

Unit – III

Advanced PL/SQL

- Exception handling
- Cursors
- Stored procedures and stored functions
- Database triggers
- Packages

Exception

An error occurs during execution of program is called **Exception**.
PL/SQL supports to catch such error with the Exception block in the program.

Two types of Exception:

- System Defined Exception (Predefined Exception):
 - In built exception handling by the database itself.
- User Defined Exception :
 - An exception handling as per the user requirement for the program.

Oracle Error	Equivalent Exception	Description
ORA-00001	DUP_VAL_ON_INDEX	Unique constraint violated.
ORA-01403	NO_DATA_FOUND	No data found
ORA-01422	TOO_MANY_ROWS	A select...Into statement matches more than one row.
ORA-01476	ZERO_DIVIDE	Division by zero
ORA-01722	INVALID_NUMBER	Conversion to a number failed
ORA-06502	VALUE_ERROR	Truncation, arithmetic error
ORA-06511	CURSOR_ALREADY_OPEN	Attempt to open a cursor that is already open.

System Defined Exception

Unique Constraint violated:

This exception is generated, when try to insert a duplicate value for the primary key field.

E.g.

```
CREATE TABLE temp ( t NUMBER , PRIMARY KEY ( t ));
```

```
INSERT INTO temp VALUES (1);
```

```
BEGIN
```

```
    INSERT INTO temp VALUES (1);
```

```
END;
```

System Defined Exception

No Data Found

This exception is generated, When the SELECT INTO statement does not return any row.

E.g.

DECLARE

 t1 temp.t%TYPE;

BEGIN

 SELECT t INTO t1 FROM temp WHERE t = 20;

END;

System Defined Exception

Illegal Cursor operation

This exception is generated, when already close cursor is close again.

E.g.

DECLARE

CURSOR c1 IS SELECT * FROM temp;

BEGIN

OPEN c1;

CLOSE c1;

CLOSE c1;

END;

System Defined Exception

Too many rows are selected

This exception is generated, when SELECT INTO statement return more than one row.

E.g.

```
INSERT INTO temp VALUES (1);
```

```
DECLARE
```

```
    num NUMBER;
```

```
BEGIN
```

```
    SELECT * INTO num FROM temp;
```

```
END;
```

System Defined Exception

Divide by zero

This exception is generated, when any number is divide by zero.

E.g.

DECLARE

num NUMBER := 4;

BEGIN

num := num / 0 ;

DBMS_OUTPUT.PUT_LINE('num' || num);

END;

System Defined Exception

Invalid number

This exception is generated, when character is passed for field having a numeric data type.

E.g.

```
INSERT INTO temp VALUES ('X');
```


System Defined Exception

Value error

This exception is generated, when value is more than the size specified for the field datatype.

E.g.

DECLARE

num NUMBER(2) ;

BEGIN

num := #

DBMS_OUTPUT.PUT_LINE('num ::---->' || num);

END;

System Defined Exception

Cursor_Already_Open

This exception is generated, when already opened cursor is open again without closing it.

DECLARE

CURSOR c1 IS SELECT * FROM temp;

BEGIN

OPEN c1;

OPEN c1;

CLOSE c1;

END;

Exception Handling

EXCEPTION

```
WHEN      Exception Name 1 THEN  
            Sequence_of_statements;  
[ WHEN      Exception Name 2 THEN  
            Sequence_of_statements;  
            .....  
            ..... ]  
[WHEN      OTHERS      THEN  
            Sequence_of_statements;  
  
]  
END;
```

Handling System Defined Exception

E.g. When the duplicate value is inserted for primary key field, user defined message to be displayed on screen.

BEGIN

INSERT INTO temp VALUES (1);

EXCEPTION

WHEN DUP_VAL_ON_INDEX THEN

DBMS_OUTPUT.PUT_LINE('Insertion is failed');

DBMS_OUTPUT.PUT_LINE('Duplicate value should not be allowed in primary key field');

END;

Handling Multiple System Defined Exception

E.g. Accept one number from user and if exception is generated than display proper message.

