5. Using PreparedStatements

Contents

[1. Introduction 1](#_Toc6424)

[2. Create and Execute Queries with a PreparedStatement 1](#_Toc8153)

[3. Using PreparedStatement's Execute Method 1](#_Toc13508)

[4. Parameterizing PreparedStatement 1](#_Toc18925)

[5. Demonstration - Parameterizing PreparedStatement 2](#_Toc24681)

[6. Summary 2](#_Toc26115)

# 1. Introduction

=>slides: Pg. 1

Welcome back to the Database Applications with JDBC in Java SE Applications course. This is the Using PreparedStatements module. So in the previous modules, we saw how to connect to a database using a connection, and how we can pass that connection a JDBC URL to allow us to load the driver and find the database to connect to. And we saw briefly using a PreparedStatement to get the data from a database.

=>slides: Pg. 2

So now in this module, we'll take this further. So we'll use the PreparedStatement to read and write to the database, we'll show that the PreparedStatement is parameterized, and we'll show how we can pass parameters into that statement. We'll see that the PreparedStatement has multiple methods on it that we can call, and that we can use either ExecuteQuery or Execute to both select data and/or update data in the database.

=>slides: Pg. 3

So JDBC has different statement interfaces, and the base interface is called Statement, and we can use this to execute statements against our database. However, the Statement interface won't be part of the examination. That's because it's highly discouraged these days. There are security risks with just using raw statements, trying to query a database. Instead, we'd either use PreparedStatement or CallableStatement, and both PreparedStatement and CallableStatement derive from Statement. PreparedStatement let's us use parameterized queries, and we'll see those in a moment, and CallableStatements let's us call stored procedures, and we'll see those in a module later in this course. =>slides: Pg. 4

So PreparedStatement represents an SQL statement that will be sent to the database, and we can use the PreparedStatement for any CRUD operation. So we can use this and do create, read, update, or delete against the database. And we can also use this for any complexity of SQL statements as well. Up until now, we've seen very simple SQL statements, but if you need a statement that has multiple joins, and multiple selects, and nested clauses, then you can use PreparedStatements for that as well. So it's basically any SQL that we could send to the database can be used inside a PreparedStatement. So PreparedStatement has methods that allow us to read data from the database, so we can bring back something known as a resultSet, and we can also use PreparedStatements to modify the data within the database as well. And we'll see examples of both of those things in this module.

=>slides: Pg. 5

Now again, just to emphasize, like connections, PreparedStatements must be closed, and again to do this, we'd use try‑with‑resources, but also remember that closing a connection will also close any associated PreparedStatements. So if I use a connection to create a PreparedStatement, and then I close that connection, the PreparedStatement will also be closed.

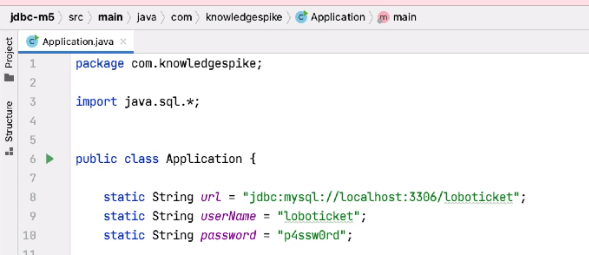
=>slides: Pg. 6

So, on PreparedStatements, we have three execute methods. We have the executeUpdate method that we use to update the database. So we'd use that to send inserts, updates, and deletes to the database. We can use executeQuery to execute a PreparedStatement that's doing a query against the database. So we use that to send select to the database. And then finally, there's a general purpose execute method, and we'll see examples of each of these in a moment. Okay, so let's take a look. Let's go and write some queries, and use PreparedStatement to execute those queries.

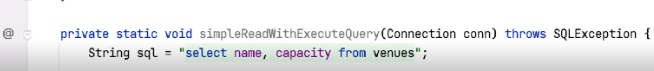
# Create and Execute Queries with a PreparedStatement

=>slides: Pg. 7

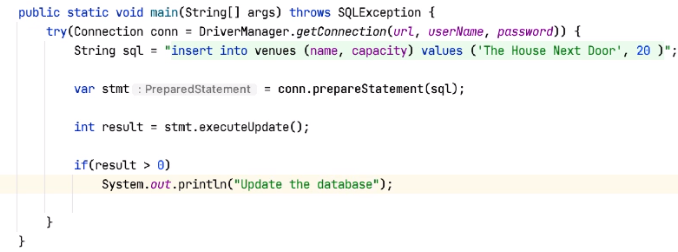
So the first thing I'm going to do is to take a look at using PreparedStatements without any parameters. So we can say something like this.



