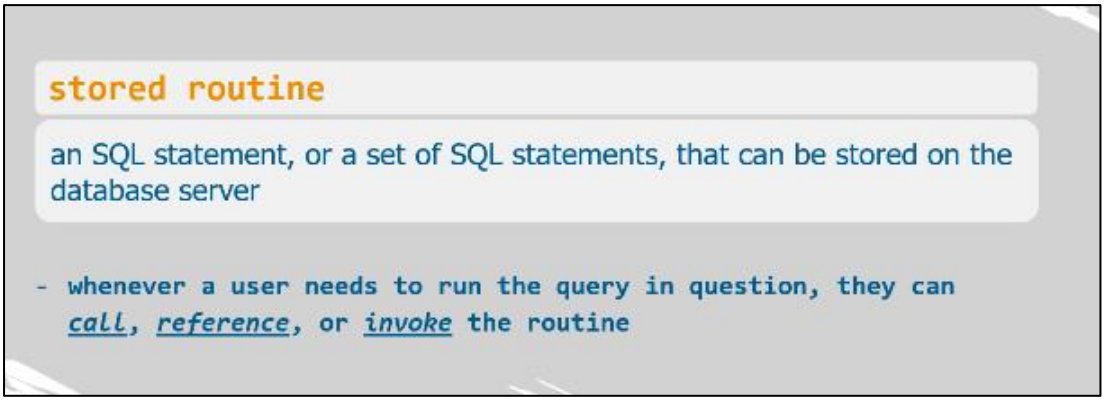
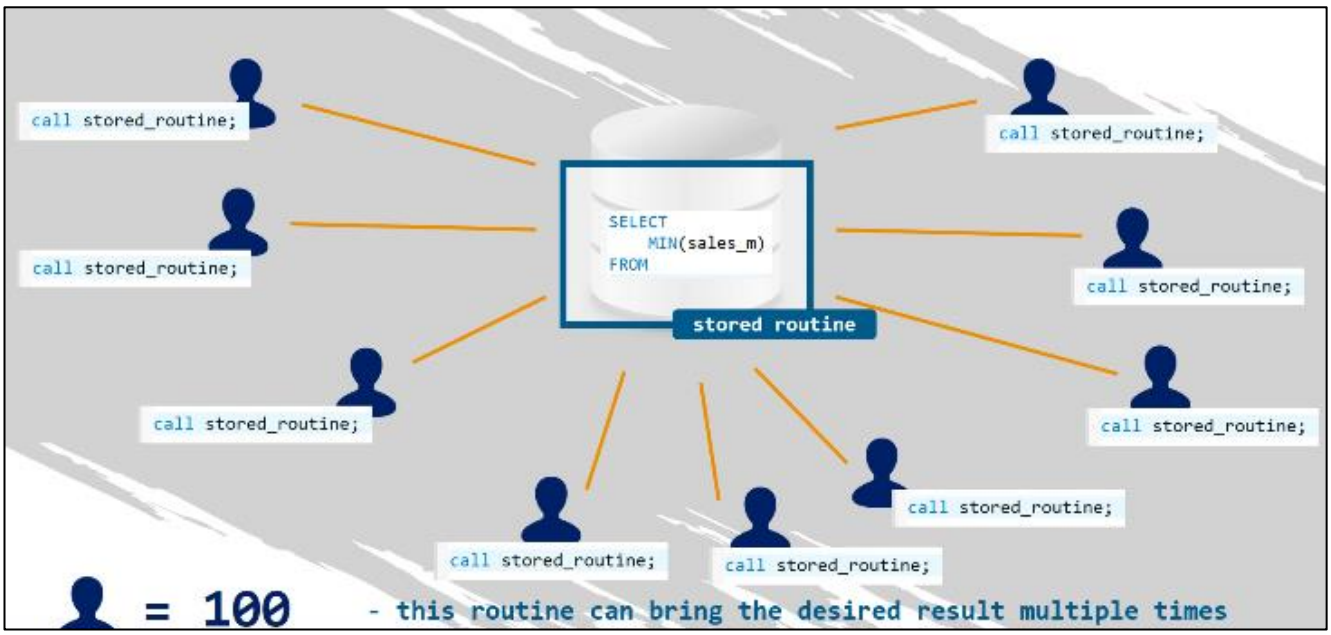
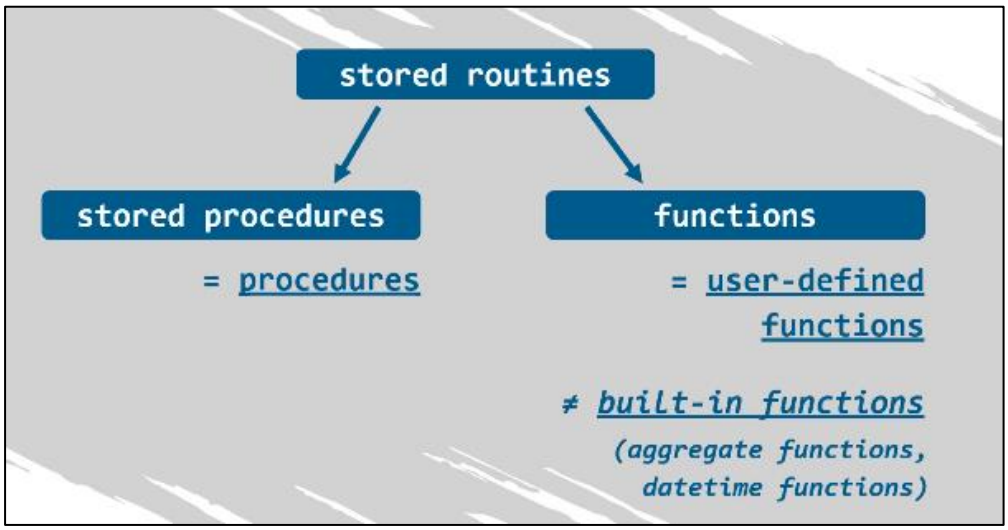
Stored Routines

