Cursors and Exception Handling

- **SELECT..INTO statement**
- **Working with cursor : introduction, types, attributes and processing (i.e. declaring, opening, fetching and closing),**
- **Using parameterized cursor**
- **Using cursor FOR loop**
- **Error Handling : introduction, advantages of exceptions, types of**
- **exceptions Working with user-defined exceptions – declaration,**
- **Raise_Application_Error, Pragma Exception_Init**
- **Sqlcode And Sqlerrm**

VPSC

SELECT INTO Statement:

(ANCHORD DECLARATION OF VARIABLE)

The PL / SQL uses % TYPE declarations attribute to anchor (attach) the variable data type. An anchor variable or column in a table can be used for anchoring. During anchoring, you tell the PL / SQL to use a variable or a table column data type as a data type for anchor variable in the program.

<VARIABLE NAME> <DATATYPE> %TYPE [: =];

Write a program that displays the use of %TYPE variable.

```
DECLARE
        VENO EMP.ENO %TYPE;
        VENAME EMP.ENAME %TYPE;
        VESAL EMP.SAL %TYPE;

   BEGIN
        VENO: = &NO1;
        SELECT ENAME, SAL INTO VENAME, VESAL FROM EMP WEHERE ENO = VENO;

   DBMS_OUTPUTPPUT_LINE ('EMP NO IS - ' || VENO);
   DBMS_OUTPUTPPUT_LINE ('EMP NAME IS - ' || VENAME);
   DBMS_OUTPUTPPUT_LINE ('EMP SAL IS - ' || VESAL);
   END;
/
```

Write a program to find out whether or not the given employee is eligible for bonus or not according to following condition. The bonus is granted if the salary is more than the average total salary of any one employee otherwise the bonus will be not granted.

```
DECLARE
                VNO EMP.EMPNO %TYPE;
                VREC EMP %ROWTYPE;
                VAVG NUMBER (10, 2);

BEGIN
                VNO: = &EMPNO;

                SELECT * INTO VREC FROM EMP WHERE EMPNO = VNO;
                SELECT AVG (SAL) INTO VAVG FROM EMP;

                IF VAVG > VREC.SAL THEN
DBMS_OUTPUT.PUT_LINE (‘THE BONUS WILL BE GRANTED);
                ELSE
DBMS_OUTPUT.PUT_LINE (‘THE BONUS WILL BE GRANTED);
                END IF;
END;
```

INTRODUCTION TO CURSOR:

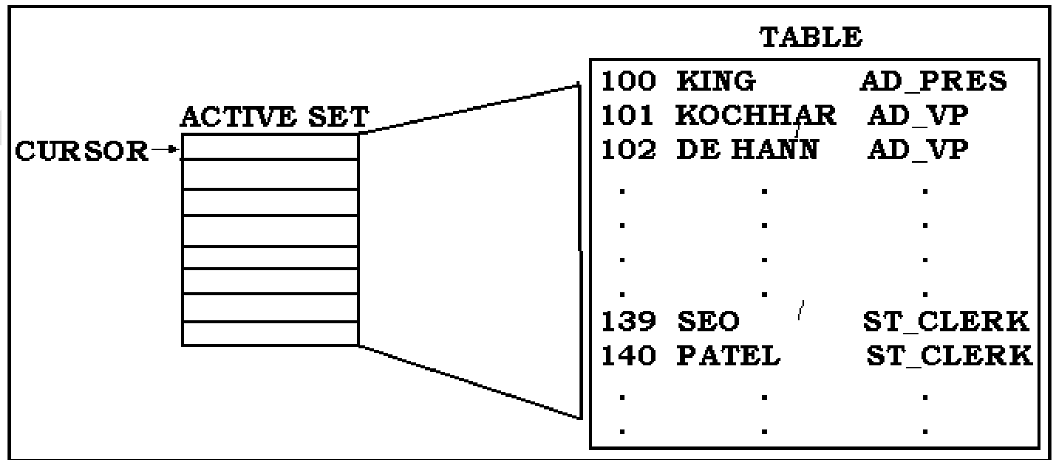
- Cursor is a memory area which store records written by SQL query. It is mainly used to perform operations on more then one row in PL block.

- When Oracle processes a SQL statement, it opens an area in the memory called Private SQL area. This area stores information responsible for executing the statement. An identifier for this area is created called a Cursor.

