

PL/SQL

("Procedural Language
extensions to SQL")

Introduction to PL/SQL

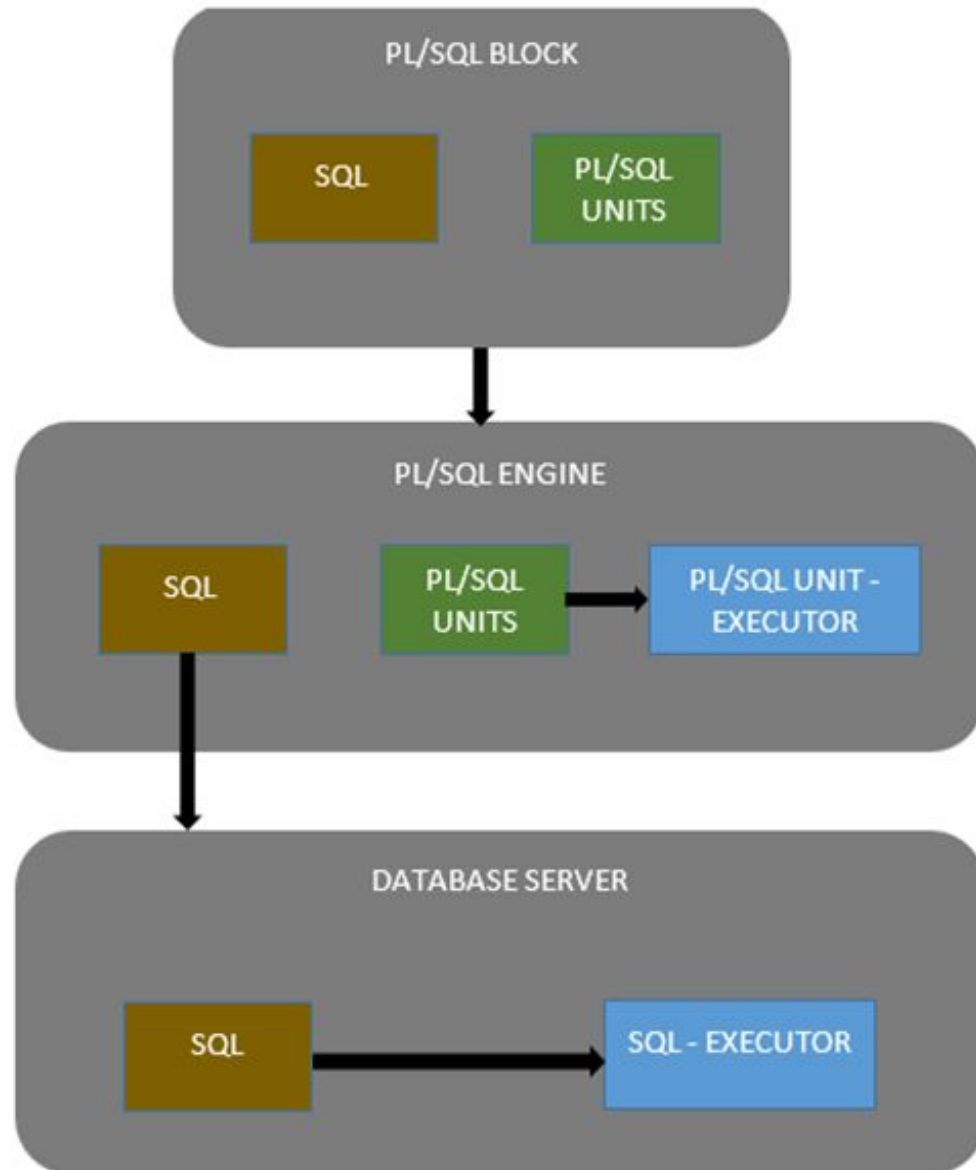
- ❖ PL SQL basically stands for "Procedural Language extensions to SQL".
- ❖ It is **Extension** of Structured Query Language (SQL) that is used in Oracle.
- ❖ Unlike SQL, PL/SQL allows the programmer to write code in **procedural format**.
- ❖ It combines the **data manipulation power of SQL** with the processing power of procedural language to **create a super powerful SQL queries**.
- ❖ It allows the programmers to instruct the compiler '**what to do**' through SQL and '**how to do**' through its procedural way.
- ❖ PL/SQL is a **completely portable, high-performance transaction-processing language**.
- ❖ PL/SQL provides a **built-in, interpreted and OS independent programming environment**.

Advantage of Using PL/SQL

- ❖ Better performance, as SQL is executed in bulk rather than a single statement
- ❖ Block Structures
- ❖ Procedural Language Capability
- ❖ Full Portability
- ❖ Support Object Oriented Programming concepts
- ❖ Error Handling

SQL	PL/SQL
SQL is a single query that is used to perform DML and DDL operations.	PL/SQL is a block of codes that used to write the entire program blocks/ procedure/ function, etc.
It is declarative, that defines what needs to be done, rather than how things need to be done.	PL/SQL is procedural that defines how the things needs to be done.
Execute as a single statement.	Execute as a whole block.
Mainly used to manipulate data.	Mainly used to create an application
Cannot contain PL/SQL code in it.	It is an extension of SQL, so it can contain SQL inside it.

