CSE – 302
Database Management System Sessional

PL/SQL

(PROCEDURAL STRUCTURED QUERY LANGUAGE)

PL/SQL

- PL/SQL stands for Procedural Language extension of SQL
- PL/SQL is a combination of SQL along with the procedural features of programming languages
- It was developed by Oracle Corporation in the early 90’s to enhance the capabilities of SQL

A SIMPLE PL/SQL BLOCK

PL/SQL Block consists of three sections:

- The Declaration section (optional)
- The Execution section (mandatory)
- The Exception Handling (or Error) section (optional)

PL/SQL BLOCK

-
  ```
  [DECLARE]
      /* Variable declaration */
  BEGIN
      /* Program Execution */
  [EXCEPTION]
      /* Exception handling */
  END;
  ```

DECLARATIVE SECTION

- Identified by the *DECLARE* keyword
- Used to define variables and constants referenced in the block
- Variable:
  - Reserve a temporary storage area in memory
  - Manipulated without accessing a physical storage medium
- Constant:
  - Its assigned value doesn‟t change during execution

EXECUTABLE SECTION

- Identified by the *BEGIN* keyword
  - Mandatory
  - Can consist of several SQL and/or PL/SQL statements
- Used to access & manipulate data within the block

EXCEPTION-HANDLING SECTION

- Identified by the EXCEPTION keyword
- Used to display messages or identify other actions to be taken when an error occurs
- Addresses errors that occur during a statement's execution
- Examples: No rows returned or divide by zero errors

END KEYWORD

- Used to close a PL/SQL block
- Always followed by a semicolon

Example

```
BEGIN
    DBMS_OUTPUT.put_line ('Hello World!');
END;
```

The classic “Hello World!” block contains an executable section that calls the DBMS_OUTPUT.PUT_LINE procedure to display text on the screen

PL/SQL PLACEHOLDERS

- Placeholders are temporary storage area
- Can be any of **variables**, **constants** and **records**
- Used to manipulate data during the execution of a PL SQL block
- Placeholders can be defined with a name and a datatype.

  Few of the datatypes used to define placeholders are - Number (n,m) , Char (n) , Varchar2 (n) , Date , Long , Long raw, Raw, Blob, Clob, Nclob, Bfile

PL/SQL VARIABLES

- These are placeholders that store the values that can change through the PL/SQL Block

- Syntax:

```
variable_name  datatype  [NOT NULL := value ];
```

PL/SQL VARIABLES

- Syntax:
  variable_name datatype [NOT NULL := value];

- Each variable declaration is a separate statement and must be terminated by a semicolon

- When a variable is specified as NOT NULL, you **MUST** initialize the variable when it is declared

- Non-numeric data types must be enclosed in single quotation marks ' '

PL/SQL VARIABLES

- The classic “Hello World!” block contains an executable section that calls the DBMS_OUTPUT.PUT_LINE procedure to display text on the screen:

```
DECLARE
l_message VARCHAR2 (100) := 'Hello World!';
 BEGIN
 DBMS_OUTPUT.put_line(l_message);
 END;
```

EXAMPLE

- The following example block demonstrates the PL/SQL ability to nest blocks within blocks as well as the use of the *concatenation* operator (||) to join together multiple strings.

```
DECLARE
    l_message VARCHAR2 (100) :=
 'Hello';
    BEGIN
    DECLARE
              l_message2 VARCHAR2 (100) := l_message || '
     World!';  BEGIN
              DBMS_OUTPUT.put_line
     (l_message2);  END;
 END;
```

Types of Blocks

- ❑ Procedure
- ❑ Function
- ❑ Anonymous block

- All the blocks I have shown you so far are “anonymous”—they have no names. If using anonymous blocks were the only way you could organize your statements, it would be very hard to use PL/SQL to build a large, complex application.

