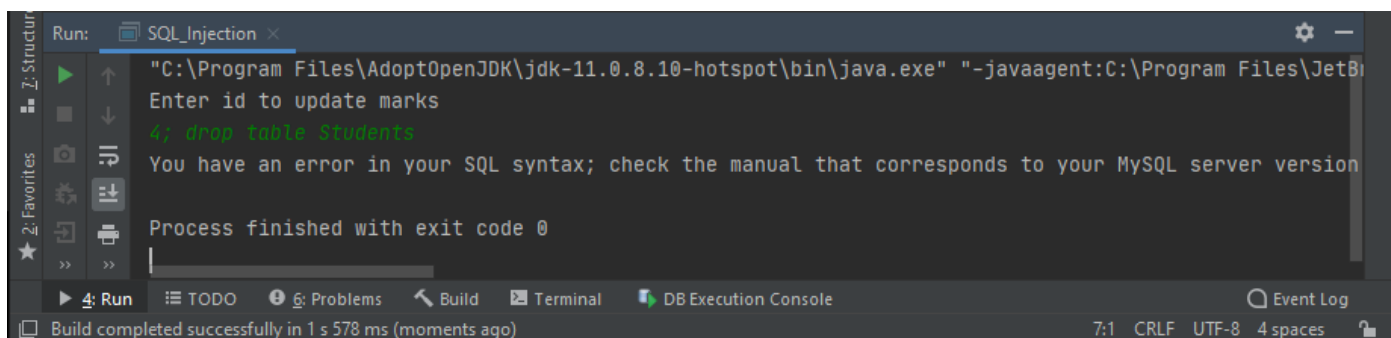
            Connection con = DriverManager.getConnection(host, uName, uPass);
            Statement st=con.createStatement();

            String query ="UPDATE Students set marks=100 WHERE id ="+id+"";
            st.executeUpdate(query);
```

```

    } catch (SQLException err) {
        System.out.println(err.getMessage());
    }
}

```



```

Run: SQL_Injection x
"C:\Program Files\AdoptOpenJDK\jdk-11.0.8.10-hotspot\bin\java.exe" "-javaagent:C:\Program Files\Jet8
Enter id to update marks
4; drop table Students
You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version
Process finished with exit code 0

```

As clearly evident we got an error message instead of running the very dangerous payload. By Reducing the privileges of the application **NO** substantially harmful query is executable.

## 2. Using Input Validation →

Before even sending the query to be executed we can before anything else perform input validation that is to check if there is any sort of abnormality present in the given string that may trigger some unwanted behaviour. If caught we can simply throw an error and end execution gracefully.

For example, to be able to execute multiple queries there needs to exist multiple semicolons (;) hence we can detect these semicolons we can prevent multiple SQL statements being executed. I used the following code snippet to check for the same.

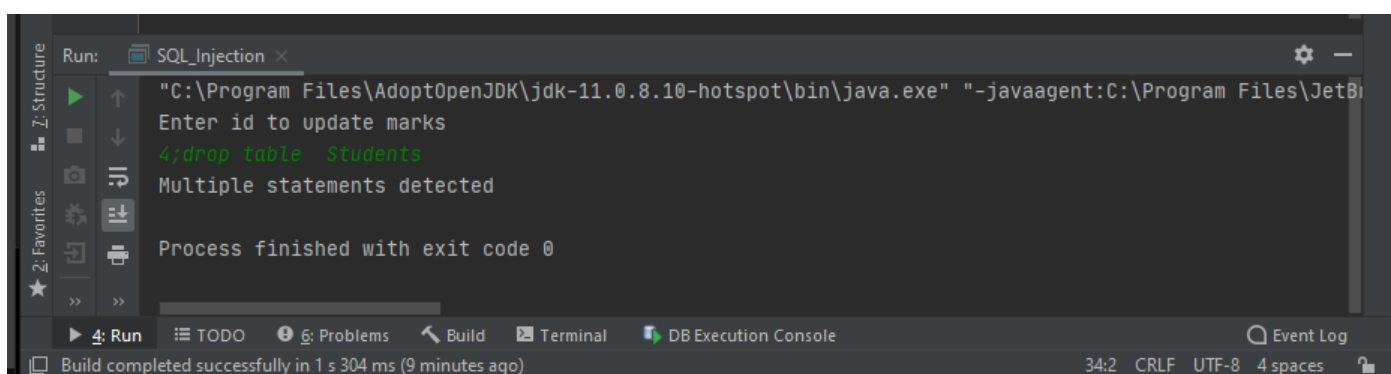
```

String query = "UPDATE Students set marks=100 WHERE id ="+id+";";
int count = 0;

for (int i = 0; i < query.length(); i++) {
    if (query.charAt(i) == ';') {
        count++;
    }
    if(count>1) throw new SQLException("Multiple statements detected");
}
st.executeUpdate(query);

```

If a user with a malicious intent or a hacker tries to exploit this vulnerability, we will be able to catch it and throw an exception error before encountering and loss.