DECLARE

t1 NUMBER(2);

BEGIN

t1 := &t1;

INSERT INTO temp VALUES (t1);

EXCEPTION

WHEN INVALID_NUMBR THEN

DBMS_OUTPUT.PUT_LINE('Please insert only numeric value');

WHEN VALUE_ERROR THEN

DBMS_OUTPUT.PUT_LINE('please insert value according to field size');

WHEN DUP_VAL_ON_INDEX THEN

DBMS_OUTPUT.PUT_LINE('Duplicate value should not be allowed');

END;

OTHERS clause to handle system defined exception

E.g. **OTHERS** will trap any other error that is not handled in exception section

DECLARE

 t1 NUMBER(2);

BEGIN

t1 := &t1;

 INSERT INTO temp VALUES (t1);

EXCEPTION

WHEN INVALID_NUMBR THEN

 DBMS_OUTPUT.PUT_LINE('Please insert only numeric value');

WHEN VALUE_ERROR THEN

 DBMS_OUTPUT.PUT_LINE('please insert value according to field size');

WHEN OTHERS THEN

 DBMS_OUTPUT.PUT_LINE('Exception is generated ');

END;

User-Defined Exception

DECLARE

Exception name **EXCEPTION;**

BEGIN

RAISE Exception name

EXCEPTION

WHEN exception name **THEN**

 Sequence_Of_Statements;

END;

E.g. restrict the number of rows in table.

User Defined Exception

// Restrict the table to store more than 4 records.

DECLARE

```
    user_exc          EXCEPTION;  
    cnt               NUMBER; -- To count total number of rows in table.  
    num               NUMBER; -- To store values of field
```

BEGIN

```
    num := &num;  
    SELECT COUNT(*) INTO cnt FROM temp;  
    IF (cnt >= 4 ) THEN  
        RAISE user_exc;  
    END IF;  
    INSERT INTO temp VALUES (num);
```

EXCEPTION

```
    WHEN user_exc THEN  
        DBMS_OUTPUT.PUT_LINE('Record Should Not Be Stored');  
        DBMS_OUTPUT.PUT_LINE('Size Of Table Is Full ');
```

END;

Exercise

- 1. Consider the table Department_Master (Dept_Id, Dept_Name). Dept_Id is a primary key.**
 - a) Write a PL/SQL block to handle duplicate value for Dept_Id field.**
 - b) Write a PL/SQL block to handle a situation when no record is exist for entered Dept_id.**
 - c) Write a PL/SQL block to restrict to store only 5 records in Department_Master.**

Cursor

- **Context Area**
- **Cursor**
- **Active Set.**
- **Two Types of cursors:**
 - Implicit cursors
 - Explicit cursors
- **Cursor Attributes**

Attribute	Type	Description
%ISOPEN	Boolean	Evaluates to TRUE if the cursor is open.
%FOUND	Boolean	Evaluates to TRUE if the most recent fetch returns a row;
%NOTFOUND	Boolean	Evaluates to TRUE if the most recent fetch does not return a row.
%ROWCOUNT	Number	Evaluates to the total number of rows affected by statement

Cursor

Oracle creates a memory area, known as **context area**, for processing an SQL statement, which contains all information needed for processing the statement, for example, number of rows processed.

A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time.

There are **two types of cursors**:

- Implicit cursors

- Explicit cursors

Implicit Cursor

- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed. Programmers cannot control the implicit cursors and the information in it.
- Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement.
- In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**.

Implicit Cursor (SQL) : %FOUND Attribute

E.g. To check that SQL statement returns any record or not.

DECLARE

```
emp_num emp_m.emp_no%type;  
emp_rec emp_m%rowtype;
```

BEGIN

```
emp_num := &Employee_Number;
```

```
SELECT * INTO emp_rec FROM EMP_M WHERE emp_no = emp_num;
```

```
IF SQL%FOUND THEN -- select succeeded
```

```
DBMS_OUTPUT.PUT_LINE ('Employee Record Is exists for employee No'|| emp_num);  
DBMS_OUTPUT.PUT_LINE('Employee No. ::' || emp_rec.emp_no);  
DBMS_OUTPUT.PUT_LINE('Employee Name :: ' || emp_rec.emp_name);
```

```
END IF;
```

```
END;
```

```
/
```

Explicit Cursor

Steps for Explicit Cursor:

- Declaring the cursor for initializing in the memory
- Opening the cursor for allocating memory
- Fetching the cursor for retrieving data
- Closing the cursor to release allocated memory

Explicit Cursor Steps

- **Declaring the Cursor**

CURSOR cursor_name IS SELECT statement

- **Opening the Cursor**

OPEN cursor_name;

- **Fetching the Cursor**

FETCH cursor_name INTO [variable name | variable names]

- **Closing the Cursor**

CLOSE cursor name;

Explicit Cursor Steps

- **Declaring the Cursor**

Declaring the cursor defines the cursor with a name and the associated SELECT statement. **E.g.**

