PLSQL

Lesson 03: Cursors, Exception handling, Procedures, Functions, Packages, Adv Packages concepts.

Lesson Objectives

To understand the following topics:

- Introduction to Cursors
- Implicit and Explicit Cursors
- Cursor attributes
- Processing Implicit Cursors and Explicit Cursors
- Error Handling
  - Predefined Exceptions
  - Numbered Exceptions
  - User Defined Exceptions
- Raising Exceptions
- Control passing to Exception Handlers

Lesson Objectives

To understand the following topics:

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

3.1: Cursors
Concept

A cursor is a “handle” or “name” for a private SQL area.

An SQL area (context area) is an area in the memory in which a parsed statement and other information for processing the statement are kept.

PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return “only one row”.

For queries that return “more than one row”, you must declare an explicit cursor.

- Thus the two types of cursors are:
  - implicit
  - explicit

3.1: Cursors
Concept

Implicit Cursor:

- The PL/SQL engine takes care of automatic processing.
- PL/SQL implicitly declares cursors for all DML statements.
- They are simple SELECT statements and are written in the BEGIN block (executable section) of the PL/SQL.
- They are easy to code, and they retrieve exactly one row

3.1: Cursors
Implicit Cursors

Processing Implicit Cursors:

- Oracle implicitly opens a cursor to process each SQL statement that is not associated with an explicitly declared cursor.
- This implicit cursor is known as SQL cursor.
  - Program cannot use the OPEN, FETCH, and CLOSE statements to control the SQL cursor. PL/SQL implicitly does those operations .
  - You can use cursor attributes to get information about the most recently executed SQL statement.
  - Implicit Cursor is used to process INSERT, UPDATE, DELETE, and single row SELECT INTO statements.

3.1: Cursors
Implicit Cursors - Example

```
BEGIN
   UPDATE dept SET dname ='Production' WHERE deptno= 50;
   IF SQL%NOTFOUND THEN
     INSERT into department_master VALUES ( 50, 'Production');
   END IF;
END;
```

```
BEGIN
  UPDATE dept SET dname ='Production' WHERE deptno = 50;
  IF SQL%ROWCOUNT = 0 THEN
     INSERT into  department_master VALUES ( 50, 'Production');
END IF;
END;
```

3.1: Cursors
Explicit Cursors

Explicit Cursor:

- The set of rows returned by a query can consist of zero, one, or multiple rows, depending on how many rows meet your search criteria.
- When a query returns multiple rows, you can explicitly declare a cursor to process the rows.
- You can declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.
- Processing has to be done by the user.

3.1: Cursors
Processing Explicit Cursors

While processing Explicit Cursors you have to perform the following four steps:

- Declare the cursor
- Open the cursor for a query
- Fetch the results into PL/SQL variables
- Close the cursor

3.1: Cursors
Processing Explicit Cursors

Declaring a Cursor:

Syntax:

```
CURSOR Cursor_Name IS Select_Statement;
```

- Any SELECT statements are legal including JOINS, UNION, and MINUS clauses.
- SELECT statement should not have an INTO clause.
- Cursor declaration can reference PL/SQL variables in the WHERE clause.
- The variables (bind variables) used in the WHERE clause must be visible at the point of the cursor.

3.1: Cursors
Processing Explicit Cursors

Opening a Cursor

- Syntax:

```
OPEN  Cursor_Name;
```

- When a cursor is opened, the following events occur:
- The values of bind variables are examined.
- The active result set is determined.
- The active result set pointer is set to the first row.

3.1: Cursors
Processing Explicit Cursors

Fetching from a Cursor

- Syntax:

```
FETCH Cursor_Name  INTO List_Of_Variables;
FETCH  Cursor_Name INTO PL/SQL _Record;
```

- The “list of variables” in the INTO clause should match the “column names list” in the SELECT clause of the CURSOR declaration, both in terms of count as well as in datatype.
- After each FETCH, the active set pointer is increased to point to the next row.
- The end of the active set can be found out by using %NOTFOUND attribute of the cursor.

3.1: Cursors
Processing Explicit Cursors

Closing a Cursor

- Syntax

```
CLOSE  Cursor_Name;
```

- Closing a Cursor frees the resources associated with the Cursor.
- You cannot FETCH from a closed Cursor.
- You cannot close an already closed Cursor.

