6. Working with Data from a PreparedStatement

Contents

[1. Introduction 1](#_Toc14984)

[2. JDBC ResultSet 1](#_Toc15157)

[3. Demonstration - Using ResultSet 3](#_Toc11025)

[4. Using JDBC's GetObject 7](#_Toc14274)

[5. Demonstration - Using GetObject 7](#_Toc10000)

[6. Binding JDBC Parameters 9](#_Toc23154)

[7. Demonstration - Binding JDBC Parameters 9](#_Toc14362)

[8. Summary 10](#_Toc1192)

# 1. Introduction

=>slides: Pg. 1-2

So in the previous module, we saw how to create PreparedStatement and to execute it, either to change data in the database or to bring back data from the database. What we didn't see was how to process that data, and we'll see that here. So we're going to take a look at how we process data using a ResultSet. We'll also look at modifying data in the database, and we'll look at how we can set null values when we send data to the database. And finally, to get data from the ResultSet, we can get data by column position and we can also get it by column name, and we'll see how we do that in this module as well..

# JDBC ResultSet

=>slides: Pg. 3

So to get a ResultSet, we execute a SELECT query either by calling execute or executeQuery. If data is returned, it's returned in this ResultSet object. So the ResultSet has a cursor. Initially, that cursor is pointing before the first row in the ResultSet. So we call next on the ResultSet to move the cursor. The call to next returns true if there's a row to process; it returns false otherwise. So either there is no data in the ResultSet, or we've gone past the last row that was returned. Once we've moved the cursor to the row, we can then get the values from the columns within the rows.

=>slides: Pg. 4

So it's something like this. So we have a SELECT query, so we're going to select name and capacity from the venues table. We prepare the statement and execute the query, and then we call rs.next, where RS is the ResultSet. Next moves the cursor. If rs.next returns true, we have a row to process. We can then get the data from that row. So where the PreparedStatement has SET methods to set data, the ResultSet has GET methods to get data. And again these are type‑based. So, rs.getString will return me the string value for the name column, rs.getInt returns me the int value for the capacity column. So here we're using the names of the columns to get the data, not the IDs of the columns to get the data, so just one and two. And then once we have this data, we can use it. So here we're just printing out those values.

=>slides: Pg. 5

So the ResultSet cursor works something like this. So this is the data we've got back from our database. We're getting back three columns, an Id column, a Name column, and a Capacity column.

=>slides: Pg. 6

When we get this data back initially, the initial cursor position will be here if you like. So, essentially pointing to some row before the first row in the data.

=>slides: Pg. 7

When we call rs.next, it will return true, and I'll be referencing the first row in the ResultSet,

=>slides: Pg. 8

.next again returns true, and references the second row,

=>slides: Pg. 9

returns true for a third time and references the third row,

=>slides: Pg. 10

but when we call rs.next again, it will now return false. And basically now we are beyond the end of the ResultSet.

=>slides: Pg. 11

Now, as well as getting the data by name from the column, you can also get the data by column number. So here we're seeing rs.getString(1) and rs.getInt(2). So there are analogous methods for getString and getInt that take either a string, which is the column name, or an integer, which is the column number.

=>slides: Pg. 12

So occasionally your ResultSet only contains one row. So, for example, you may be using a count or an aggregate function that simply returns one value. In that case, rather than calling while rs.next, you can simply call if rs.next.

=>slides: Pg. 13

And we can see an example here. So, the query this time is a count, so select count(\*) from venues. We create our prepared statement, we then execute it, and then we call if(rs.next). So that will move the cursor and check the value is true. And then if it is, we can get back the value. In this case, we call rs.getInt(1). We might not know the name of the column now, but we definitely know that column 1 contains that integer value. And again, once we have that, we can print out the value.

=>slides: Pg. 14

We're showing the same example here, but this time we are naming the column. So we use select count(\*) as count, so that names this aggregate column as count, and then we can call rs.getInt count to get back the value. So this code is essentially the same as the code on the previous slide, but using the column named version of the GET method rather, than the column index version of that method. Either one will work equally as well.

