

# Using Prepared Statements

This page covers the following topics:

- [Overview of Prepared Statements](#)
- [Creating a PreparedStatement Object](#)
- [Supplying Values for PreparedStatement Parameters](#)

## Overview of Prepared Statements

Sometimes it is more convenient to use a `PreparedStatement` object for sending SQL statements to the database. This special type of statement is derived from the more general class, `Statement`, that you already know.

If you want to execute a `Statement` object many times, it usually reduces execution time to use a `PreparedStatement` object instead.

The main feature of a `PreparedStatement` object is that, unlike a `Statement` object, it is given a SQL statement when it is created. The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled. As a result, the `PreparedStatement` object contains not just a SQL statement, but a SQL statement that has been precompiled. This means that when the `PreparedStatement` is executed, the DBMS can just run the `PreparedStatement` SQL statement without having to compile it first.

Although `PreparedStatement` objects can be used for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters. The advantage of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it. Examples of this are in the following sections.

The following method, `CoffeesTable.updateCoffeeSales`, stores the number of pounds of coffee sold in the current week in the `SALES` column for each type of coffee, and updates the total number of pounds of coffee sold in the `TOTAL` column for each type of coffee:

```
public void updateCoffeeSales(HashMap<String, Integer> salesForWeek)
    throws SQLException {

    PreparedStatement updateSales = null;
    PreparedStatement updateTotal = null;

    String updateString =
        "update " + dbName + ".COFFEES " +
        "set SALES = ? where COF_NAME = ?";

    String updateStatement =
        "update " + dbName + ".COFFEES " +
        "set TOTAL = TOTAL + ? " +
        "where COF_NAME = ?";

    try {
        con.setAutoCommit(false);
        updateSales = con.prepareStatement(updateString);
        updateTotal = con.prepareStatement(updateStatement);

        for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
            updateSales.setInt(1, e.getValue().intValue());
            updateSales.setString(2, e.getKey());
            updateSales.executeUpdate();
            updateTotal.setInt(1, e.getValue().intValue());
            updateTotal.setString(2, e.getKey());
            updateTotal.executeUpdate();
            con.commit();
        }
    } catch (SQLException e) {
        JDBCUtilities.printSQLException(e);
        if (con != null) {
            try {
                System.err.print("Transaction is being rolled back");
                con.rollback();
            } catch (SQLException excep) {
                JDBCUtilities.printSQLException(excep);
            }
        }
    } finally {
```

```

        if (updateSales != null) {
            updateSales.close();
        }
        if (updateTotal != null) {
            updateTotal.close();
        }
        con.setAutoCommit(true);
    }
}

```

## Creating a PreparedStatement Object

The following creates a `PreparedStatement` object that takes two input parameters:

```

String updateString =
    "update " + dbName + ".COFFEES " +
    "set SALES = ? where COF_NAME = ?";
updateSales = con.prepareStatement(updateString);

```

## Supplying Values for PreparedStatement Parameters

You must supply values in place of the question mark placeholders (if there are any) before you can execute a `PreparedStatement` object. Do this by calling one of the setter methods defined in the `PreparedStatement` class. The following statements supply the two question mark placeholders in the `PreparedStatement` named `updateSales`:

```

updateSales.setInt(1, e.getValue().intValue());
updateSales.setString(2, e.getKey());

```

The first argument for each of these setter methods specifies the question mark placeholder. In this example, `setInt` specifies the first placeholder and `setString` specifies the second placeholder.

After a parameter has been set with a value, it retains that value until it is reset to another value, or the method `clearParameters` is called. Using the `PreparedStatement` object `updateSales`, the following code fragment illustrates reusing a prepared statement after resetting the value of one of its parameters and leaving the other one the same:

```

// changes SALES column of French Roast
//row to 100

updateSales.setInt(1, 100);
updateSales.setString(2, "French_Roast");
updateSales.executeUpdate();

// changes SALES column of Espresso row to 100
// (the first parameter stayed 100, and the second
// parameter was reset to "Espresso")

updateSales.setString(2, "Espresso");
updateSales.executeUpdate();

```

## Using Loops to Set Values

You can often make coding easier by using a `for` loop or a `while` loop to set values for input parameters.

The `CoffeesTable.updateCoffeeSales` method uses a for-each loop to repeatedly set values in the `PreparedStatement` objects `updateSales` and `updateTotal`:

```

for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {

    updateSales.setInt(1, e.getValue().intValue());
    updateSales.setString(2, e.getKey());

    // ...
}

```

The method `CoffeesTable.updateCoffeeSales` takes one argument, `HashMap`. Each element in the `HashMap` argument contains the name of one type of coffee and the number of pounds of that type of coffee sold during the current week. The for-each loop iterates through each element of the `HashMap` argument and sets the appropriate question mark placeholders in `updateSales` and `updateTotal`.

## Executing PreparedStatement Objects

As with `Statement` objects, to execute a `PreparedStatement` object, call an execute statement: `executeQuery` if the query returns only one `ResultSet` (such as a `SELECT` SQL statement), `executeUpdate` if the query does not return a `ResultSet` (such as an `UPDATE` SQL statement), or `execute` if the query might return more than one `ResultSet` object. Both `PreparedStatement` objects in `CoffeesTable.updateCoffeeSales` contain `UPDATE` SQL statements, so both are executed by calling `executeUpdate`:

```
updateSales.setInt(1, e.getValue().intValue());
updateSales.setString(2, e.getKey());
updateSales.executeUpdate();

updateTotal.setInt(1, e.getValue().intValue());
updateTotal.setString(2, e.getKey());
updateTotal.executeUpdate();
con.commit();
```

No arguments are supplied to `executeUpdate` when they are used to execute `updateSales` and `updateTotals`; both `PreparedStatement` objects already contain the SQL statement to be executed.

**Note:** At the beginning of `CoffeesTable.updateCoffeeSales`, the auto-commit mode is set to `false`:

```
con.setAutoCommit(false);
```

Consequently, no SQL statements are committed until the method `commit` is called. For more information about the auto-commit mode, see [Transactions](#).

### Return Values for the executeUpdate Method

Whereas `executeQuery` returns a `ResultSet` object containing the results of the query sent to the DBMS, the return value for `executeUpdate` is an `int` value that indicates how many rows of a table were updated. For instance, the following code shows the return value of `executeUpdate` being assigned to the variable `n`:

```
updateSales.setInt(1, 50);
updateSales.setString(2, "Espresso");
int n = updateSales.executeUpdate();
// n = 1 because one row had a change in it
```

The table `COFFEES` is updated; the value 50 replaces the value in the column `SALES` in the row for `Espresso`. That update affects one row in the table, so `n` is equal to 1.

When the method `executeUpdate` is used to execute a DDL (data definition language) statement, such as in creating a table, it returns the `int` value of 0. Consequently, in the following code fragment, which executes the DDL statement used to create the table `COFFEES`, `n` is assigned a value of 0:

```
// n = 0
int n = executeUpdate(createTableCoffees);
```

Note that when the return value for `executeUpdate` is 0, it can mean one of two things:

- The statement executed was an update statement that affected zero rows.
- The statement executed was a DDL statement.