

Theory: JDBC Prepared Statements

⌚ 16 minutes 0 / 5 problems solved

Skip this topic

Start practicing

274 users solved this topic. Latest completion was about 2 hours ago.

§1. What is a Prepared Statement?

We have already observed JDBC statements and how to use them to interact with the database. However, as you could've noticed, the JDBC `Statement` interface is not so flexible. Since you have to specify each statement as a string, there is no way to set statement parameters such as a record id or a column value. Another disadvantage of the `Statement` interface is that each time you execute it, the database compiles the SQL statement without caching it. It leads to an execution time increase.

To cope with such problems, JDBC API provides a special type of statement that is called a **prepared statement**, represented by the `java.sql.PreparedStatement` interface. Similarly to the statements, prepared statements can create, select, update, and delete database data. However, prepared statements have several advantages.

§2. Prepared Statement parameters

To understand the difference between `Statement` and `PreparedStatement` and see how `PreparedStatement` handles SQL query parameters, let's consider an example. First, we create an SQLite database called `music` and add the `artists` table to the database:

```
1 CREATE TABLE music.artists
2 (
3     id          INTEGER PRIMARY KEY,
4     name        TEXT    NOT NULL,
5     origin       TEXT,
6     songs_number INTEGER
7 );
```

To add the most influential band of all time to the `artists` table by using `Statement` interface we can write the following code:

```
1 try (Statement statement = con.createStatement()) {
2     statement.executeUpdate("INSERT INTO artists (name, origin, songs_number) VALU
ES " +
3         "('The Beatles', 'Liverpool, England', 213)");
4 }
```

Now if we need to add more bands, we have to provide a new string with a changed `name`, `origin`, and `songs_number` values. However, `PreparedStatement` provides a more elegant way to do so:

```
1
String insert = "INSERT INTO artists (name, origin, songs_number) VALUES (?, ?, ?)";
2
3 try (PreparedStatement preparedStatement = con.prepareStatement(insert)) {
4     preparedStatement.setString(1, "The Beatles");
5     preparedStatement.setString(2, "Liverpool, England");
6     preparedStatement.setInt(3, 213);
7
8     preparedStatement.executeUpdate();
9 }
```

To create a `PreparedStatement` object, we used the `prepareStatement(String sql)` method of the `Connection` object. Note, how the value of the insert variable that we passed as a parameter to the `prepareStatement` method contains the question mark (`?`) symbols. Such symbols are placeholders for the values that will be passed to the query. The first placeholder corresponds to the name of the artist, the second one to its origin, and the third one to the number of artist songs.

Current topic:

JDBC Prepared Statements ...

Topic depends on:

✗ `JDBC Statements` ...

Topic is required for:

JDBC Transactions ...

Table of contents:

1 JDBC Prepared Statements

§1. What is a Prepared Statement?

§2. Prepared Statement parameters

§3. Prepared Statement setters

§4. Conclusion

Feedback & Comments

§3. Prepared Statement setters

`PreparedStatement` interface provides setter methods to replace an SQL query placeholder with the value. The signature of the setters is the following:

1) The name of the method is `setXXX`, where XXX is a type. We need to use the appropriate `setXXX` method for a given SQL type. In the example above, we've used the `setString` method since the SQL type of artist name is `TEXT`.

2) Each `PreparedStatement.setXXX` method accepts at least two parameters. The first one indicates the parameter index, while the second one — the parameter value of the specific type. In the example above, the `songs_number` corresponds to the third `?` placeholder and has an `INTEGER` type in the database, which corresponds to the Java `int` type.

Sometimes you need to set the parameter value to `NULL`. For such cases, `PreparedStatement` provides a `setNull` method that accepts the parameter index and an SQL type code defined in Java. For example, we can set the artist's origin to `NULL` in the following way:

```
1 String updateOrigin = "UPDATE artists SET origin = ? WHERE id = ?";
2
3
4 try (PreparedStatement preparedStatement = con.prepareStatement(updateOrigin)) {
5     preparedStatement.setNull(1, Types.VARCHAR);
6     preparedStatement.setInt(2, 1);
7     preparedStatement.executeUpdate();
8 }
```

Another possible situation is when you do not know the exact type of the parameter at runtime. In such a case, you can use the `setObject` method that is the most general one. It can accept a parameter index and any `Object` as an argument. Let's see an example:

```
1 String insert = "INSERT INTO artists (name, songs_number) VALUES (?, ?)";
2 try (PreparedStatement preparedStatement = con.prepareStatement(insert)) {
3     preparedStatement.setObject(1, new StringBuilder("Madonna"));
4     preparedStatement.setObject(2, new Long(88));
5
6     preparedStatement.executeUpdate();
7 }
```

Statement placeholders can be used for values only, which means that you cannot replace the SQL keyword with a placeholder.

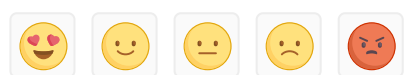
§4. Conclusion

`PreparedStatement` interface represents a precompiled SQL statement that provides methods to bind the query parameters to the values. In contrast to `Statement`, `PreparedStatement` is cached by the database. This means that if you execute a statement for the first time, the database will compile, cache, and execute it. Then, if you execute the same statement with other parameters, from the database perspective the statement hasn't been changed, so it will not recompile the statement because the database contains a cached one. It leads to boosting database execution performance.

All things considered, using prepared statements reduces the execution time, eases inserting parameters into SQL statements, and allows us to reuse a prepared statement with new parameters. In addition, prepared statements protect against SQL injection through a separation of the query code and parameter values.

 Report a typo

31 users liked this theory. 0 didn't like it. What about you?



Start practicing

[Comments \(2\)](#)

[Hints \(0\)](#)

[Useful links \(0\)](#)

[Show discussion](#)