Oracle for Developers PL/SQL)

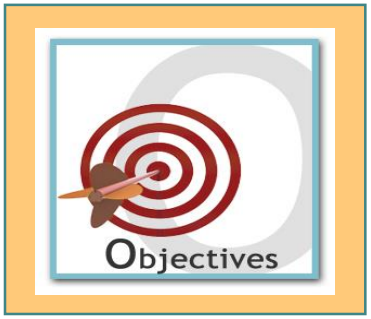
Procedures, Functions, and Packages

Capgemini

Lesson Objectives

To understand the following topics:

- Subprograms in PL/SQL
- Anonymous blocks versus Stored Subprograms
- Procedure
  - Subprogram Parameter modes
- Functions
- Packages
  - Package Specification and Package Body
- Autonomous Transactions



Introduction

A subprogram is a named block of PL/SQL

There are two types of subprograms in PL/SQL, namely: Procedures and Functions

Each subprogram has:

- A declarative part
- An executable part or body, and
- An exception handling part (which is optional)

A function is used to perform an action and return a single value

Anonymous Blocks & Stored Subprograms Comparison

Anonymous Blocks	Stored Subprograms/Named Blocks
1. Anonymous Blocks do not have names.	1. Stored subprograms are named PL/SQL blocks.
2. They are interactively executed. The block needs to be compiled every time it is run.	2. They are compiled at the time of creation and stored in the database itself. Source code is also stored in the database.
3. Only the user who created the block can use the block.	3. Necessary privileges are required to execute the block.

Procedures

A procedure is used to perform an action.

It is illegal to constrain datatypes with size.

Syntax:

```
CREATE or  REPLACE PROCEDURE  Proc_Name
   (Parameter {IN | OUT | IN OUT} datatype := value,...)
IS | AS
   Variable_Declaration ;
   Cursor_Declaration ;
   Exception_Declaration ;
BEGIN
    PL/SQL_Statements ;
EXCEPTION
    Exception_Definition ;
END Proc_Name ;
```

Subprogram Parameter Modes

IN	OUT	IN OUT
The default	Must be specified	Must be specified
Used to pass values to the procedure.	Used to return values to the caller.	Used to pass initial values to the procedure and return updated values to the caller.
Formal parameter acts like a constant.	Formal parameter acts like an uninitialized variable.	Formal parameter acts like an initialized and updated variable.
Formal parameter cannot be assigned a value.	Formal parameter cannot be used in an expression, but should be assigned a value.	Formal parameter should be assigned a value.
Actual parameter can be a constant, literal, initialized variable, or expression.	Actual parameter must be a variable.	Actual parameter must be a variable.
Actual parameter is passed by reference (a pointer to the value is passed in).	Actual parameter is passed by value (a copy of the value is passed out) unless NOCOPY is specified.	Actual parameter is passed by value (a copy of the value is passed in and out) unless NOCOPY is specified.

Example on Procedures

Example 1:

```
CREATE OR REPLACE PROCEDURE Raise_Salary
  ( s_no IN number, raise_sal IN number) IS
  v_cur_salary   number ;
   missing_salary exception;
BEGIN
    SELECT staff_sal INTO v_cur_salary FROM staff_master
    WHERE staff_code=s_no;
  IF  v_cur_salary IS NULL THEN
    RAISE  missing_salary;
  END IF ;
        UPDATE staff_master SET staff_sal = v_cur_salary + raise_sal
        WHERE staff_code =  s_no ;
EXCEPTION
          WHEN  missing_salary THEN
          INSERT into emp_audit VALUES( sno, 'salary is missing');
 END raise_salary;
```

Example on Procedures

Example 2:

```
CREATE OR REPLACE PROCEDURE
    Get_Details(s_code IN number,
    s_name OUT varchar2,s_sal OUT number ) IS
BEGIN
  SELECT staff_name, staff_sal INTO s_name, s_sal
  FROM staff_master WHERE staff_code=s_code;
EXCEPTION
          WHEN no_data_found  THEN
          INSERT into auditstaff
          VALUES( 'No employee with id ' || s_code);
          s_name := null;
          s_sal := null;
    END get_details ;
```