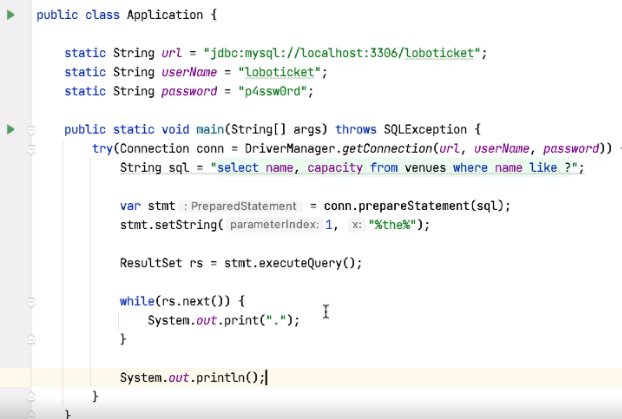
=>slides: Pg. 15

Okay, so some dos and dont's. Always check the return value of rs.next to make sure we have data to process. You can't access any data before you call rs.next, so you must always call this at least once. Make sure you use a valid column number if you're using the GET methods that take an integer value. If you use an invalid column number, then it will throw a SQL exception, and this is the same as the SET methods. We tried calling set string with an invalid parameter number for the prepared statement, then we saw it throws a SQL exception. Similarly with names. So make sure you use a valid column name, and again, if you don't, it will throw a SQL exception. Okay, so let's go take a look at some code and see how we do this.

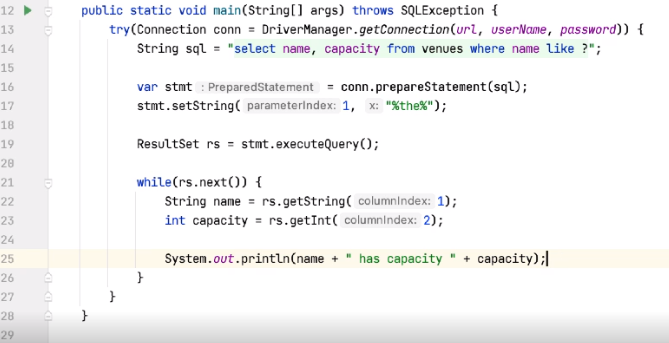
# Demonstration - Using ResultSet

=>slides: Pg. 16

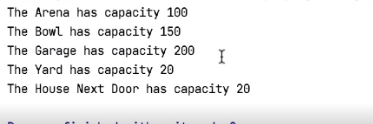




So earlier, we saw this code where we're selecting data from the venues table. We have a parameter, so the question mark, we call conn.prepareStatement, we set that parameter, we select anything from the venue where the name contains the characters "the," then execute of the query, we get back a ResultSet, we call rs.next, and then we print out a full stop. So now we can see how to actually get at the data in this ResultSet.



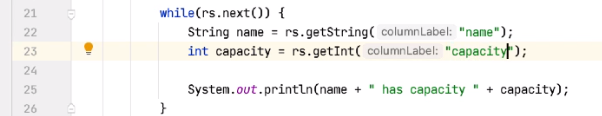
So we're getting back two columns here, name and capacity. The first one is a string, and the second one is an integer. So here I can say String name = rs.get, and let's say getString, so this is going to be the first value we get back the name. We can use the column index mechanism for this. And just like setting parameters, getting parameters starts at index 1, so I can just say 1. I can then say int capacity = rs.getInt, and it's time for column number 2. And when we have that, let's go and print out this information. And let's say something like name has capacity capacity. So we don't need the System.out.println at the end now, as this is now printing things on a new line each time.



So again, if we go and execute this, we get five lines back. So the Arena, the Bowl, the Garage, the Yard, the House, and the different capacities for each of these arenas.

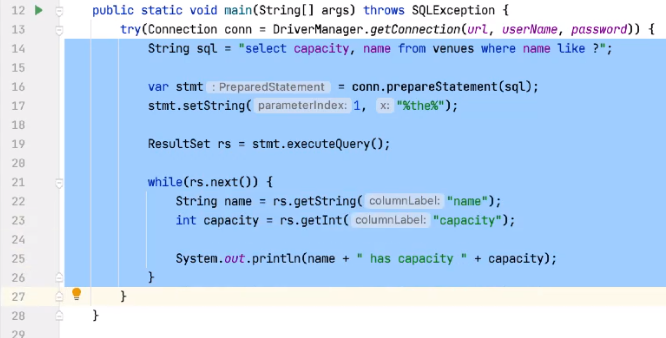


So again, let me go and save this away, and I'll just grab this code again to reuse it. So as well as using column IDs, we can use column names.