Procedure

- Also called “Stored Procedures”
- Named block
- Can process several variables
- Returns no values
- Interacts with application program using IN, OUT, or IN OUT parameters

Procedure

Syntax

```
CREATE [OR REPLACE] PROCEDURE name
    [( argument [IN|OUT|IN OUT]
    datatype

 AS
   /* declaration section */
 BEGIN
   /* executable section - required */
 EXCEPTION
  /* error handling statements */
 END;
```

```
CREATE OR REPLACE PROCEDURE
hello_world
AS
l_message VARCHAR2 (100) := 'Hello
World!';

BEGIN
DBMS_OUTPUT.put_line (l_message);
END;
```

Now, you can call your own subprogram inside a PL/SQL block:

```
BEGIN
hello_world;
```

Parameter Modes in PL/SQL Subprograms

IN

- An IN parameter lets you pass a value to the subprogram.
- It is a read-only parameter.
- Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value.
- You can pass a constant, literal, initialized variable, or expression as an IN parameter.
- You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call.

Parameter Modes in PL/SQL Subprograms (Contd.)

OUT

- An OUT parameter returns a value to the calling program.
- Inside the subprogram, an OUT parameter acts like a variable.
- You can change its value and reference the value after assigning it.
- **The actual parameter must be variable and it is passed by value**.

Parameter Modes in PL/SQL Subprograms (Contd.)

IN OUT

- An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read.
- The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression.

IN & OUT Mode Example 1

This program finds the minimum of two values, here procedure takes two numbers using IN mode and returns their minimum using OUT parameters.

```
CREATE PROCEDURE findMin(x IN number, y IN number, z OUT number)
AS
BEGIN
        IF x < y THEN
        z:= x;
        ELSE z:= y;
        END IF;
END;

DECLARE
a number;
b number;
c number;
  BEGIN
        a:= 23; b:= 45;
        findMin(a, b, c);
       dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
```

IN & OUT Mode Example 2

- This procedure computes the square of value of a passed value. This example shows how we can use same parameter to accept a value and then return another result.

```
DECLARE
      a number;
PROCEDURE squareNum(x IN OUT number)
AS
BEGIN
      x := x * x;
 END;

 BEGIN
      a:= 23;
       squareNum(a);
      dbms_output.put_line(' Square of (23): ' || a);
 END;
```

Procedure

An approach is to analyze the procedure and identify which parts stay the same when the message needs to change and which parts change then pass the changing parts as parameters and have a single procedure that can be used under different circumstances.

```
CREATE OR REPLACE PROCEDURE
hello_place (place_in IN VARCHAR2)
AS
 l_message VARCHAR2 (100);
BEGIN
 l_message := 'Hello ' || place_in;
 DBMS_OUTPUT.put_line(l_message);
```

```
BEGIN
hello_place ('World');
hello_place ('Universe');
END;
```

FUNCTION

- Named block that is stored on the Oracle9i server
- Accepts zero or more input parameters
- Returns one value

BASIC SYNTAX:

```
CREATE [OR REPLACE] FUNCTION name
[( argument datatype
[{, argument datatype}] )]  RETURN
datatype
AS
/* declaration section */  BEGIN
/* executable section - required */  EXCEPTION
/* error handling statements */
```

FUNCTION

```
CREATE OR REPLACE FUNCTION
hello_message (place_in IN VARCHAR2)  RETURN
VARCHAR2
AS  BEGIN
RETURN 'Hello ' || place_in;
END;
```

- The type of program is now FUNCTION, not PROCEDURE.
- The subprogram name now describes the data being returned, not the action being taken.
- The body or implementation of the subprogram now contains a RETURN clause that

constructs the message and passes it back to the calling block.

- The RETURN clause after the parameter list sets the type of data returned by the function

FUNCTION

Now, call the function to retrieve the message and assign it to a variable:

```
DECLARE
l_message VARCHAR2 (100);
BEGIN
 l_message := hello_message ('Universe');
END;
```

Running SQL onside PL/SQL Blocks

Suppose, for example, that thereis a table named employees, Now we can then see the last name of the employee with ID 138, as follows:

```
SELECT last_name
FROM employees
WHERE employee_id = '138';
```

Now we need to run this same query inside my PL/SQL block and display the name.

```
DECLARE
 l_name employees.last_name%TYPE;
BEGIN
SELECT last_name INTO l_name FROM employees
WHERE employee_id = '138';
DBMS_OUTPUT.put_line (l_name);
END;
```

ASSIGNING VALUE TO A VARIABLE

- We can assign values to variables in the two ways

  1) Assign values to variables:

  ```
  variable_name:= value;
  ```

  2) Assign values to variables directly from the database columns by using a SELECT.. INTO statement.