Executing a Procedure

Executing the Procedure from SQL*PLUS environment,

- Create a bind variables salary and name SQLPLUS by using VARIABLE command as follows:

```
variable salary number
variable name varchar2(20)
```

- After execution, use SQL*PLUS PRINT command to view results

```
EXECUTE Get_Details(100003,:Salary, :Name)
```

```
print salary
print name
```

Positional notation: Example

```
CREATE OR REPLACE PROCEDURE Create_Dept(deptno number,dname
varchar2,location varchar2) as
BEGIN
INSERT INTO  dept VALUES(deptno,dname,location);
END;
```

Executing a procedure using positional parameter notation is as follows:

```
SQL>execute  Create_Dept(90,'sales','mumbai');
```

Named notation: Example

```
CREATE OR REPLACE PROCEDURE Create_Dept(deptno number,dname
varchar2,location varchar2) as
BEGIN
INSERT INTO  dept VALUES(deptno,dname,location);
END;
```

Executing a procedure using named parameter notation is as follows:

```
SQL>execute  Create_Dept(deptno=>90,dname=>'sales',location=>'mumbai');
```

Following procedure call is also valid :

```
SQL>execute  Create_Dept(location=>'mumbai‘, deptno=>90,dname=>'sales');
```

Mixed Notation Example:

```
CREATE OR REPLACE PROCEDURE Create_Dept(deptno number,dname
varchar2,location varchar2) as
BEGIN
INSERT INTO  dept VALUES(deptno,dname,location);
END;
```

Executing a procedure using mixed parameter notation is as follows:

```
SQL>execute  Create_Dept(90, location=>'mumbai‘, dname=>'sales');
```

Functions

A function is similar to a procedure.

A function is used to compute a value.

- A function accepts one or more parameters, and returns a single value by using a return value.
- A function can return multiple values by using OUT parameters.
- A function is used as part of an expression, and can be called as

  Lvalue := Function_Name(Param1, Param2, …….).
- Functions returning a single value for a row can be used with SQL statements.

Functions

Syntax :

```
CREATE  or  REPLACE FUNCTION Func_Name(
 Param  IN|OUT|IN OUT datatype :=  value,..)  RETURN datatype1
 IS|AS
       Variable_Declaration ;
       Cursor_Declaration ;
       Exception_Declaration ;
 BEGIN
          PL/SQL_Statements ;
          RETURN Variable_Or_Value_Of_Type_Datatype1 ;
 EXCEPTION
          Exception_Definition ;
 END Func_Name ;
```

Examples on Functions

Example 1:

```
CREATE FUNCTION Crt_Dept(dno number,
  dname varchar2)  RETURN number AS
BEGIN
        INSERT into department_master
        VALUES (dno,dname) ;
                return 1 ;
EXCEPTION
        WHEN others THEN
                return 0 ;
END crt_dept ;
```

Executing a Function

Executing functions from SQL*PLUS:

- Create a bind variable Avg salary in SQLPLUS by using VARIABLE command as follows:
  - Execute the Function with EXECUTE command:
  - After execution, use SQL*PLUS PRINT command to view results.

```
variable flag number
```

```
EXECUTE :flag:=Crt_Dept(60,'Production');
```

```
PRINT flag;
```

Exceptions handling in Procedures and Functions

If procedure has no exception handler for any error, the control immediately passes out of the procedure 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.

Packages

A package is a schema object that groups all the logically related PL/SQL types, items, and subprograms.

- Packages usually have two parts, **a specification** and **a body**, although sometimes the body is unnecessary.
  - The specification (spec for short) is the interface to your applications. It declares the types, variables, constants, exceptions, cursors, and subprograms available for use.
  - The body fully defines cursors and subprograms, and so implements the spec.
- Each part is separately stored in a Data Dictionary.