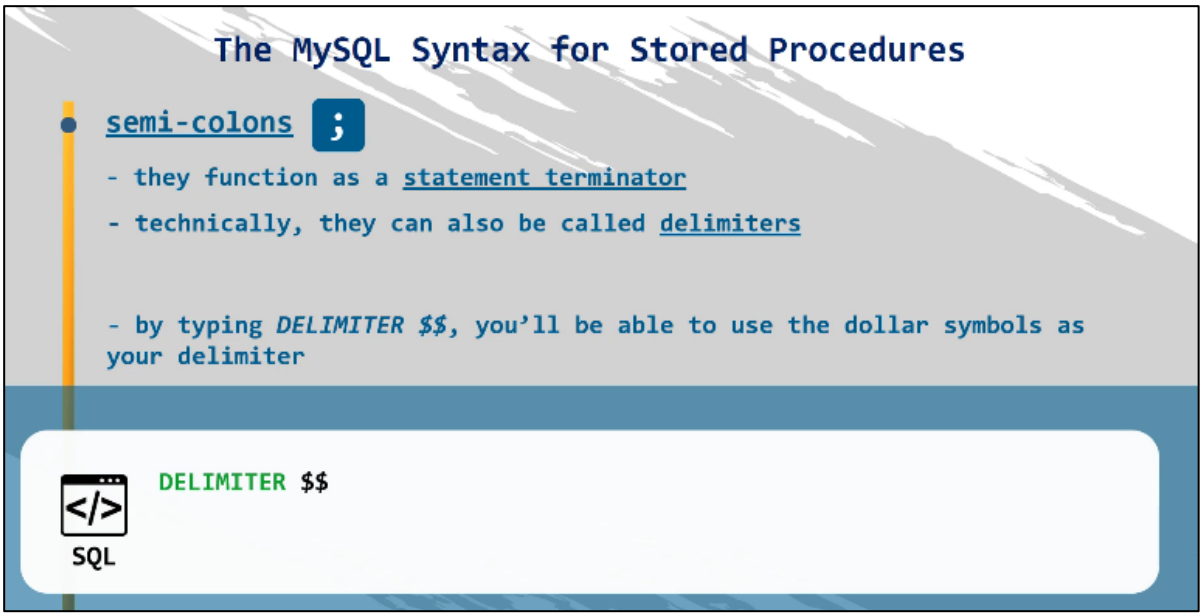


We essentially use this when there is a particular query we run multiple times a day – so instead of constantly having to rewrite it, we just call it instead.



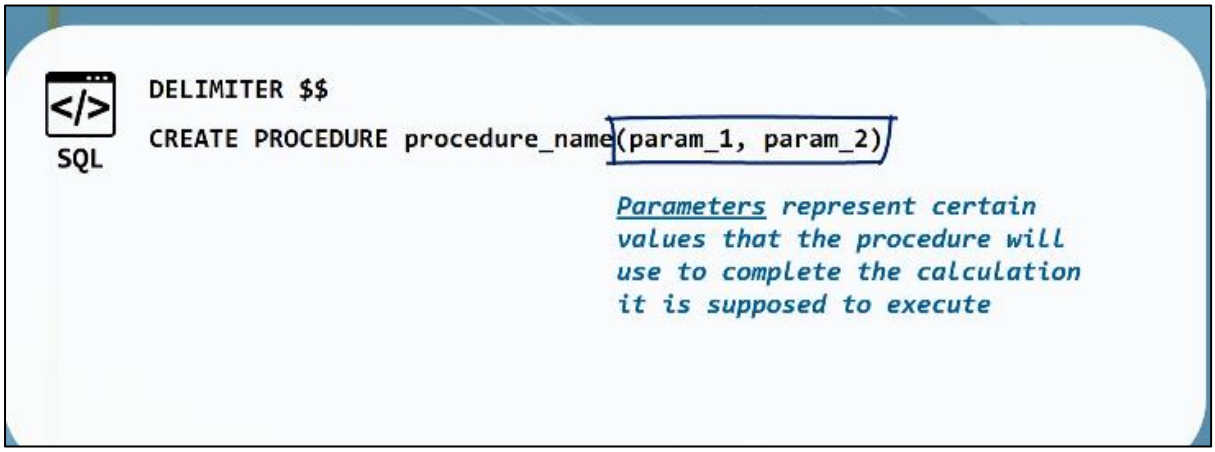
Lecturer notes: We can have two types of stored routines: stored procedures and functions. Funny enough, all procedures are stored. Therefore, we could simply refer to them as procedures if we prefer. Functions, though, it can be of various types; they could be programmed manually, and in that case, they will act like stored routines. These are the user-defined functions. Alternatively, we can have built-in functions in MySQL, which means that these functions have already been defined inside MySQL. We've already used some of them, like the aggregate functions or the date-time functions. So when we say functions, only the context will clarify which type of functions we are talking about.