- When you use SQL*Plus to select database records, the SELECT statement is sufficient to display the rows of tables or views that meet the specified criteria. Unfortunately, this is not true when you user PL/SQL. When a PL/SQL block, trigger or procedure uses a SELECT statement that returns more than one row, Oracle displays an error message and invokes the TOO_MANY_ROWS exception. To resolve this problem, Oracle uses a mechanism called a Cursor.

- The set of rows returned by a cursor is called an Active Data Set (Result Set). The row that is being processes is called the Current Row. The Current Row inside the Active Data Set (Result Set) is identified by the cursor, which allows the individual processing of each of them.

- The data that is stored in the cursor is called the **Active Data Set.**



- Cursor can be used for retrieve & update the records in the table.

- Cursors are used when the SQL SELECT statement is expected to return more than one row.

Example, a query like, SELECT * FROM CSTMAST WHERE CITY='UDP'; or
SELECT * FROM EMP WHERE SAL>=3000;
So, Cursor is a buffer, which will store the results of the recent query.

- A Cursor must be declared and its definition contains the query. The cursor must be defined in the DECLARE section of the program. A cursor must be opened before processing and close after processing.

```
CURSOR cursor_name IS
    select_statement;
```

TYPES OF CURSOR:

There are basically two types of cursor are there:

(1) Implicit Cursor

(2) Explicit Cursor (User-defined Cursor)

(1) *Implicit Cursor:*

The Oracle engine implicitly opens a cursor on the Server to process each SQL statement. Since the Implicit Cursor is opened and managed by Oracle engine internally, the function of reserving an area in memory, populating this area with appropriate data, processing the data in the memory area, releasing the memory area when the processing is complete is taken care of by the Oracle engine. The result data is then passed to the client through the network.

Implicit Cursor attributes can be used to access information about the status of last insert, update, delete or single row select statement. This can be done by preceding the Implicit Cursor attribute with the cursor name SQL. The values of the cursor attributes always refer to the SQL command that was executed most recently. Before Oracle opens Implicit SQL cursor, the attributes of the implicit cursors will have NULL values in their fields. The following are the attributes of Implicit Cursor.

VPSC

ATTRIBUTES	DESCRIPTION
SQL%ISOPEN	The Oracle engine automatically opens and closes the SQL cursor after executing its associated select, insert, update or delete SQL statement has been processed in case of implicit cursors. Thus SQL%ISOPEN attribute of an implicit cursor cannot be referenced outside of SQL statement. As a result, SQL%ISOPEN always evaluates to FALSE.
SQL%FOUND	It is TRUE if an insert, update or delete affected one or more rows, or a single row select returned one or more rows. Otherwise, it evaluates to FALSE.
SQL%NOTFOUND	It is logical opposite of SQL%FOUND. It evaluates to TRUE, if an insert, update, or delete affected no rows, or a single row select statement returns no row. Otherwise, it evaluates to FALSE.
SQL%ROWCOUNT	It returns number of rows affected by an insert, update or delete or select into statement.

(2) *Explicit Cursor:*

When individual records in a table have to be processed inside a PL/SQL code block a cursor is used. This cursor will be declared and mapped to an SQL query in the Declare Section of PL/SQL block and used within the Executable Section. A cursor thus created and used is known as an Explicit Cursor.

The Explicit Cursor is declared, named and managed by programmer. The Oracle engine does not automatically manage it. The following are the attributes of Explicit Cursor.

