LAB NO.: 10 **Date:**

PROCEDURES, FUNCTIONS & TRIGGERS

Objectives:

In this lab, student will be able to:

- Understand the different types of subprograms: Procedures and Function.
- Use Procedures and Functions to perform specific tasks.
- Understand the concept of Packages.

Subprograms: A PL/SQL subprogram is a named PL/SQL block that can be invoked repeatedly. If the subprogram has parameters, their values can differ for each invocation. PL/SQL has two types of subprograms:

1. **Procedure:** used to perform an action.
2. **Function:** to compute and return a value.

Uses of Subprograms:

- Modularity
- Easier Application Design
- Maintainability
- Reusability
- Better Performance

Since the stored subprograms run in the database server, a single invocation over the network can start a large job.

Part of a Subprogram:

- Declarative part(optional)
- Executable part(required)
- Exception handling part(optional)

PROCEDURE

A procedure is a subprogram that performs a specific action. A procedure invocation (or call) is a statement. In a procedure a return statement returns the control to the invoker and we cannot specify any expression. The declarative part of a subprogram does not begin with the keyword DECLARE, as the declarative part of an anonymous block does.

Syntax:

```
CREATE OR REPLACE PROCEDURE procedure_name(parameters list)    is
        BEGIN
        //statements
        END;
        /
```

The procedure can be executed by just calling the procedure name with parameters in other PL/SQL block. e.g.: procedure_name (actual values for the parameters);

Example: Create a procedure to print Hello world and execute the procedure.

```
create or replace procedure print_hello is
        begin
        dbms_output.put_line('Hello World');
```

```
        end;
  /
```

In another PL/SQL block procedure is called as shown below:

```
declare
        begin
        print_hello;
        end;
/
```

Output:
Hello World

FUNCTIONS

A function has a same structure as a procedure except:

- A function must include a RETURN clause, which specifies the data type of the value that the function returns (A procedure heading cannot have a RETURN clause)
- In the executable part of a function, every execution path must lead to a RETURN statement. Otherwise, the PL/SQL compiler issues a compile-time warning.

Syntax:

```
CREATE OR REPLACE FUNCTION function_name(variable_name datatype)
        RETURN datatype
        AS
           //declare section
        BEGIN
        …….
        RETURN var2 //variable of the return datatype
        END;
        /
```

Example 1: Create a function to return the sum of two numbers

```
        create or replace function sum_number(a number, b number)
         return number as
         tot number;
         begin
            tot := a + b;
        return tot;
        end;
        /
```

In another PL/SQL Block

```
        setserveroutput on;
        declare
        begin
        dbms_output.put_line(sum_number(5,4));
        end;
        /
```

Output: 9

Example 2: A function that, given the name of a department, returns the count of the number of instructors in that department. (Using University database schema)

```
create function dept count(dept name varchar)
returns integer
begin
declare d count integer;
select count(*) into d count
from instructor
where instructor.dept name= dept name
return d count;
end
```

This function can be used in a query that returns names and budgets of all departments with more than 12 instructors:

```
select dept name, budget
from instructor
where dept count(dept name) > 12;
```

PL/SQL Subprogram Parameter Modes

IN	OUT	IN OUT
Default mode	Must be specified.	Must be specified.
Passes a value to the subprogram.	Returns a value to the invoker.	Passes an initial value to the subprogram and returns an updated value to the invoker.
Formal parameter acts like a constant: When the subprogram begins, its value is that of either its actual parameter or default value, and the subprogram cannot change this value.	Formal parameter is initialized to the default value of its type. The default value of the type is `NULL` except for a record type with a non-`NULL` default value When the subprogram begins, the formal parameter has its initial value regardless of the value of its actual parameter. Oracle recommends that the subprogram assign a value to the formal parameter.	Formal parameter acts like an initialized variable: When the subprogram begins, its value is that of its actual parameter. Oracle recommends that the subprogram update its value.
Actual parameter can be a constant, initialized variable, literal, or expression.	If the default value of the formal parameter type is `NULL`, then the actual parameter must be a variable whose data type is not defined as `NOT NULL`.	Actual parameter must be a variable (typically, it is a string buffer or numeric accumulator).
Actual parameter is passed by reference.	By default, actual parameter is passed by value; if you specify `NOCOPY`, it might be passed by reference.	By default, actual parameter is passed by value (in both directions); if you specify `NOCOPY`, it might be passed by reference.