```

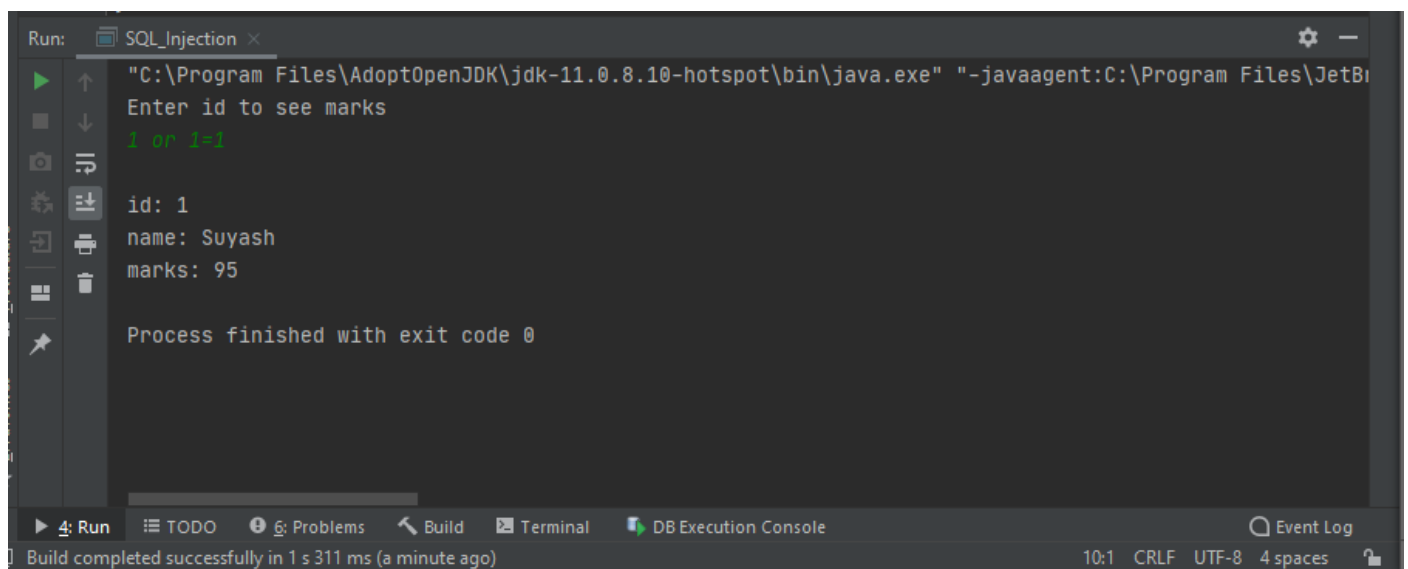
Run: SQL_Injection x
"C:\Program Files\AdoptOpenJDK\jdk-11.0.8.10-hotspot\bin\java.exe" "-javaagent:C:\Program Files\Jet8
Enter id to update marks
4;drop table Students
Multiple statements detected
Process finished with exit code 0

```



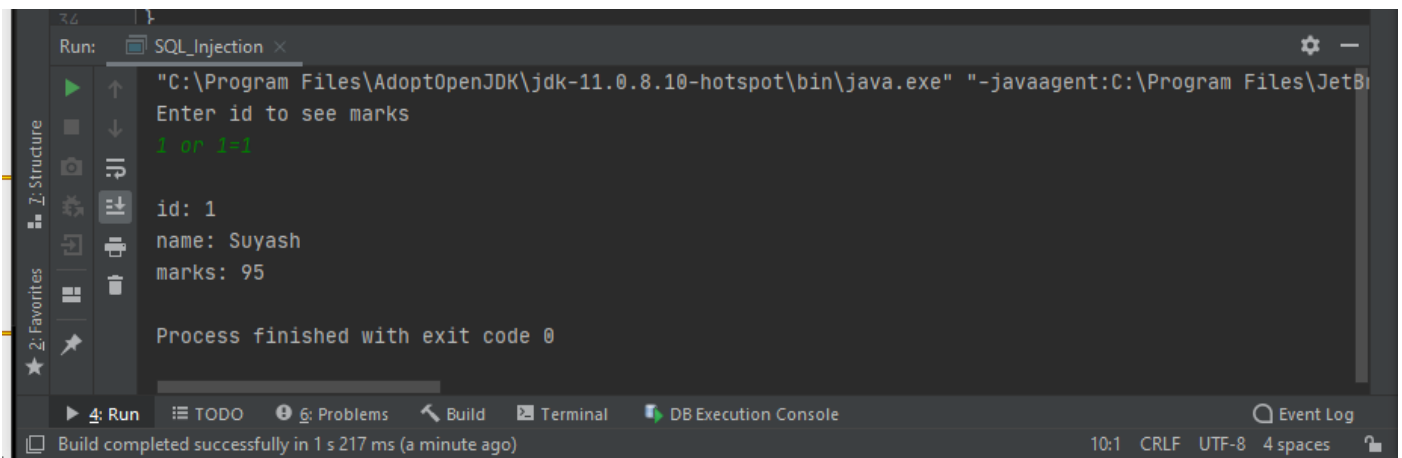
Coming over to our first example where a user could see his or her marks but the hacker was able to get away with more when he used a query with conjunction **OR 1=1** we can prevent this by taking only 1 word as input or as in the case of id only an integer value. Any deviation from this format will lead to an error or ignoring of the remaining input buffer. and prevention of exploitation of any kind of vulnerability.