VPSC

ATTRIBUTES	DESCRIPTION

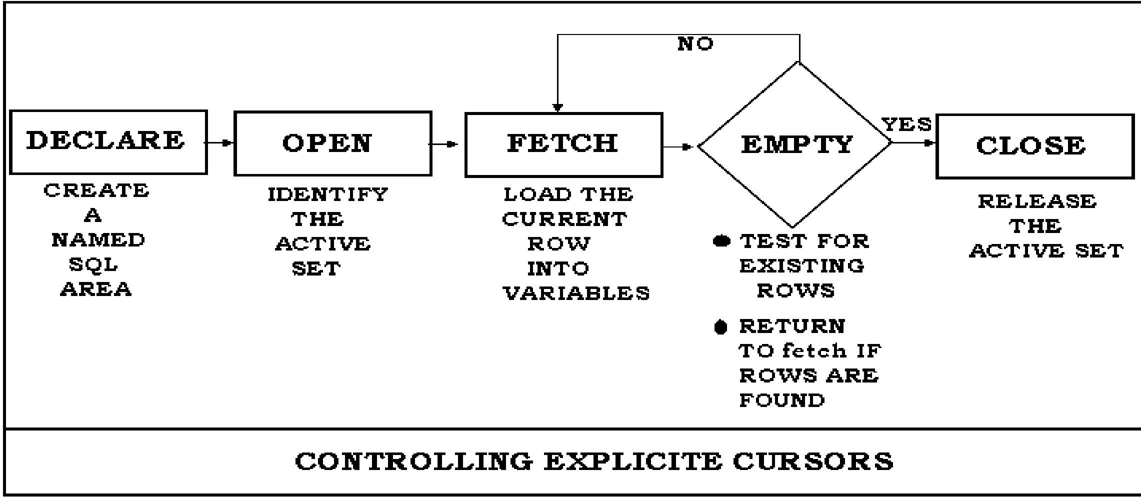
%ISOPEN	To determine if a cursor is open.
	ISOPEN is TRUE if a cursor is open.
	ISOPEN is FALSE if a cursor is not open.
	Ex.
	IF CUR_NAME%ISOPEN THEN
	<Statements>;
	END IF;

%FOUND	To determine if a row was retrieved. Used after FETCH. Ex. WHILE CUR_ NAME %FOUND LOOP <Statements>; END LOOP;
%NOTFOUND	To determine if a row was retrieved. Use it after FETCH. NOTFOUND is TRUE if a row was not retrieved. NOTFOUND is FALSE if a row was retrieved. Ex. EXIT WHEN CUR_ NAME %NOTFOUND;
%ROWCOUNT	ROWCOUNT is zero when the cursor is opened. ROWCOUNT returns the number of rows retrieved. Ex. EXIT WHEN (CUR_ NAME %ROWCOUND>5);

To ret rive the data from the cursor on has to user FETCH command. The cursor management requires the following step-by-step procedure.

1. Declare the cursor.
2. Open the cursor.
3. Fetch the cursor into PL variable.
4. Check for EOF.
5. Perform operations in PL block.
6. Close the cursor.

VPSC



DIFFERENTIATE BETWEEN IMPLICIT AND EXPLICIT CURSOR:

IMPLICIT CURSOR	EXPLICIT CURSOR
(1). If Oracle engine for its internal processing has opened a cursor they are known as Implicit Cursors.	(1). A user can also open a cursor for processing data as required. Such user-defined cursors are known as Explicit Cursors.
(2). Oracle by default creates the cursor with name SQL.	(2). User can create the explicit cursor with any name and also can change the name of the cursor.
(3). We have to put SQL prefix before accessing any attributes of the Implicit Cursor.	(3). If we want to use any Explicit Cursors attributes then we have to put user defined cursor's name as a prefix.
(4). User cannot require OPENING and CLOSING of the implicit cursor.	(4). User must have to OPEN the cursor for fetching the data and CLOSE the cursor after completion of operation.
(5). We cannot use the FETCH command with Implicit Cursor.	(5). We have to use the FETCH command with Explicit Cursor.

(6). There are no other types of implicit cursors.	(6). There are two types of explicit cursors are there: (i) Normal Cursor. (ii) Parameterized Cursor.

SYNTAX to define the cursor:

```
CURSOR cursor_name IS
    select_statement;
```

CURSOR <CORSOR-NAME> IS <SELECT STATEMENT>;

Ex.

CURSOR CUR_SAL IS
SELECT EMP_NO, NAME FROM EMP WHERE SAL>5000;

CURSOR CUR_SAL IS
SELECT * FROM EMP WHERE SAL>6570;

SYNTAX for OPENING the CURSOR:

```
OPEN  cursor_name;
```

OPEN <CURSOR-NAME>;

Ex.

OPEN CUR_SAL;

SYNTAX to store the date in to the CURSOR:

```
FETCH cursor_name INTO  [variable1, variable2, ...]
                        | record_name];
```

FETCH <CURSOR-NAME> INTO <VAR1>, <VAR2>, <VAR3>…

Or

FETCH <CURSOR_NAME> INTO <RECORD-NAME>;

Ex.

FETCH CUR_SAL INTO mEMPNO, mEMPNAME;
FETCH CUR_SAL INTO EMP_REC;

SYNTAX to CLOSETING the CURSOR:

```
CLOSE   cursor_name;
```

CLOSE <CURSOR-NAME>;

Ex.

