Practical – 6

Packages and Exception Handling

Packages

A **package** is a schema object that groups 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.

PL/SQL packages are schema objects that groups logically related PL/SQL types, variables and subprograms.

A package will have two mandatory parts:

- Package specification
- Package body or definition

Package Specification

The specification is the interface to the package. It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

PACKAGE SPECIFICATION:

```
CREATE [OR REPLACE] PACKAGE package_name
{ IS | AS }
   [definitions of public TYPES
   ,declarations of public variables, types, and objects
   ,declarations of exceptions
   ,pragmas
   ,declarations of cursors, procedures, and functions
   ,headers of procedures and functions]
END [package_name];
```

Example

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

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

```
Package created.
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from code outside the package.

The CREATE PACKAGE BODY Statement is used for creating the package body. The following code snippet shows the package body declaration for the *cust_sal* package created above.

PACKAGE BODY:

```
CREATE [OR REPLACE] PACKAGE BODY package_name
   { IS | AS }

   [definitions of private TYPEs
   ,declarations of private variables, types, and objects
   ,full definitions of cursors
   ,full definitions of procedures and functions]
[BEGIN
   sequence_of_statements

[EXCEPTION
   exception_handlers ] ]

END [package_name];
```

Example

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS
   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
      SELECT salary INTO c_sal
      FROM customers
      WHERE id = c_id;
      dbms_output.put_line('Salary: '|| c_sal);
   END find_sal;
END cust_sal;
/
```

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

```
Package body created.
```

Using the Package Elements

The package elements (variables, procedures or functions) are accessed with the following syntax:

```
package_name.element_name;
```

Consider, we already have created above package in our database schema; the following program uses the *find_sal* method of the *cust_sal* package:

```
DECLARE
   code customers.id%type := &cc_id;
BEGIN
   cust_sal.find_sal(code);
END;
/
```

When the above code is executed at SQL prompt, it prompt to enter customer ID and when you enter an ID, it displays corresponding salary as follows:

```
Enter value for cc_id: 1
Salary: 3000

PL/SQL procedure successfully completed.
```

Package Alter Syntax

```
ALTER PACKAGE <Package Name> COMPILE BODY;
/
```

Package Alter Code:

```
SQL>ALTER PACKAGE pkg1 COMPILE BODY;

Package body Altered.
```

Package Drop

Package Drop Syntax:

```
DROP PACKAGE <Package Name>;
```

Package Drop Code:

```
SQL>DROP PACKAGE pkg1;

Package dropped.
```

Exercise:

1	Create one package named as emp_data which has two functions get_empinfo and emp_totsal which takes one parameter emp_id and function returns the value. Execute package with two functions and display result to user.
2	Create one package named as emp_mgmt which has three procedures insert, update and fetch data to and from employee table and if data not found than handle exception otherwise execute all pl/sql statements with package.
4	Create a package which has one function elgible_for_discount that checks if customer has minimum balance > 1000 in account than give 10% of bonus to his/her account and then return value true otherwise false. Execute package and handle appropriate exceptions for package.

Exception Handling

- An error condition during a program execution is called an exception in PL/SQL. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions:

  - System-defined exceptions
    - Named Exceptions
    - Unnamed Exceptions
  - User-defined exceptions

- Using Exception Handling we can test the code and avoid it from exiting abruptly. When an exception occurs a message which explains its cause is received. PL/SQL Exception message consists of three parts.
  1)TypeofException
  2)AnErrorCode
  3) A message

Syntax:

```
DECLARE
      Declaration section
 BEGIN
       Exception section
 EXCEPTION
      WHEN ex_name1 THEN
               -Error handling statements
     WHEN ex_name2 THEN
              -Error handling statements
 WHEN Others THEN
             -Error handling statements
END;
```

System Defined Exception:

1) Named System Exceptions System exceptions are automatically raised by Oracle, when a program violates a RDBMS rule. There are some system exceptions which are raised frequently, so they are pre-defined and given a name in Oracle which are known as Named System Exceptions.

For example: NO_DATA_FOUND and ZERO_DIVIDE are called Named System exceptions. Named system exceptions are:

1) Not Declared explicitly
2)Raised implicitly when a predefined Oracle error occurs
3) caught by referencing the standard name within an exception-handling routine.

Exception	Oracle Error	SQLCODE	Description
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.
CURSOR_ALREADY_OPEN	06511	-6511	When you open a cursor that is already open.