  SELECT *column_name*

  INTO *variable_name* FROM *table_name* [WHERE condition];

Example:

The below program will get the salary of an employee with id 'E_001' and display it on the screen.

```
DECLARE
    var_salary number(6);
    var_emp_id varchar2(50) := 'E_001';
BEGIN
    SELECT salary  INTO var_salary  FROM employee WHERE
    emp_id = var_emp_id;
    dbms_output.put_line(var_salary);
    dbms_output.put_line ('The employee ' || var_emp_id || ' has
    salary ' || var_salary);
END;
```

SCOPE OF PL/SQL VARIABLES

- PL/SQL allows the nesting of Blocks within Blocks

- Based on their declaration we can classify variables into two types.

  Local variables - These are declared in a inner block and cannot be referenced by outside Blocks.

  Global variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.

SCOPE OF PL/SQL VARIABLES

```
DECLARE
    var_num1 number (10);
    BEGIN
    var_num1 := 100;
    DECLARE
    var_mult number (10);
    BEGIN
    var_mult := var_num1 * 100;
    END;
END;
```

SCOPE OF PL/SQL VARIABLES

```
DECLARE
	var_num1 number(10);
BEGIN
	var_num1 := 100;
	DECLARE
	var_mult number (10);
	BEGIN
	var_mult := var_num1 * 100;
	END;
	var_mult := 900;
END;
```

ORA-06550: line 12, column 1: PLS-00201: identifier 'VAR_MULT' must be declared

PL/SQL CONSTANTS

constant_name CONSTANT datatype := VALUE;

- *constant_name* is the name of the constant i.e. similar to a variable name.
- The word *CONSTANT* is a reserved word and ensures that the value does not change.
- *VALUE* - It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

PL/SQL RECORDS

- Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc.
- Each scalar data types in the record holds a value.
- A record can be visualized as a row of data. It can contain all the contents of a row.

PL/SQL RECORDS

- General Syntax:

  TYPE record_type_name

  IS

  RECORD (first_col_name column_datatype, second_col_name column_datatype, ...);

- *record_type_name* – it is the name of the composite type you want to define.
- *first_col_name, second_col_name, etc.,- it is the* names the fields/columns within the record.
- *column_datatype* defines the scalar datatype of the fields.

PL/SQL RECORDS DECLARATION

```
DECLARE
TYPE employee_type
IS
RECORD (
        employee_id number(5),
        employee_first_name  varchar2(25),
        employee_last_name  employee.last_name%type,
        employee_dept employee.dept%type );

        employee_salary employee.salary%type;
        employee_rec employee_type;
```

PL/SQL RECORDS DECLARATION

- If all the fields of a record are based on the columns of a table, we can declare the record as follows:

```
record_name   table_name%ROWTYPE;
```

Example,

```
DECLARE
employee_rec   employee%ROWTYPE;
```

CONDITIONAL STATEMENTS – IF..ELSE

```
IF condition 1
THEN
    statement 1;
    statement 2;
ELSIF condtion2
THEN
   statement 3;
ELSE
   statement 4;
END IF;
```

CONDITIONAL STATEMENTS – IF..ELSE

```
DECLARE
price number(8);
outputString varchar2(100);

BEGIN
SELECT price INTO price FROM FOOD WHERE Food_id = 'F_002' ;

IF price < 100 THEN
    outputString := 'Price value is less than 100' ;
ELSIF price > 700 THEN
    outputString := 'Price value is greater than 700' ;
ELSE
    outputString := 'Price value is between 100 and 700' ;
END IF;

dbms_output.put_line(outputString);
END;
```