Lecturer notes: Think of how semicolons are used in SQL. Initially, we established that the function was a statement terminator, but technically, they can also be called delimiters. And by typing 'delimiter' and the dollar symbol two times, we'll be able to use the dollar symbol as our delimiter. The semicolon isn't your dilemma anymore.

Well, think of the long sheets of code we can have in the SQL editor. There, every query is terminated by a semicolon, right? As we know, the SQL engine will run only the first of the statements in our procedure, and we'll move on to the next query that is beyond the procedure. It is not going to read the code after the first semicolon. To avoid this problem, we need a temporary delimiter different from the standard semicolon. There are various symbols you can use, a double dollar sign or a double forward slash, for instance. It doesn't really matter which one you choose. I will opt for the double dollar symbol.

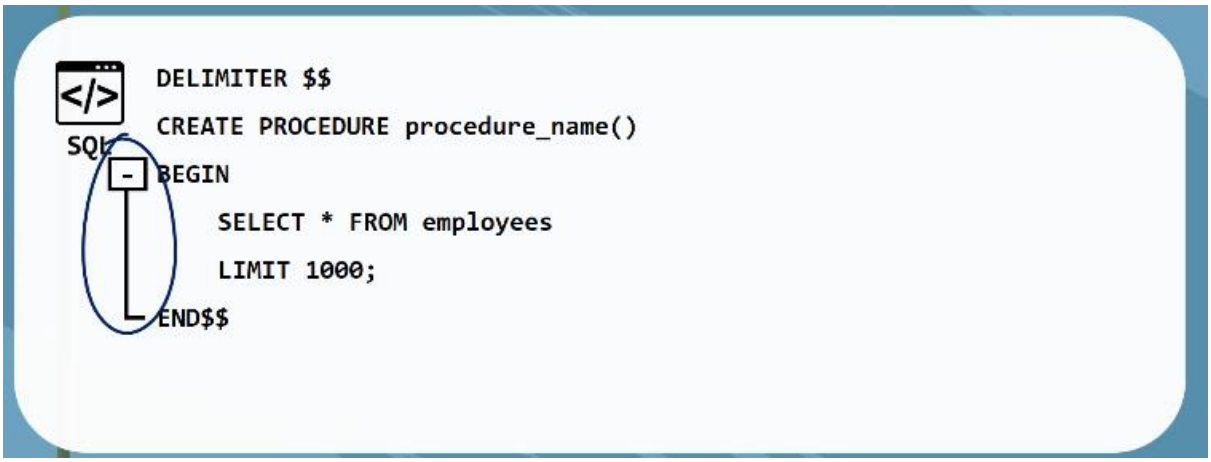


Lecturer notes: We must then write, 'create procedure' and attach the name we would like to assign to it. Next to the name, remember that we **must** always **open** and **close parentheses**. They are inherent to the syntax for creating a procedure because, within these parentheses, you would typically insert parameters. What do parameters do? They represent certain values that the procedure will use to complete the calculation it is supposed to execute. However, please remember that a procedure can be created without parameters too. Nevertheless, the parentheses must always be attached to its name. Otherwise, MySQL will display an error message.

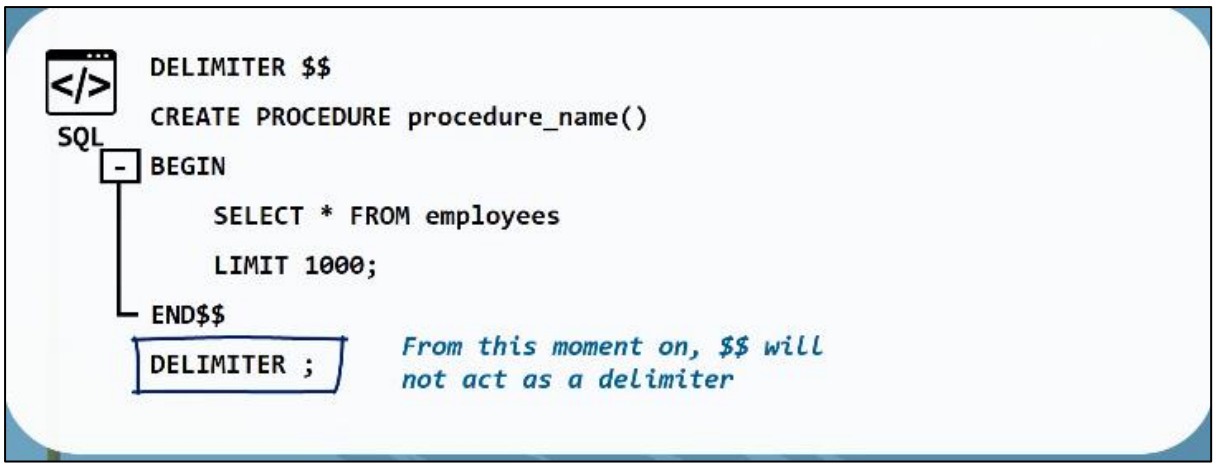
```
CREATE PROCEDURE procedure_name()
```

What follows is the body of the procedure. It is always enclosed between the keyword begin, the keyword end, and the temporary delimiter, which in our case is a double dollar sign. My SQL

Workbench will display a black vertical line along the left side of the body of the procedure with a tiny box on top. By clicking on the minus or the plus sign located within this box, you can hide or expand the code of the body.