Examples of internally defined exceptions include division by zero and out of memory. Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR. The other internal exceptions can be given names.

When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system.

Buit-In Example Code:

```
SQL> DECLARE
              temp employee%rowtype;
          BEGIN
            SELECT * INTO temp FROM employee WHERE empid=3;
      EXCEPTION
              WHEN no_data_found THEN
                 dbms_output.put_line('Table Does Not have Data');
           END;
  /
```

Table Does Not have Data
PL/SQL procedure successfully completed.

2) **Unnamed System Exceptions:**
Those system exception for which oracle does not provide a name is known as unnamed system exception. These exceptions do not occur frequently. These Exceptions have a code and an associated message.

There are two ways to handle unnamed system exceptions:

1. By using the WHEN OTHERS exception handler, or
2. By associating the exception code to a name and using it as a named Exception

We can assign a name to unnamed system exceptions using a **Pragma** called **EXCEPTION_INIT.**

EXCEPTION_INIT will associate a predefined Oracle error number to a programmer_defined exception name.

Steps to be followed to use unnamed system exceptions are
• they are raised implicitly.
• If they are not handled in WHEN others they must be handled explicitly.
• To handle the exception explicitly, they must be declared using Pragma

EXCEPTION_INIT as given above and handled referencing the user-defined exception name in the exception section.
The general syntax to declare unnamed system exception using EXCEPTION_INIT is:

Syntax:

```
DECLARE
        < Exception Name> EXCEPTION;
        PRAGMA EXCEPTION_INIT(exception Name,-20015);
BEGIN
        <SQL Statement>
        IF <condition> THEN
                RAISE <Exception Name>
        END IF;
EXCEPTION
WHEN < exception Name> THEN
        <User Defined Action to be taken>
END;
```

Example:

```
DECLARE
      myex EXCEPTION;
      PRAGMA EXCEPTION_INIT (myex,-20015);
      n NUMBER := &n;
BEGIN
      FOR i IN 1..n LOOP
      dbms_output.put_line(i);
            IF i=n THEN
                  RAISE myex;
            END IF;
      END LOOP;
EXCEPTION
      WHEN myex THEN
      dbms_output.put_line('loop is end');
END;
 /
Enter value for n: 6
old   4:  n NUMBER := &n;
new   4:  n NUMBER := 6;
1
2
3
4
5
6
loop is end

PL/SQL procedure successfully completed.
```

3) User Defined Exceptions:

Apart from system exceptions we can explicitly define exceptions based on business rules. These are known as user-defined exceptions.

Steps to be followed to use user-defined exceptions:
- They should be explicitly declared in the declaration section.
- They should be explicitly raised in the Execution Section.

• They should be handled by referencing the user-defined exception name in the exception section.

Syntax:

```
DECLARE
        <Exception Name> EXCEPTION;
BEGIN
        <SQL Statement>
        IF <condition> THEN
                RAISE <Exception Name>
        END IF;
EXCEPTION
        WHEN <Exception Name> THEN
                <User Defined Action to be Taken>
END;
```

Example:

```
DECLARE
        c_id customers.id%type := &cc_id;
        c_name  customers.name%type;
         c_addr customers.address%type;

         -- user defined exception
        ex_invalid_id  EXCEPTION;
BEGIN
         IF c_id <= 0 THEN
                RAISE ex_invalid_id;
        ELSE
                SELECT name, address INTO c_name, c_addr
                FROM customers
                WHERE id = c_id;
                 DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
                 DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
        END IF;
EXCEPTION
           WHEN ex_invalid_id THEN
              dbms_output.put_line('ID must be greater than zero!');
           WHEN no_data_found THEN
              dbms_output.put_line('No such customer!');
           WHEN others THEN
              dbms_output.put_line('Error!');
END;
/
```

Exercise:

Account (act_id, act_nm,cust_id,cur_bal)

1	Create a pl/sql block which handle divided by zero error with named system exception.
2	Create a pl/sql block which calculates total marks and percentage of students whose percentage less than 35 than raise an error. [Marks are less error message with error code -21000 use pragma_exception_init].
3	Create a pl/sql block which takes the workerid as input and if worker id is not available in table raise the exception (system defined). For the same worker id count the total number of days that he has worked on different projects. If total no_of_days is less than 50 then raise a userdefine exception which print the massage " To less work".