14. Demonstrate Cursors, Exception and Composite Data Types in PL SQL.

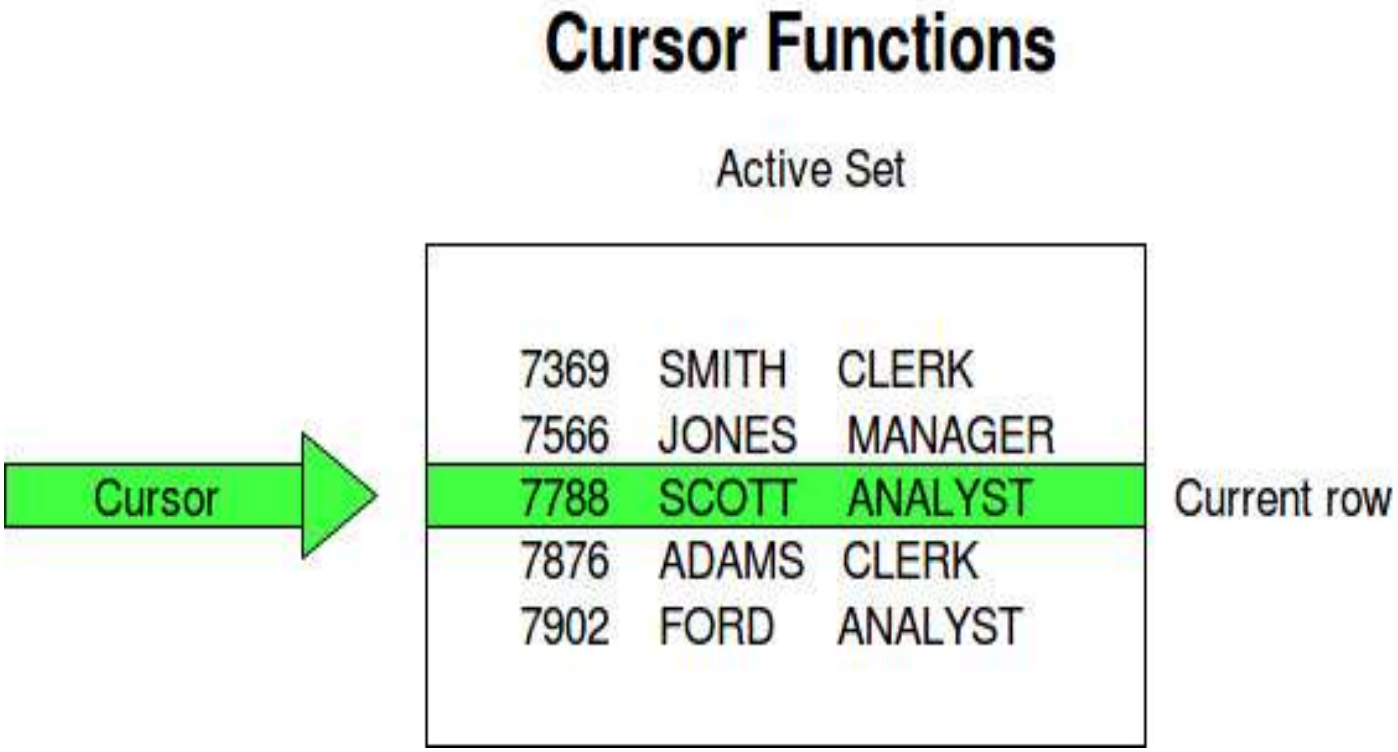
Cursors

- When an SQL statement is processed, Oracle creates a memory area known as context area. Also known as temporary memory area or private memory area.
- It contains all the information needed for processing the SQL statement.

Cursors

- A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.
- The major function of a cursor is to retrieve data, one row at a time, from a result set.

Cursors


Cursor Functions
Active Set
7369 SMITH CLERK
7566 JONES MANAGER
7788 SCOTT ANALYST
7876 ADAMS CLERK
7902 FORD ANALYST
Cursor
Current row

Cursors

There are two types of cursors

Implicit cursors

- Whenever Oracle executes an SQL statement such as SELECT , INSERT , UPDATE , and DELETE , it automatically creates an implicit cursor.
- Oracle internally manages the whole execution cycle of implicit cursors and reveals only the cursor's information and statuses such as SQL%ROWCOUNT, SQL%ISOPEN, SQL%FOUND, and SQL%NOTFOUND.