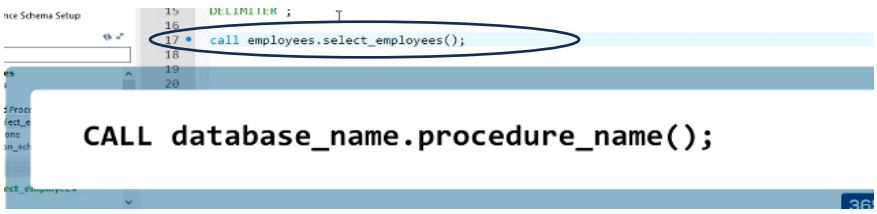


Lecturer notes: The body of the procedure is composed of a query, and this query is the reason we are creating the entire procedure in the first place. It will be placed between the begin and end keywords. More importantly, however, at the end of this query, we'll have the usual delimiter: the semicolon, not the double dollar sign. How come? Well, if we use the delimiter again, the creation of the procedure will stop here, and MySQL will show an error.



Lecturer notes: Finally, we can't forget to reset the delimiter to the classical semicolon symbol. If you forget to do that, you risk making the opposite mistake; not running any of the code succeeding the line where we are calling the procedure. And from this moment on, the double dollar sign will not act as a delimiter. Once again, the semicolon will have this role.

At this stage, we can talk about invoking the procedure. Essentially, there are three primary methods to achieve this. The first one involves the following syntax. Call the name of the database the stored procedure is applied to:



When executed, this line of code will deliver the first 1,000 rows from the employee's table, as outlined in the procedure's body of code.

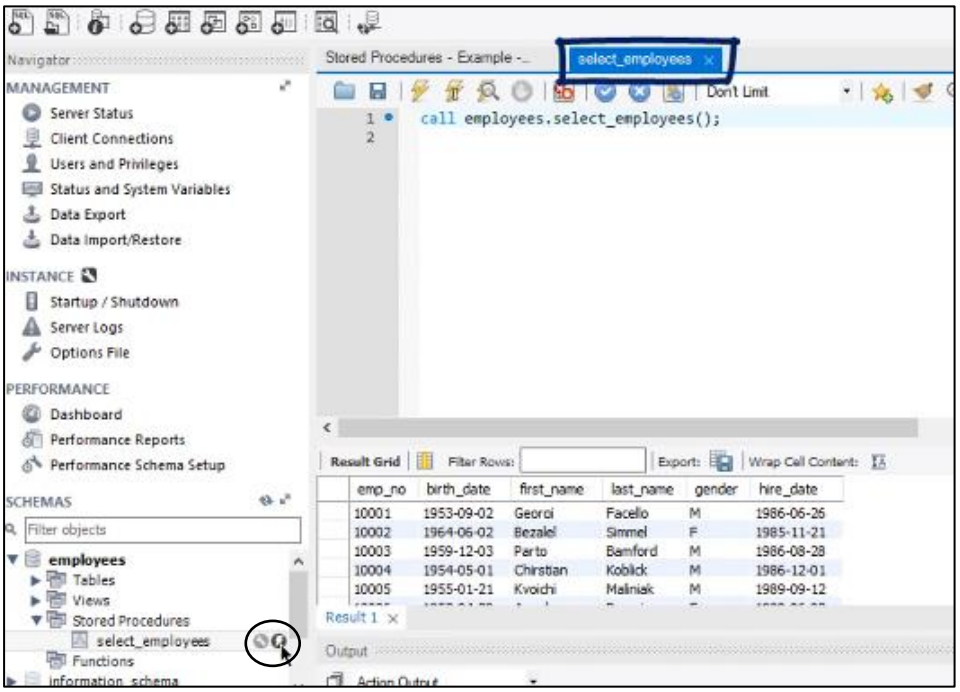
```
269    DELIMITER // #Refer back to our word notes to understand why we use dolar signs
270    CREATE PROCEDURE select_employees()
271    BEGIN
272
273    SELECT *
274    FROM employees
275    LIMIT 1000;
276
277    END //
278    DELIMITER ; #SQL is a weirdo, it wont explicitly tell us our procedure is working like it does with everything else because it's a procedure
279
280    SELECT 1; # A way to confirm our procedure works
```

The second way to invoke "select employees" would be to take advantage of the fact that we have already selected employees as our default database: "USE employees;". In other words, we can skip the database name part and call the procedure name directly.