CONDITIONAL STATEMENTS – NESTED IF

```
IF condition1 THEN
     statement1;
ELSE
    IF condition2 THEN
           statement2;
    ELSIF condition3 THEN
    statement3;
     END IF;
END IF;
```

CONDITIONAL STATEMENTS – NESTED IF

```
DECLARE
price number(8);
outputString varchar2(100);

BEGIN
SELECT price INTO price FROM FOOD WHERE Food_id = 'F_002' ;

IF price < 100 THEN
     outputString := 'Price value is less than 100' ;
ELSE
       IF price < 750 THEN
            outputString := 'Price value is less than 750' ;
       ELSIF price = 750 THEN
            outputString := 'Price value is 750' ;
       END IF;
END IF;
dbms_output.put_line(outputString);
END;
```

ITERATIVE STATEMENTS - LOOP

- There are three types of loops in PL/SQL:
  - Simple Loop
  - While Loop
  - For Loop

SIMPLE LOOP

- A Simple Loop is used when a set of statements is to be executed at least once before the loop terminates.
- An EXIT condition must be specified in the loop, otherwise the loop will get into an infinite number of iterations.
- When the EXIT condition is satisfied the process exits from the loop.

SIMPLE LOOP

- Syntax:

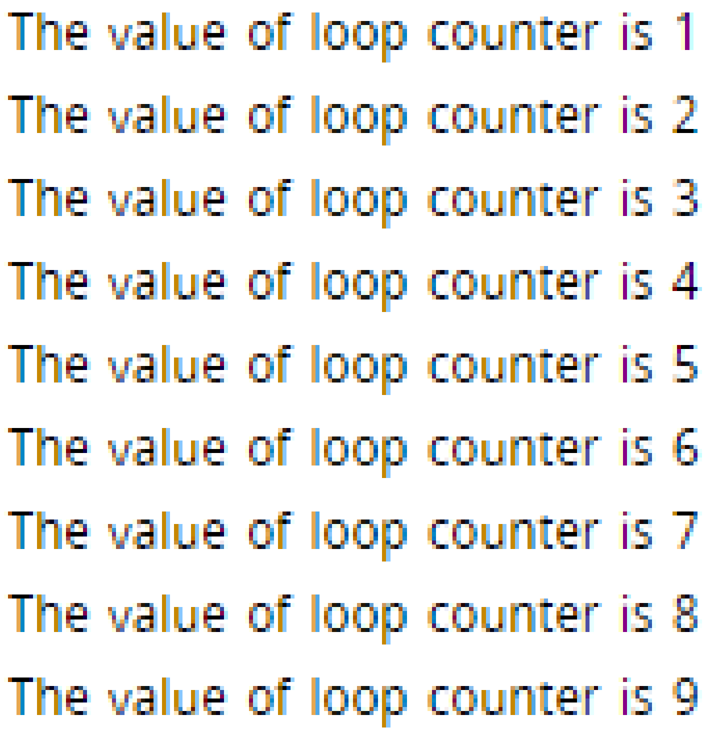
```
LOOP
      statements;
EXIT;
{or EXIT WHEN condition;}

END LOOP;
```

SIMPLE LOOP

```
DECLARE
   loop_counter number(6) := 1;
BEGIN
    LOOP
         dbms_output.put_line('The value of loop counter is ' ||
           loop_counter);
         loop_counter := loop_counter + 1;
    EXIT WHEN loop_counter = 10;
    END LOOP;
END;
```

SIMPLE LOOP

Output:



```
The value of loop counter is 1
The value of loop counter is 2
The value of loop counter is 3
The value of loop counter is 4
The value of loop counter is 5
The value of loop counter is 6
The value of loop counter is 7
The value of loop counter is 8
The value of loop counter is 9
```

SIMPLE LOOP

- These are the important steps to be followed while using Simple Loop:

  1) Initialise a variable before the loop body.
  2) Increment the variable in the loop.
  3) Use a EXIT WHEN statement to exit from the Loop.