So we have some SQL, and let's say select name, capacity from venues. So we have the connection created in a try block above. So to get a PreparedStatement, I can say var stmt = conn.PrepareStatement, and I can pass it the SQL. And notice that the statement variable here is of type PreparedStatement. So once I have the statement, I can execute it. So I can say stmt.executeQuery because we know this is a SQL query, and this returns me a resultSet. Once I have that resultSet, I can iterate over it, and we'll cover exactly what happens here in the next module, but for now, let me just do this. I'll just print out a dot for every row we get back in this resultSet, and then print a new line at the end. Okay, so once I have this, I can execute the code. And notice we get four dots here, so there's four rows in my venues table. So we can see that we can do a select here, so what about inserts, updates, and deletes?

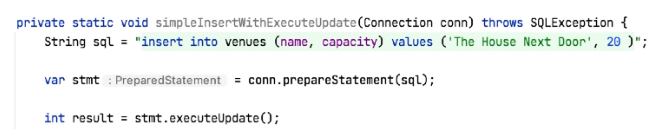


Well first of all, let me just extract this into a method, and I'll call this simpleReadWithExecuteQuery.

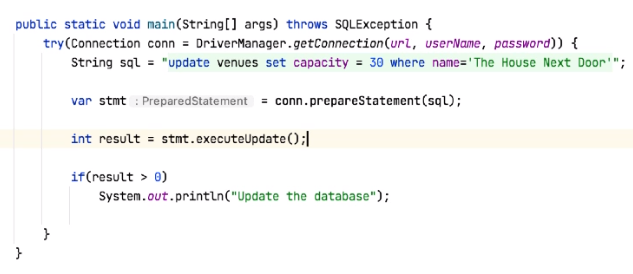


Okay, so for an insert, I can say String sql =, so we'll say insert into venues (name, capacity), we'll need the values for these columns, and the values are going to be The House Next Door, and the capacity can be 20. So once I have that, we'll do something similar to the select. We can say var statement equals conn.prepareStatement, and we can pass it the sql. But now rather than calling executeQuery, we can call executeUpdate, and executeUpdate returns a count of the number of rows that have been affected by this query. So let's say int result = stmt.executeUpdate. And then if result is greater than 0, System.out.println updated the database. So if I run this again,



then indeed we've updated the database. 

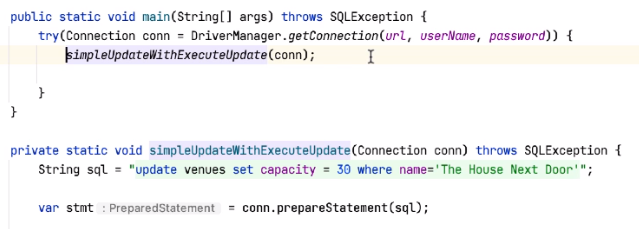
So again, let me extract this into a method. This time we'll call it simpleInsertWithExecuteUpdate. Okay, and we can do similar things for deletes and updates.



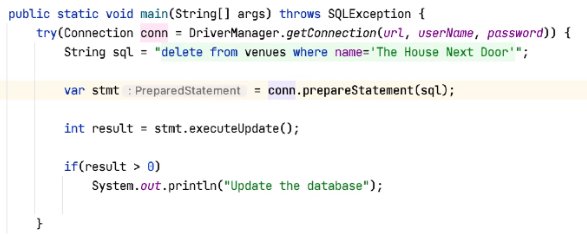
So keeping the code the same, we can say update venues, set capacity = 30 where name = The House Next Door. Again, we call PreparedStatement, again we call executeUpdate, and again we can check the result value.



And if I execute this code, then again we've updated the database.



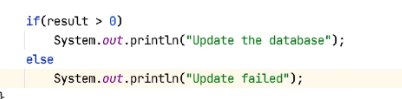
So again we refactor the code to say simpleUpdateWithExecuteUpdate, and then we can do the same thing with delete.



So instead of doing update venues, we can do delete from venues, where name = The House Next Door, prepare the statement, execute the update, and check the results, and again if I rerun this,



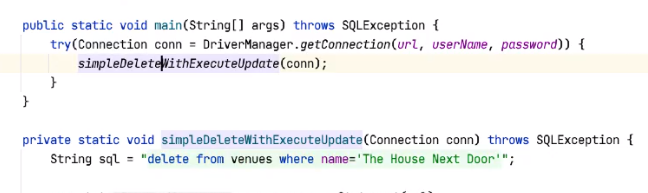
we say Update the database,



and then you put an else clause here and say Update failed. So now if I run this again, the results should be 0, so no rows should have been changed, and we shouldn't be updating the database.