Note: Do not use OUT and IN OUT for function parameters. Ideally, a function takes zero or more parameters and returns a single value. A function with IN OUT parameters returns multiple values and has side effects. Regardless of how an OUT or IN OUT parameter is passed:

- If the subprogram exits successfully, then the value of the actual parameter is the final value assigned to the formal parameter. (The formal parameter is assigned at least one value—the initial value.)
- If the subprogram ends with an exception, then the value of the actual parameter is undefined.

- Formal OUT and IN OUT parameters can be returned in any order. In this example, the final values of x and y are undefined:

```
CREATE OR REPLACE PROCEDURE p (x OUT INTEGER, y OUT INTEGER)
AS
BEGIN
x := 17; y := 93;
END;
/
```

Example 1:
A procedure that, given the name of a department, returns the count of the number of instructors in that department.

```
create procedure dept_count_proc(in dept_name varchar,
out d_count integer)
begin
select count(*) into d_count
from instructor
where instructor.dept_name= dept_count_proc.dept_name
end
```

Procedures can be invoked either from an SQL procedure or from embedded SQL by the call statement:

```
declare d_count integer;
call dept_count_proc('Physics', d_count);
```

Example 2: The procedure p has two IN parameters, one OUT parameter, and one IN OUT parameter. The OUT and IN OUT parameters are passed by value (the default). The anonymous block invokes p twice, with different actual parameters. Before each invocation, the anonymous block prints the values of the actual parameters. The procedure p prints the initial values of its formal parameters. After each invocation, the anonymous block prints the values of the actual parameters again.

```
CREATE OR REPLACE PROCEDURE p (
a PLS_INTEGER, -- IN by default
b IN PLS_INTEGER,
c OUT PLS_INTEGER,
d IN OUT BINARY_FLOAT
) IS
BEGIN
-- Print values of parameters:
DBMS_OUTPUT.PUT_LINE('Inside procedure p:');
DBMS_OUTPUT.PUT('IN a = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(a), 'NULL'));
DBMS_OUTPUT.PUT('IN b = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(b), 'NULL'));
DBMS_OUTPUT.PUT('OUT c = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(c), 'NULL'));
```

```
DBMS_OUTPUT.PUT_LINE('IN OUT d = ' || TO_CHAR(d));
-- Can reference IN parameters a and b,
-- but cannot assign values to them.
c := a+10; -- Assign value to OUT parameter
d := 10/b; -- Assign value to IN OUT parameter
END;
/
DECLARE
aa CONSTANT PLS_INTEGER := 1;
bb PLS_INTEGER := 2;
cc PLS_INTEGER := 3;
dd BINARY_FLOAT := 4;
ee PLS_INTEGER;
ff BINARY_FLOAT := 5;
BEGIN
DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');
DBMS_OUTPUT.PUT('aa = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));
DBMS_OUTPUT.PUT('bb = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));
DBMS_OUTPUT.PUT('cc = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));
DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));
p (aa, -- constant
bb, -- initialized variable
cc, -- initialized variable
dd -- initialized variable
);
DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');
DBMS_OUTPUT.PUT('aa = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa), 'NULL'));
DBMS_OUTPUT.PUT('bb = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(bb), 'NULL'));
DBMS_OUTPUT.PUT('cc = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(cc), 'NULL'));
DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));
DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');
DBMS_OUTPUT.PUT('ee = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));
DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));
p (1, -- literal
(bb+3)*4, -- expression
ee, -- uninitialized variable
ff -- initialized variable
);
DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');
DBMS_OUTPUT.PUT('ee = ');
DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(ee), 'NULL'));
DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));
END;
```

/

Output:

```
Before invoking procedure p:
aa = 1
bb = 2
cc = 3
dd = 4.0E+000
Inside procedure p:
IN a = 1
IN b = 2
OUT c = NULL
IN OUT d = 4.0E+000
After invoking procedure p:
aa = 1
bb = 2
cc = 11
dd = 5.0E+000
Before invoking procedure p:
ee = NULL
ff = 5.0E+000
Inside procedure p:
IN a = 1
IN b = 20
OUT c = NULL
IN OUT d = 5.0E+000
After invoking procedure p:
ee = 11
ff = 5.0E-001
```

EXERCISE:

Procedures:

1. Based on the University Database Schema in Lab 2, write a procedure which takes the dept_name as input parameter and lists all the instructors associated with the department as well as list all the courses offered by the department. Also, write an anonymous block with the procedure call.
2. Based on the University Database Schema in Lab 2, write a Pl/Sql block of code that lists the most popular course (highest number of students take it) for each of the departments. It should make use of a procedure course_popular which finds the most popular course in the given department.

