Lesson-03 : (Part-2) Procedure, Functions and Packages

Overview of Subprograms

A subprogram:
- Is a NAMED PL/SQL block that can accept parameters and be invoked from a calling environment
- Is of TWO types:
  - A PROCEDURE that performs an action
  - A FUNCTION that computes a value
- Is based on STANDARD PL/SQL block structure
- Provides modularity, reusability, extensibility, and maintainability
- Provides easy maintenance, improved data security and integrity, improved performance, and improved code clarity

Block Structure for PL/SQL Subprograms

```
<header>                    } Subprogram
IS | AS                     } Specification
  Declaration section            }---
BEGIN                            }  |
  Executable section             }  Subprogram
EXCEPTION (optional)             }  Body
  Exception section              }  |
END;                             }---
```

Comparison

	Anonymous Block	Subprogram / Named Block
[1]	Do not have names	STORED subprograms are NAMED Blocks
[2]	They are interactively executed. The block needs to be compiled every time it is run.	They are compiled at the time of creation and stored in the Oracle DB. Source code is also stored in DB
[3]	Only the user who created the block can use the block	Necessary privileges are required to execute the block

What is a Procedure?

- A procedure is a type of subprogram that PERFORMS AN ACTION.
- A procedure is stored in the DB, as a schema object, for repeated execution.

Creating a Procedure

Syntax:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter1 [mode1] datatype1, parameter2 [mode2] datatype2,
. . .)]
IS|AS
PL/SQL Block;
```

- PL/SQL block starts with either BEGIN or the declaration of local variables and ends with either END or END procedure_name.

Formal v/s Actual Parameters

- Formal parameters: variables declared in the parameter list of a subprogram specification
  Example:
  ```
  CREATE PROCEDURE raise_sal(p_id NUMBER, p_amount NUMBER)
  ...
  END raise_sal;
  ```
- Actual parameters: variables or expressions referenced in the parameter list of a subprogram call
  Example:
  ```
  raise_sal(v_id, 2000)
  ```

Procedural Parameter Modes

The THREE types of procedural parameters are as follows:

- IN parameter
- OUT parameter
- IN OUT parameter

	IN	OUT	IN OUT
Default mode	Must be specified		Must be specified
Value is passed into the subprogram	Returned to the calling environment		Passed into the subprg; returned to the calling environment
Formal parameters act as a constant	Uninitialized variables		Initialized variables
Actual parameters can be literals, expr. or initialized var.	Must be variable		Must be variable
Can be assigned a default value.	Cannot be assigned a default value		Cannot be assigned a default value.

Example:

```
CREATE OR REPLACE PROCEDURE raise_salary
  ( p_id IN employees.employee_id%TYPE,
      p_amount IN NUMBER)
IS

BEGIN
      -- Updating the salary of an employee based on the given ID
      -- and amount
      UPDATE employees
            SET salary = salary + p_amount
            WHERE employee_id = p_id;
END raise_salary;
```

Executing a Procedure
~~~~~~~~~~~~~~~~~~~~
To execute a procedure in the SQL*Plus environment, the EXECUTE command is used.

```
SQL> EXECUTE raise_salary(206, 2000)

PL/SQL procedure successfully completed.
```

However, if we desire to pass variables of the SQL*Plus environment to the PL/SQL procedure or function; they need to be created first in the SQL*Plus environment. Done using the VARIABLE command.

These variables are called BIND or HOST variables.

The bind or host variables are referenced in the PL/SQL block of code by preceding them with a COLON.

Example:

```
SQL> -- Creating TWO variables in the host environment.
SQL> VARIABLE fname VARCHAR2(20)
SQL> VARIABLE salary NUMBER
SQL>
SQL>
SQL> -- Now, executing the 'get_details' procedure; which gets the
     -- first_name and salary of an employee
SQL> EXECUTE get_details(206, :fname, :salary)

PL/SQL procedure successfully completed.

SQL> -- Viewing the got details
SQL> PRINT fname

FNAME
--------------------------------------------------------------------------------
William

SQL> PRINT salary

    SALARY
----------
      8300
```

Procedure with IN OUT Parameter
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```
CREATE OR REPLACE PROCEDURE format_phone
  ( p_phone_no IN OUT VARCHAR2 )
IS

BEGIN
      p_phone_no := '(' || SUBSTR(p_phone_no, 1, 3) ||
                           ')' || SUBSTR(p_phone_no, 4, 3) ||
                           '-' || SUBSTR(p_phone_no, 7);
END format_phone;
/
```

Now, executing the above procedure in SQL*Plus environment.

```
SQL> VARIABLE g_phone_no VARCHAR2(15)
SQL> BEGIN
  2   :g_phone_no := '8006330575';
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL> PRINT g_phone_no

G_PHONE_NO
---------------------------------------------------------------------
8006330575

SQL> EXECUTE format_phone( :g_phone_no )

PL/SQL procedure successfully completed.