Packages

Note that:

- Packages variables ~ global variables
- Functions and Procedures ~ accessible to users having access to the package
- Private Subprograms written in package body ~ not accessible to users

Packages

Syntax of Package Specification:

```
CREATE or REPLACE PACKAGE Package_Name
IS|AS
   variable_declaration ;
   cursor_declaration ;
   type_declaration
   exception_declaration
   FUNCTION Func_Name(param datatype,..) return   datatype1 ;
   PROCEDURE Proc_Name(param {IN|OUT|IN OUT}   datatype,...);
END package_name ;
```

Packages

Syntax of Package Body:

```
CREATE or REPLACE PACKAGE BODY Package_Name
IS|AS
   variable_declaration ;
   cursor_declaration ;
   type_declaration
   exception_declaration
PROCEDURE Proc_Name(param {IN|OUT|IN OUT} datatype,...} IS
BEGIN
            pl/sql_statements ;
END proc_name ;
FUNCTION Func_Name(param datatype,...) is
      BEGIN
            pl/sql_statements ;
END func_name ;
END package_name ;
```

Example of Package

Creating Package Specification

```
CREATE OR REPLACE PACKAGE Pack1 AS
        PROCEDURE Proc1;
        FUNCTION Fun1 return varchar2;
END pack1;
```

Example of Package

Creating Package Body

```
CREATE OR REPLACE PACKAGE BODY Pack1 AS
   PROCEDURE Proc1 IS
   BEGIN
            dbms_output.put_line('hi a message frm procedure');
   END Proc1;
            function Fun1 return varchar2 IS
   BEGIN
            return ('hello from fun1');
   END Fun1;
END Pack1;
```

Executing a Package

Executing Procedure from a package:

```
EXEC Pack1.Proc1
Hi a message frm procedure
```

- **Executing Function from a package:**

```
SELECT Pack1.Fun1 FROM dual;

FUN1
--------------------------------------
hello from fun1
```

Package Instantiation

Package Instantiation:

- The packaged procedures and functions have to be prefixed with package names.
- The first time a package is called, it is instantiated.

Subprograms and Ref Type Cursors

You can declare Cursor Variables as the formal parameters of Functions and Procedures.

```
CREATE OR REPLACE PACKAGE Staff_Data AS
   TYPE staffcurtyp is ref cursor return
     staff_master%rowtype;
 PROCEDURE Open_Staff_Cur(staff_cur IN OUT
  staffcurtyp);
END Staff_Data;
```

Subprograms and Ref Type Cursors

```
CREATE OR REPLACE PACKAGE BODY Staff_Data AS
PROCEDURE Open_Staff_Cur (staff_cur IN OUT staffcurtyp) IS
BEGIN
      OPEN staff_cur  for SELECT * FROM staff_master;
      end Open_Staff_Cur;
END Staff_Data;
```

Note: Cursor Variable as the formal parameter should be in IN OUT mode.

Subprograms and Ref Type Cursors

Execution in SQL*PLUS:

- Step 1: Declare a bind variable in a PL/SQL host environment of type REFCURSOR.

```
SQL> VARIABLE cv REFCURSOR
```

- Step 2: SET AUTOPRINT ON to automatically display the query results.

```
SQL> set autoprint on
```

Subprograms and Ref Type Cursors

- Step 3: Execute the package with the specified procedure along with the cursor as follows:

```
SQL> execute Staff_Data.Open_Staff_Cur(:cv);
```

Subprograms and Ref Type Cursors

Passing a Cursor Variable as IN parameter to a stored procedure:

- Step 1: Create a Package Specification

```
CREATE OR REPLACE PACKAGE StaffData AS
        TYPE cur_type is REF CURSOR;
        TYPE staffcurtyp is REF CURSOR
        return staff%rowtype;
        PROCEDURE Ret_Data (staff_cur IN OUT staffcurtyp,
        choice in number);
 END StaffData;
```

Subprograms and Ref Type Cursors

- Step 2: Create a Package Body:

```
CREATE OR REPLACE PACKAGE BODY StaffData AS
   PROCEDURE Ret_Data (staff_cur IN OUT staffcurtyp,
        choice IN number) is
BEGIN
   IF choice = 1 THEN
            OPEN staff_cur for select * FROM staff_master
              WHERE staff_dob is not null;
   ELSIF choice = 2 THEN
             OPEN staff_cur for SELECT * FROM staff_master
          WHERE staff_sal > 2500;
```