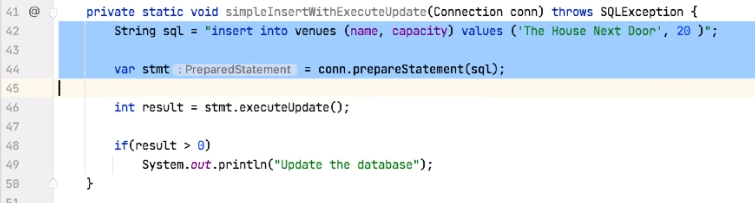


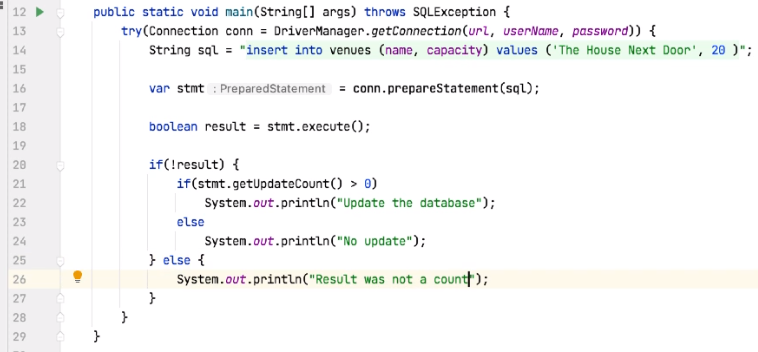
So I run this again, sure enough, Update failed. So if the insert, update, or delete succeed, then we get back a count of the number of rows that have been changed. If nothing is changed, we get back a count of 0.



So again, let me extract this into its own method, and we'll call this simpleDeleteWithExecuteUpdate.

# Using PreparedStatement's Execute Method

So if I grab this insert code first of all, and then here, rather than calling executeUpdate, we'll just call execute.



Now you notice straightaway, execute doesn't return an integer, in fact, it returns a Boolean, and the Boolean tells me whether the result of the execute returns a ResultSet or not. So if the Boolean is false, there's no ResultSet, and if it's true, there is a ResultSet. So here I can say if not result, and then I can say stmt.getUpdateCount, and this again will tell me how many rows were changed in the database. So now I can say if stmt.getUpdateCount is greater than 0, then we update the database, else No update. And also for the result we can add an else to say Result was not a count.

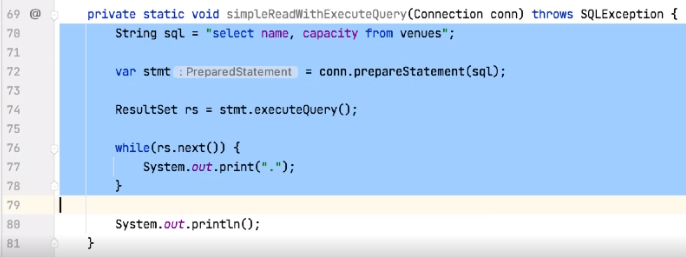


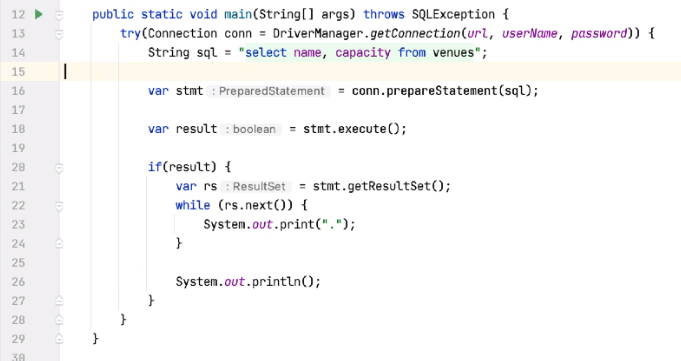
So, again, if I run this, we do indeed update the database. But now just to confirm, the execute returns a Boolean, we check that to make sure we're doing an insert, update or delete, then we can call getUpdateCount to get back the value which tells us how many rows have changed. And this will be true if we're using insert, update or delete, so I won't repeat those demonstrations here. So what about our ResultSet?





So again if I extract this,

and go and grab the code we used to do the query.



So similarly here, rather than calling executeQuery, we can call execute. Again, we don't get back a ResultSet, we get back a Boolean. So let's say var result equals stmt.execute. And then I can say if(result) var rs equals stmt.getResultSet, and then I can iterate around the ResultSet.