CLOSE CUR_SAL;

Write a program that uses a cursor attribute SQL%FOUND to raise the salary of employees by 20% and also display the appropriate message based on the existence to the record in the EMP table.

```
BEGIN
        UPDATE EMP SET SAL = SAL * 0.2O
                WHERE EMP_NO = &EMP_NO;

        IF SQL%FOUND THEN
                DBMS_OUTPUT.PUT_LINE ('Employee Record Updated');
        ELSE
                DBMS_OUTPUT.PUT_LINE ('Employee number does not exist');
        END IF;
END;
/
```

Write a program that uses a cursor attribute SQL%ROWCOUNT to raise the salary of employees by 10% that are working in department number 10 and also display the appropriate message based on the existence to the record in the EMP table.

```
DECLARE
        ROWS_UPDATED CHAR (4);
BEGIN
        UPDATE EMP SET SAL = SAL * 0.10
                WHERE DEPT_NO = 10;

        ROWS_UPDATED: = TO_CHAR (SQL%ROWCOUNT);

IF SQL%ROWCOUNT>0 THEN
DBMS_OUTPUT.PUT_LINE (ROWS_UPDATED || 'Employees Records are updated');
        ELSE
DBMS_OUTPUT.PUT_LINE ('nobody is working in DEPT_NO 10');
        END IF;
END;
/
```

Write a program that displays the deletion of records using an IMPLICIT CURSOR.

```
BEGIN

        DELETE FROM EMP WHERE DEPT_NO = 10;

        IF SQL%FOUND THEN
                DBMS_OUTPUT.PUT_LINE (‘RECORDS FOR DEPT_NO 10 EXISTS’);
        ELSE
                DBMS_OUTPUT.PUT_LINE (‘NO RECORDS DEPT_NO 10’);
        END IF;

        IF SQL%ROWCOUNT > 0 THEN
                DBMS_OUTPUT.PUT_LINE (‘TOTAL DELETED RECORDS: =’||SQL%ROWCOUNT);
        ELSE
                DBMS_OUTPUT.PUT_LINE (‘NO RECORDS ARE FOUND’);
        END IF;
END;
/
```

Write a program that uses a cursor attribute %ISOPEN and %NOTFOUND to raise the salary of employees of department number 20 by 5% and also display the appropriate message based on the existence to the record in the EMP table. Whenever any such raise is given to the employees, a record for the same is maintained in the emp_update table.

```
DECLARE

CURSOR CUR_EMP IS SELECT EMP_NO, SAL FROM EMP WHERE DEPT_NO = 20;
        V_NO EMP.EMP_NO%TYPE;
        V_SAL EMP.SAL%TYPE;
BEGIN

OPEN CUR_EMP;

        IF CUR_EMP%ISOPEN THEN

        LOOP
                FETCH CUR_EMP INTO V_NO, V_SAL;
                EXIT WHEN CUR_EMP%NOTFOUND;
                UPDATE EMP SET SAL = V_SAL + (V_SAL*0.05) WHERE EMP_NO = V_NO;
                INSERT INTO EMP_UPDATE VALUES (V_NO, V_SAL*0.05);
        END LOOP;
                COMMIT;
                CLOSE CUR_EMP;
                DBMS_OUTPUT.PUT_LINE (‘RECORDS ARE UPDATED FOR DEPT_NO = 20’);
        ELSE
                DBMS_OUTPUT.PUT_LINE (‘UNABLE TO OPEN A CURSOR FOR THIS DEPT’);
        END IF;
END;
/
```

Write a program using a cursor to display no., name, and the basic salary of 3 highest paid employees.

```
DECLARE
        VENO SALARY.NO %TYPE;
        VENAME EMP.NAME %TYPE;
        VESAL SALARY.BASIC %TYPE;

        CURSOR CUR_SAL IS SELECT DISTINCT SALARY.NO, EMP.NAME, SALARY.BASIC
        FROM SALARY, EMP WHERE SALARY.NO=EMP.ENO ORDER BY BASIC DESC;
BEGIN
        OPEN CUR_SAL;
        LOOP
                FETCH CUR_SAL INTO VENO, VENAME, VESAL;

                EXIT WHEN CUR_SAL %NOTFOUND;
                EXIT WHEN (CUR_SAL %ROWCOUNT > 3);
                DBMS_OUTPUT.PUT_LINE ('NO -' || VENO || ' NAME -' || VENAME || 'SAL -' || VESAL);
        END LOOP;

        CLOSE CUR_SAL;
END;
/
```

CURSOR FOR LOOP:

The cursor FOR loop can be used to process multiple records. The advantage of cursor FOR loop is that the loop itself will OPEN a cursor, FETCHES (reads) the records into the cursor from the table until EOF is encounters, then it CLOSES the cursor.