Cursors

- **Explicit cursors**
- Explicit cursors are programmer-defined cursors for gaining more control over the context area.
- An explicit cursor should be defined in the declaration section of the PL/SQL Block.
- It is created on a SELECT Statement which returns more than one row.

Explicit cursors

There are four steps in using an Explicit Cursor.

1. **DECLARE** the cursor in the Declaration section.
2. **OPEN** the cursor in the Execution Section.
3. **FETCH** the data from the cursor into PL/SQL variables or records in the Execution Section.
4. **CLOSE** the cursor in the Execution Section before you end the PL/SQL Block.

Explicit cursors

1. **Declare a cursor**

   Before using an explicit cursor, you must declare it in the declaration section of pl/sql block

   Syntax:

   CURSOR <cursor_name> **IS** <SELECT_statement>;

   Example:

   CURSOR c1 **IS** **SELECT** eno, ename, sal from emp;

Explicit cursors

2. open a cursor

Before start fetching rows from the cursor, you must open it.

Syntax:

OPEN cursor_name;

- **Example**: **OPEN** c1;

Explicit cursors

3. Fetch rows from cursor

- This statement is used after declaring and opening your cursor. The FETCH statement places the contents of the current row into variables.

  Syntax

  FETCH cursor_name **INTO** variable_list;

Explicit cursors

4.close cursor

- CLOSE statement is used to close the cursor once you have finished using it.

  Syntax

  CLOSE cursor_name;

- **Example: CLOSE** c1;

Explicit cursors

Using cursor in a PL/SQL block

```
DECLARE
variables;
 create a cursor;
BEGIN
OPEN cursor;
FETCH cursor;
process the records;
CLOSE cursor;
END;
```

- Pl/SQL BLOCK to Display the employee no, name , salary using cursor.

```
DECLARE
v_eno   emp.eno  %type;
v_ename   emp.ename  %type;
v_sal   emp.sal   %type;
CURSOR C1  IS  SELECT eno, ename, sal  FROM  emp;
BEGIN
OPEN C1;
LOOP
FETCH C1 INTO v_eno , v_ename, v_sal ;
EXIT  WHEN C1  %  NOTFOUND;
dbms_output.put_line( v_eno|| '   ' || v_ename || '   ' || v_sal );
END LOOP;
CLOSE C1;
END;
  /
```

Explicit Cursor Attributes

- A cursor has four attributes which you can reference in the following format.
- **cursor_name % ATTRIBUTE**

1) % ISOPEN
2) % FOUND
3) % NOTFOUND
4) % ROWCOUNT

Explicit Cursor Attributes

% FOUND

- Returns **TRUE** if a record was fetched successfully.
- **FALSE** if no row is fetched.
- **INVALID_CURSOR** if the cursor is not opened

Explicit Cursor Attributes

% NOTFOUND

This attribute has values:

- **FALSE** if a record was fetched successfully.
- **TRUE** if no row is returned.
- **INVALID_CURSOR** if the cursor is not opened.

Explicit Cursor Attributes

% ROWCOUNT

- This attribute returns the **number of rows** fetched from the cursor.
- If the cursor is not opened, this attribute returns INVALID_CURSOR.

% ISOPEN

- This attribute is **TRUE** if the cursor is open or **FALSE** if it is not.

- PL/SQL BLOCK to display the employee name using cursor.

```
DECLARE
v_ename   emp.ename  %type;
CURSOR C1  IS  SELECT ename  FROM  emp;
BEGIN
OPEN C1;
If  C1 %ISOPEN = TRUE THEN
dbms_output.put_line(' cursor is opened');
end if;
LOOP
FETCH  C1  INTO v_ename;
EXIT  WHEN C1  %  NOTFOUND;
dbms_output.put_line( 'emp name  is ' || v_ename  );
dbms_output.put_line( 'rows found is '  || C1%rowcount);
END LOOP;
CLOSE C1;
END;
```

Implicit Cursor

- update the salary of each employee in emp table i.e. increase the salary of each employee by 500 and use the **SQL % ROWCOUNT** attribute to determine the number of rows affected.