So, again, execute in the code, we now get back five rows from the table, as we've inserted a row in the previous example. And again, just to emphasize, when we call execute, we get back a Boolean, we check that result. If it's true, we have a ResultSet, so I can call stmt.getResultSet, and I can then iterate over that ResultSet. So execute is typically used when you want to execute multiple queries at the same time, so a collection of inserts, reads, writes, and deletes, and maybe sending them all to the database in one statement. If you do that, you can execute them and then get back the result for each query that you've used.

=>slides: Pg. 8

So to review what we've just seen, PreparedStatement has three execute type methods. There's a general purpose execute method, and the return type for this is a Boolean. That Boolean is set to true if we execute a SELECT, but is set to false if we've executed an INSERT, an UPDATE or a DELETE. For an executeQuery, the return type is the ResultSet itself; if we use a SELECT, then we return the data. If we call executeUpdate, then the return type of this is an int, and this is used for INSERTs, UPDATEs, and DELETEs, and the int value we get back is the number of rows that have been changed.

# Parameterizing PreparedStatement

=>slides: Pg. 9

So, so far, the SQL statements we've seen inside PreparedStatement have all used hard‑coded values. So the insert we did is use a hard‑coded string with a name and a hard‑coded number for the capacity, the same for the update and for the deletes. But we did say that with PreparedStatements, a PreparedStatement can be parameterized. So the idea is this. We can pass parameters into a query, and those parameters could be, for example, filters inside of a where clause, or they could be values to pass into the insert, update, or delete. When we create the statement string, we represent those parameters by using a question mark inside the string. And we'll see an example in a moment. Parameters we want to pass in have a type. So they could be an int parameter, a string parameter, for example. And before we execute the query, we have to set the value of the parameter, and we set the value dependent on its type. So we can set the value of the parameter based on its position. So each question mark in our query string has a position, but those positions are 1‑based, not 0. So the first question mark in the string is number one, the second question mark is number two, and so on and so forth.

=>slides: Pg. 10

So we can do something like this. So here we're saying, insert into venues. We want to insert a name and a capacity value. But now, rather than hard coding the values, we're using question marks inside the SQL string. So we define the statement, so that's insert into venues (name, Capacity) values are unknown. We then create the PreparedStatement, and this is in a try‑with‑resources block, so we're doing conn.prepareStatement, passing in the sql. And then we pass in the values we want to set. So the first value is a string value, so we call it setString on the PreparedStatement. It's the first value, so its position is 1, and the value we're setting is just The Coliseum. The second value is an integer value. Its position is number 2, and the value you want to set for this is 300. So we set both of those values, and we then execute the statement. And again, the execute will return the number of rows that have been updated based on this insert.

=>slides: Pg. 11

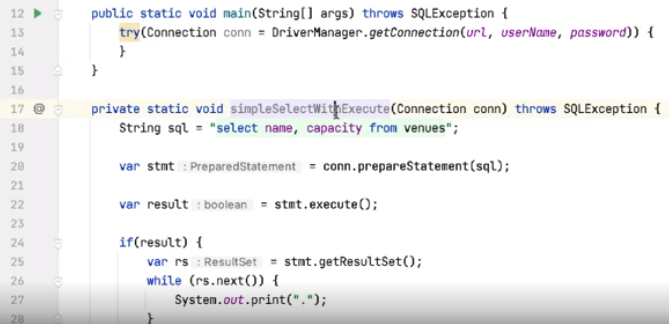
So parameters are used for all the query types, so whether it's create, read, update, or delete. They are 1‑based, so the first parameter is number 1, not number 0, so 1, 2, 3, 4, etc. You don't have to set the parameters in any specific order. So if you wanted to, you could set parameter three before parameter one. But all the values have to be set; otherwise, the query will fail.

=>slides: Pg. 12

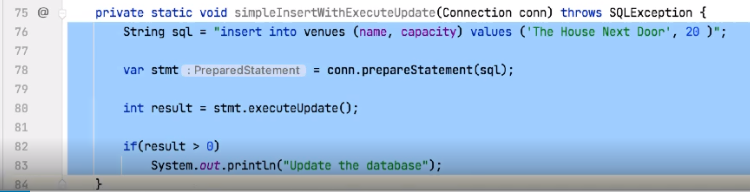
You can also use the PreparedStatement multiple times. So here again, we're doing an insert into venues with the name and a capacity and two values. We then create the PreparedStatement, and we set the value of both variables. So we set the first parameter, number 1, to be The Colosseum, and we set the value of the second parameter to be 300. We then execute the update. So this will insert a value into our table. What I can then do is reuse the PreparedStatement. So I call setString on parameter one, so I change the name of parameter one to be The Garden, and then I call executeUpdate again. So this will insert a new row into the table, but this time it will have the name of The Garden and a capacity of 300. So we're inserting two rows, but in between the inserts we are only changing the name. So let's take a look at some code and see how we do this.