CURSOR c_emp IS SELECT no, fname, lname FROM emp;

- **Opening the Cursor**

Opening the cursor allocates memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. **E.g.**

OPEN c_emp;

- **Fetching the Cursor**

Fetching the cursor involves accessing one row at a time. **E.g.**

FETCH c_emp INTO e_id, e_fname, e_lname;

- **Closing the Cursor**

Closing the cursor means releasing the allocated memory. **E.g.**

CLOSE c_emp;

Explicit Cursor : While Loop

Que. : Display all rows of emp table with the use of record type variable. (Using While Loop)

Emp_M (emp_no, emp_name)

```
DECLARE
    EMPREC EMP_M%ROWTYPE;
    CURSOR C1 IS SELECT * FROM EMP_M;
BEGIN
    OPEN C1 ;
    IF C1%ISOPEN THEN
        FETCH C1 INTO EMPREC ;
        While ( C1%FOUND )
            LOOP

                DBMS_OUTPUT.PUT_LINE('NO :=' || EMPREC.emp_no);
                DBMS_OUTPUT.PUT_LINE('NAME :=' || EMPREC.emp_name );
                DBMS_OUTPUT.PUT_LINE('NO. OF ROWS ACCESSED' || C1%ROWCOUNT);

            FETCH C1 INTO EMPREC ;

        END LOOP;
    END IF;
    CLOSE C1;
END;
```

Explicit Cursor : For Loop

- It is very useful because it automatically do the following steps of explicit cursor.
 - Open Cursor
 - Fetch the record
 - Close Cursor
- **FOR** *record_index* **IN** *cursor_name*
LOOP
 ...statements...
END LOOP;

Cursor Example (For Loop)

Que. : Display all rows of emp table with the use of record type variable. (Using While Loop)

Emp_M (Emp_No, Emp_Name)

DECLARE

CURSOR c1 IS SELECT * FROM EMP_M;

BEGIN

FOR v_empdata IN c1

LOOP

DBMS_OUTPUT.PUT_LINE('NO := ' || v_empdata.emp_no);

DBMS_OUTPUT.PUT_LINE('NAME := ' || v_empdata.emp_name);

DBMS_OUTPUT.PUT_LINE('NO. OF ROWS ACCESSED' || c1%rowcount);

END LOOP;

END;

PROCEDURE

It is a group of **SQL** statements and logic to perform a specific task.

A **procedure** is compiled once and can be called multiple times without being compiled.

Syntax:

```
CREATE [ OR REPLACE ] PROCEDURE procedure-name
```

```
[  
  ( argument-name [ {IN | OUT | INOUT} ] data-type ,  
    .....  
    .....  
    argument-name [ {IN | OUT | INOUT} ] data-type  
  )  
]  
{ IS | AS }  
procedure body
```

Procedure Without Arguments

E.g.

```
CREATE OR REPLACE PROCEDURE P1 as
BEGIN
    dbms_output.put_line ('Without argument procedure');
END;
```

Execute statement

```
EXECUTE P1;
```

PL/SQL block

```
BEGIN
    P1;
END;
```

Procedure – With Arguments

```
CREATE TABLE emp_det (no NUMBER , sal NUMBER(10,2));
```

E.g. A procedure to insert new record in emp table

```
CREATE OR REPLACE PROCEDURE P1  
    (v_num NUMBER, v_sal NUMBER(10,2) ) is  
BEGIN  
    insert into emp_det values (v_num , v_sal);  
END;
```

Procedure – With Argument

[1]

```
EXECUTE P1(1,10000);
```

[2]

```
BEGIN
```

```
    P1(2,20000);
```

```
END;
```

[3]

```
declare
```

```
    num          emp_det.no%type;
```

```
    sal          emp_det.sal%type;
```

```
BEGIN
```

```
    num := &num;
```

```
    sal := &sal;
```

```
    P1(num,sal);
```

```
END;
```

FUNCTION

- Functions are stored code and are very similar to procedures.
- Function *returns* a single value.
- The datatype of the return value must be declared in the function.
- A function has output that needs to be assigned to a variable, or it can be used in a SELECT statement.
- Functions can accept one, many, or no parameters, but a function must have a return clause in the executable section of the function.