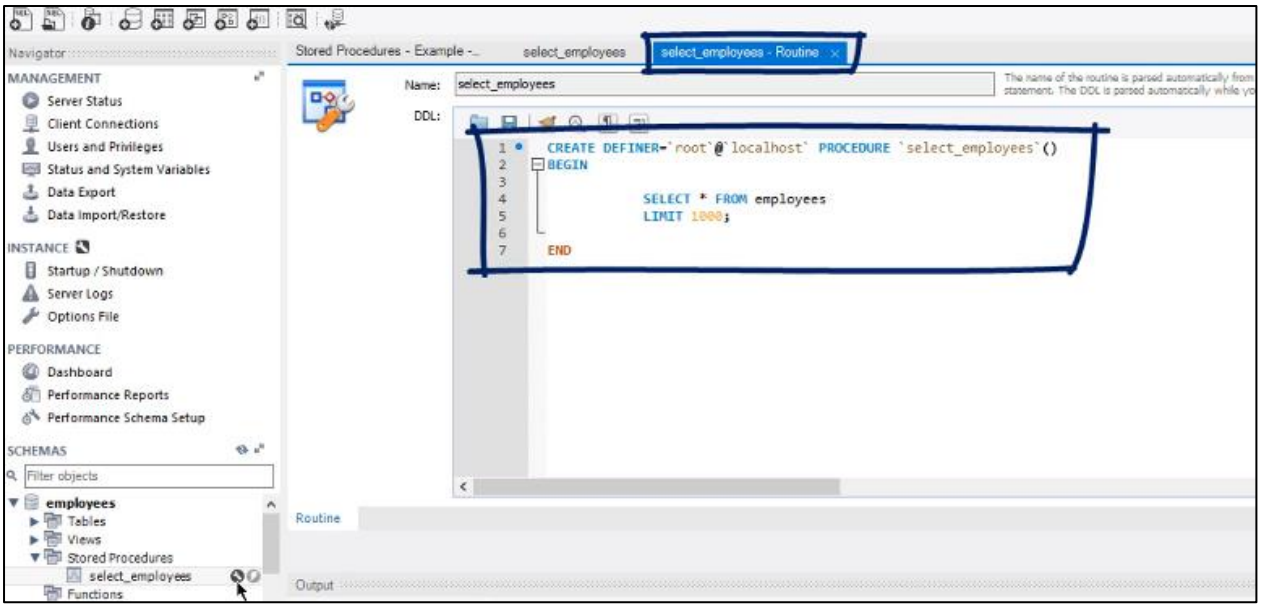
```
call employees.select_employees();

call select_employees();
```

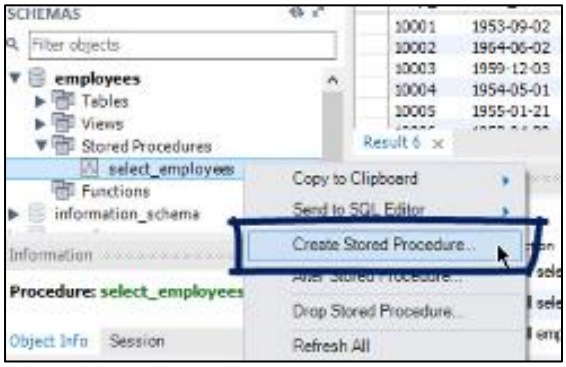
The third way to invoke a procedure is to click on this tiny lightning symbol that turns up as you hover over the name select employees in the schema section in Workbench. After you press this button, a new tab will open to the right. In its top part, you see a newly started SQL window whose only line of code is identical to the first option for invoking the procedure we discussed. Logically, in the middle part of this tab, you can see the results set obtained, the first 1,000 rows of the employee's table, awesome. These are the three ways to invoke a procedure in MySQL.



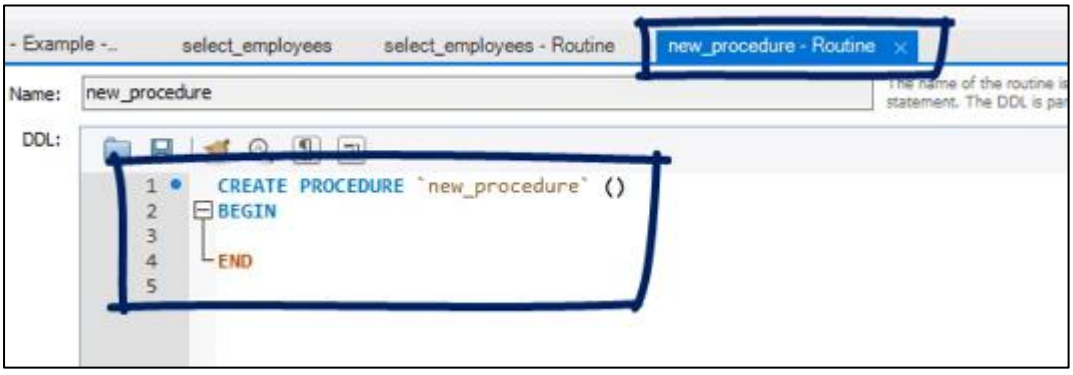
Next to the lightning symbol we used, you can see a tiny icon depicting a wrench. If you click on it, a new separate tab will appear. It shows the whole procedure in question. This is an amazingly useful tool, as it channels your search directly towards the code of the stored procedure and allows for a quicker and more purposeful correction of its structure whenever necessary.



Another way of creating procedures is by doing it through the schemas tab:

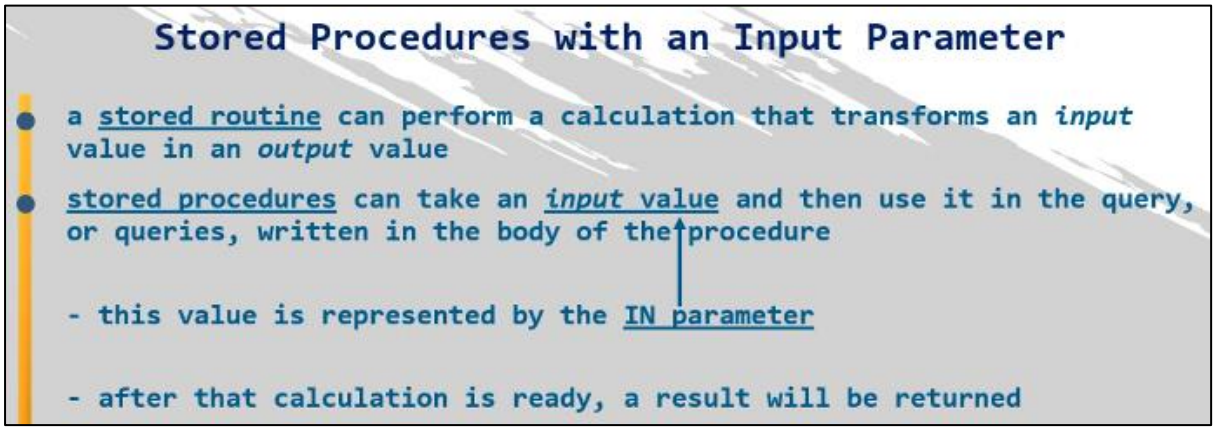


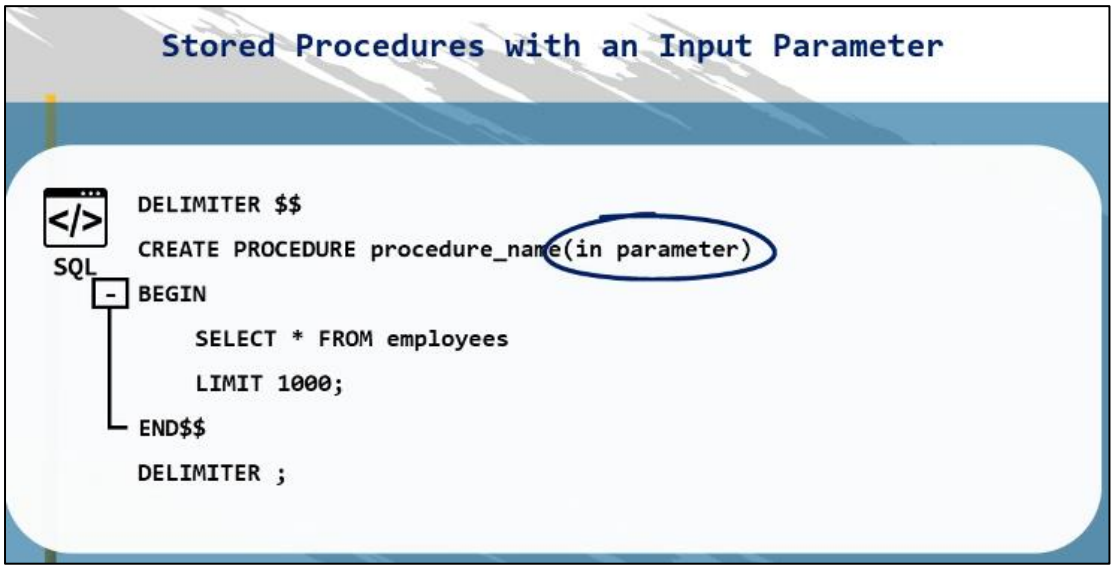
A new table will then pop up with the skeleton of a procedure for you to fill in:



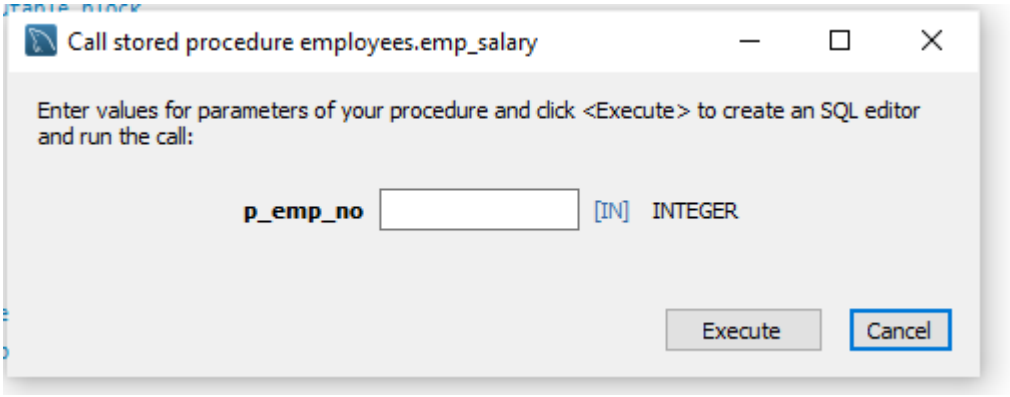
So much easierrrrr!

NEW EXPANSION ON TOPIC:





This is essentially the result of using an IN parameter:



The code, along with the breakdown, is below – incredibly interesting and shows how we're now moving from writing static queries to creating **reusable, parameter-driven logic.** This is exactly how SQL starts behaving like **a real programming language**.

```
324     DELIMITER $$
325 ●   # Temporarily changes the SQL statement delimiter from ';' to '$$' ('//' can also be used to achieve the same outcome.
326     # This is required so MySQL does not end the procedure definition
327     # when it encounters semicolons inside the procedure body.
328
329     USE employees $$
330     # Explicitly confirms that the procedure is created inside the employees database.
331
332 ●   CREATE PROCEDURE emp_salary(IN p_emp_no INTEGER)
333     # Creates a stored procedure named emp_salary.
334     # It accepts one INPUT parameter:
335     # p_emp_no → the employee number supplied when the procedure is called.
336
```

```sql
BEGIN
# Marks the beginning of the procedure's executable block.

SELECT
    e.first_name,
    e.last_name,
    s.salary,
    s.from_date,
    s.to_date
# Specifies the data we want returned when the procedure is executed:
# employee name details and their salary history.

FROM
    employees e
# Uses the employees table and assigns it the alias 'e' for readability.

JOIN
    salaries s ON e.emp_no = s.emp_no
# Joins the salaries table to employees using emp_no,
# linking each employee to their salary records.

WHERE
    e.emp_no = p_emp_no;
# Filters the result to ONLY the employee whose emp_no
# matches the value passed into the procedure.

END$$
# Ends the procedure definition.
# '$$' is used instead of ';' because we changed the delimiter earlier.

DELIMITER ;
# Resets the delimiter back to the standard semicolon.
# This tells MySQL we are finished defining the procedure.
```