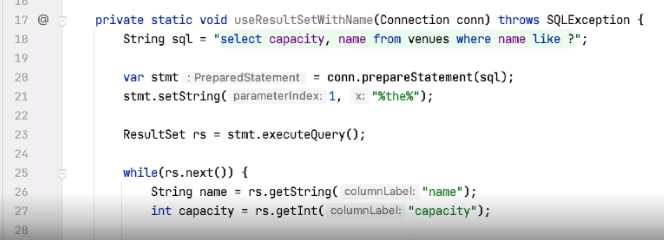


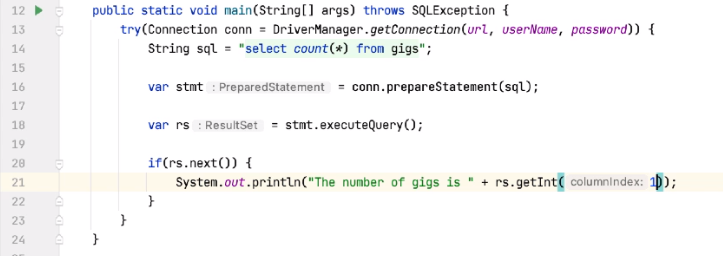
So I can say I want the value of the name column and I want the value of the capacity column. And again, running this code, we should see the same output. And indeed, we do. This is generally a better idea.



There's nothing stopping us changing the order of these, for example, inside the SELECT, but using the column names will still work. But if we use the column indices, we'd have to change the index number. 

And again, let me just go and put this away.

So useResultSetWithName. So we showed on the slides that if we get back a single row, we can just use an if statement rather than a while statement. So how about something like this?



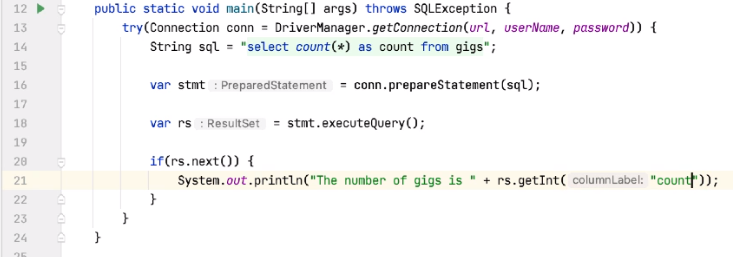
Let's say String sql = select count(\*) from gigs. So how many gigs do we have? And then I can say var stmt = conn.prepareStatement, passing it the SQL. There's no parameters here. That's fine. But now I can say var rs = stmt.executeQuery. And then, if(rs.next) System.out.println, the number of gigs is, and let's say, rs.getInt. And this time we'll use a columnIndex, so a columnIndex of 1, run the code.



So there are 15 gigs. Now, if we wanted to use a column name here, we could, but we'd need to change the query.



So, having saved that previous query away,



what we can say in our SQL is select count(\*) as count, and that will name the column for us. And then here, rather than using the column index, we can use the column name of count.



And again, running the code, and sure enough, this still works. So this shows us a couple of things. Obviously within the SQL, we can use standard SQL, as this is just sent to the database. And then within the ResultSet, we're only getting back one row here, so we'll quite happily say if(rs.next) just to get that first row back.

# Using JDBC's GetObject

=>slides: Pg. 17

So we've seen that with the resultSet that we have a set of GET methods that we can use to extract the values from the data that's returned to us. We saw getString and getInt, for example, but there's also getBoolean, getDouble, and getLong. And similarly to the SetObject on the prepared statement, there's a GetObject. And again, this can be used as a general purpose accessor method to this data, although using it is slightly more complex.

