PL/SQL: PROCEDURES & FUNCTIONS

Modular code is a way to build a program from distinct parts (modules), each of which performs a specific function or task toward the program's final objective. As soon as modular code is stored on the database server, it becomes a database object, or subprogram, that is available to other program units for repeated execution. To save code to the database, the source code needs to be sent to the server so that it can be compiled into p-code and stored in the database.

Benefits of Modular Code

A PL/SQL module is any complete logical unit of work. The five types of PL/SQL modules are - **anonymous blocks** that are run with a text script, **procedures**, **functions**, **packages**, and **triggers**. Using modular code offers two main benefits: It is more reusable, and it is more manageable.

The block structure is common for all the module types. The block begins with a header (for named blocks only), which consists of the module's name and a parameter list (if used).

Anonymous Block

Anonymous blocks are much like modules, except that anonymous blocks do not have headers. The anonymous blocks have no name and thus cannot be called by another block. They are not stored in the database and must be compiled and then run each time the script is loaded.

The PL/SQL block in a subprogram is a named block that can accept parameters and that can be invoked from an application that can communicate with the Oracle database server. A subprogram can be compiled and stored in the database. This allows the programmer to reuse the program. It also allows for easier code maintenance. Subprograms are either procedures or functions.

A function that is stored in the database. It is a named PL/SQL block that can take parameters and be invoked. The significant difference is that a function is a PL/SQL block that returns a single value. Functions can accept one, many, or no parameters, but a function must have a return clause in the executable section of the function. The datatype of the return value must be declared in the header of the function. A function is not a stand-alone executable in the same way a procedure is: It must be used in some context.

```
CREATE [OR REPLACE] FUNCTION function_name (parameter list)
RETURN datatype [IS | AS]
      <local variables>
BEGIN
      <body>
      RETURN (return_value);
END;
```

The function does not necessarily have any parameters, but it must have a RETURN value whose datatype is declared in the header, and it must return values for all the varying possible execution streams. The RETURN statement does not have to appear as the last line of the main execution section, and there may be more than one RETURN statement (there should be a RETURN statement for each exception). A function may have IN, OUT, or IN OUT parameters, but we rarely see anything except IN parameters because it is bad programming practice to do otherwise.

```
CREATE OR REPLACE FUNCTION SHOW_STAFF_DETAILS
      (STAFF_ID STAFF.SID%TYPE)
RETURN VARCHAR2 AS
      V_DETAIL VARCHAR2(80);
BEGIN
      SELECT FNAME||© ©||LNAME||© : ©||DESG INTO V_DETAIL
            FROM STAFF
            WHERE SID = STAFF_ID;
      RETURN V_DETAIL;
EXCEPTION
      WHEN NO_DATA_FOUND THEN
            RETURN(©The Staff does not Exist..©);
      WHEN OTHERS THEN
            RETURN(©Error in Exceuting Procedure..©);
END;
```

A function can be made to return a boolean value.

```
CREATE OR REPLACE FUNCTION IS_VALID_SID
      (STAFF_ID IN STAFF.SID%TYPE)
RETURN BOOLEAN AS
      V_KNT NUMBER;
BEGIN
      SELECT COUNT(*) INTO V_KNT
            FROM STAFF WHERE SID = STAFF_ID;
      RETURN 1 = V_KNT;        -- if V_KNT is 1, it evaluates to TRUE
EXCEPTION
      WHEN OTHERS THEN
            RETURN FALSE;
END;
```

Invoking Functions

Functions return a single value and can be very useful in a SELECT statement.

```
SELECT SID, SHOW_STAFF_DETAILS(SID) AS STAFF_INFO
      FROM STAFF;
```

- A function may be called in a PL/SQL block.

```
DECLARE
      V_DESCRIPT VARCHAR2(80);
BEGIN
      V_DESCRIPT := SHOW_STAFF_DETAILS(&STAFF_ID);
      DBMS_OUTPUT.PUT_LINE(V_DESCRIPT);
      DBMS_OUTPUT.PUT_LINE(©.. DONE ..©);
END;
```

Important Notes...