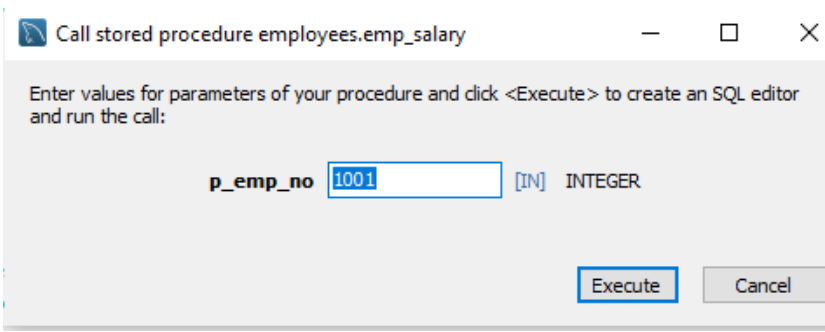
My extended notes: Stored procedures such as this allow businesses to centralise logic in the database, accept dynamic input, and expose controlled, reusable operations instead of raw data access. In a business environment, HR analysts and payroll teams frequently need to retrieve salary histories for individual employees without exposing the full salary table or rewriting complex SQL queries.

By creating a stored procedure such as emp_salary, the organisation encapsulates this logic within the database itself. The procedure accepts an employee number as an input parameter and returns only the relevant salary information for that employee. So if we wanted salary information about employee 10001, we'd enter it into the parentheses of the stored procedure, and this is what we'd get as a result:

```sql
call employees.emp_salary(10001);
```

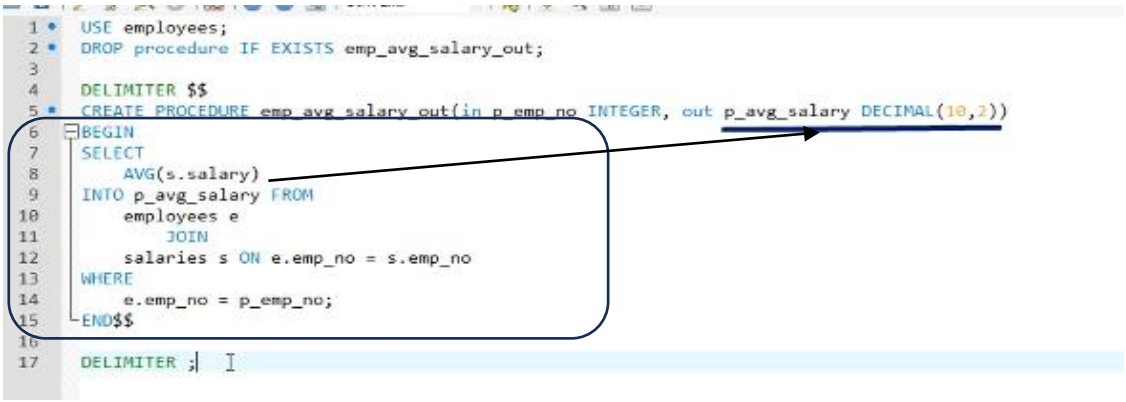
We then get all the contracts Georgi Facello has had since joining the company:

first_name	last_name	salary	from_date	to_date
Georgi	Facello	60117	1986-06-26	1987-06-26
Georgi	Facello	62102	1987-06-26	1988-06-25
Georgi	Facello	66074	1988-06-25	1989-06-25
Georgi	Facello	66596	1989-06-25	1990-06-25
Georgi	Facello	66961	1990-06-25	1991-06-25
Georgi	Facello	71046	1991-06-25	1992-06-24
Georgi	Facello	74333	1992-06-24	1993-06-24
Georgi	Facello	75286	1993-06-24	1994-06-24
Georgi	Facello	75994	1994-06-24	1995-06-24