FUNCTION – Syntax

CREATE [OR REPLACE] FUNCTION function-name

```
[  
  ( argument-name [ {IN | OUT | INOUT} ] data-type ,  
    .....  
    .....  
    argument-name [ {IN | OUT | INOUT} ] data-type  
  )  
]
```

RETURN return_type

{ IS | AS }

function body

Example of Function

```
CREATE OR REPLACE FUNCTION F ( no in Number )  
RETURN NUMBER AS  
    f number(5) := 1 ;  
BEGIN  
    for i in 1.. no loop  
        f := f * i ;  
    end loop;  
    return ( f ) ;  
END;
```

Function - Example

To Call a function in a PL/SQL block

=====

Declare

 v_out number ;

begin

 v_out := f(2);

 dbms_output.put_line('value of v_out is ' || v_out);

end;

/

Exercise

- 1. Consider the table Department_Master (Dept_Id, Dept_Name). Dept_Id is a primary key.**
 - a) Write a procedure to insert new record in Department_Master table. (Without any arguments)**
 - b) Write a procedure to insert new record in Department_Master table. (With an arguments)**
 - c) Write a function that return Dept_name for passing Dept_Id as an argument.**
 - d) Write a function that return total number of records in a Department_Master table.**

PACKAGE

- A **package** is a schema object that groups
 - Variables,
 - Exceptions,
 - Cursors,
 - Procedure and
 - Functions.
- Enable the Oracle server to read multiple objects into memory at once
- A **package** is compiled and stored in the database
- It consists of two parts :
 - A specification (Package Header)
 - A body (Package Body)

Exercise

- **Employee** (Emp_Id , f_name , m_name , l_name, birth_date)
- **Create a package for the following.**
 - I. Procedure to display the name in format: 'A. P. Shah'
 - II. Procedure to display the name in format: 'Amit P. Shah'
 - III. Procedure to display the name in format: 'Amit Parimalbhai Shah'
 - IV. Function to return name of employee whose employee id is passed.
 - V. Function to count the employee having birthday today.

Trigger

- A procedure that is run implicitly when Insert, Delete or Update statement is issued for a table / view or a when database system action occurs.
- You can enable and disable a trigger, but you cannot explicitly invoke it.
- Database triggers can be associated with a table, schema, or database. They are implicitly fired when:
 - **DML** statements are executed (INSERT, UPDATE, DELETE) against an associated table
 - Certain **DDL** statements are executed.
 - Example : (ALTER, CREATE, DROP) on objects within a database or schema
 - A specified **database event** occurs
 - Example: STARTUP, SHUTDOWN, SERVERERROR

Statement Level Trigger

- A **statement-level trigger** is fired whenever a **trigger** event occurs on a **table** regardless of how many rows are affected.
- For example, if you update 1000 rows in a **table**, then a **statement-level trigger** on that **table** would only be executed once.
- The following hierarchy is followed when a trigger is fired.
 - 1) BEFORE statement trigger fires first.
 - 2) AFTER statement level trigger fires.

Trigger : Syntax

```
CREATE [OR REPLACE ] TRIGGER trigger_name
    {BEFORE | AFTER }
    {INSERT [OR] | UPDATE [OR] | DELETE}
    ON table_name
BEGIN
    --- sql statements
END;
```

Example

Write Statement Level triggers for product table.

- CREATE TABLE product
(product_id number(5) primary key,
product_name varchar2(40)) ;

Insert into product values (101,'l1')

Insert into product values (102,'l2');

Insert into product values (103,'l3');

Insert into product values (104,'l4');

Before Statement Level Trigger

- CREATE or REPLACE TRIGGER **Before_Update_Stat_product**
BEFORE UPDATE
ON product
BEGIN
DBMS_OUTPUT.PUT_LINE ('Before update, statement level');
END;

After Statement Level Trigger

- CREATE or REPLACE TRIGGER **After_Update_Stat_product**
AFTER UPDATE
ON product
BEGIN
DBMS_OUTPUT.PUT_LINE ('After update, statement level');
End;

Update Statement

UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 101 ;

PPR

Trigger : Example

Restrict the product table to store only 5 records.

- CREATE or REPLACE TRIGGER **Before_Insert_product**

BEFORE INSERT

ON product

BEGIN

SELECT COUNT(*) INTO v_cnt FROM Product;

IF v_cnt > 5 THEN

RAISE_APPLICATION_ERROR(-20100,'Cannot insert record
because product table limit is only 5 records');

END IF;

End•

Trigger

- Enable/Disable Triger
 - ALTER TRIGGER Triger-Name ENABLE;
 - ALTER TRIGGER Triger-Name DISABLE;
- Drop Trigger
 - Drop Trigger Trigger-Name