# Demonstration - Parameterizing PreparedStatement

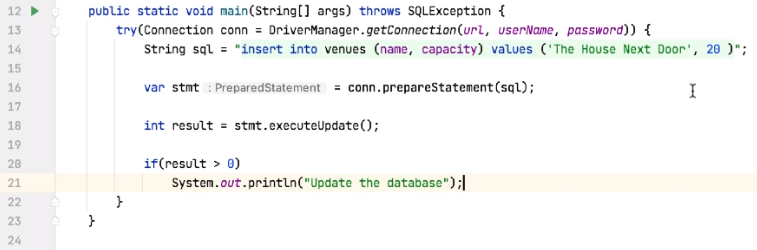
=>slides: Pg. 13



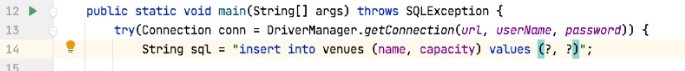
So I've extracted the previous code into this method, simpleSelectWithExecute. So what we'll do now is take a look at how we parameterize some of these queries.



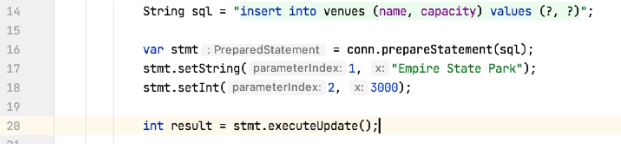
So the first thing I want to do is to do the insert. So let me just grab this code.



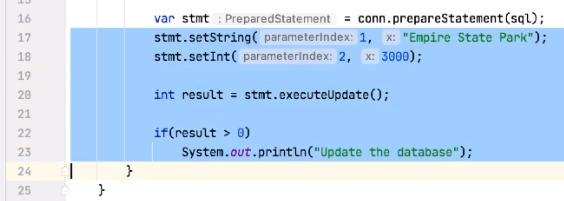
So this is the insert code we saw a little earlier. So we're inserting into venues, we ought to insert the name and the capacity values, and the values we're setting with The House Next Door with the capacity of 20.



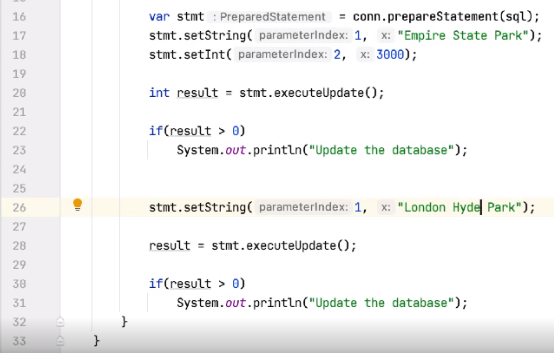
What I'd like to do is to parameterize these two values. And we say we want to add parameters by using question marks, so I'm telling prepareStatement to expect two parameters to be added here. Once we have the statement, I can say stmt.set.



Now, remember, I can set these parameters in any order I want to, but let's set them in numerical order first of all. So the first parameter type is a string, so I say setString. Parameter number 1, remember, not 0, and we give this a value. Let's call this Empire State Park, let's say. I then want to set the second value. So, stmt.setInt this time. So the second parameter is an integer. Give it the number 2, that's the parameter index, and then the value. Let's set the capacity of this to be 3000. And I can then call stmt.executeUpdate to update the database. We run the code. Sure enough, the database has been updated. So what happens if I switch these two? So now we set parameter 2 before we set parameter 1. Run the code. We still update the database. Remember, I can also reuse the values.



So let me go and grab this block of code and just duplicate it, correct the compiler error.

