Concepts of Cursors

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

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 Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. **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. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The following table provides the description of the most used attributes –

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Any SQL cursor attribute will be accessed as **sql%attribute_name** as shown below in the example.

Example

We will be using the CUSTOMERS table.

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected −

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers
   SET salary = salary + 500;
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

6 customers selected

PL/SQL procedure successfully completed.

If you check the records in customers table, you will find that the rows have been updated –

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

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.

The **syntax** for creating an explicit cursor is –

CURSOR cursor_name IS select_statement;

Working with an explicit cursor includes the following steps –

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

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example –

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows –

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows –

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows –

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
   c_id customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
```

5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.

Exception handling in PL/SQL

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

By handling the exceptions we can ensure a PL/SQL block does not exit abruptly.

Structure of Exception Handling.

General Syntax for coding the exception section

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

General PL/SQL statements can be used in the Exception Block.
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 for 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

a) 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 is known as Named System Exceptions.
For example: NO_DATA_FOUND and ZERO_DIVIDE are called 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.

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

For Example: Suppose a NO_DATA_FOUND exception is raised in a program, 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;
```

b) Unnamed System Exceptions

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

We can assign a name to unnamed system exceptions using a **Pragma** called **EXCEPTION_INIT.**

EXCEPTION_INIT will associate a predefined Oracle error number to a programmer defined exception name.

Steps to be followed to use unnamed system exceptions are

- They are raised implicitly.
- If they are not handled in WHEN others they must be handled explicitly.
- To handle the exception explicitly, they must be declared using Pragma

EXCEPTION_INIT as given above and handled the user-defined exception name in the exception section.

The general syntax to declare unnamed system exception using EXCEPTION_INIT is:

```
DECLARE
        Exception_name EXCEPTION;
        PRAGMA EXCEPTION_INIT (exception_name, Err_code);
BEGIN
        Execution section
EXCEPTION
        WHEN exception_name THEN handle the exception
END;
```

For Example: Let's consider the customer table and orders table. Here cid is a primary key in customer table and a foreign key in orders table.

If we try to delete cid from the customer table when it has child records in orders table an exception will be thrown with oracle code number -2292.

We can provide a name to this exception and handle it in the exception section as given below.

```
DECLARE
Child_rec_exception EXCEPTION;
```

```
PRAGMA EXCEPTION_INIT (Child_rec_exception, -2292);
BEGIN
Delete FROM customer where cid= 101;
EXCEPTION
WHEN Child_rec_exception
THEN Dbms_output.put_line ('Child records are present for this cid.');
END;
/
```

c) User-defined Exceptions

Apart from sytem 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.

Syntax:

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

For Example: This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid_id** is raised.

```
DECLARE
  c_id customers.id%type := &c_id;
  c_name customerS.Name%type;
  c_addr customers.address%type;
  -- user defined exception
  ex_invalid_id  EXCEPTION;
BEGIN
  IF c_id <= 0 THEN
    RAISE ex_invalid_id;
  ELSE
    SELECT  name, address INTO  c_name, c_addr
    FROM customers
    WHERE id = c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
```

```
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
  END IF;

EXCEPTION
  WHEN ex_invalid_id THEN
    dbms_output.put_line('ID must be greater than zero!');
  WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

RAISE_APPLICATION_ERROR ( )

RAISE_APPLICATION_ERROR is a built-in procedure in oracle which is used to display the user-defined error messages along with the error number whose range is in between -20000 and -20999.
Whenever a message is displayed using RAISE_APPLICATION_ERROR, all previous transactions which are not committed within the PL/SQL Block are rolled back automatically (i.e. change due to INSERT, UPDATE, or DELETE statements). RAISE_APPLICATION_ERROR raises an exception but does not handle it.
RAISE_APPLICATION_ERROR is used for the following reasons,
a) to create a unique id for an user-defined exception.

b) to make the user-defined exception look like an Oracle error. The General Syntax to use this procedure is:
RAISE_APPLICATION_ERROR (error_number, error_message);

- The Error number must be between -20000 and -20999
- The Error_message is the message you want to display when the error occurs.

Steps to be folowed to use RAISE_APPLICATION_ERROR procedure:

1. Declare a user-defined exception in the declaration section.
2. Raise the user-defined exception based on a specific business rule in the execution section.
3. Finally, catch the exception and link the exception to a user-defined error number in RAISE_APPLICATION_ERROR.

Example:

```
SET SERVEROUTPUT ON;
DECLARE
        kerror exception;
        Balance integer := &balance;
BEGIN
        if balance<100 then
```

```
                raise kerror;
        else
                dbms_output.put_line('balance:'||balance);
        end if;
EXCEPTION
        when kerror then
                raise_application_error(-20343,'The balance is too low');
END;
/
```

OR

```
SET SERVEROUTPUT ON;
DECLARE
        balance NUMBER:=&balance;
BEGIN
        If balance<=100 then
                Raise_application_error(-20343,'the balance is too low');
        End if;
END;
/
```