```
Scanner sc = new Scanner(System.in);  
System.out.println("Enter id to see marks");  
int id = sc.nextInt();  
  
try{  
    String host = "jdbc:mysql://localhost:3306/ISA";  
    String uName = "root";  
    String uPass = "password";  
    Connection con = DriverManager.getConnection(host, uName, uPass);  
    Statement st=con.createStatement();  
  
    String query ="select * from Students where id ="+id+"";  
    ResultSet rs=st.executeQuery(query);
```



Only integer part was extracted and rest was ignored. We can also use just one input at a time or **list-input validation** so we don't take in more parameters.

```
Scanner sc = new Scanner(System.in);  
System.out.println("Enter id to see marks");  
String id = sc.next();  
  
try{  
    String host = "jdbc:mysql://localhost:3306/ISA";  
    String uName = "root";  
    String uPass = "password";  
    Connection con = DriverManager.getConnection(host, uName, uPass);  
    Statement st=con.createStatement();  
  
    String query ="select * from Students where id ="+id+"";  
    ResultSet rs=st.executeQuery(query);
```



```
Run: SQL_Injection x
"C:\Program Files\AdoptOpenJDK\jdk-11.0.8.10-hotspot\bin\java.exe" "-javaagent:C:\Program Files\JetB
Enter id to see marks
1 or 1=1