Implicit Cursor

```
DECLARE
v_total  NUMBER(3);
BEGIN
UPDATE emp SET sal = sal + 500;
IF SQL % NOTFOUND THEN
dbms_output.put_line('NO EMPLOYEES UPDATED');
ELSIF SQL % FOUND THEN
v_total :=  SQL  %ROWCOUNT;
dbms_output.put_line(v_total  || ' EMPLOYEES  SAL
   UPDATED');
END IF;
END;
```

Exception

- An exception is an error condition during a program execution.
- An exception is an error which disrupts the normal flow of program execution.
- PL/SQL provides us the exception block which raises the exception thus helping the programmer to find out the fault and resolve it.

Exception

- There are two types of exceptions defined in PL/SQL

1. User defined exception.
2. System defined /predefined exceptions.

Syntax of the Exception-handling section:

```
BEGIN
----------set of stmts;
EXCEPTION
      WHEN  e1 THEN
            exception_handler1
       WHEN  e2 THEN
            exception_handler2
      WHEN  OTHERS THEN
            other_exception_handler
END;
```

Exception

- A PL/SQL block can have an exception-handling section, which can have one or more exception handlers.
- The code that you write to handle exceptions is called an exception handler.
- When an exception occurs in the executable section(begin), the execution of the current block stops, and control transfers to the exception handling section.

Exception

- If the exception e1 occurred, the exception_handler1 runs.

- If the exception e2 occurred, the exception_handler2 executes.

- In case any other exception arises, then the other_exception_handler runs.

Predefined Exceptions

- PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program.
- Some of the predefined are
- NO_DATA_FOUND
- TOO_MANY_ROWS
- VALUE_ERROR
- ZERO_DIVIDE

Predefined Exceptions

NO_DATA_FOUND:

- It is raised WHEN a SELECT- INTO statement returns *no* rows.

Predefined Exceptions

- Display the employee name given the employee number.

```
DECLARE
        v_eno   emp.eno %type := 8;
        v_ename   emp.ename %type;
 BEGIN
      SELECT eno, ename INTO v_eno, v_ename FROM emp
      WHERE eno = v_eno;
      DBMS_OUTPUT.PUT_LINE ('Name: '|| v_ename);
EXCEPTION
WHEN NO_DATA_FOUND THEN
dbms_output.put_line('No such employee!');
WHEN OTHERS THEN
dbms_output.put_line('Error!');
END;
/
```

Predefined Exceptions

ZERO_DIVIDE

- It is raised when an attempt is made to divide a number by zero.

PL/SQL program to perform division of two numbers using EXCEPTION

```
DECLARE
   a NUMBER  := 10;
   b  NUMBER := 0;
   result  NUMBER;
BEGIN
   result := a/b;
 DBMS_OUTPUT.PUT_LINE('THE RESULT AFTER DIVISION IS'||result );
EXCEPTION
   WHEN zero_divide THEN
      dbms_output.put_line(' division  by zero is not possible ');
      dbms_output.put_line('the value of a is '||a);
      dbms_output.put_line('the value of b is '||b);
END;
```

User-defined Exceptions

- PL/SQL facilitates the users to define their own exceptions according to the need of the program.
- A user-defined exception can be raised explicitly, using either a **RAISE** statement

Syntax for User Defined Exception

```
DECLARE
    exception_name  EXCEPTION;
BEGIN
   IF   condition THEN
     RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

PL/SQL Program to get employee name and salary using eid.