3.1: Cursors Attributes

Cursor Attributes:

- Explicit cursor attributes return information about the execution of a multi-row query.
- When an “Explicit cursor” or a “cursor variable” is opened, the rows that satisfy the associated query are identified and form the result set.
- Rows are fetched from the result set.
- Examples: %ISOPEN, %FOUND, %NOTFOUND, %ROWCOUNT, etc.

3.1: Cursors
Types of Cursor Attributes

The different types of cursor attributes are described in brief, as follows:

%ISOPEN

%ISOPEN returns TRUE if its cursor or cursor variable is open. Otherwise it returns FALSE.

Syntax:

```
Cur_Name%ISOPEN
```

3.1: Cursors
Types of Cursor Attributes

Example:

```
      DECLARE
                        cursor c1 is
                  select_statement ;
       BEGIN
            IF c1%ISOPEN THEN
                        pl/sql_statements ;
            END IF;
        END ;
```

3.1: Cursors
Types of Cursor Attributes

%FOUND

%FOUND yields NULL after a cursor or cursor variable is opened but before the first fetch.

Thereafter, it yields:

TRUE if the last fetch has returned a row, or

FALSE if the last fetch has failed to return a row

Syntax:

```
cur_Name%FOUND
```

3.1: Cursors
Types of Cursor Attributes

Example:

```
DECLARE section;
    open c1 ;
    fetch c1 into var_list ;
IF c1%FOUND THEN
      pl/sql_statements ;
END IF ;
```

3.1: Cursors
Types of Cursor Attributes

%NOTFOUND

- %NOTFOUND is the logical opposite of %FOUND.
- %NOTFOUND yields:
  - FALSE if the last fetch has returned a row, or
  - TRUE if the last fetch has failed to return a row
- It is mostly used as an exit condition.
- Syntax:

```
cur_Name%NOTFOUND
```

3.1: Cursors
Types of Cursor Attributes

%ROWCOUNT

- %ROWCOUNT returns number of rows fetched from the cursor area using FETCH command.
- %ROWCOUNT is zeroed when its cursor or cursor variable is opened.
  - Before the first fetch, %ROWCOUNT yields 0.
  - Thereafter, it yields the number of rows fetched at that point of time.
- The number is incremented if the last FETCH has returned a row.
- Syntax:

```
cur_Name%NOTFOUND
```

3.1: Cursors
Cursor FETCH loops

They are examples of simple loop statements.

The FETCH statement should be followed by the EXIT condition to avoid infinite looping.

Condition to be checked is cursor%NOTFOUND.

Examples: LOOP .. END LOOP, WHILE LOOP, etc

3.1: Cursors
Cursor using LOOP … END LOOP:

```
DECLARE
        cursor c1 is ……..
BEGIN
       open cursor c1;  /* open the cursor and identify the active result set.*/
LOOP
       fetch c1 into  variable_list ;
       -- exit out of the loop when there are no more rows.
       /* exit is done before processing to prevent handling of null rows.*/
       EXIT WHEN C1%NOTFOUND ;
       /* Process the fetched rows using variables and  PL/SQLstatements */
END LOOP;
       -- Free resources used by the cursor
       close  c1;
       -- commit
       commit;
END;
```

3.1: Cursors
FOR Cursor LOOP

FOR Cursor Loop

```
FOR Variable in Cursor_Name
    LOOP
            Process the variables
    END LOOP;
```

You can pass parameters to the cursor in a CURSOR FOR loop.

```
FOR Variable in Cursor_Name ( PARAM1 , PARAM 2 ….)
    LOOP
            Process the variables
    END LOOP;
```

3.1: Cursors
SELECT… FOR UPDATE

SELECT … FOR UPDATE cursor:

- The method of locking records which are selected for modification, consists of two parts:
- The FOR UPDATE clause in CURSOR declaration.
- The WHERE CURRENT OF clause in an UPDATE or DELETE statement.
- Syntax: FOR UPDATE

```
CURSOR   Cursor_Name IS  SELECT  ….. FROM … WHERE   .. ORDER
BY     FOR UPDATE  [OF column names ] [ NOWAIT]
```

where column names are the names of the columns in the table against which the query is fired. The column names are optional.

3.1: Cursors
SELECT… FOR UPDATE

If the cursor is declared with a FOR UPDATE clause, the WHERE CURRENT OF clause can be used in an UPDATE or DELETE statement.

Syntax: WHERE CURRENT OF

```
WHERE CURRENT OF Cursor_Name
```

The WHERE CURRENT OF clause evaluates up to the row that was just retrieved by the cursor.