SQL> PRINT g_phone_no

G_PHONE_NO
---------------------------------------------------------------------
(800)633-0575
```

Invoking a Procedure from Anonymous PL/SQL block
==========

We can invoke a procedure from an anonymous PL/SQL block just by using its name and passing the required parameters.

Example:

```
-- Calling the 'raise_salary' procedure
raise_salary(v_id, v_amount);

DBMS_OUTPUT.PUT_LINE('Phone no. BEFORE formatting : ' || v_phone_no );
format_phone( v_phone_no ); -- invoking the 'format_phone' procedure
DBMS_OUTPUT.PUT_LINE('Phone no. AFTER formatting : ' || v_phone_no );
```

==========

What is a Function?
==========

- A function is a named PL/SQL block that RETURNs a VALUE.
- A function is stored in the DB as a schema object for repeated execution.
- A function is called as part of an expression.

Creating Functions
~~~~~~~~~~

Syntax:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter1 [mode1] datatype1, parameter2 [mode2] datatype2,
. . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

NOTE: The PL/SQL block must have at least one RETURN statement.
~~~~~~~~~~

Example:

```
CREATE OR REPLACE FUNCTION get_sal
  ( p_id         employees.employee_id%TYPE )
RETURN NUMBER
AS
      v_salary    employees.salary%TYPE := 0;
BEGIN
      SELECT salary INTO v_salary
      FROM employees
      WHERE employee_id = p_id;

      RETURN v_salary;
END get_sal;
```

Executing a Function

~~~~~~~~~~~~~~~~~~~

To execute the function in the SQL*Plus environment.

```
SQL> VARIABLE g_sal  NUMBER

SQL> EXECUTE :g_sal := get_sal( 206 )

PL/SQL procedure successfully completed.

SQL> PRINT g_sal

     G_SAL
----------
      8300
```

Example-2

```
CREATE OR REPLACE FUNCTION tax ( p_value IN NUMBER )
RETURN NUMBER
AS
BEGIN
      RETURN p_value * 0.08;
END tax;
```

Invoking the 'tax' function in SQL*Plus environment.

```
SQL> VARIABLE tax_amt NUMBER
SQL> EXECUTE :tax_amt  :=  tax(5000)

PL/SQL procedure successfully completed.

SQL> PRINT tax_amt

   TAX_AMT
----------
       400

SQL> SELECT first_name, salary, tax(salary) "TAX"
  2  FROM employees
  3  WHERE department_id >= 100;

FIRST_NAME               SALARY        TAX
-------------------- ---------- ----------
Nancy                     12008     960.64
Daniel                     9000        720
```
~~~~~~~~~~~~~~~~~~~

Exception Handling in Procedures and Functions

If a procedure/function has no exception handler for any error, the control immediately passes out of the procedure/function to the calling environment.

Values of OUT and IN OUT formal parameters are not returned to actual parameters.

Actual parameters will retain their old values.

Comparing Procedure and Function

Procedure	Function
Execute as a PL/SQL statement	Invoked as part of an expression
Do not contain RETURN clause in the header	Must contain a RETURN clause in the header
Can return none, one or many values	Must return a SINGLE value
Can contain a RETURN statement	Must contain at least one RETURN statement

Packages

Overview of Packages

Packages:

- Group logically related PL/SQL types, items, and subprograms
- Consist of two parts:
  - Specification
  - Body
- Allow the Oracle DB server to read multiple objects into memory at once

Creating a Package Specification

To create a Package specification the Syntax is as follows:

```
CREATE PACKAGE package_name AS
 variable_declaration
 cursor_declaration
 FUNCTION func_name param datatype ,..) return datatype1 ;
 PROCEDURE proc_name param in|out|in out} datatype ,...);
END package_name;
```

Example:

```
CREATE OR REPLACE PACKAGE pack1 AS
  PROCEDURE proc1;
  FUNCTION  fun1 return varchar2;
END pack1;
```

Creating a Package Body

To create a Package Body, the syntax is as follows:

```
CREATE PACKAGE BODY package_name AS
 variable_declaration
 cursor_declaration;
 PROCEDURE proc_name (param {IN|OUT|IN OUT} datatype ,...) IS
 BEGIN
      pl/sql_statements
 END proc_name;
 FUNCTION func_name (param datatype ,...) RETURN datatype is
 BEGIN
      pl/sql_statements
 END func_name;
END package_name;

Example:
CREATE OR REPLACE PACKAGE BODY pack1 AS
  PROCEDURE proc1 is
  BEGIN
      dbms_output.put_line('Hi a message from procedure..');
  END proc1;
  FUNCTION fun1 return varchar2 is
  BEGIN
      return ('Hello from fun1..');
  END fun1;
END pack1;

How to execute package components?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
After the package specification and package body is successfully compiled
we can execute the procedure(s) and function(s) from the package using
the
FQN (fully-qualified name) i.e. package_name.proc/func

Executing the procedure and function in the SQL*Plus environment

SQL> EXECUTE  pack1.proc1
Hi - A message from proc1 procedure...

PL/SQL procedure successfully completed.

SQL> VAR  g_text  VARCHAR2(50)
SQL> EXECUTE :g_text := pack1.fun1()

PL/SQL procedure successfully completed.

SQL> PRINT g_text

G_TEXT
--------------------------------------------------------------------
Hello from fun1.

================================================================================
Ref Type Cursors
================
TDB
      There is a more detailed discussion on Ref. Type Cursor in
Adv. PL/SQL - Lesson-02
```