Department of Information Systems and Technologies

CTIS259 Database Management Systems and Applications

2025 – 2026 Fall

Lab Guide 14

Instructor : Nimet Ceren SERİM **Week:** 10
Assistant : Engin Zafer KIRAÇBEDEL, Hatice Zehra YILMAZ **Date:** 17-18.11.2025

Aim of this lab session: **1.** PL/SQL Programming Language

ORACLE Server Configurations:

IP Address: 139.179.33.231

Port number: 1522

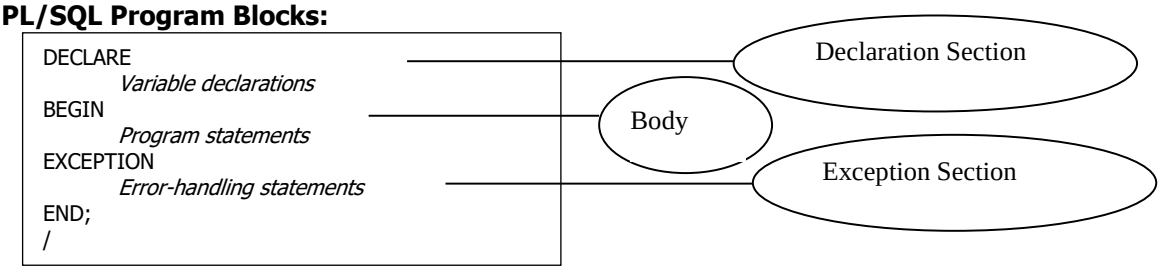
SID: orclctis

Please use student (fYourStudentId) accounts

PL/SQL

PL/SQL is a full-featured programming language. PL/SQL keywords are not case-sensitive, so lower-case letters are equivalent to corresponding upper-case letters except within string and character literals.

PL/SQL Program Blocks:



Declaring Variables:

Ex:

```
DECLARE
s_id NUMBER(4);
name VARCHAR2(15);
```

Scalar data Types:

Data Type	Description	Sample Declaration
VARCHAR2	Variable-length character string	`current_s_last VARCHAR2(30);`
CHAR	Fixed-length character string	`student_gender CHAR(1);`
DATE	Date and time	`todays_date DATE;`
INTERVAL	Time interval	`curr_time INTERVAL YEAR TO MONTH;` `curr_elapsed_time INTERVAL DAY TO SECOND;`
NUMBER	Floating-point, fixed-point, or integer number	`current_price NUMBER(5,2);`
Integer number subtypes (BINARY_INTEGER, INTEGER, INT, SMALLINT)	Integer	`counter BINARY_INTEGER;`
Decimal number subtypes (DEC, DECIMAL, DOUBLE PRECISION, NUMERIC, REAL)	Numeric value with varying precision and scale	`Student_gpa REAL;`
BOOLEAN	True/False value	`Order_flag BOOLEAN;`

BINARY_INTEGER data type is generally used for loop counters. It's values stored in binary format, which takes slightly less storage space than the NUMBER data type, so the system perform calculations more quickly on BINARY_INTEGER values.

Composite Variables:

A **data structure** is a data object made up of multiple individual data elements. A **composite variable** references a data structure that contains multiple scalar variables, such as a record or a table.

Reference Variables:

Reference Variables: Reference variables directly reference a specific database field or record and assume the data type of the associated field or record.

Reference Data Type	Description	Sample Declaration
%TYPE	Assumes the data type of a database field.	`Cust_address Customer.c_address%TYPE;`
%ROWTYPE	Assumes the data type of a database record.	`Order_row orders%ROWTYPE;`

LOB Data Types:

LOB data types declare variables that reference binary data objects such as images and sounds. They can store either binary or character data up to four gigabytes in size. LOB values in PL/SQL programs must be manipulated using a special set of programs, called the DBMS_LOB package.

Assignment Statements:

```
DECLARE
     variable1 NUMBER;
     variable2 NUMBER := 0;
BEGIN
     variable1 := variable2 + 1;
END;
```

Displaying PL/SQL Program Output in SQL*Plus:

- Before you begin to write your program you have to type the following in order to display something as an output:
  SET SERVEROUTPUT ON; (Optional size of server output is 2.000 bytes. If you wish more, add SIZE statement. Eg: SET SERVEROUTPUT ON SIZE 4000;)
- To display program output use the DBMS_OUTPUT.PUT_LINE procedure, which has the following syntax: DBMS_OUTPUT.PUT_LINE("*variable_name*"); (Maximum size for PUT_LINE is 255 characters.)