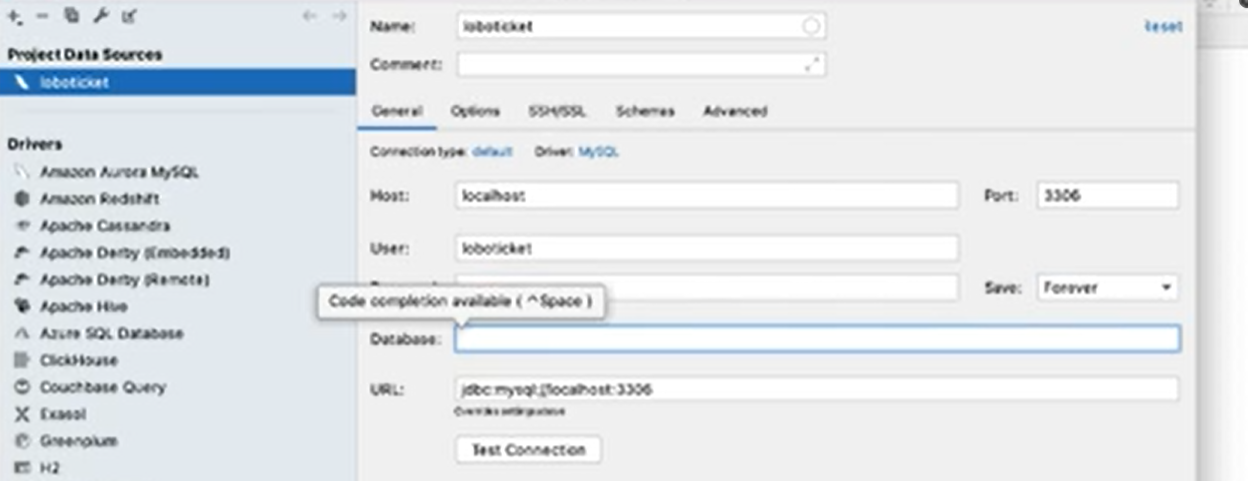


But now I'm not going to set the capacity for the second time. So we change our first parameter to be London Hyde Park and leave the second parameter alone. So we're inserting Empire State Park for the third time, so there will now be three of these in the database, which obviously may cause issues. It has a capacity of 3000. We then set parameter 1 to be London Hyde Park, but we don't change the capacity, and then we call executeUpdate again. So let's see what happens.

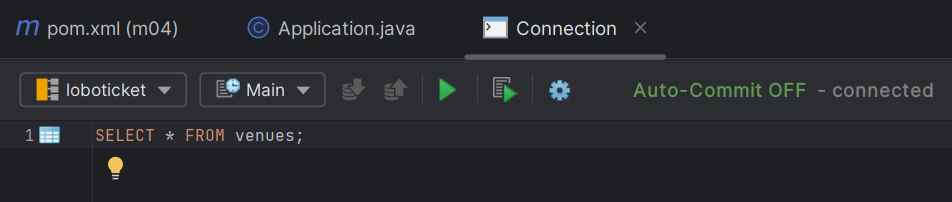


So we update the database twice. So just to prove that, let me go and look and see what's in the database.

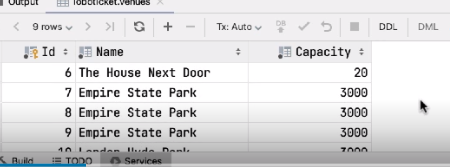




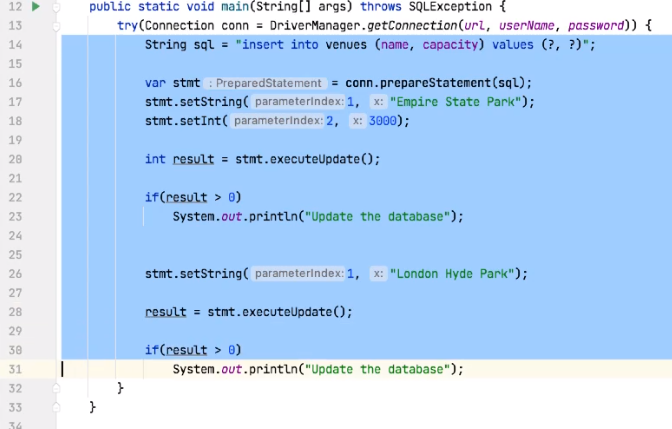
I can do that from inside IntelliJ IDEA. So on the database window here, I click on the plus button and go to Data Source and say MySQL, and give this a name. I will just call it loboticket. My user, remember, was loboticket, and the password was p4ssw0rd. And the database we're connecting to is also called loboticket. Test the connection. That's all good. Say OK. And this opens up a console window for me.



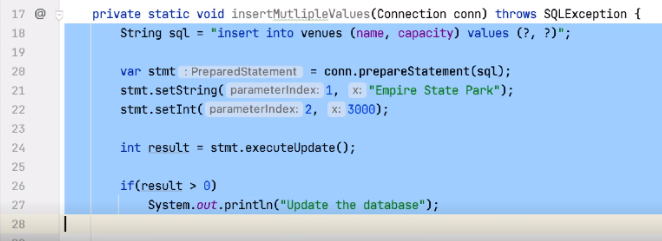
So from here, I can do a select\* from venues.



