Department of Information Systems and Technologies

CTIS259 Database Management Systems and Applications

2025 – 2026 Fall

Lab Guide 15

Instructor	: Nimet Ceren SERİM	**Week:**	10
Assistant	: Engin Zafer KIRAÇBEDEL, Hatice Zehra YILMAZ	**Date:**	20-21.11.2025

Aim of this lab session: **1.** PL/SQL Programming Language;
- Creating named blocks for a stored procedure and functions
- Case Statement, Extract, Accept,

ORACLE Server Configurations:

IP Address: 139.179.33.231

Port number: 1522

SID: orclctis

Please use ora (oraxx) accounts

Creating a Procedure

A procedure is a group of PL/SQL statements that you can call by name. A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
END procedure_name;
```

Where,

- *Procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *Procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

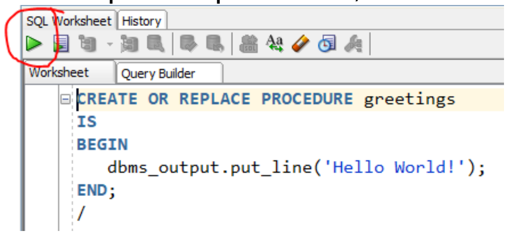
Example:

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

PLSQL_Lab15_1.sql

```
CREATE OR REPLACE PROCEDURE greetings
IS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

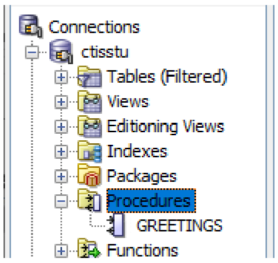
To compile the procedure, click the **Run Statement** button:



```
CREATE OR REPLACE PROCEDURE greetings
IS
BEGIN
   dbms_output.put_line('Hello World!');
END;
/
```

When the above code is compiled using the SQL prompt, it will produce the following result and it will be added to the **procedures** node:

```
Procedure GREETINGS created.
```



Executing a Procedure

A procedure can be called in two ways;

- Using the **EXECUTE** keyword;

```
EXECUTE procedure_name( arguments);
```

- Calling the name of the procedure from a PL/SQL block

The above procedure named **'greetings'** can be called with the EXECUTE keyword as:

```
EXECUTE greetings;
```

The above call will display;

```
PL/SQL procedure successfully completed.
Hello World
```

The procedure can also be called from another PL/SQL block

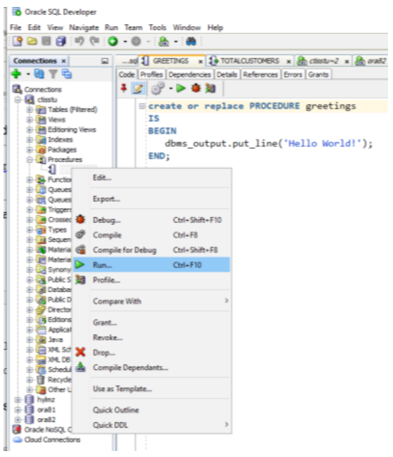
```
BEGIN
   Greetings();
END;
/
```

The above call will display;

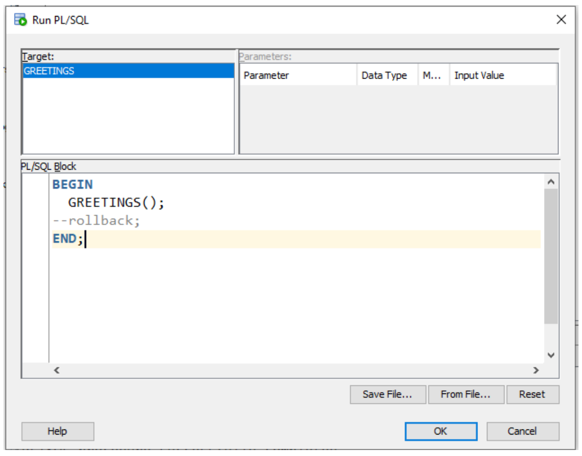
```
PL/SQL procedure successfully completed.
Hello World
```

