Lab 10. Procedures and Functions

Databases II

10.1 Procedures

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.

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 (this is the type you have used so far), procedures, functions, packages, and triggers. Using modular code offers two main benefits: It is more reusable, and it is more manageable.

BLOCK STRUCTURE

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). The declaration section consists of variables, cursors, and subblocks that are needed in the next section. The main part of the module is the executable section, which is where all the calculations and processing are performed. This section contains executable code such as IF-THEN-ELSE, loops, calls to other PL/SQL modules, and so on. The last section of the module is an optional exception-handling section, which is where the code to handle exceptions is placed.

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 procedure is a module that performs one or more actions; it does not need to return any values. The syntax for creating a procedure is as follows:

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

Every procedure has three parts: the header portion, which comes before AS (sometimes you see IS; they are inter- changeable); the keyword, which contains the procedure name and parameter list; and the body, which is everything after the AS keyword. The word REPLACE is optional. When REPLACE is not used in the header of the procedure, to change the code in the procedure, you must drop and then re-create the procedure. Because it is very common to change a procedure's code, especially when it is under development, it is strongly recommended that you use the OR REPLACE option.

EXAMPLE

```
CREATE OR REPLACE PROCEDURE Discount AS
 CURSOR c_group_discount IS
   SELECT distinct s.course_no, c.description
     FROM section s, enrollment e, course c
    WHERE s.section_id = e.section_id AND c.course_no = s.course_no
    GROUP BY s.course_no, c.description, e.section_id, s.section_id
    HAVING COUNT(*) >=8;
 BEGIN
   FOR r_group_discount IN c_group_discount LOOP
     UPDATE course
       SET cost = cost * .95
      WHERE course_no = r_group_discount.course_no;
     DBMS_OUTPUT.PUT_LINE ('A 5% discount has been given to '||
              r_group_discount.course_no||' '|| r_group_discount.description);
   END LOOP;
 END;
/
EXEC Discount
/
```

Query the Data Dictionary for Information on Procedures

Two main views in the data dictionary provide information on stored code. USER_OBJECTS shows you information about the objects, and USER_SOURCE shows you the text of the source code. The data dictionary also has ALL_ and DBA_ versions of these views.

Example:

```
SELECT object_name, object_type, status
  FROM user_objects
 WHERE object_name = 'name_procedure';
```

10.2 Passing Parameters into and out of Procedures

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.
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.

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

PASSING CONSTRAINTS (DATATYPE) WITH PARAMETER VALUES

Formal parameters do not require constraints in the datatype. For example, instead of specifying a constraint such as VARCHAR2(60), you just say VARCHAR2 against the parameter name in the formal parameter list. The constraint is passed with the value when a call is made.

MATCHING ACTUAL AND FORMAL 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 you mix notation, list positional notation before named notation.
Default values can be used if a call to the program does not include a value in the parameter list. Note that it makes no difference which style is used; they function similarly.

EXAMPLE

```
CREATE OR REPLACE PROCEDURE find_sname (i_student_id IN NUMBER,o_first_name OUT
          VARCHAR2, o_last_name OUT VARCHAR2) AS
  BEGIN
  SELECT first_name, last_name
    INTO o_first_name, o_last_name
    FROM student
   WHERE student_id = i_student_id;
EXCEPTION
  WHEN OTHERS
  THEN
    DBMS_OUTPUT.PUT_LINE('Error in finding student_id: '||i_student_id);
END find_sname;
```

10.3 Functions

Functions are another type of stored code and are very similar to procedures. 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 has output that needs to be assigned to a variable, or it can be used in a SELECT statement.

The syntax for creating a function is as follows:

```
CREATE [OR REPLACE] FUNCTION function_name (parameter list) RETURN datatype IS

    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 you rarely see anything except IN parameters because it is bad programming practice to do otherwise.

EXAMPLE

```
CREATE OR REPLACE FUNCTION show_description(i_course_no course.course_no%TYPE)
     RETURN varchar2 AS

  v_description varchar2(50);
  BEGIN
    SELECT description
      INTO v_description
      FROM course
     WHERE course_no = i_course_no;
    RETURN v_description;
    EXCEPTION
       WHEN NO_DATA_FOUND THEN RETURN('The Course is not in the database');
       WHEN OTHERS THEN RETURN('Error in running show_description');
     END;
```

Exercices:

1. 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.
2. Write a procedure that takes in a zip code, city, and state and inserts the values into the zip code table. It should check to see if the zip 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 your zip code.
3. Write a stored function called new_student_id that takes in no parameters and returns a student.student_id%TYPE. The value returned will be used when inserting a new student into the application. It will be derived by using the formula student_id_seq.NEXTVAL.
4. Write a stored function called zip_does_not_exist that takes in a zipcode.zip%TYPE and returns a Boolean. The function will return TRUE if the zip code passed into it does not exist. It will return a FALSE if the zip code does exist. Hint: Here's an example of how this might be used:

   ```
   DECLARE
     cons_zip CONSTANT zipcode.zip%TYPE := '&sv_zipcode';
     e_zipcode_is_not_valid EXCEPTION;
   BEGIN
     IF zipcode_does_not_exist(cons_zip) THEN
       RAISE e_zipcode_is_not_valid;
      ELSE
        -- An insert of an instructor's record which
        -- makes use of the checked zipcode might go here.
        NULL;
    END IF;
   EXCEPTION
     WHEN e_zipcode_is_not_valid THEN DBMS_OUTPUT.PUT_LINE ('Could not find
          zipcode');
   ```
5. Create a new function. For a given instructor, determine how many sections he or she is teaching. If the number is greater than or equal to 3, return a message saying that the instructor needs a vacation. Otherwise, return a message saying how many sections this instructor is teaching.