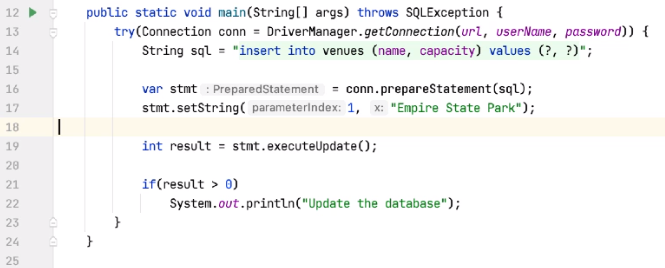
Run the query, and if I scroll down here, we can see we have Empire State Park inserted three times because I've run the query three times. We also have London Hyde Park inserted as well. So for these two inserts, first one, I set both the name and the capacity. For the second one, I only set the name. The capacity was still held inside the PreparedStatement.



So let me just save this moment.



So we say, insertMultipleValues and then grab this code again.



So what happens if I don't set a parameter? So we set parameter 1, but we don't set parameter 2, and then try and run the code. So if I do that, we get an exception.



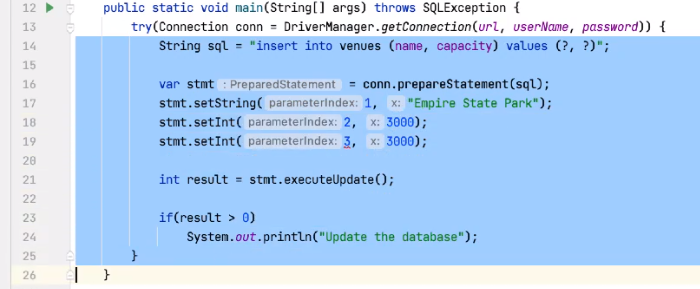
So we get a SQL exception, and the text of the exception is No value specified for parameter 2. So it's not even possible to execute the query unless we've set all the parameters. So you put that back.



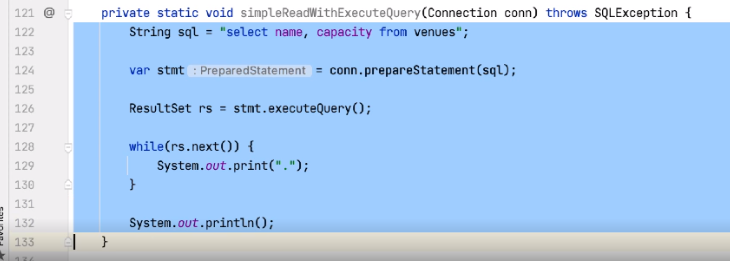
So suppose I try and set a spurious parameter 3. So,STS is smart enough to pick this up for me, first of all, so it knows we only have two parameters specified in the query. That's IDE‑specific. This will compile,



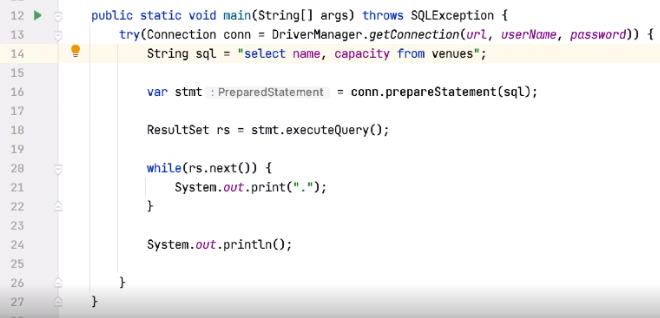
and if I try and run this code, again, we get a SQL exception, but this time it says 3 is greater than the number of parameters, which is 2. So you could try and execute the query with too few parameters or too many parameters, but that will always fail with a SQL exception.



So again, let me save this. So what about SELECT?



So this code was the first code we did, we used for SELECT, so let me go and grab that and

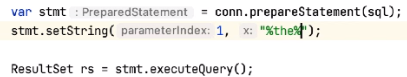


paste that in here. So SELECT can be parameterized as well. So we were selecting all the rows from the venues table.



Let's filter this. So let's say where name like question mark.

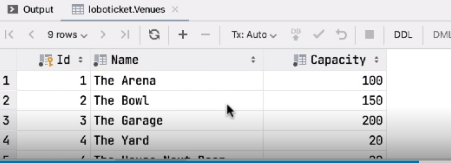


So if I execute this now, sure enough, we get an exception. There's no value specified for parameter 1. 

So after I prepare the statement, I can say stmt.setString, it's parameter 1, and then let's set the value to %the%. So, any venue whose name contains the characters t, h, or e.