You can also execute a procedure from the Oracle SQL Developer using the following steps:

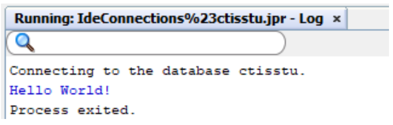
- Right-click the procedure name and choose **Run…**menu item.



```
create or replace PROCEDURE greetings
IS
BEGIN
   dbms_output.put_line('Hello World!');
END;
```



```
BEGIN
  GREETINGS();
--rollback;
END;
```



```
Connecting to the database ctisstu.
Hello World!
Process exited.
```

In order to see the errors, execute the following command and examine the error.

```
SHOW ERRORS PROCEDURE GREETINGS;
```

Deleting a Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is;

```
DROP PROCEDURE procedure-name;
```

You can drop the greetings procedure by using the following statement;

```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms:

Parameter Mode & Description
IN An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
OUT An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
IN OUT An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

A function is same as a procedure except that it **returns a value** by return statement. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A stored function (also called a user function or user-defined function) is a set of PL/SQL statements you can call by name. Stored functions are very similar to procedures, except that a function returns a value to the environment in which it is called. User functions can be used as part of a SQL expression.

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE FUNCTION** statement is as follows;

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example:

The following example illustrates how to create and call a function.

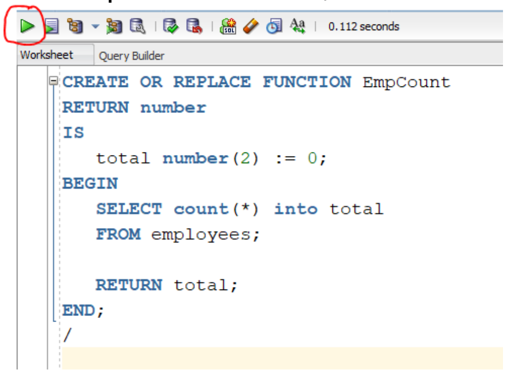
➢ Write a function that returns the total number of EMPLOYEES in the employees table.

Lab15_2.sql

```
CREATE OR REPLACE FUNCTION EmpCount
RETURN number
IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM employees;

   RETURN total;
END;
/
```

To compile the function, click the **Run Statement** button:

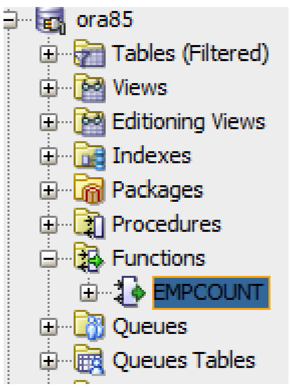


```
CREATE OR REPLACE FUNCTION EmpCount
RETURN number
IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total
   FROM employees;

   RETURN total;
END;
/
```

When the above code is compiled using the SQL prompt, it will produce the following result and it will be added to the **functions** node:

```
Function EMPCOUNT compiled
```



In order to see the errors, execute the following command and examine the error.

```
SHOW ERRORS FUNCTION EMPCOUNT;
```

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

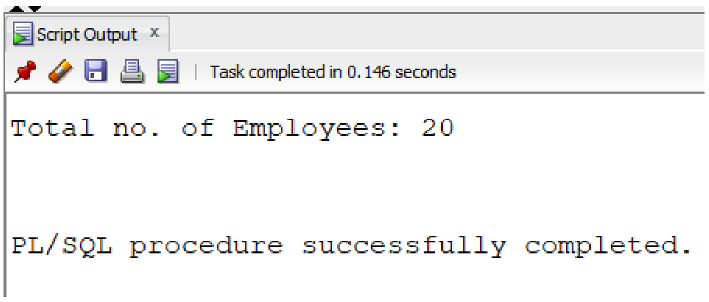
A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value.

Following program calls the function **EMPCOUNT** from an anonymous block;