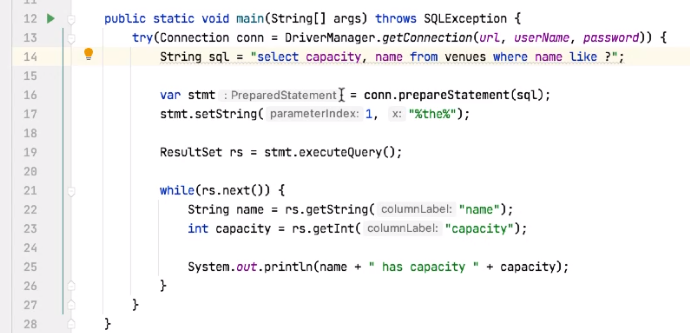
=>slides: Pg. 18

So if we use GetObject, we might do something like this. So here we have a while(rs.next)loop. We use rs.getObject on both the name column and the capacity column. But then to turn these values into the actual field values, we're using instanceof to check the type of these things. So we're checking to see if the name field is a string, and if we are, we're casting nameField to be of type string and storing it in the name variable, and similarly, we're checking to see if the capacity field is an integer, and if it is, we're casting that to be of type int and storing it in the capacity variable.

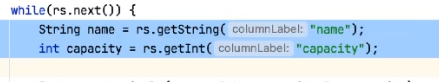
# Demonstration - Using GetObject

=>slides: Pg. 19

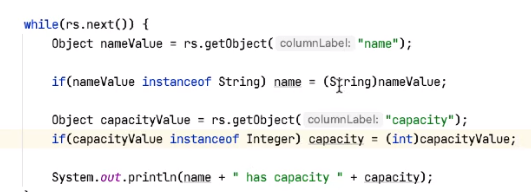
So let's see how we might do that.



So this is the select code that we had earlier. And here we're using getString and getInt to get the values from the columns.



And as we saw on the slides, instead of this, we can use getObject to get these values. And that's going to make the code slightly more complex, so let's see how we do that.



The first thing I want to do is to declare two variables to hold these values. And as we'll see in a moment here, rather than having name of type String, we'll declare a variable of type Object called nameValue, and rather than getString, we'll call getObject. And the same with capacity. So rather than having an int here, we'll have Object, and again, we'll call this capacityValue. And again, rather than having getInt, we can have getObject. So once we have these two objects, we have to extract the value from them, but first of all, we have to check that they are the right types. So to do that, we can say something like if nameValue instanceof String, so if value instanceof String, then cast the name value to be a String into the name variable. And we can do the same thing with capacityValue. So if capacityValue instanceof\_Integer, which will be the java.lang type, we can say capacity = int cast of capacityValue. Then once we have the two values extracted, we can use them in the same way. So having done that, if I run this code, we get out the same data. So reading with a capacity of 100, and so on and so forth. So this is a more verbose way of getting the data and also has worse performance than as getting the strings and the integers, for example, directly from the data. However, this does give us a lot of flexibility in our code.

# Binding JDBC Parameters

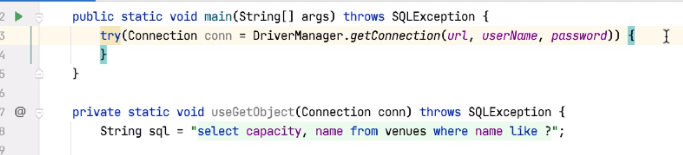
=>slides: Pg. 20

So as we've seen previously, we can also bind parameters in a SELECT, and we'll show some more examples of this in a moment.

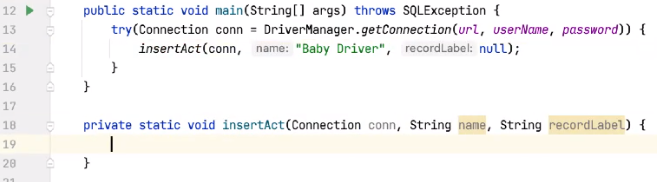
=>slides: Pg. 21  
So the code to do that looks something like this. Here we create the query, we create the PreparedStatement, and then we call setInt on the PreparedStatement to set the value. Noticing in the query we have a question mark, that's our placeholder, and we're setting the capacity here to be greater than 120. We can then execute the query, and then do the same thing with ResultSet to check the cursor, move, get the values, and use those values.

=>slides: Pg. 22  
Now another thing we might want to do is to bind nulls. So if our column in the database is able to hold null values, we need a way of setting that null value. So here the way we are doing that is that we're checking to see if the value passed in is a null, and if it is, rather then calling setString to set the value of that parameter to pass into the SQL statement, we call setNull on that parameter. When we call setNull, we also have to specify the type of parameter that we'd like to set. So Types.CHAR basically says this is a string parameter.