<u>SYNTAX:</u>

```
FOR <VARIABLE> IN <CURSOR_NAME> LOOP
        <STATEMENTS>;
END LOOP;
```

```
FOR record_name IN cursor_name LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

The VARIABLE used in the cursor FOR loop is not necessary to declare. It will be automatically created by the cursor FOR loop.

Write a PL/SQL block with the use of cursor FOR loop to display the information of EMP table.

```
DECLARE
    CURSOR CUR_EMP4 IS SELECT * FROM EMP;
    M_EMPREC EMP%ROWTYPE;

BEGIN
    FOR M_EMPRECH IN CUR_EMP4 LOOP
        DBMS_OUTPUT.PUT_LINE (RPAD (M_EMPREC.EMP_NAME,15) || M_EMPREC.SAL);
    END LOOP;
END;
/
```

Write a program using a cursor to raise the salary of employees of department number 20 by 5% and also display the appropriate message based on the existence to the record in the EMP table. Whenever any such raise is given to the employees, a record for the same is maintained in the emp_update table.

```
DECLARE

CURSOR CUR_EMP3 IS SELECT EMP_NO, SAL FROM EMP WHERE DEPT_NO = 20;

BEGIN
    FOR EMP_REC IN CUR_EMP3 LOOP
        UPDATE EMP SET SAL = V_SAL + (V_SAL*0.5) WHERE EMP_NO = V_NO;
        INSERT INTO EMP_UPDATE VALUES (V_NO, V_SAL*0.05);
    END LOOP;
    COMMIT;
        DBMS_OUTPUT.PUT_LINE  ('RECORDS ARE UPDATED FOR DEPT_NO = 20');
    ELSE
        DBMS_OUTPUT.PUT_LINE  ('UNABLE TO OPEN A CURSOR FOR THIS DEPT');
    END IF;
END;
/
```

PARAMETERIZED CURSOR:

A cursor can be declared with parameters, which allows you to pass values to the cursor. The values are passed to the cursor when it is opened, and they are used in the query when it is executed. With the use of parameters you can open and close a cursor many times with different values. The cursor with different values will then return different active sets each time it is opened. When parameters are passed, you need not worry about the scope of variable. The general syntax for parameterized cursor is:

SYNTAX:

```
CURSOR <CURSOR_NAME>
    [(PARAMETER1 DATATYPE, PARMETER2 DATATYPE,….)]
IS <SELECT STATEMENT>;
```

Declaring the parameterized cursor:

```
CURSOR cursor_name
  [(parameter_name datatype, ...)]
IS
  select_statement;
```

Opening a parameterized cursor:

```
OPEN  cursor_name(parameter_value,.....) ;
```

Write a program using a parameterized cursor that deals with bank transactions. If amount is deposited into the bank then that amount is added into the BALANCE, otherwise if some amount is withdrawn from the account then BALACE will be reduced.

CREATE THE FOLLOWING TABLES:

BANK_MASTER	
ACCNO	**BALANCE**
10	1000
20	800
30	7000
40	1000

BANK_DETAIL		
ACCNO	**TYPE**	**AMOUNT**
10	D	1000
20	W	800

```
DECLARE

CURSOR C_ACC (ACC_NO NUMBER) IS SELECT BALANCE FROM BANK_MASTER WHERE ACCNO =
ACC_NO;
CURSOR C_TRAN (NO NUMBER, TTYPE CHAR) IS SELECT * FROM BANK_DETAIL WHERE ACCNO =
NO AND TYPE = TTYPE;

        MACCNO BANK_DETAIL.ACCNO%TYPE;
        MTYPE  BANK_DETAIL.TYPE%TYPE;
        MAMT   BANK_DETAIL.AMOUNT%TYPE;

        MBAL   BANK_MASTER.BALANCE%TYPE;