Functions:

3. Write a function to return the Square of a given number and call it from an anonymous block.

4. Based on the University Database Schema in Lab 2, write a Pl/Sql block of code that lists the highest paid Instructor in each of the Department. It should make use of a function department_highest which returns the highest paid Instructor for the given branch.

TRIGGERS

Objectives:

In this lab, student will be able to:

- Use triggers to enforce business logic in a database.
- Use Row triggers to enforce any constraint on a table.
- Understand the usage of "Instead Of" Triggers.

Triggers

A trigger is a named PL/SQL block that is stored in the database and run in response to an event that occurs in the database. The user can specify the event (insert, update or delete)

- whether the trigger fires before or after the event
- whether the trigger runs for each event or for each row affected by the event

Uses of Triggers

1. Enforce referential integrity constraints when child and parent tables are on different nodes of the distributed database.
2. Prevent DML operations on a table after regular business hours.
3. Modify table data when DML statements are issued against views.
4. Enforce complex business rules that you cannot define with constraints.

Triggers vs Constraints

- A trigger applies on new data only while constraint holds true for the entire data Eg.: a trigger can prevent a DML statement from inserting a NULL but the column couldhave null prior.
- Constraints are easier to write and less error prone.
- But triggers can enforce complex business rules that constraints cannot.

Syntax:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE(AFTER) INSERT OR UPDATE OF list of columns
OR DELETE on tablename
FOR EACH ROW
begin
CASE
WHEN INSERTING THEN
//set of actions
WHEN UPDATING(column1) THEN
//set of actions
WHEN DELETING THEN
//set of actions
END CASE;
END;
/
```

:NEW AND :OLD

- The trigger is fired when DML operations (INSERT, UPDATE, and DELETE statements) are performed on the table.
- The user can choose what combination of operations should fire the trigger.
- Because the trigger uses BEFORE keyword, it can access the new values before they go into the table, and can change the values if there is an easily-corrected error

by assigning to :NEW.column_name.

- NEW.attribute is accessible on insertion and updation.
- OLD.attribute is accessible on deletion and updation.

Example: A trigger to insert the records into the emp_delete table before deleting the records from the employee table, based on the HR schema provided in the appendix.

```
CREATE OR REPLACE trigger emp_trigger
BEFORE DELETE ON employee
FOR EACH ROW
BEGIN
insert into emp_delete values(:OLD.emp_id,
:OLD.emp_name,
:OLD.emp_sal);
END;
/
```

Output:

```
delete from employee where emp_id=1101;
When you execute the above statement in the terminal the same
record is inserted in the emp_delete table.
```

INSTEAD OF TRIGGERS:

- It is created to perform data manipulation of views which cannot be performed ingeneral.
- We cannot perform deletion or insertion on views if the view contains aggregate functions or joins.
- An INSTEAD OF trigger is the only way to update a view that is not inherently updatable.
- Consider a view which is joined on tables A and B. A delete operation on the view is equivalent to deletion on A separately and B separately.

Syntax:

```
Create or replace trigger trigger_name
INSTEAD OF DELETE on view_name
FOR EACH ROW
BEGIN
//Set of Actions
END;
```

EXERCISE:

Row Triggers

1. Based on the University database Schema in Lab 2, write a row trigger that records along with the time any change made in the Takes (ID, course-id, sec-id, semester, year, grade) table in log_change_Takes (Time_Of_Change, ID, courseid,sec-id, semester, year, grade).

2. Based on the University database schema in Lab: 2, write a row trigger to insert the existing values of the Instructor (ID, name, dept-name, salary) table into a new table Old_ Data_Instructor (ID, name, dept-name, salary) when the salary table is updated.

Database Triggers

3. Based on the University Schema, write a database trigger on Instructor that checks the following:
   - The name of the instructor is a valid name containing only alphabets.
   - The salary of an instructor is not zero and is positive

- The salary does not exceed the budget of the department to which the instructor belongs.

4. Create a transparent audit system for a table Client_master (client_no, name, address, Bal_due). The system must keep track of the records that are being deleted or updated. The functionality being when a record is deleted or modified the original record details and the date of operation are stored in the auditclient (client_no, name, bal_due, operation, userid, opdate) table, then the delete or update is allowed to go through.

Instead of Triggers

5. Based on the University database Schema in Lab 2, create a view Advisor_Student which is a natural join on Advisor, Student and Instructor tables. Create an INSTEAD OF trigger on Advisor_Student to enable the user to delete the corresponding entries in Advisor table.