  If you use a EXIT statement without WHEN condition, the statements in the loop is executed only once.

WHILE LOOP

- A WHILE LOOP is used when a set of statements has to be executed as long as a condition is true.

- The condition is evaluated at the beginning of each iteration. The iteration continues until the condition becomes false.

SYNTAX:

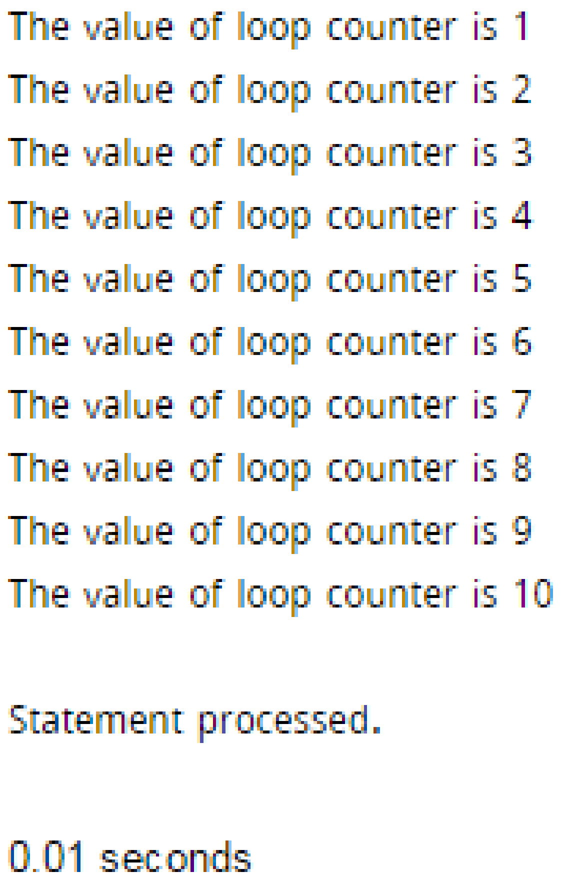
```
WHILE <condition>
LOOP statements;
END LOOP;
```

WHILE LOOP

```
DECLARE
   loop_counter number(6) := 1;
BEGIN
    WHILE loop_counter <= 10
    LOOP
        dbms_output.put_line('The value of loop counter is ' ||
          loop_counter);
        loop_counter := loop_counter + 1;
    END LOOP;
END;
```

WHILE LOOP

Output:



```
The value of loop counter is 1
The value of loop counter is 2
The value of loop counter is 3
The value of loop counter is 4
The value of loop counter is 5
The value of loop counter is 6
The value of loop counter is 7
The value of loop counter is 8
The value of loop counter is 9
The value of loop counter is 10

Statement processed.


0.01 seconds
```

WHILE LOOP

Important steps to follow when executing a while loop:

1) Initialise a variable before the loop body.
2) Increment the variable in the loop.
3) EXIT WHEN statement and EXIT statements can be used in while loops but it's not done oftenly.

FOR LOOP

- A FOR LOOP is used to execute a set of statements for a predetermined number of times.
- Iteration occurs between the start and end integer values given.
- The counter is always incremented by 1. The loop exits when the counter reaches the value of the end integer.

FOR LOOP

- SYNTAX -

```
FOR counter IN start_val..end_val
LOOP statements;
END LOOP;
```

start_val - Start integer value.

end_val - End integer value.

FOR LOOP

```
DECLARE
BEGIN
    FOR counter IN 1 .. 10
    LOOP
    dbms_output.put_line('The value of loop counter is ' ||
    counter);
    END LOOP;
END;
```

FOR LOOP

- Important steps to follow when executing a while loop:

  1) The counter variable is implicitly declared in the declaration section, so it's not necessary to declare it explicity.

  2) The counter variable is incremented by 1 and does not need to be incremented explicitly.

  3) EXIT WHEN statement and EXIT statements can be used in FOR loops but it's not done oftenly.