```
DECLARE
        V_eno  Number:= 0;
        V_ename   emp.ename  %type;
        V_sal  emp.sal  %type;
        -- user defined exception
         myexcep  EXCEPTION;
BEGIN
         IF V_eno <= 0 THEN
           RAISE myexcep;
         ELSE
           SELECT  ename, sal INTO  V_ename, v_sal  FROM emp   WHERE eno =
    V_eno;
           DBMS_OUTPUT.PUT_LINE ('Name: '||  V_ename);
           DBMS_OUTPUT.PUT_LINE ('salary: ' || v_sal);
  END IF;
EXCEPTION
      WHEN myexcep THEN
      dbms_output.put_line('ID must be greater than zero!');
      WHEN others THEN
      dbms_output.put_line('Error!');
END;
```

Composite Data types

Oracle supports the following composite data types:

- PL/SQL collections.
    1. varray
    2. Nested table
    3. Associative array (index-by table)
- PL/SQL Records

Composite Data types

- **Collection** is an ordered group of logically related elements.
- In a collection, the elements are of same datatype.
- To access element, we use its index with variable name.

varray

- **VARRAY** stands for the variable-sized array.
- Varray is used to store an ordered collection of data

1. Declare a VARRAY type

syntax:

TYPE <type_name> **IS VARRAY**(size) **OF** <datatype> ;

- **EX**: TYPE array_type IS VARRAY(7) OF VARCHAR2(20);

varray

2. Declare and initialize VARRAY variables

<variablename> <type_name> := <type_name>(values);

Ex: **days** array_type := array_type(‘mon’, ‘tues’);

3. To access an element

Days(n);

- n is the index of the element, which begins with 1 and ends with the max_element.

Varray Example:

```
DECLARE
    TYPE  array_type  IS VARRAY(7) OF VARCHAR2(20);
    days  array_type  :=  array_type('mon', 'tues');
 BEGIN
      dbms_output.put_line('day- 1  is '| |day(1));
      dbms_output.put_line('day- 2 is '| |day(2));
      dbms_output.put_line('total no of elements '||
   day.count);
END;
/
```

output:

```
day- 1 is mon
day- 2 is tues
total no of elements  2
```

Nested Table

- A **nested table** is like a one-dimensional structure which is unbounded in nature
- It can hold any number of elements.
- A nested table differs from an array in the following aspects –
- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.

Nested Table Example

```
DECLARE
    TYPE  my_nesttab  IS TABLE  OF NUMBER;
 v_nt  my_nesttab := my_nesttab(3,6,9,12,15,18);
 BEGIN
dbms_output.put_line(
            'value at index 1 is '||v_nt(1));
END;
/
```

O/P: value at index 1 is 3.

Nested Table Example

```
DECLARE
    TYPE  my_nesttab  IS  TABLE  OF NUMBER;
    v_nt  my_nesttab := my_nesttab(3,6,9,12,15,18);
 BEGIN
         FOR i IN 1.. V_nt.count
        LOOP
        dbms_output.put_line(
       'value at index ' || i || ' is ' ||  v_nt(i));
       END LOOP;
END;
/
```

OUTPUT:

value at index 1 is 3
value at index 2 is 6 …….

Associative array

- An **associative array**(also called index-by table) holds the elements of same datatype as a set of **key-value** pairs.
- It is unbounded collection which means that can hold any number of elements.
- Each **key** is unique and is used to refer the corresponding **value**. The key can be either an integer or a string.

Associative array Example

```
DECLARE
    TYPE  books  IS   TABLE  OF NUMBER
                      INDEX BY VARCHAR2(20);
     Isbn books;
 BEGIN
       isbn('oracle')  := 101;
       isbn('mysql')  := 102;
dbms_output.put_line(
              'value is '|| isbn('oracle') );
END;
/
```

O/P: value is 101

Associative array Example

```
DECLARE
    TYPE  books  IS TABLE  OF NUMBER   INDEX  BY VARCHAR2(20);
     isbn  books;
     flag varchar2(20);
 BEGIN
       isbn('oracle')  := 101;
       isbn('mysql')  := 102;
       isbn('db2')  := 103;
       flag := isbn.FIRST;
      WHILE flag IS NOT NULL
      LOOP
     dbms_output.put_line( 'key  is '|| flag||' value stored  ' ||
    isbn(flag) );
     Flag  : = isbn.NEXT(flag);
      END LOOP;
END;
/
```

PL/SQL RECORDS

- A **record** is a data structure that can hold data items of different kinds.
- A Record consist of different fields, similar to a row of a database table.
- PL/SQL can handle the following types of records
  1. Table-based
  2. Cursor-based records
  3. User-defined records

Table-Based Records

%ROWTYPE

- Lets you declare a record variable that represents a row in a table.
- All the fields from referenced table are inherited to record variable.
- For each column in the referenced table, the record variable has a column with same name and datatype.
- To reference a field in the record, use **recordname.fieldname**

Find employee name and salary using PL/SQL block