1. When a function is created the Oracle – (a) compiles a function, (b) stores the compiled code in database, (c) stores the source code for function in database [use USER_SOURCE view's TEXT column].
2. Verify creation of function by querying USER_OBJECTS view. To know whether the created function is in **valid state** or otherwise, use the STATUS column.
3. When a function is compiled and Oracle returns the message `ªFunction created with compilation errors.°`, use SHOW ERRORS command to view the errors.
4. The errors for a function in **invalid state** [i.e., one with compilation errors] can be viewed by querying USER_ERRORS view.
5. Local variables in function definition are not preceded by DECLARE keyword.
6. It is not allowed to specify the datatype for the parameters and for the data type of RETURN clause (before AS or IS). It will result in an error.
7. A function can be dropped as like any other oracle objects.

```
DROP FUNCTION function_name;
```

8. Functions cannot use DDL statements within their body.
9. A function may contain several RETURN statements but only one of them returns the value.

A procedure is a module that performs one or more actions; it does not need to return any values. A procedure may have zero to many parameters .

```
CREATE OR REPLACE PROCEDURE procedure_name
      [(parameter[, parameter, ...])]
      AS
      [local declarations]
BEGIN
      executable statements
[EXCEPTION
      exception handlers]
END [procedure_name];
```

Every procedure has three parts: the header portion, which comes before AS (sometimes IS; they are interchangeable); the keyword, which contains the procedure name and parameter list; and the body, which is everything after the AS keyword.

```
CREATE OR REPLACE PROCEDURE HOW_MANY_PRODUCTS_BY_VENDOR
      (V_VENDOR PRODUCT.V_CODE%TYPE) AS
      V_PRODUCT_COUNT NUMBER;
BEGIN
      SELECT COUNT(P_CODE) INTO V_PRODUCT_COUNT
            FROM PRODUCT
            WHERE V_VODE = V_VENDOR;
      DBMS_OUTPUT.PUT_LINE(©Vendor ID ©||V_VENDOR||© supplies ©||
            V_PRODUCT_COUNT||© products..©);
EXCEPTION
      WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE(©Vendor ID ©||V_VENDOR||
                  © does not exist or do not supply any product..©);
END;
```

Invoking a Procedure

1. Using a PL/SQL block structure [may be named block].

```
DECLARE
      V_VENDOR VENDOR.V_CODE%TYPE := &VENDOR_NUMBER;
BEGIN
      HOW_MANY_PRODUCTS_BY_VENDOR(V_VENDOR);
      DBMS_OUTPUT.PUT_LINE(©.. DONE ..©);
END;
```

2. At SQL prompt or any procedural 3GL code.

```
EXEC HOW_MANY_PRODUCTS_BY_VENDOR(&VENDOR_NAME);
```

A procedure can become invalid if the table it is based on is deleted or changed. An invalid procedure can be recompile d using the command:

```
ALTER PROCEDURE procedure_name COMPILE ;
```

As like functions, the source code for a porcedure can be viewed by querying USER_SOURCE view. The procedure state and other details can be checked by querying USER_OBJECTS view.

Parameters are the means to pass values to and from the calling environment to the server. These are the values that are processed or returned by executing the procedure. The three types of parameter modes are IN, OUT, and IN OUT. Modes specify whether the parameter passed is read in or a receptacle for what comes out.

Mode	Description	Usage
IN	Passes a value into the program Constants, literals, expressions Cannot be changed within the program's default mode	Read-only value
OUT	Passes a value back from the program Cannot assign default values Must be a variable A value is assigned only if the program is successful	Write-only value
IN OUT	Passes values in and also sends values back	Has to be a variable

Formal & Actual Parameters

Formal parameters are the names specified in parentheses as part of a module's header. Actual parameters are the values or expressions specified in parentheses as a parameter list when the module is called. The formal parameter and the related actual parameter must be of the same or compatible datatypes .