```
SET SERVEROUTPUT ON;
DECLARE
   c number(2);
BEGIN
   c := EMPCOUNT ();
   dbms_output.put_line('Total no. of Employees: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result:



```
Total no. of Employees: 20


PL/SQL procedure successfully completed.
```

Deleting a Function

A function is deleted with the **DROP FUNCTION** statement. Syntax for deleting a function is;

```
DROP FUNCTION function-name;
```

You can drop the **EMPCOUNT** procedure by using the following statement;

```
DROP FUNCTION EMPCOUNT;
```

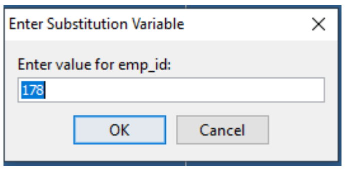
Please use student (fYourStudentId) accounts

CASE Statement: The PL/SQL CASE statement allows you to execute a sequence of statements based on a selector. A selector can be anything such as variable, function, or expression that the CASE statement evaluates to a Boolean value. The syntax is given below:

```
[<<label_name>>]
CASE [TRUE | selector]
   WHEN expression1 THEN
     sequence_of_statements1;
   ...
  [ELSE sequence_of_statements;]
END CASE [label_name];
```

PLSQL_Lab15_3.sql

➢ Write a PL/SQL program block that displays a message according to the **commission_pct** of the specified employee. (Commission_pct 0: **N/A**, 0.1: **Low**, 0.4: **High**, in all other cases **Fair**).



```
PL/SQL procedure successfully completed.

Employee 178 commission .15 which is Fair
```

```
SET SERVEROUTPUT ON;
DECLARE
  n_pct    employees.commission_pct%TYPE;
  v_eval   varchar2(10);
  n_emp_id employees.employee_id%TYPE := &emp_id;
BEGIN
  SELECT commission_pct INTO n_pct
  FROM employees
  WHERE employee_id = n_emp_id;

  CASE nvl(n_pct, 0)
    WHEN 0 THEN
      v_eval := 'N/A';
    WHEN 0.1 THEN
      v_eval := 'Low';
    WHEN 0.4 THEN
      v_eval := 'High';
    ELSE
      v_eval := 'Fair';
  END CASE;
DBMS_OUTPUT.PUT_LINE('Employee '||n_emp_id||' commission '||n_pct||' which is '||v_eval);
END;
/
```

Searched CASE statement: PL/SQL provides a special CASE statement called searched CASE statement. The syntax is as follows:

```
[<<label_name>>]
CASE
   WHEN search_condition_1 THEN sequence_of_statements_1;
   WHEN search_condition_2 THEN sequence_of_statements_2;
   ...
  [ELSE sequence_of_statements;]
END CASE [label_name];
```

The searched CASE statement has no selector. Each WHEN clause in the searched CASE statement contains a search condition that returns a Boolean value.

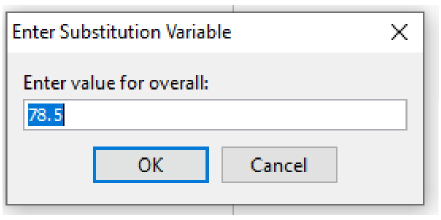
`PLSQL_Lab15_4.sql`

➢ Write the procedure **`find_letter`** that gets the overall grade as input parameter, finds and returns the equivalent letter grade as output parameter.