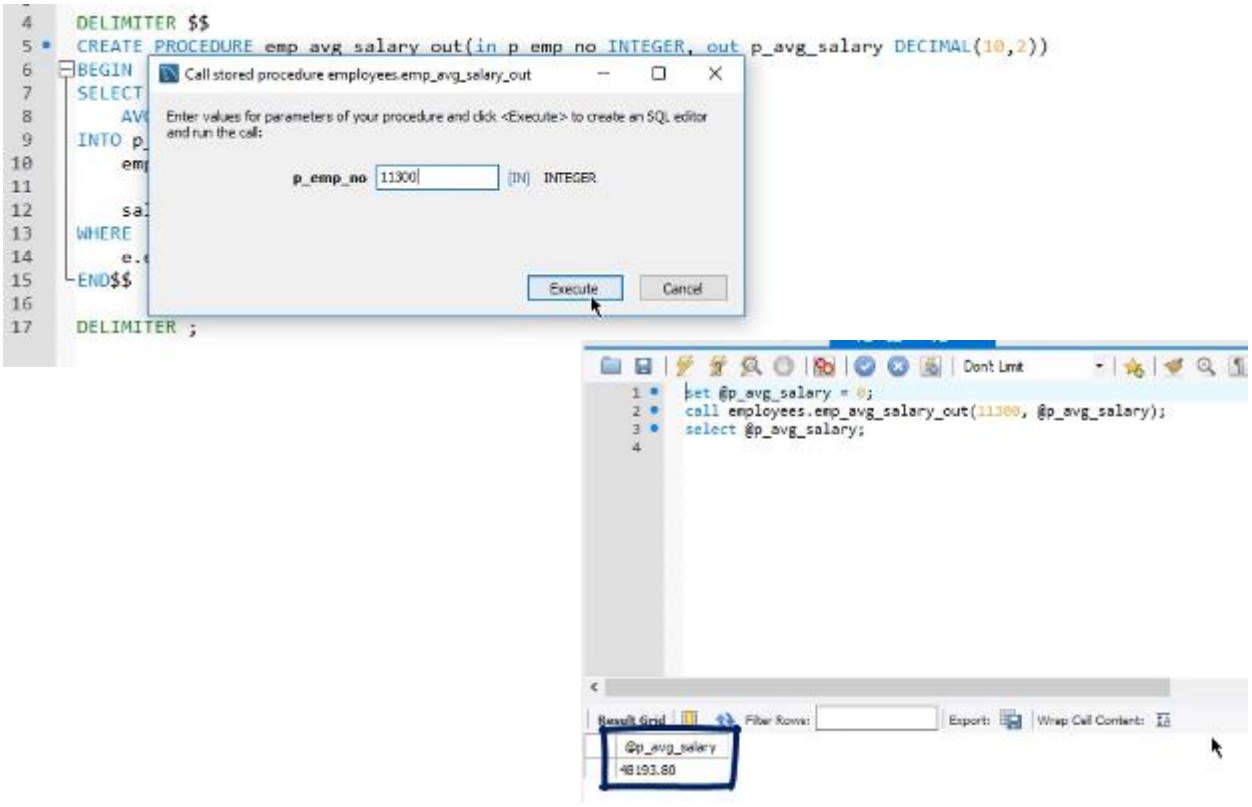
Stored Procedures with INPUT AND OUTPUT PROCEDURES:

Lecturer notes: in-depthOkay, so it is clear how procedures with zero or one parameters work. In cases where the select statement is used in the procedure's body, the output will be displayed to the user readily, and they will be able to treat the result as they like. However, if the outcome is supposed to be stored in another variable that can be recorded in the database and that can be used in outside applications, a second parameter must be defined within the parentheses. This parameter will be called an OUT parameter. It will represent the variable containing the output value of the operation executed by the query of the stored procedure.



We will use P employee number as an in parameter again, and we will add P average salary as an out parameter. It will be of the DECIMAL type because it will define a monetary value, let it be of a precision of 10, and more importantly, a scale of two. Second, look at the body of the procedure. The query must reflect our idea to store just a single value in our parameter. That's why we will need only one selection: The average amount obtained from the salary column in the salaries table. We must then insert this value into the out parameter we just declared. This is the philosopher's stone of my SQL stored procedures. When out parameters are in play. Every time you create a procedure containing both an in and an out parameter, remember that you must use the SELECT INTO structure in the query of this object's body. And **this time outcome of 48,193.80 cents is not just displayed for the user but is also stored in the P average salary parameter.**

The result of the IN & OUT parameters:



```sql
#QUESTION: Create a procedure called 'emp_info' that uses as parameters the first and the last name of an individual,
-- and returns their employee number.
DROP PROCEDURE IF EXISTS emp_info;

DELIMITER //
CREATE PROCEDURE emp_info(IN p_first_name VARCHAR(50), IN p_last_name VARCHAR(50), OUT pemp_no INT) #So Workbench will show an input box for the OUT parameter, which feels like it's asking us
-- to type the OUTCOME – but it's actually asking: "What variable should I store the output in?" I see it as just SQL mental reminder to return the employee number
BEGIN
SELECT
e.emp_no
INTO pemp_no FROM
employees e
WHERE e.first_name = p_first_name AND
e.last_name = p_last_name
LIMIT 1; #We use the LIMIT 1 feature because we have some recurring names e.g. 'Georgi Facello' in our database and this procedure is only programmed to retrieve 1 row, so if it encounters mu
-- with the same value it will get confused and return ERROR, so this just ensures that even if it encounters multiple rows, it should still return 1 row.
END//
DELIMITER ;

CALL emp_info('Georgi', 'Facello', @pemp_no);
SELECT @pemp_no;
```

Action Output

#	Time	Action	Message
75	13:21:46	set @pemp_no = 0	0 row(s) affected
76	13:21:46	call employees.emp_info('Georgi', 'Facello', @pemp_no)	Error Code: 1172. Result consisted of more than one row
77	13:22:25	set @pemp_no = 0	0 row(s) affected
78	13:22:25	call employees.emp_info('Georgi', 'Facello', @pemp_no)	Error Code: 1172. Result consisted of more than one row
79	13:30:05	DROP PROCEDURE IF EXISTS emp_info	0 row(s) affected
80	13:30:13	CREATE PROCEDURE emp_info(IN p_first_name VARCHAR(50), IN p_last_name VARCHAR(50), OUT pemp_no IN...	0 row(s) affected