Subprograms and Ref Type Cursors

Step 2 (contd.):

```
        ELSIF choice = 3 THEN
                    OPEN staff_cur for SELECT * FROM
          staff_master WHERE dept_code = 20;
        END IF;
 END Ret_Data;
END StaffData;
```

Autonomous Transactions

Autonomous transactions are useful for implementing:

- transaction logging,
- counters, and
- other such actions, which needs to be performed independent of whether the calling transaction is committed or rolled-back

Autonomous transactions:

- are independent of the parent transaction.
- do not inherit the characteristic of the parent (calling) transaction.

Autonomous Transactions

Note that:

- Any changes made cannot be seen by the calling transaction unless they are committed.
- Rollback of the parent does not rollback the called transaction. There are no limits other than the resource limits on how many Autonomous transactions may be nested.
- Autonomous transactions must be explicitly committed or rolled-back, otherwise an error is generated.

Autonomous Transactions - Example

The following example shows how to define an Autonomous block.

```
CREATE PROCEDURE  Log_Usage ( staff_no IN number,
            msg_in IN varchar2)
IS
PRAGMA        AUTONOMOUS_TRANSACTION;
 contd.
```

Autonomous Transactions - Example

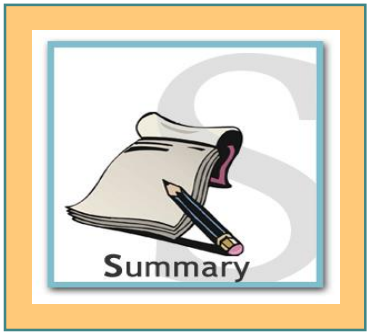
```
BEGIN
        INSERT into log1 VALUES (staff_no, msg_in);
        commit;
END LOG_USAGE;

CREATE PROCEDURE Chg_Emp
IS
BEGIN
        Log_Usage(7566,'Changing salary ');   -- ←
        UPDATE staff_master
      SET staff_sal = sal + 250
         WHERE staff_code = 100003;
END chg_emp;
```

Summary

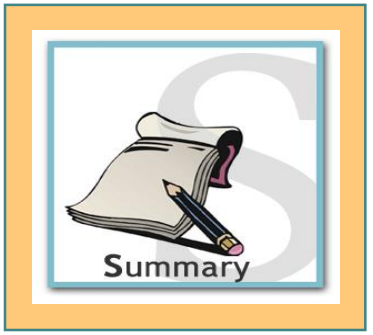
In this lesson, you have learnt:

- Subprograms in PL/SQL are named PL/SQL blocks.
- There are two types of subprograms, namely: Procedures and Functions
- Procedure is used to perform an action
  - Procedures have three subprogram parameter modes, namely: IN, OUT, and INOUT



Summary

- Functions are used to compute a value
  - A function accepts one or more parameters, and returns a single value by using a return value
  - A function can return multiple values by using OUT parameters
- Packages are schema objects that groups all the logically related PL/SQL types, items, and subprograms
  - Packages usually have two parts, a specification and a body



Review – Questions

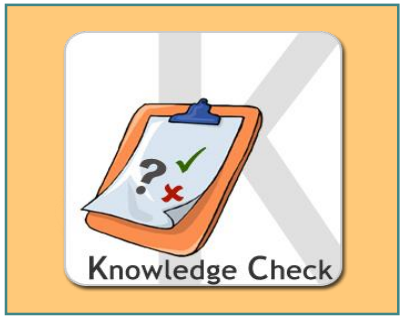
Question 1: Anonymous Blocks do not have names.

- True / False

Question 2: A function can return multiple values by using OUT parameters

- True / False

Question 3: A Package consists of "Package Specification" and "Package Body", each of them is stored in a Data Dictionary named DBMS_package.



Review – Questions

Question 4: An ___ parameter returns a value to the caller of a subprogram.

Question 5: A procedure contains two parts: ___ and ___.

Question 6: In ___ notation, the order of the parameters is not significant.