Grading Criteria:

```
A: 85-100
B: 75-84
C: 60-74
D: 45-59
F: 0-44
```

➢ Write a PL/SQL program block that reads the overall grade from the user and displays the equivalent letter grade using the procedure.



```
PL/SQL procedure successfully completed.

Your Letter grade is: B
```

```
create or replace procedure find_letter
(overall IN number, letter OUT char)
IS
BEGIN
CASE
  WHEN overall >= 85 THEN
    letter := 'A' ;
  WHEN overall >= 75 THEN
    letter := 'B' ;
  WHEN overall >= 60 THEN
    letter := 'C' ;
  WHEN overall >= 45 THEN
      letter := 'D' ;
  ELSE
    letter := 'F' ;
  END CASE;
END;
/
```

```
SET SERVEROUTPUT ON;

DECLARE
  c_grade number(5,2) := &overall;
  c_letter CHAR(1);
BEGIN
  find_letter(c_grade, c_letter);
dbms_output.put_line('Your Letter grade is: '|| c_letter);
END;
/
```

EXTRACT Function

The EXTRACT function extracts a value from a date or interval value.

Syntax
```
EXTRACT (
{ YEAR | MONTH | DAY | HOUR | MINUTE | SECOND }
| { TIMEZONE_HOUR | TIMEZONE_MINUTE }
| { TIMEZONE_REGION | TIMEZONE_ABBR }
FROM { date_value | interval_value } )
```

Note
- You can only extract YEAR, MONTH, and DAY from a DATE.
- You can only extract TIMEZONE_HOUR and TIMEZONE_MINUTE from a timestamp with a time zone datatype.
- The EXTRACT function returns a numeric value when the following parameters are provided: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, TIMEZONE_MINUTE.

Examples:
```
select EXTRACT(YEAR FROM DATE '2020-04-21')
from dual;
Result: 2020
```

```
select EXTRACT(MONTH FROM DATE '2020-04-21')
from dual;
Result: 4
```

```
select EXTRACT(DAY FROM DATE '2020-04-21')
from dual;
Result: 21
```

```
select last_name, hire_date, extract(year from hire_date)
from employees;
```

ACCEPT Command

Reads a line of input and stores it in a given substitution variable. The ACCEPT command permits to interact with the user through the console and to create variable from the user input.

Syntax
```
ACC[EPT] variable [NUM[BER] | CHAR | DATE | BINARY_FLOAT | BINARY_DOUBLE] [FOR[MAT]
format] [DEF[AULT] default] [PROMPT text|NOPR[OMPT]] [HIDE]
```

Examples:
- To display the prompt "Enter weekly salary: " and place the reply in a NUMBER variable named SALARY with a default of 000.0, enter:

```
ACCEPT salary NUMBER FORMAT '999.99' DEFAULT '000.0' PROMPT 'Enter weekly salary:';
```

- To display the prompt "Password: ", place the reply in a CHAR variable named PSWD, and suppress the display, enter:

```
ACCEPT pswd CHAR PROMPT 'Type your Password:  ' HIDE;
```

- To display the prompt "Enter date hired: " and place the reply in a DATE variable, HIRED, with the format "dd/mm/yyyy" and a default of "01/01/2003", enter:

```
ACCEPT hired DATE FORMAT 'dd/mm/yyyy' DEFAULT '01/01/2003' PROMPT 'Enter date hired:';
```

HOW TO DELETE A SUBSTITUTION VARIABLE?

To delete a substitution variable, use the command UNDEFINE followed by the variable name.

```
UNDEFINE MySubstitutionVariable
```

SUPPRESS THE SUBSTITUTION VARIABLE CONTROL

Lists each line of the script before and after substitution.

```
SET VERIFY ON
```

You can suppress this listing by setting the SET command variable VERIFY to OFF.

```
SET VERIFY OFF
```

PLSQL_Lab15_5.sql

- Display all the user tables in the database.

```
SET SERVEROUTPUT ON;
BEGIN
   dbms_output.put_line  (user || ' Tables in the database:');

   FOR t IN (SELECT table_name FROM user_tables)
   LOOP
      dbms_output.put_line(t.table_name);
   END LOOP;
END;
/
```