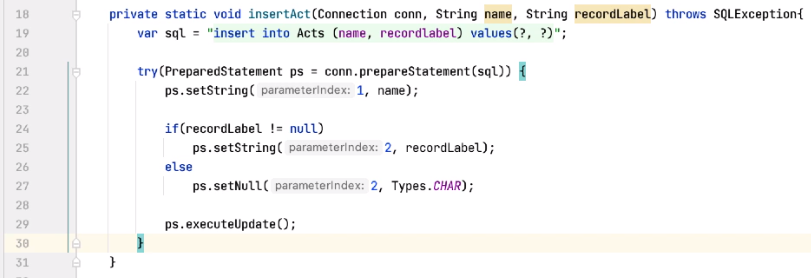
# Demonstration - Binding JDBC Parameters

=>slides: Pg. 23  


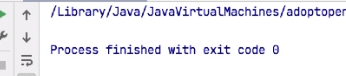
So let's go and take a look and see how we do that. So, what I'm going to do is insert an act into my database, act of a name, and they have an optional recordLabel.



So let's write a method here called insertAct, and the insertAct will take the connection. We're going to pass it the name of the act int to insert, and let's call this Baby Driver, say, and then we're going to pass it a value for the recordLabel,, and that value could be null. So let me generate this method. So you have a method called insertAct that takes a connection, the name of the act, and the recordLabel.



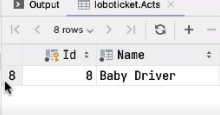
So the SQL for this method looks like this. We'll do an insert into Acts, we'll pass in the name, and a recordLabel, and we need to specify those values. So inside here, we're going to use a PreparedStatement. So in all the previous examples we've seen, we've wrapped our connection in a try‑with‑resources block, but inside that connection, we've used a PreparedStatement, and we've used a ResultSet. We've said that when the connection gets closed, the PreparedStatement gets closed, and if the PreparedStatement gets closed, it closes any associated ResultSets. Here, we'd like to manage the PreparedStatement independently of the connection, so I can wrap my PreparedStatement inside a try‑with‑resources. So we can say try, but inside here, we say PreparedStatement, ps = conn.prepareStatement, and we can pass it the SQL. PrepareStatement, prepareStatement could throw a SQL exception, so we need to handle that in some way, and we're going to handle it just by adding the throws clause to the method. So once we have the PreparedStatement, we need to set these two values. So I can say ps.setString, setting the name column, which is parameter 1, and giving it the value, which is the name of the act. But for the second parameter, this could be null. So I'm going to say if recordLabel is not equal to null, we do a ps.setString of parameter 2, and pass it the recordLabel, but if recordLabel is null, then I want to say ps.setNull on parameter 2 and specify the type. So I can say Types.CHAR, and types is part of the java.sql namespace.



So what happens when I run this? So we create a Connection inside our try‑with‑resources, we call insertAct, we set up the SQL, we call PreparedStatement inside a try‑with‑resources, we set the parameters, then we have to call ps.executeUpdate to execute that statement. Once you've done that, the PreparedStatement is closed, we return from this call, and then the connection is closed. So if I execute this code, we see no exceptions, so I'm assuming it's worked.



If I load up the Database console, and do a select \* from Acts,



then in here we see the Baby Driver act, and sure enough, the recordLabel value has been set to null. =>slides: Pg. 24

So just another note on closing resources. We've just seen that we can wrap connections and prepared statements in a try‑with‑resources block, and that closes the connection and the PreparedStatement, but remember the result sets must also be closed, and we could use try‑with‑resources to do that as well; however, remember that closing the PreparedStatement will also close any associated result set.

# Summary

=>slides: Pg. 25

So what have we just seen? We've seen that when executing queries with a PreparedStatement, this returns a ResultSet. We also talked about the fact that ResultSets have a cursor. We must move the cursor before accessing the data within the ResultSet. When we access the data, we access it by column, and that's either by name or by index. And remember that the index is 1‑based, so we call a getString, getInt, etc., but using 1‑based indices.

=>slides: Pg. 26

So next up we'll take a look at working with CallableStatements, so how do we call stored procedures within the database?  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*