NESTED LOOPS

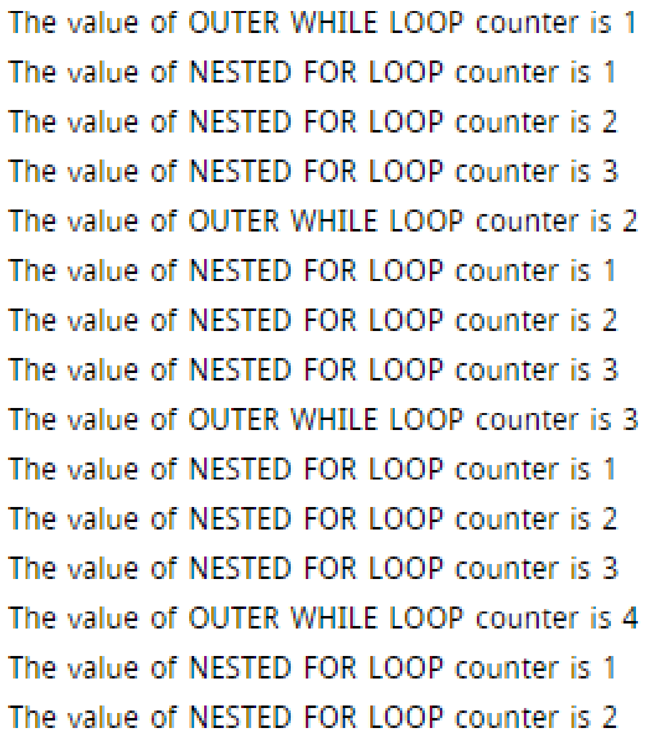
- Any type of loop can be nested inside another loop
- Execution of the inner loop must be completed before control is returned to the outer loop

NESTED LOOPS

```
DECLARE
   loop_counter number(10) := 1;
BEGIN

    WHILE loop_counter <= 10
    LOOP
        dbms_output.put_line('The value of OUTER WHILE LOOP counter is
          ' || loop_counter);
        FOR counter IN 1 .. 3
            LOOP dbms_output.put_line('The value of NESTED FOR LOOP
              counter is ' || counter);
        END LOOP;
        loop_counter := loop_counter + 1;
    END LOOP;

END;
```

NESTED LOOPS

Output:



```
The value of OUTER WHILE LOOP counter is 1
The value of NESTED FOR LOOP counter is 1
The value of NESTED FOR LOOP counter is 2
The value of NESTED FOR LOOP counter is 3
The value of OUTER WHILE LOOP counter is 2
The value of NESTED FOR LOOP counter is 1
The value of NESTED FOR LOOP counter is 2
The value of NESTED FOR LOOP counter is 3
The value of OUTER WHILE LOOP counter is 3
The value of NESTED FOR LOOP counter is 1
The value of NESTED FOR LOOP counter is 2
The value of NESTED FOR LOOP counter is 3
The value of OUTER WHILE LOOP counter is 4
The value of NESTED FOR LOOP counter is 1
The value of NESTED FOR LOOP counter is 2
```

⋮

CURSOR

- A cursor is a temporary work area created in the system memory when a SQL statement is executed.
- A cursor contains information on a select statement and the rows of data accessed by it.
- This temporary work area is used to store the data retrieved from the database, and manipulate this data.
- A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the *active* set.

CURSOR

- Cursors are defined and manipulated using –

    DECLARE

    OPEN

    FETCH

    CLOSE

DECLARING CURSORS

- **SYNTAX**

  CURSOR <cursor name> IS <select-expression>;

- **EXAMPLE**

  CURSOR emp_cursor

  IS

  SELECT employee_id, employee_name FROM employee WHERE employee_name LIKE 'E%';

OPENING A CURSOR

- Opens a cursor (which must be closed)
- Gets the query result from the database
- The rows returned become the cursor's current active set
- Sets the cursor to position before the first row. This becomes the current row. You must use the same cursor name if you want data from that cursor.
- OPEN <cursor name>;

  Example: OPEN emp_cursor;

FETCHING A ROW

- Moves the cursor to the next row in the current active set
- Assigns values to the host variables