EXERCISE 1:

Type and execute the following program:

```
--PL/SQL Program to display the current date
SET SERVEROUTPUT ON;
DECLARE
   todays_date DATE;
BEGIN
   todays_date := SYSDATE;
      DBMS_OUTPUT.PUT_LINE('Today''s date is ' );
      DBMS_OUTPUT.PUT_LINE(todays_date);
END;
/
```

(The slash at the end, instructs the PL/SQL interpreter to execute the program code)

DataBase Values:
You can use the SELECT statement to have Oracle assign values to a variable. For each item in the select list, there must be a corresponding, type-compatible variable in the INTO list. An example follows:

```
DECLARE
   emp_id   emp.empno%TYPE;
   emp_name emp.ename%TYPE;
   wages    NUMBER(7,2);
BEGIN
   ...
   SELECT ename, sal + comm
      INTO emp_name, wages FROM emp
      WHERE empno = emp_id;
   ...
END;
```

Data Conversion Functions:

PL/SQL sometimes performs implicit data conversions i.e. while printing date converts it to a string. But, as a programmer, prefer to make explicit data conversion.

Data Conversion Function	Description	Example
TO_CHAR	Converts either a number or date value to a sting using a specific format model.	`TO_CHAR(2.98, '$999.99');` `TO_CHAR(SYSDATE, 'MM/DD/YYYY');`
TO_DATE	Converts a string to a date using a specific format model.	`TO_DATE('07/14/2003', 'MM/DD/YYYY');`
TO_NUMBER	Converts a string to a number	`TO_NUMBER('2');`

Manipulating Character Strings:

- Concatenating Character Strings : To concatenate two strings, use double bar (| |) operator :

```
new_string := string1 || string2;
```

- Removing Blank Leading and Trailing Spaces from Strings : To remove blank leading spaces, use LTRIM function. To remove blank trailing spaces, use RTRIM function.

```
string := LTRIM(string_variable _name);
string := RTRIM(string_variable _name);
```

- Finding the length of Character Strings : To find number of characters in the string use LENGTH function.

```
string_length := LENGTH(string_variable _name);
```

- Character String Case Functions:

```
string := UPPER(string_variable _name);
string := LOWER(string_variable _name);
string := INITCAP(string_variable _name);
```

- The INSTR Function: searches a string for a specific substring. If the function finds the substring, it returns an integer that represents the starting position of the substring within the original string. If the function does not find the substring, it returns 0.

```
start_position := INSTR(original_string, substring);
```

- The SUBSTR Function: extracts a specific number of characters from a character string, starting at a given point.

```
Extracted_string := SUBSTR(string_variable, starting_point, number_of_characters);
```

EXERCISE 2-a:

Write a PL/SQL program to declare a string and initialize it to "CTIS259-Database Management Systems". Then find the department ("CTIS") and course number ("259") and output them.

EXERCISE 2-b:

Modify the PL/SQL program in **EXERCISE 2-a** so the program will display the department in different format (Capitalizes the first letter) also searches for the word 'Data' and displays its position.

Conditional Control: IF Statements
Often, it is necessary to take alternative actions depending on circumstances. The IF statement lets you execute a sequence of statements conditionally. That is, whether the sequence is executed or not depends on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF.

IF-THEN-ELSE
The form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as follows:

```
IF condition THEN
   sequence_of_statements1
ELSE
   sequence_of_statements2
END IF;
```

Note that there is a usage of ELSIF statement in the nested IF blocks.

Iterative Control:

LOOP and EXIT Statements
LOOP statements let you execute a sequence of statements multiple times. There are three forms of LOOP statements: LOOP, WHILE-LOOP, and FOR-LOOP.

LOOP
The simplest form of LOOP statement is the basic (or infinite) loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
   sequence_of_statements  (contains EXIT for some conditions, in order to breakdown a loop)
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop. If further processing is undesirable or impossible, you can use an EXIT statement to complete the loop. You can place one or more EXIT statements anywhere inside a loop, but nowhere outside a loop. There are two forms of EXIT statements: EXIT and EXIT-WHEN.

EXIT
The EXIT statement forces a loop to complete unconditionally. When an EXIT statement is encountered, the loop completes immediately and control passes to the next statement. An example follows:

```
LOOP
   ...
   IF credit_rating < 3 THEN
      ...
      EXIT;  -- exit loop immediately
   END IF;
END LOOP;
-- control resumes here
```

The next example shows that you cannot use the EXIT statement to complete a PL/SQL block:

```
BEGIN
   ...
   IF credit_rating < 3 THEN
      ...
      EXIT;  -- illegal
   END IF;
END;
/
```

Remember, the EXIT statement must be placed inside a loop. To complete a PL/SQL block before its normal end is reached, you can use the RETURN statement.