When querying multiple tables Rows in a table are locked only if the FOR UPDATE OF clause refers to a column in that table.

contd.

3.1: Cursors
SELECT... FOR UPDATE

For example: Following query locks the staff_master table but not the department_master table.

```
CURSOR C1 is SELECT staff_code, job, dname from
emp, dept WHERE emp.deptno=dept.deptno FOR
UPDATE OF sal;
```

Using primary key simulates the WHERE CURRENT OF clause but does not create any locks.

3.1: Cursors
SELECT… FOR UPDATE - Examples

To promote professors who earn more than 20000

```
    DECLARE
    CURSOR c_staff  is SELECT staff_code, staff_master.design_code
     FROM staff_master,designation_master
     WHERE design_name = ‘Professor' and staff_sal > 20000
    and staff_master.design_code =designation_master.design_code
    FOR UPDATE OF design_code NOWAIT;
    d_code  designation_master.design_code%type;
BEGIN
       SELECT design_code into d_code FROM designation_master
             WHERE design_name=‘Director’;
           FOR v_rec in c_staff
           LOOP
               UPDATE staff_master SET design_code = d_code
                WHERE current of c_staff;
           END LOOP;
  END;
```

3.2: Exception Handling
Understanding Exception Handling in PL/SQL

Error Handling:

- In PL/SQL, a warning or error condition is called an “exception”.
  - Exceptions can be internally defined (by the run-time system) or user defined.
  - Examples of internally defined exceptions:
- division by zero
- out of memory
  - Some common internal exceptions have predefined names, namely:
- ZERO_DIVIDE
- STORAGE_ERROR

3.2: Exception Handling
Understanding Exception Handling in PL/SQL

The other exceptions can be given user-defined names.

Exceptions can be defined in the declarative part of any PL/SQL block, subprogram, or package. These are user-defined exceptions.

3.2: Exception Handling
Declaring Exception

Exception is an error that is defined by the program.

- It could be an error with the data, as well.

There are three types of exceptions in Oracle:

- Predefined exceptions
- Numbered exceptions
- User defined exceptions

3.2: Exception Handling
Predefined Exception

Predefined Exceptions correspond to the most common Oracle errors.

- They are always available to the program. Hence there is no need to declare them.
- They are automatically raised by ORACLE whenever that particular error condition occurs.
- Examples: NO_DATA_FOUND,CURSOR_ALREADY_OPEN, PROGRAM_ERROR

3.2: Exception Handling
Predefined Exception - Example

In the following example, the built in exception is handled.

```
 DECLARE
        v_staffno   staff_master.staff_code%type;
        v_name      staff_master.staff_name%type;
BEGIN
        SELECT staff_name into v_name FROM staff_master
        WHERE staff_code=&v_staffno;
        dbms_output.put_line(v_name);
EXCEPTION
        WHEN  NO_DATA_FOUND THEN
        dbms_output.put_line('Not Found');
 END;
 /
```

3.2: Exception Handling
Numbered Exception

An exception name can be associated with an ORACLE error.

- This gives us the ability to trap the error specifically to ORACLE errors
- This is done with the help of “compiler directives” –
- PRAGMA EXCEPTION_INIT

3.2: Exception Handling
Numbered Exception

PRAGMA EXCEPTION_INIT:

- A PRAGMA is a compiler directive that is processed at compile time, not at run time. It is used to name an exception.
- In PL/SQL, the PRAGMA EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number.
- This arrangement lets you refer to any internal exception(error) by name, and to write a specific handler for it.
- When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.

3.2: Exception Handling
Numbered Exception (Contd.)

User defined exceptions can be named with error number between -20000 and -20999.

The naming is declared in Declaration section.

It is valid within the PL/SQL blocks only.

Syntax is:

```
PRAGMA EXCEPTION_INIT(Exception
Name,Error_Number);
```

3.2: Exception Handling
Numbered Exception - Example

A PL/SQL block to handle Numbered Exceptions

```
DECLARE
         v_bookno number := 10000008;
        child_rec_found EXCEPTION;
        PRAGMA EXCEPTION_INIT (child_rec_found, -2292);
BEGIN
              DELETE from book_master
             WHERE book_code = v_bookno;
EXCEPTION
              WHEN child_rec_found THEN
       INSERT into error_log
              VALUES (‘Book entries exist for book:' || v_bookno);
END;
```

3.2: Exception Handling
User-defined Exception

User-defined Exceptions are:

- declared in the Declaration section,
- raised in the Executable section, and
- handled in the Exception section

3.2: Exception Handling
User-defined Exception - Example

Here is an example of User Defined Exception:

```
DECLARE
      E_Balance_Not_Sufficient EXCEPTION;
      E_Comm_Too_Larage EXCEPTION;
      …
BEGIN
      NULL;
END;
```

3.2: Exception Handling
Raising Exceptions

Raising Exceptions:

Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that are associated with an Oracle error number using EXCEPTION_INIT.