Matching Formal & Actual Parameters

There are two methods to match actual and formal parameters: positional notation and named notation. Positional notation is simply association by position: The order of the parameters used when executing the procedure matches the order in the procedure's header. Named notation is explicit association using the symbol =>:

```
formal_parameter_name => argument_value
```

In named notation, the order does not matter. If the notations are mixed, the positional notation are listed before the named notation.
Default values can be used if a call to the program does not include a value in the parameter list.

```
CREATE OR REPLACE PROCEDURE LIST_STUDENT_NAME
      (V_SNO IN STUDENT.ROLL%TYPE, V_NAME OUT VARCHAR2) AS
BEGIN
      SELECT FNAME||LNMAE INTO V_NAME
            FROM STUDENT
            WHERE ROLL = V_SNO;
      EXCEPTION
            WHEN OTHERS THEN
                  DBMS_OUTPUT.PUT_LINE(©Error finding Roll Number: ©
                        ||V_SNO);
END;
```

Calling the **LIST_STUDENT_NAME** procedure

```
DECLARE
      V_NAME VARCHAR2(30);
BEGIN
      LIST_STUDENT_NAME(4001, V_NAME);
      DBMS_OUTPUT.PUT_LINE (©Roll 4001 : ©||V_NAME);
END;
```

Write a procedure with no parameters. The procedure should say whether the current day is a weekend or weekday. Additionally, it should tell you the user's name and the current time. It also should specify how many valid and invalid procedures are in the database.

```
CREATE OR REPLACE PROCEDURE SYSTEM_INFORMATION AS
      IS_WEEKEND NUMBER;
      VALID_P NUMBER;
      INVAL_P NUMBER;
      CURR_TM CHAR(9);
BEGIN
      SELECT COUNT(STATUS) INTO VALID_P
            FROM USER_OBJECTS
            WHERE OBJECT_TYPE = ©PROCEDURE© AND STATUS = ©VALID©;
      SELECT COUNT(STATUS) INTO INVAL_P
            FROM USER_OBJECTS
            WHERE OBJECT_TYPE = ©PROCEDURE© AND STATUS = ©INVALID©;

      IS_WEEKEND := TO_CHAR(SYSDATE, ©D©);
      CURR_TM := TO_CHAR(SYSDATE, ©HH24:MI:SS©);

      DBMS_OUTPUT.PUT_LINE(©Hello User : ©||USERNAME);
      DBMS_OUTPUT.PUT_LINE(©   Current Time : ©||CURR_TM);

      IF IS_WEEKEND IN (1, 7) THEN
            DBMS_OUTPUT.PUT_LINE(©   It is a Weekend..©);
      ELSE
            DBMS_OUTPUT.PUT_LINE(©   It is a Weekday..©);

      DBMS_OUTPUT.PUT_LINE(©   In all ©||VALID_P||© valid procedures. ©);
      DBMS_OUTPUT.PUT_LINE(© In all ©||INVAL_P||© invalid procedures. ©);
      DBMS_OUTPUT.PUT_LINE(©.. Done..©);
END;
```

Test the procedure..

```
EXEC SYSTEM_INFORMATION;
```

Try this out...

Create a table ALL_PIN_CODES to store PINCODE (6 digits numeric), AREA (alphanemeric 25 characters) and STATE (2 character codes). The valid state codes are – MH, CG, DL, TS, WB, MP, UP. Add some records to ALL_PIN_CODES.
Now, write a procedure that takes in a pin code, city, and state and inserts the values into the ALL_PIN_CODES table. It should check to see if the pin code is already in the database. If it is, an exception should be raised, and an error message should be displayed. Write an anonymous block that uses the procedure and inserts the intended pin code.