BEGIN

        MACCNO: = &MACCNO;
        MTYPE: = '&MTYPE';
        MAMT: = &AMOUNT;

        OPEN C_ACC (MACCNO);
        FETCH C_ACC INTO MBAL;
```

```
        IF C_ACC%FOUND THEN

                IF MAMT < MBAL THEN
                        IF UPPER (MTYPE) = 'W' THEN
                                UPDATE BANK_MASTER SET BALANCE = MBAL - MAMT WHERE
                                MACCNO = ACCNO;
                                INSERT INTO BANK_DETAIL (ACCNO, TYPE, AMOUNT)
                                VALUES (MACCNO, MTYPE, MAMT);
                        ELSE
                                UPDATE BANK_MASTER SET BALANCE = MBAL + MAMT WHERE
                                MACCNO = ACCNO;
                                INSERT INTO BANK_DETAIL (ACCNO, TYPE, AMOUNT) VALUES
                                (MACCNO, MTYPE, MAMT);
                        END IF;

                CLOSE C_ACC;
                COMMIT;

                ELSE
                        DBMS_OUTPUT.PUT_LINE
                         ('THE WITHDRAWAL AMOUNT IS GREATER THEN BALACE');
                END IF;

        ELSE
                DBMS_OUTPUT.PUT_LINE
                ('RECORD FOR ACCOUNT NUMBER' ||' '|| MACCNO||' ' || 'DOES NOT EXIST');
        END IF;

        FOR BANKREC IN C_TRAN (MACCNO, MTYPE) LOOP
                DBMS_OUTPUT.PUT_LINE
                (BANKREC.ACCNO ||' '|| BANKREC.TYPE || ' ' || BANKREC.AMOUNT);
        END LOOP;
END;
/
SELECT * FROM BANK_MASTER;
```

VPSC

EXCEPTIONS:

Run-time errors arise from design faults, coding mistakes, hardware failures, and many other sources. Although you cannot anticipate all possible errors, you can plan to handle certain kinds of errors meaningful to your PL/SQL program

An Exception is an error situation, which arises during program execution. When an error occurs exception is raised, normal execution is stopped and control transfers to exception-handling part. Exception handlers are routines written to handle the exception

TYPE OF EXCEPTIONS:

Generally the exceptions are roughly divided into two categories

(1) Predefined Oracle errors /System Name errors
(2) Undefined Oracle errors / System UnName errors
(3) User-defined errors

(1) Predefined Oracle errors

Predefined exception is raised automatically whenever there is a violation of Oracle coding rules. Predefined exceptions are those like ZERO_DIVIDE, which is raised automatically when we try to divide a number by zero.

Other built-in exceptions are given below. You can handle unexpected Oracle errors using OTHERS handler. It can handle all raised exceptions that are not handled by any other handler. It must always be written as the last handler in exception block

1. **CUROSR_ALREADY_OPEN (ORA – 06511):**

   This exception is displayed when user tries to open a cursor that is already open.

2. **INVALID_CUROSR (ORA – 01001):**

   This exception is raised whenever a user references an invalid cursor or attempts an illegal cursor operation.

3. **DUP_VAL_ON_INDEX (ORA – 00001):**

   Whenever a user tries to insert a duplicate value into a unique column then this error is raised

4. **INVALID_NUMBER (ORA – 01722):**

   When the user tires to use something other then the number is numeric field (column) then this exception is raised.

5. **LOGIN_DENIED (ORA – 01017):**

   When user tires to login into the oracle with invalid username & password then this exception is raised.

6. **NO_DATA_FOUND (ORA – 01403):**

   When single row select statement returns no data then this exception is raised.

7. **NOT_LOGGED_ON (ORA – 01012):**

   This exception is raised when user is not connected to oracle.

8. **PORGRAM_ERROR (ORA – 06501):**

   This exception is raised when PL/SQL block has an internal error.

9. **STORAGE_ERROR (ORA – 06500):**

   This exception is raised when PL/SQL have insufficient memory to execute the PL/SQL block.

10. **TOO_MANY_ROWS (ORA – 01422):**

    This exception is raised when a single row select statement returns more then one row.

11. **VALUE_ERROR (ORA – 06502):**

    This exception is raised whenever the user encounters an arithmetic, conversion or constraints error.

12. **ZERO_DIVIDE (ORA – 01476):**

    This error is raised whenever PL/SQL block tries to divide any value by zero.

13. **OTHERS:**

    This flag is used to catch any error situation not coded by the programmer in the exception section. Therefore, it must appear last in the exception section.

VPSC

SYNTAX:

```
Exception
    WHEN <ExceptionName>THEN
        <User define Action to be carried out>
```

Predefined exception handlers are declared globally in package STANDARD.Hence we need not have to define them rather just use them.

The biggest advantage of exception handling is it improves readability and reliability of the code. Errors from many statements of code can be handles with a single handler. Instead of checking for an error at every point we can just add an exception handler and if any exception is raised it is handled by that. For checking errors at a specific spot it is always better to have those statements in a separate begin – end block.

EXAMPLES OF EXCEPTION:

Write a program that explains the use of NO_DATA_FOUND exception.

```
DECLARE
    SALARY NUMBER;

BEGIN
    SELECT SAL INTO SALARY FROM EMP WHERE EMPNO = 100;

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE ('No records are found for empno 100');

END;
```

Write a program that explains the use of TOO_MANY_ROWS exception.

```
DECLARE
    SALARY NUMBER;

BEGIN
    SELECT SAL INTO SALARY FROM EMP WHERE DEPT_NO = 10;

EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE ('More then one rows are selected for department 10');

END;
/
```

Write a program that explains the use of ZERO_DIVIDE exception.

```
DECLARE
    NO NUMBER (3);
```

VPSC

```
BEGIN
        NO:=10/0;

EXCEPTION
        WHEN ZERO_DIVIDE THEN
                DBMS_OUTPUT.PUT_LINE ('You can not divide the number by 0');

END;
```

(2) Undefined Oracle errors

To handle error conditions (typically ORA- messages) that have no predefined name, you must use the OTHERS handler or the pragma EXCEPTION_INIT. A **pragma** is a compiler directive that is processed at compile time, not at run time.

In PL/SQL, the pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle error number. That lets you refer to any internal exception 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

You must code the pragma EXCEPTION_INIT in the declarative part of a PL/SQL block, subprogram, or package

SYNTAX:

```
DECLARE
        <ExceptionName> EXCEPTION;
        PRAGMA EXCEPTION_INIT(<ExceptionName>,<ErrorCodeNo>);
BEGIN
        Statement
EXCEPTION
WHEN <ExceptionName> THEN
        <Action>
END;
```

PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number);

Where exception_name is the name of a previously declared exception and the number is a negative value corresponding to an ORA- error number.

The pragma must appear somewhere after the exception declaration in the same declarative section, When you use EXCEPTION_INIT, you must supply a literal number for the second argument of the pragma call. By explicitly naming this system exception, the purpose of the exception handler is self-evident.

The EXCEPTION_INIT pragma improves the readability of your programs by assigning names to otherwise obscure error numbers. You can employ the EXCEPTION_INIT pragma more than once in your program. You can even assign more than one exception name to the same error number.

Write a program that explains the use of PRAGMA EXCEPTION_INIT exception.

```
create table org_level(company_id number(8)     not null,
org_level varchar2(1) not null );

DECLARE
        Invalid_org_level EXCEPTION;

      PRAGMA EXCEPTION_INIT(invalid_org_level,-2290);

BEGIN

  INSERT INTO org_level VALUES (1001,'P');
  COMMIT;

EXCEPTION
      WHEN invalid_org_level THEN
            DBMS_OUTPUT.PUT_LINE ('Organization Level');