- FETCH <cursor name> INTO <host variables>;

  Example: FETCH emp_cursor INTO e_id, e_name;

CLOSING THE CURSOR

- Closes the cursor (which must be open)
- There is no longer an active set
- Reopening the same cursor will reset it to point to the beginning of the returned table

- CLOSE <cursor name>;

  Example: CLOSE emp_cursor;

EXAMPLE

```
DECLARE
    CURSOR food_cursor
    IS
    SELECT food_name, price FROM food WHERE food_name LIKE
      'P%';

    food_val food_cursor%ROWTYPE;

BEGIN
    OPEN food_cursor ;
    FETCH food_cursor INTO food_val;
    DBMS_OUTPUT.PUT_LINE(food_val.food_name);
    CLOSE food_cursor ;
END;
```

TYPES OF CURSOR

There are two types of cursors in PL/SQL:

- **Implicit cursors**
- **Explicit cursors**

IMPLICIT CURSORS

- These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed.

- They are also created when a SELECT statement that returns just one row is executed.

EXPLICIT CURSORS

- They must be created when you are executing a SELECT statement that returns more than one row.
- Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.
- Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

CURSOR PROPERTIES

- Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations.

- The cursor attributes available are

  %FOUND, %NOTFOUND: a record can/cannot be fetched from the cursor
  %ROWCOUNT: the number of rows fetched from the cursor so far
  %ISOPEN: the cursor has been opened

STATUS OF THE CURSOR

Attributes	Return Value	Example
%FOUND	The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row.	SQL%FOUND
	The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT….INTO statement do not return a row.	

STATUS OF THE CURSOR

Attributes	Return Value	Example
%NOTFOUND	The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE at least one row and if SELECT ….INTO statement return at least one row.	SQL%NOTFOUND
	The return value is TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECT ….INTO statement does not return a row.	
%ROWCOUNT	Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT	SQL%ROWCOUNT

IMPLICIT CURSOR EXAMPLE

```
DECLARE
var_rows number(5);
BEGIN
  UPDATE food
  SET price = price + 1000;
  IF SQL%NOTFOUND THEN
    dbms_output.put_line('None of the prices where updated');
  ELSIF SQL%FOUND THEN
    var_rows := SQL%ROWCOUNT;
    dbms_output.put_line('Prices for ' || var_rows || ' foods are updated');
  END IF;
END;
```

SIMPLE CURSOR LOOPS

```
DECLARE
    CURSOR Food_cursor
    IS
    SELECT Food_name, price FROM Food WHERE Food_name LIKE
        'P%';
    Food_val Food_cursor%ROWTYPE;
BEGIN
    OPEN Food_cursor;
    LOOP FETCH Food_cursor INTO Food_val;
    EXIT WHEN Food_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(Food_val.Food_name);
    END LOOP;
    CLOSE Food_cursor;
END;
```

CURSOR FOR LOOP

```
DECLARE
    CURSOR food_cursor
    IS
    SELECT Food_name, Price FROM Food WHERE
       Food_name LIKE 'P%';
BEGIN
    FOR food_val IN food_cursor
    LOOP
    EXIT WHEN food_cursor %NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(food_val.Food_name);
    END LOOP;
END;
```

EXCEPTION HANDLING

- PL/SQL provides a feature to handle the **Exceptions** which occur in a PL/SQL Block known as exception Handling.
- Using Exception Handling we can test the code and avoid it from exiting abruptly.
- When an exception occurs a messages which explains its cause is received.
- PL/SQL Exception message consists of three parts.
  1) Type of Exception
  2) An Error Code
  3) A message

STRUCTURE OF EXCEPTION HANDLING

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

EXCEPTION HANDLING

- When an exception is raised, Oracle searches for an appropriate exception handler in the exception section.
- For example in the above example, if the error raised is 'ex_name1 ', then the error is handled according to the statements under it.
- Since, it is not possible to determine all the possible runtime errors during testing of the code, the 'WHEN Others' exception is used to manage the exceptions that are not explicitly handled.
- Only one exception can be raised in a Block and the control does not return to the Execution Section after the error is handled.