EXIT-WHEN
The EXIT-WHEN statement allows a loop to complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop. An example follows:

```
DECLARE
   Loop_count BINARY_INTEGER :=1;
BEGIN
LOOP
   INSERT INTO count_table VALUES(loop_count);
   Loop_count := loop_count +1;
   EXIT WHEN loop_count =6 ;              ---------- exit if condition is true
END LOOP;
END;
/
```

Loop Labels
Like PL/SQL blocks, loops can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement, as follows:

```
<<label_name>>
LOOP
   sequence_of_statements
END LOOP;
```

Optionally, the label name can also appear at the end of the LOOP statement, as the following example shows:

```
<<my_loop>>
LOOP
   ...
END LOOP my_loop;
```

WHILE-LOOP
The `WHILE-LOOP` statement associates a condition with a sequence of statements enclosed by the keywords LOOP and END LOOP, as follows:

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

FOR-LOOP
The `FOR-LOOP` statement lets you specify a range of integers, then execute a sequence of statements once for each integer in the range.

The syntax follows:

```
FOR counter IN [REVERSE] lower_bound..higher_bound LOOP
   sequence_of_statements
END LOOP;
```

Dynamic Ranges
PL/SQL lets you determine the loop range dynamically at run time, as the following example shows:

```
SELECT COUNT(empno) INTO emp_count FROM emp;
FOR i IN 1..emp_count LOOP
   ...
END LOOP;
```

GOTO Statement
The GOTO statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the GOTO statement transfers control to the labeled statement or block. In the following example, you go to an executable statement farther down in a sequence of statements:

```
BEGIN
   ...
   GOTO insert_row;
   ...
   <<insert_row>>
   INSERT INTO emp VALUES ...
END;
```

Exercise 3: Find and display the date **n** months later from now.

```
SET SERVEROUTPUT ON;
DECLARE
   today date := sysdate;
   nMonths number := &enterN;

BEGIN
   dbms_output.put_line(nMonths||' months later date will be:');
   dbms_output.put_line(ADD_MONTHS(today, nMonths));
END;
/
```

Exercise 4: Calculate the area and perimeter of a circle. The radius will be given by user. (pi: 3.14)

```
SET SERVEROUTPUT ON;
DECLARE
    pi constant number(3,2) := 3.14;
    radius number(7,2);
    area number(13,2);
    perimeter number(13,2);
BEGIN
    radius := &n;
    perimeter:= 2*pi*radius;
    area := pi* power(radius,2);
    dbms_output.put_line ('Area: '||area||' '||'perimeter: '||perimeter);
END;
/
```

Exercise 5: Generate the reverse of a given number

```
set serveroutput on;
DECLARE
  num NUMBER(7) := &n;
  rev NUMBER(7) := 0;
BEGIN
  WHILE(num > 0)
  LOOP
    rev := rev*10 + mod(num, 10);
    num := floor(num/10);
  END LOOP;

    dbms_output.put_line('Reverse of number is '|| rev);
  End;
/
```

Exercise 6: Calculate and display the sum of the numbers between 1-100.

```
SET SERVEROUTPUT ON;
DECLARE
  v_i number(3) := 1;
  v_sum  number(5) := 0;
BEGIN
  LOOP
    v_sum := v_sum + v_i;
    v_i := v_i + 1;
    exit when v_i > 100 ;
  END LOOP;

   dbms_output.put_line ('The sum of the numbers: '||v_sum);
END;
/
```

USING SQL QUERIES IN PL/SQL PROGRAMS

SQL Data Definition language (DDL) commands, such as CREATE, ALTER, DROP cannot be used in PL/SQL programs.

Data Manipulation Language commands such as SELECT, INSERT, UPDATE, DELETE manipulates data values in tables and they can be used in PL/SQL programs.
Transaction control statements COMMIT, ROLLBACK, SAVEPOINT organizes DML commands into logical transactions and can be used in PL/SQL statements.

CURSORS

A cursor is a pointer to a memory location on the database server that the DBMS uses to process a SQL query. You use cursors to retrieve and manipulate database data in PL/SQL programs. There are two kinds of cursors implicit and explicit.

IMPLICIT Cursors

Whenever you execute an INSERT, UPDATE, DELETE, SELECT query, the DBMS allocates memory location on the database server called **context area**. The implicit cursor is a pointer to the context area. You can use implicit cursor to assign the output of a SELECT query to PL/SQL program variables when you are sure that the query will return one and only one record. If the query returns more than one record, or does not return any records, an error occurs (ORA-01422: extact fetch returns more than requested number of rows). To retrieve data values from a query's implicit cursor into PL/SQL program variables, add an INTO clause to the SELECT query.

```
SELECT field1, field2,….
INTO variable1, variable2,…
FROM table1, table2,…
WHERE join_conditions AND search_conditions_to_retrieve_1_record;
```

Exercise 7: Display the name of the department that John Micc is working for**.**

```
SET SERVEROUTPUT ON;

DECLARE
    current_dname department.dname%TYPE;
BEGIN
     SELECT dname
     INTO current_dname
     FROM department, employee
     WHERE dnumber = dno AND lname='MICC' AND fname='JOHN';
     DBMS_OUTPUT.PUT_LINE('The department name is: ' || current_dname);
END;
/
```

EXPLICIT Cursors:

You create an explicit cursor to retrieve and display data in PL/SQL programs for a query that might retrieve multiple records, or that might return no records at all. The steps to use explicit cursors:

1. Declare the cursor (explicitly in your program)
2. Open the cursor
3. Fetch the records (data rows)
4. Close the cursor.