```
DECLARE
    v_ename varchar2(20);
    v_salary Number;
BEGIN
    SELECT  ename , sal
    INTO   v_ename, v_salary
    FROM emp WHERE eno = 101;
    DBMS_OUTPUT.PUT_LINE ('Emp name is' ||
     v_ename || ' And salary is' || v_salary);
END;
```

Table-Based Records - Example

Find employee name and salary using PL/SQL block

```
DECLARE
        v_emp_record   emp %ROWTYPE;
BEGIN
        SELECT  * INTO v_emp_record
            FROM emp WHERE eno = 101;

        DBMS_OUTPUT.PUT_LINE (‘ Emp name is  ’ ||
                                        v_emp_record.ename);
        DBMS_OUTPUT.PUT_LINE (‘ salary  is  ’ ||
                                        v_emp_record.sal);
END;
```

Table-Based Records – Example2

```
-- CREATE TABLE EMP2 AS SELECT * FROM EMP
   WHERE 1=2;

DECLARE
      V_EMP_RECORD  emp %ROWTYPE;
BEGIN
      SELECT  * INTO v_emp_record
          FROM emp WHERE eno = 101;
       INSERT INTO EMP2 VALUES V_EMP_RECORD;
END;
```

Table-Based Records – Example3

```
DECLARE
      v_emp_record  emp %ROWTYPE;
BEGIN
     SELECT * INTO v_emp_record  FROM emp WHERE eno = 101;
      v_emp_record.SAL :=  v_emp_record.SAL+100;
   UPDATE EMP SET ROW = V_emp_record WHERE
   eno = 101;
END;
/
```

Cursor-based records

```
DECLARE
        CURSOR C1  IS  SELECT eno, ename, sal  FROM  emp;
        Emp_record  C1 % ROWTYPE;
BEGIN
      OPEN C1;
      loop
      FETCH C1 INTO  Emp_record;
      EXIT  WHEN C1  %  NOTFOUND;
     dbms_output.put_line( ' EMP NO =  ' || Emp_record. eno);
     dbms_output.put_line( ' NAME =  ' || Emp_record.ename);
     dbms_output.put_line( ' SALARY =  ' || Emp_record.sal);
    end loop;
    CLOSE C1;
END;
```

```
DECLARE
v_eno   emp.eno  %type;
v_ename   emp.ename  %type;
v_sal   emp.sal   %type;
CURSOR C1  IS  SELECT eno, ename, sal  FROM  emp;
BEGIN
OPEN C1;
LOOP
FETCH C1 INTO v_eno , v_ename, v_sal ;
EXIT  WHEN C1  %  NOTFOUND;
dbms_output.put_line( v_eno|| '   ' || v_ename || '   ' || v_sal );
END LOOP;
CLOSE C1;
END;
  /
```

User-Defined Records

- PL/SQL provides a user-defined record type that allows you to define the record structure.
- These records consist of different fields.

User-Defined Records

Defining a Record

TYPE <type_name> IS RECORD

(column1 datatype1 ,

column2 datatype2 ,

…….. columnN datatypeN);

Variable syntax:

- recordvariable-name type_name;

User defined Records– Example1

```
DECLARE
        TYPE emp_rec_type IS RECORD
        (empname   varchar2(100), empsal  NUMBER);
       -- variable declaration
       emp1  emp_rec_type;
BEGIN
        Emp1.empname := ' siva ';
        Emp1.empsal := 1000;
  dbms_output.put_line( ' EMPNAME =  ' ||Emp1.empname );
  dbms_output.put_line( ' SALARY =  ' || Emp1.empsal);
END;
```

User defined Records– Example2

```
DECLARE
        TYPE emp_rec_type IS RECORD
        (empname   varchar2(100), empsal  NUMBER);
       emp1  emp_rec_type;
BEGIN
        SELECT  ename, sal  INTO emp1
            FROM emp WHERE eno = 101;
 dbms_output.put_line(' EMPNAME = ‘||Emp1.empname);
  dbms_output.put_line( ' SALARY =  ' || Emp1.empsal);
END;
```