TYPES OF EXCEPTION

- There are 3 types of Exceptions -

  a) Named System Exceptions
  b) Unnamed System Exceptions
  c) User-defined Exceptions

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.

- 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.

NAMED SYSTEM EXCEPTIONS

Exception Name	Reason	Error Number
CURSOR_ALREADY_OPEN	When you open a cursor that is already open.	ORA-06511
INVALID_CURSOR	When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened.	ORA-01001
NO_DATA_FOUND	When a SELECT...INTO clause does not return any row from a table.	ORA-01403
TOO_MANY_ROWS	When you SELECT or fetch more than one row into a record or variable.	ORA-01422
ZERO_DIVIDE	When you attempt to divide a number by zero.	ORA-01476

NAMED SYSTEM EXCEPTIONS

Example:

Suppose a NO_DATA_FOUND exception is raised in a proc, we can write a code to handle the exception as given below.

```
BEGIN
//Execution section
EXCEPTION
    WHEN NO_DATA_FOUND
    THEN
    dbms_output.put_line ('A SELECT...INTO did not
    return any row.');
END;
```

UNNAMED SYSTEM EXCEPTIONS

- Those system exception for which oracle does not provide a name is known as unnamed system exception.
- These exception 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.

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.

USER-DEFINED EXCEPTIONS

```
DECLARE
Emp_sal NUMBER(10,3);
Emp_no VARCHAR2(12);
too_high_sal EXCEPTION;
BEGIN
       SELECT employee_id, salary INTO emp_no, emp_sal FROM employee WHERE
       employee_name = ‘E_Y’;
       IF emp_sal * 1.05 > 2000 THEN
          RAISE too_high_sal;
       END IF;
  EXCEPTION
          WHEN NO_DATA_FOUND
          THEN dbms_output.put_line(‘No data found’);
          WHEN too_high_sal
          THEN dbms_output.put_line(‘High Salary’);
END;
```

RAISE APPLICATION ERROR()

- It is also possible to use procedure raise_application_error().
- This procedure has two parameters <error number> and <message text>.
- <error number> is a negative integer defined by the user and must range between -20000 and -20999.
- <error message> is a string with a length up to 2048 characters.

- If the procedure raise application error is called from a PL/SQL block, processing the PL/SQL block terminates and all database modifications are undone, that is, an implicit rollback is performed in addition to displaying the error message.

RAISE APPLICATION ERROR

```
DECLARE
    emp_sal NUMBER(10,3);
    emp_no VARCHAR2(12);
BEGIN
    SELECT employee_id, salary INTO emp_no, emp_sal
    FROM employee WHERE employee_name = ‘E_Y’;
    IF emp_sal * 1.05 > 2000
      THEN raise_application_error(-20010, ‘‘Salary is too
    high‘‘);
    END IF;
END;
```

%TYPE & %ROWTYPE

- %TYPE Takes the data type from the table
- %TYPE Form is <variable> <table>.<column>%TYPE
- %ROWTYPE Creates a record with fields for each column of the specified table

```
DECLARE
     variable name data type;
     row_variable table %ROWTYPE;
BEGIN
   SELECT column name1, column name2, ………… INTO
   row_variable FROM table name WHERE column name =
   variable name;
```

- The variables are then accessed as: row_variable.column name

SQL COMMAND CATEGORIES

SQL commands are grouped into four major categories depending on their functionality

- **Data Definition Language (DDL)**

  These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.
- **Data Manipulation Language (DML)**

  These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.
- **Transaction Control Language (TCL)**

  These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.
- **Data Control Language (DCL)**

  These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.

OTHERS

- If the output is not being displayed:

  ```
  SET SERVEROUTPUT ON;
  ```

SELF STUDY

- How to use record
  http://plsql-tutorial.com/plsql-records.htm

REFERENCES

1. http://plsql-tutorial.com
2. http://www.oracle.com/technetwork/database/features/plsql/index.html
3. http://www.tutorialspoint.com/plsql/

Thank You