Other user-defined exceptions must be raised explicitly by RAISE statements.

The syntax is:

```
RAISE  Exception_Name;
```

3.2: Exception Handling
Raising Exceptions - Example

An exception is defined and raised as shown below:

```
DECLARE
        ...
        retired_emp EXCEPTION ;
BEGIN
        pl/sql_statements ;
        if error_condition then
        RAISE retired_emp ;
        pl/sql_statements ;
EXCEPTION
        WHEN retired_emp THEN
        pl/sql_statements ;
END ;
```

Control passing to Exception Handler

Control passing to Exception Handler :

- When an exception is raised, normal execution of your PL/SQL block or subprogram stops, and control passes to its exception-handling part.
- To catch the raised exceptions, you write “exception handlers”.
- Each exception handler consists of a WHEN clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised.
- These statements complete execution of the block or subprogram, however, the control does not return to where the exception was raised. In other words, you cannot resume processing where you left off.

3.2: Exception Handling
User-defined Exception - Example

User Defined Exception Handling:

```
    DECLARE
         dup_deptno EXCEPTION;
         v_counter binary_integer;
         v_department number(2) := 50;
BEGIN
        SELECT count(*) into v_counter  FROM department_master
         WHERE dept_code=50;
     IF v_counter > 0 THEN
                RAISE dup_deptno ;
    END IF;
          INSERT into department_master
          VALUES (v_department ,'new name');
EXCEPTION
            WHEN dup_deptno THEN
               INSERT into error_log
               VALUES (‘Dept: ‘|| v_department ||' already exists”);
      END ;
```

Others Exception Handler

OTHERS Exception Handler:

- The optional OTHERS exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions that are not specifically named in the Exception section.
- A block or subprogram can have only one OTHERS handler.
- To handle a specific case within the OTHERS handler, predefined functions SQLCODE and SQLERRM are used.
- SQLCODE returns the current error code. And SQLERRM returns the current error message text.
- The values of SQLCODE and SQLERRM should be assigned to local variables before using it within a SQL statement.

3.2: Exception Handling
Others Exception Handler - Example

```
    DECLARE
      v_dummy varchar2(1);
      v_designation number(2) := 109;
    BEGIN
          SELECT 'x' into v_dummy FROM designation_master
          WHERE design_code= v_designation;
           INSERT into error_log
          VALUES (‘Designation: ' || v_designation || 'already exists');
  EXCEPTION
        WHEN no_data_found THEN
              insert into designation_master  values (v_designation,’newdesig’);
         WHEN OTHERS THEN
               Err_Num = SQLCODE;
               Err_Msg =SUBSTR( SQLERRM, 1, 100);
                INSERT into errors  VALUES( err_num, err_msg );
END ;
```

3.2: Exception Handling
Raise_Application_Error

RAISE_APPLICATION_ERROR:

The procedure RAISE_APPLICATION_ERROR lets you issue user-defined ORA- error messages from stored subprograms.

In this way, you can report errors to your application and avoid returning unhandled exceptions.

Syntax:

```
        RAISE_APPLICATION_ERROR( Error_Number,
        Error_Message);
```

where:

Error_Number is a parameter between -20000 and -20999

Error_Message is the text associated with this error

3.2: Exception Handling
Raise_Application_Error - Example

Here is an example of Raise Application Error:

```
DECLARE
        /* VARIABLES */
BEGIN
        ……..
        ……..
EXCEPTION
        WHEN OTHERS THEN
        -- Will transfer the error to the calling environment
        RAISE_APPLICATION_ERROR( -20999 ,'Contact
DBA');
END ;
```

3.3: Procedures
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.

3.3: Procedures Comparison

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.

3.3: Procedures
Procedures

A procedure is used to perform an action.

It is illegal to constrain datatypes.

Syntax:

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

3.3: Procedures
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 uninitialized 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.

