PL/SQL - PROCEDURES

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

A subprogram can be created:

- At schema level
- Inside a package
- Inside a PL/SQL block

A schema level subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter 'PL/SQL - Packages'.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms:

- **Functions**: these subprograms return a single value, mainly used to compute and return a value.
- **Procedures**: these subprograms do not return a value directly, mainly used to perform an action.

This chapter is going to cover important aspects of a **PL/SQL procedure** and we will cover **PL/SQL function** in next chapter.

Parts of a PL/SQL Subprogram

Each PL/SQL subprogram has a name, and may have a parameter list. Like anonymous PL/SQL blocks and, the named blocks a subprograms will also have following three parts:

S.N.	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

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 modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter 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.

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

When above code is executed using SQL prompt, it will produce the following result:

```
Procedure created.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways:

- Using the EXECUTE keyword
- 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 would display:

```
Hello World

PL/SQL procedure successfully completed.
```

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

```
BEGIN
   greetings;
END;
/
```

The above call would display:

```
Hello World

PL/SQL procedure successfully completed.
```

Deleting a Standalone Procedure

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

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

So you can drop *greetings* procedure by using the following statement:

```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms

S.N.	Parameter Mode & Description
1	**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.**
2	**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**.
2	**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 its 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.**

IN & OUT Mode Example 1

This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

```
DECLARE
   a number;
   b number;
   c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < y THEN
      z:= x;
   ELSE
      z:= y;
   END IF;
END;

BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

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

```
 Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use same parameter to accept a value and then return another result.

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

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

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

Methods for Passing Parameters

Actual parameters could be passed in three ways:

- Positional notation
- Named notation
- Mixed notation

POSITIONAL NOTATION

In positional notation, you can call the procedure as:

```
findMin(a, b, c, d);
```

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, a is substituted for x, b is substituted for y, c is substituted for z and d is substituted for m.

NAMED NOTATION

In named notation, the actual parameter is associated with the formal parameter using the arrow symbol => . So the procedure call would look like:

```
findMin(x=>a, y=>b, z=>c, m=>d);
```

MIXED NOTATION

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal:

```
findMin(a, b, c, m=>d);
```

But this is not legal:

```
findMin(x=>a, b, c, d);
```