Decleration:
```
CURSOR cursor_name IS select_query;
```
Opening:
```
OPEN cursor_name;
```
Fetching Data rows:
```
LOOP
     FETCH cursor_name INTO variable_name(s);
     EXIT WHEN cursor_name%NOTFOUND;
END LOOP;
```
Closing:
```
CLOSE cursor_name;
```

Exercise 8: Display the names of employees who are working for Department 3.

```
SET SERVEROUTPUT ON;
DECLARE
     e_lname employee.lname%TYPE;
     e_fname employee.fname%TYPE;
     deptno department.dnumber%TYPE;
     CURSOR emp_cursor IS
               SELECT fname, lname
               FROM employee
              WHERE dno = deptno;
BEGIN
     deptno := 3;
     OPEN emp_cursor;
     LOOP
          FETCH emp_cursor INTO e_fname, e_lname;
          EXIT WHEN emp_cursor%NOTFOUND;
          DBMS_OUTPUT.PUT_LINE ('Employee: ' || e_fname || ' '|| e_lname);
     END LOOP;
     CLOSE emp_cursor;
END;
/
```

Note that, if you want you may define %ROWTYPE variable instead of 2 different variables and you may use selected fields of the employee record by this ROWTYPE variable.

emps emp_cursor%ROWTYPE; and use them as**: emps.lname** and **emps.fname**
You may also use following type of for loops:

```
FOR variable_name(s) IN cursor_name LOOP
     Processing commands
END LOOP;
```

Exercise 9:

```
SET SERVEROUTPUT ON;
DECLARE
     deptno department.dnumber%TYPE;
     CURSOR emp_cursor IS
               SELECT fname, lname
               FROM employee
               WHERE dno = deptno;
     e_row emp_cursor%ROWTYPE;

BEGIN
     deptno := 3;
     FOR e_row IN emp_cursor LOOP
          DBMS_OUTPUT.PUT_LINE ('Employee: ' || e_row.fname || ' ' || e_row.lname);
     END LOOP;
END;
/
```

EXCEPTIONS:
For the programmers, it is impossible to do something if a user's computer fails but, they should do everything possible to prevent users from entering incorrect data and keep incorrect keystrokes from damaging the system. Well-written programs help users avoid errors, inform users when an error occurs, and provide advice for correcting errors. PL/SQL supports **exception handling**, whereby programmers place commands for displaying error messages and giving users operations for fixing errors in the program's exception section.

When a runtime error occurs, an **exception**, or unwanted event, is **raised**. Program control immediately transfers to the program's exception section, where exception handler exits to deal with, or handle, different error situations. Most common predefined exceptions:

ERROR CODE	Exception Name	Description
ORA-0001	DUP_VAL_ON_INDEX	Command violates primary key unique constraint
ORA-01403	NO_DATA_FOUND	Query retrieves no records
ORA-01422	TOO_MANY_ROWS	Query returns more rows than anticipated
ORA-01476	ZERO_DIVIDE	Division by zero
ORA-01722	INVALID_NUMBER	Invalid number conversion such as trying to convert 2B to a number.
ORA-06502	VALUE_ERROR	Error in truncation, arithmetic, or data conversion operation.

You can create exception handlers to display alternate error messages for predefined exceptions.
General syntax of an exception handler:

```
EXCEPTION
     WHEN exception1_name THEN
          exception1 handler commands;

          WHEN exception2_name THEN
          exception2 handler commands;
          ……

         WHEN OTHERS THEN
          Other handler commands;
END;
```

Exercise_10.sql

Display the name of the specified employee. Employee numbers can be 100-9999, so write a user-defined exception and display and appropriate message on the screen.

```
SET SERVEROUTPUT ON;

DECLARE
   Emp_name           VARCHAR2(10);
   Emp_number         INTEGER;
   Empno_out_of_range EXCEPTION;
BEGIN
   Emp_number := 10001;
   IF Emp_number > 9999 OR Emp_number < 100 THEN
      RAISE Empno_out_of_range;
   ELSE
      SELECT fname INTO Emp_name FROM employee
         WHERE ssn = Emp_number;
      dbms_output.put_line ('Employee name is ' || Emp_name);
END IF;
EXCEPTION
   WHEN Empno_out_of_range THEN
      dbms_output.put_line ('Employee number  ' || Emp_number || '  is out of range.');
END;
/
```