id: 1
name: Suyash
marks: 95

Process finished with exit code 0

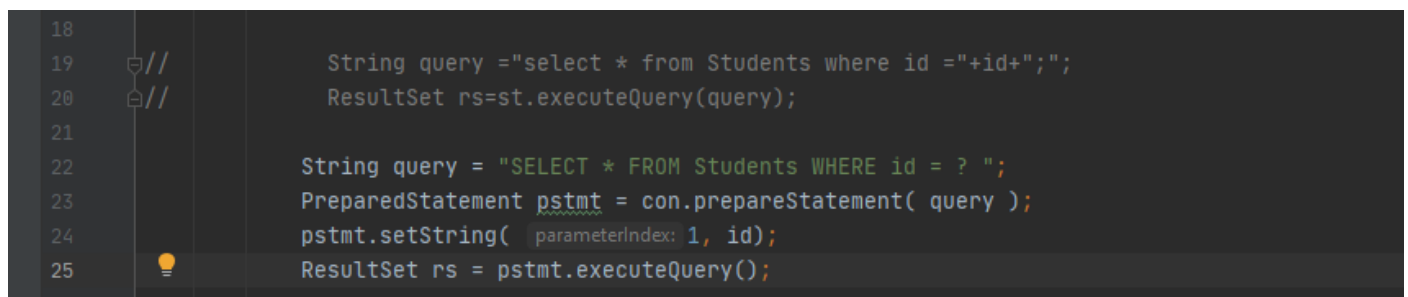
Build completed successfully in 1 s 217 ms (a minute ago) 10:1 CRLF UTF-8 4 spaces
```

Once the hacker was unable to exploit the vulnerability and couldn't extract any more data than intended.

### 3. Use of Prepared Statements (with Parameterized Queries)

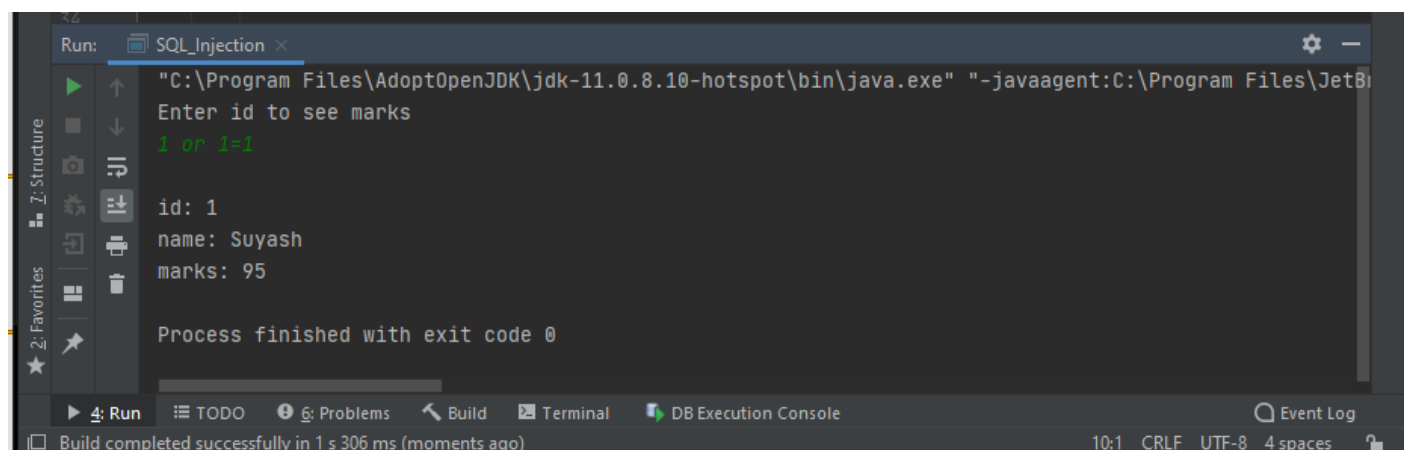
The use of prepared statements with variable binding (aka parameterized queries) is how all developers should first be taught how to write database queries. They are simple to write, and easier to understand than dynamic queries. Parameterized queries force the developer to first define all the SQL code, and then pass in each parameter to the query later. This coding style allows the database to distinguish between code and data, regardless of what user input is supplied.

Prepared statements ensure that an attacker is not able to change the intent of a query, even if SQL commands are inserted by an attacker. In the safe example below, if an attacker were to enter the userID of tom' or '1'=1, the parameterized query would not be vulnerable and would instead look for a username which literally matched the entire string tom' or '1'=1.



```
18 // String query ="select * from Students where id ="+id+";";
19 // ResultSet rs=st.executeQuery(query);
20
21
22 String query = "SELECT * FROM Students WHERE id = ? ";
23 PreparedStatement pstmt = con.prepareStatement( query );
24 pstmt.setString( parameterIndex: 1, id);
25 ResultSet rs = pstmt.executeQuery();
```

For example, as shown above the commented part is what we had been using so far but we now switched to **PreparedStatement** offered by the JAVA language to implement parameterization of queries to execute the same database query.



```
Run: SQL_Injection x
"C:\Program Files\AdoptOpenJDK\jdk-11.0.8.10-hotspot\bin\java.exe" "-javaagent:C:\Program Files\JetB
Enter id to see marks
1 or 1=1

id: 1
name: Suyash
marks: 95

Process finished with exit code 0

Build completed successfully in 1 s 306 ms (moments ago) 10:1 CRLF UTF-8 4 spaces
```

Once again, we were able to prevent the hacker from gaining any sensitive information whatsoever.