END;
```

- **EXCEPTION TRAPPING FUNCTIONS:**

When an exception occurs in your program, you can find the error code for the error and its associated message. Once you know the error code and the message, you can modify your program to take action based on the error. The two functions to identify the error code and error message.

SQLCODE: The SQLCODE function returns the number of the error code. The number can be assigned to a variable of NUMBER datatype.

SQLERRM: The SQLERRM function returns the error message associated with the error code. It can be assigned to a VARCHAR2 type variable.

(3) User Defined Exceptions:

Exceptions that are defined and raised in the server by the programmer is known as User Defined Exceptions.

A User-defined exception has to be defined by the programmer. User-defined exceptions are declared in the declaration section with their type as exception.

They must be raised explicitly using RAISE statement, unlike pre-defined exceptions that are raised implicitly. RAISE statement can also be used to raise internal exceptions.

SYNTAX

1)Declaring Exception:
DECLARE
<Exceptionname> EXCEPTION;
BEGIN
----
2) Raising Exception:
BEGIN
RAISE <Exceptionname>;
----
3) Handling Exception:
BEGIN

EXCEPTION
WHEN <Exceptionname> THEN
Statements;
END;

User defined exceptions must be declared in the DECLARE section with the reserve word EXCEPTION.

<EXCEPTION_NAME> EXCEPTION;

This exception can be brought into action by the following command in executable section

RAISE <EXCEPTION_NAME>;

When the exception is raised, processing control is passed to the EXCEPTION section of the PL/SQL block.

Therefore, the code for the exception must be defined in the EXCEPTION part of the PL/SQL block.

WHEN <EXCEPTION_NAME> THEN
<ACTION>;

VPSC

Points To Remember:

- An Exception cannot be declared twice in the same block.
- Exceptions declared in a block are considered as local to that block and global to its sub-blocks.
- An enclosing block cannot access Exceptions declared in its sub-block. Where as it possible for a sub-block to refer its enclosing Exceptions.

➢ RAISE_APPLICATION_ERROR PROCEDURE:

The RAISE_APPLICATION_ERROR procedure, is one of the Oracle utilities, which helps the user to manage the error condition in the application by specifying user defined error number and message.

To call `RAISE_APPLICATION_ERROR`, use the syntax

Raise_application_error(error_number, message[, {TRUE | FALSE}]);

Where error_number is a negative integer in the range -20000... -20999 and message is a character string up to 2048 bytes long. If the optional third parameter is TRUE, the error is placed on the stack of previous errors. If the parameter is FALSE (the default), the error replaces all previous errors. RAISE_APPLICATION_ERROR is part of package DBMS_STANDARD, and as with package STANDARD, you do not need to qualify references to it.

Write a program that explains the use of NO_DATA_FOUND exception and RAISE_APPLICATION_ERROR procedure.

```
DECLARE
        SALARY NUMBER;
BEGIN
        SELECT SAL INTO SALARY FROM EMP WHERE EMPNO = 100;
EXCEPTION
        WHEN NO_DATA_FOUND THEN
RAISE_APPLICATION_ERROR (-20100, ' No employees are found for given number');

END;
```

Write a program that explains the use of USER DEFINED exception.

```
DECLARE
        I NUMBER (2): =0;
        ENO NUMBER (4);
        ENAME VARCHAR2 (20);
        DNO NUMBER (3);
```

```
        CURSOR CUR_EMP IS SELECT EMP_NO, EMP_NAME FROM EMP WHERE DEPT_NO = 10;
        NO_DEPT_NO_FOUND EXCEPTION;

BEGIN
        OPEN CUR_EMP;
        LOOP
                FETCH CUR_EMP INTO ENO, ENAME;
                EXIT WHEN CUR_EMP%NOTFOUND;
                I : = I+1;
                DBMS_OUTPUT.PUT_LINE (I|| 'RECORD IS INSERTED INTO EMP_BACKUP TABLE');
                DBMS_OUTPUT.PUT_LINE (ENO || ENAME);
                INSERT INTO EMP_BACKUP VALUES (ENO, ENAME);
        END LOOP;

        IF CUR_EMP%ROWCOUNT = 0 THEN
                CLOSE CUR_EMP;
                RAISE NO_DEPT_NO_FOUND;
        END IF;

        DBMS_OUTPUT.PUT_LINE ('TOTAL NO. OF RECORDS INSERTED INTO EMP_BACKUP
        TABLE ARE' || I);
        CLOSE CUR_EMP;

EXCEPTION
        WHEN NO_DEPT_NO_FOUND THEN
DBMS_OUTPUT.PUT_LINE ('NO RECORD FOR DEPT NO. 10');
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE ('PL/SQL CODE ENCOUNTERED AN ERROR');

END;
/
```

Write a program that explains the use of USER DEFINED exception.

```
DECLARE
        COMISSION NUMBER (5,2);
        NULL_COMISSION EXCEPTION;
BEGIN
        SELECT EMP_COMISSION INTO COMISSION FROM EMP WHERE EMP_NO = 100;

        IF COMISSION IS NULL THEN
                RAISE NULL_COMMISSION;
        END IF;
EXCEPTION
        WHEN NULL_COMISSION THEN
                DBMS_OUTPUT.PUT_LINE ('NO COMISSION FOUND FOR GIVEN EMPLOYEE');
END;
/
```