3.3: Procedures
Examples

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;
```

3.3: Procedures
Examples

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 ;
```

3.3: Procedures
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)
```

Execute the procedure with EXECUTE command

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

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

```
print salary
print name
```

Example - 2

```
CREATE OR REPLACE PROCEDURE spEmp
    (nEmpno IN employee.empno%TYPE,
     nSal IN OUT NUMBER)
    AS
    nMinSal NUMBER;
    BEGIN
       SELECT min(sal) INTO nMinSal FROM employee;
       IF nSal<nMinSal
       THEN
              nSal:=nSal+nSal*.3;
      END IF;
   END spEmp;
   /
```

Example - 2

```
DECLARE
       salno NUMBER;
   BEGIN
       salno:=&salno;
       spEmp(&empno,salno);
       DBMS_OUTPUT.PUT_LINE(salno);
    END;
    /
```

3.4: Functions
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.

3.4: Functions
Functions

Syntax :

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

3.4: Functions
Examples

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 ;
```

3.4: Functions
Executing a Function

Executing functions from SQL*PLUS:

Create a bind variable Avg salary in SQLPLUS by using VARIABLE command as follows:

```
variable flag number
```

Execute the Function with EXECUTE command:

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

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

```
PRINT  flag;
```

3.4: Functions
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.

3.5: Packages
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.

3.5: Packages
Packages

Note that:

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

3.5: Packages
Packages

Syntax of Package Specification:

```
CREATE PACKAGE package_name  AS
     variable_declaration ;
     cursor_declaration ;
      FUNCTION func_name(param datatype,..) return
datatype1 ;
      PROCEDURE proc_name(param {in|out|in
out}datatype,...);
END package_name ;
```

3.5: Packages
Packages

Syntax of Package Body:

```
 CREATE PACKAGE BODY package_name AS
      variable_declaration ;
      cursor_declaration ;
PROCEDURE proc_name(param {IN|OUT|INOUT}
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 ;
```

3.5: Packages
Example

Creating Package Specification

```
CREATE OR REPLACE PACKAGE pack1 AS
        PROCEDURE proc1;
        FUNCTION fun1 return varchar2;
END pack1;
```

Example

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;
```

3.5: Packages
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
```

3.5: Packages
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.

3.6: Adv Package Concepts
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 INOUT staffcurtyp);
 END staff_data;
```

3.6: Adv Package Concepts
Subprograms and Ref Type Cursors

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

```
CREATE OR REPLACE PACKAGE BODY staff_data AS
PROCEDURE open_staff_cur (staff_cur INOUT staffcurtyp) IS
BEGIN
          OPEN staff_cur  for SELECT * FROM staff_master;
          end open_staff_cur;
END emp_data;
```

3.6: Adv Package Concepts
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
```

3.6: Adv Package Concepts
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);
```

3.6: Adv Package Concepts
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 INOUT staffcurtyp,
        choice in number);
 END staffdata;
```

3.6: Adv Package Concepts
Subprograms and Ref Type Cursors (Contd.)

Step 2: Create a Package Body:

```
CREATE OR REPLACE PACKAGE BODY staffdata AS
    PROCEDURE ret_data (staff_cur INOUT 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: Create a Package Body (Contd.)

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

SUMMARY

In this lesson, you have learnt:

- Cursor is a “handle” or “name” for a private SQL area.
- Implicit cursors are declared for queries that return only one row.
- Explicit cursors are declared for queries that return more than one row.
- Exception Handling
- User-defined Exceptions
- Predefined Exceptions
- Control passing to Exception Handler
- OTHERS exception handler
  - Association of Exception name to Oracle errors
  - RAISE_APPLICATION_ERROR procedure
  - Procedures and functions
  - Packages and Adv Packages concepts.

Review Question

Question 1: %COUNT returns number of rows fetched from the cursor area by using FETCH command.

- True / False

Question 2: Implicit SQL cursor is opened or closed by the program.

- True / False

Question 3: PL/SQL provides a shortcut via a ___ Loop, which implicitly handles the cursor processing.

Review Question

Question 4: The procedure ___ lets you issue user-defined ORA-error messages from stored subprograms

Question 5: The ___ tells the compiler to associate an exception name with an Oracle error number.

Question 6: ___ returns the current error code. And ___ returns the current error message text.

Review Question

Question 7: Anonymous Blocks do not have names.

- True / False

Question 8: A function can return multiple values by using OUT parameters

- True / False

Question 9: A Package consists of "Package Specification" and "Package Body", each of them is stored in a Data Dictionary named DBMS_package.