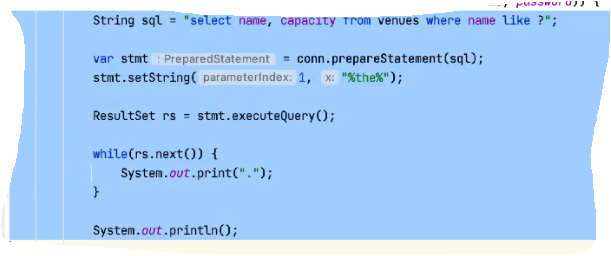
And if I go to the console for this and execute the query,



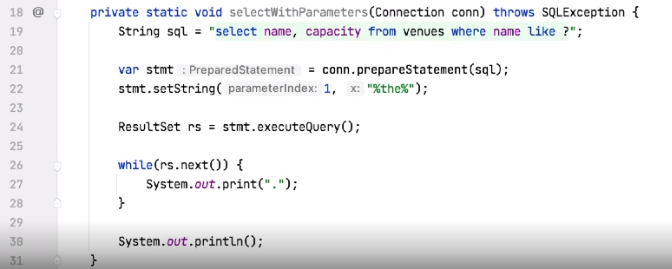
we have The Arena, The Bowl, The Garage, The Yard, and The House Next Door. So there should be five values that come back.



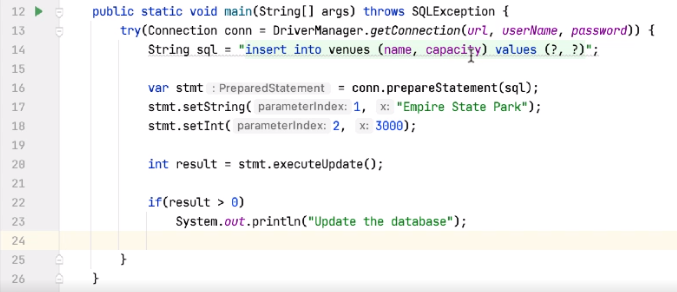
So if I execute the code, sure enough, we get 1, 2, 3, 4, 5 dots. Now, we haven't looked at the resultSet in detail yet to see how we get the data out; we'll see that in the next module. So again, just like INSERTs, UPDATEs, and DELETEs, we can parameterize queries, and we can add as many parameters as we need to here. So, I just wanted to look at one more thing.



Let me go and extract this into a method.



We'll call that selectWithParameters.



And I'm going to grab the INSERT statements. So this code here, we're inserting into venues, we're giving it a name and a capacity. We're setting two values with a string and an integer.



We could also, rather than calling setString, call setObject. Now, we could do that for both of these. If we use setObject, this is a sort of general purpose method where we can pass in any object type. Yes, at any type of at all. So if we're passing a string for Empire State Park, we're passing an integer value of 3000 for the capacity. So again, if I run this, the update works.



Again, if we look at the database console and run the query,



then the last entry here is the one we just inserted. And sure enough, it's added the name and the capacity correctly. So setObject can be useful if we're writing a helper functionality, and we're never quite sure of the type that's being passed to us.

=>slides: Pg. 14

So PreparedStatement has set methods of each data type you might want to add into a row and column in the database. So we've seen setString and setInt, but there's also setBoolean, which has a Boolean type, setDouble, Double, SetLong, which has a Long, and finally, setObject. So with setObject, we can add any type we need to, so it's like a general purpose value we can pass in here. On the right‑hand side of this table, I've also shown the database types that sort of correspond to these Java types. Now, be aware that for the examination, you don't need to worry about these types. You might be examined on the method and the parameter type, but not the database type. They're just shown here for completeness sake.

# Summary

=>slides: Pg. 15

So what have we seen in this module? We've seen how to use a PreparedStatement to execute SQL. We've also seen that this PreparedStatement can be parameterized and those parameters are specified with question marks in the SQL string. These parameters, remember, are 1 based, not 0 based. So when I call the set methods, we pass the values 1, 2, 3, etc., not 0, 1, 2, 3. When we set the values, we set them with a specific type, so setString, setInt, setBoolean, and there are other types that we can use as well. We also saw that we can execute the statement multiple times. So I can execute it multiple times with the same values, or I could change the values in between and then re‑execute the statement. We've also seen that there are different methods depending on the query. So we could call executeUpdate to do inserts, updates, or deletes, executeQuery to execute select statements, or there's a general purpose executeStatement, which is often used when we want to do multiple things to the database.

=>slides: Pg. 16

Okay, so we've seen how to execute prepared statements, what we haven't yet seen is how to work with the data that comes back from a PreparedStatement in the forms of a ResultSet, and we'll see that next.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*