PROCEDURES, FUNCTIONS & CURSORS

Objective: Introduction to procedures,functions & cursors.

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

At the 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.

- **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.

Procedures:

The procedure contains a header and a body.

- Header: The header contains the name of the procedure and the parameters or variables passed to the procedure.
- Body: The body contains a declaration section, execution section and an exception section similar to a general PL/SQL block.

When you want to create a procedure or function, you have to define parameters .There are three ways to pass parameters in procedure:

1. **IN parameters:** The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:** The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.

3. **INOUT parameters:** The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the 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 the 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 the value of the passed value. This example shows how we can use the 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 the SQL prompt, it produces the following result
− Square of (23): 529
PL/SQL procedure successfully completed.

Functions:

The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and function is a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

Syntax:

```
CREATE [OR REPLACE] FUNCTION function_name [parameters]
[(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 modifying an existing function.
- 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.

Example:

/*Creation of function*/

```
CREATE OR REPLACE FUNCTION totalProducts
RETURN number IS
   total number(2) := 0;
```

```
BEGIN
   SELECT count(*) into total
   FROM product_master;
   RETURN total;
END;
```

```
/*Calling a function*/
DECLARE
   p number(2);
BEGIN
   p := totalProducts();
   dbms_output.put_line('Total no. of Products: ' || p);
END;
```

Output :

```
Total no. of Products: 6
PL/SQL procedure successfully completed.
```

Cursor:

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK_ROWCOUNT** and **%BULK_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides a description of the most used attributes −

1. **%FOUND :** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2. **%NOTFOUND :** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3. **%ISOPEN :** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4. **%ROWCOUNT :** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

The following program will update the table and increase the salary of each salesman by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE sales_master
  SET sal_amt = sal_amt + 500;
  IF sql%notfound THEN
     dbms_output.put_line('no salesman selected');
  ELSIF sql%found THEN
     total_rows := sql%rowcount;
     dbms_output.put_line( total_rows || ' salesmen selected ');
  END IF;
END;
```

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

```
4 salesmen selected
PL/SQL procedure successfully completed.
```

If you check the records in sales_master table, you will find that the rows have been updated.

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

```
CURSOR client_cursor IS
  SELECT cleint_id, name, address FROM client_master;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

```
OPEN client_cursor;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

```
FETCH client_cursor INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

```
CLOSE client_cursor;
```

Example:

```
DECLARE
  c_id clients_master.client_id%type;
  c_name clients_master.name%type;
  c_addr clients_master.address%type;
  CURSOR client_cursor is
    SELECT cleint_id, name, address FROM client_master;
BEGIN
  OPEN client_cursor;
  LOOP
  FETCH client_cursor into c_id, c_name, c_addr;
    EXIT WHEN client_cursor%notfound;
    dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
  END LOOP;
  CLOSE client_cursor;
END;
```

Assignment # 10

1. creates a view that returns the sales revenues by clients. The values of the credit column are 5% of the total sales revenues.Add new column credit_limit to the client_master table for revenues.

   Suppose you need to develop a block that:

   I. Reset credit limits of all clients to zero.

   II. Fetch clients sorted by sales in descending order and gives them new credit limits from a budget of 1 million.