Architecture of PL/SQL



Block Structure of PL/SQL

DECLARE
Variable declaration (Optional)

BEGIN
Program Execution
SQL Statement (Mandatory)

EXCEPTION
Exception handling (Optional)

END; (Mandatory)

Displaying user messages on screen

dbms_output : It is a package that includes a number of procedures and functions that accumulate information in a buffer so that it can be retrieved later.

put_line: puts a piece of information in the package buffer followed by an end-of-line marker. Used to display message onscreen.

Display messages on screen:

Set serveroutput on

PL/SQL First Program: Hello World

set serveroutput on

BEGIN

 dbms_output.put_line('Hello World');

END;

/

Output

Hello World

PL/SQL procedure successfully completed.

PL/SQL Variables

- ❖ It needs to declare the variable first in the declaration section of a PL/SQL block before using it.
- ❖ By default, variable names are not case sensitive.
- ❖ A reserved PL/SQL keyword cannot be used as a variable name.
- ❖ **Syntax for declaring variable:**

***variable_name* datatype(size) [NOT NULL] [:= value];**

- ✓ *variable_name* is the name of the variable.
- ✓ *datatype* is a valid SQL datatype.
- ✓ *NOT NULL* is an optional specification on the variable.
- ✓ *value or DEFAULT value* is also an optional specification, where you can initialize a variable.
- ✓ Each variable declaration is a separate statement and must be terminated by a semicolon.

PL/SQL Variables

❖ Example declaring variable:

- ✓ For example, if you want to store the current salary of an employee
- ✓ When a variable is specified as NOT NULL, you must initialize the variable when it is declared.

DECLARE

```
salary number(4);
```

```
dept varchar2(10) NOT NULL := "Comp";
```

```
Desg varchar2(10) := "HR";
```

"Hello World" using the variables

DECLARE

```
message varchar2(20):= 'Hello World!';
```

BEGIN

```
dbms_output.put_line(message);
```

END;

/

Output

Hello World!

PL/SQL procedure successfully completed.

Example of initializing variable

DECLARE

a integer := 30;

b integer := 40;

c integer;

f real;

BEGIN

c := a + b;

dbms_output.put_line('Value of c: ' || c);

f := 100.0/3.0;

dbms_output.put_line('Value of f: ' || f);

END;

/

OUTPUT

Value of c: 70

Value of f: 33.33333333333333333333

PL/SQL procedure successfully completed.

Assign values to Variables

- ❖ We can assign values to variables in the two ways given below.
- We can directly assign values to variables.
- ✓ The General Syntax is:

`variable_name := value;`

- We can assign values to variables directly from the database columns by using a SELECT.. INTO statement.
- ✓ The General Syntax is:

```
SELECT column_name  
INTO variable_name  
FROM table_name  
[WHERE condition];
```

Assign values to Variables

DECLARE

```
var_salary number(6);  
var_emp_id number(6) := 101;
```

BEGIN

```
SELECT salary INTO var_salary FROM employee  
WHERE emp_id = var_emp_id;  
dbms_output.put_line('The employee ' || var_emp_id || ' has  
salary ' || var_salary);  
END;  
/
```

OUTPUT

```
The employee 101 has salary 2500  
PL/SQL procedure successfully completed.
```


Variable Scope in PL/SQL

- ❖ PL/SQL allows nesting of blocks.
- ❖ A program block can contain another inner block.
- ❖ If you declare a variable within an inner block, it is not accessible to an outer block.
- ❖ There are two types of variable scope:
 - **Local Variable:** Local variables are the inner block variables which are not accessible to outer blocks.
 - **Global Variable:** Global variables are declared in outermost block.

Variable Scope in PL/SQL

DECLARE

-- Global variables

```
a integer := 10;  
b integer := 20;  
c integer;
```

BEGIN

```
dbms_output.put_line('Outer Variable a: ' || a);  
dbms_output.put_line('Outer Variable b: ' || b);  
c := a + b;  
dbms_output.put_line('Value of c: ' || c);
```

DECLARE

-- Local variables

```
a integer := 40;  
b integer := 30;  
d integer;
```

BEGIN

```
dbms_output.put_line('Inner Variable a: ' || a);  
dbms_output.put_line('Inner Variable b: ' || b);  
d:= a - b;  
dbms_output.put_line('Value of d: ' || d);
```

END;

END;

/

OUTPUT

```
Outer Variable a: 10  
Outer Variable b: 20  
Value of c: 30  
Inner Variable a: 40  
Inner Variable b: 30  
Value of d: 10
```

PL/SQL procedure
successfully completed.

PL/SQL Constants

❖ Syntax to declare a constant:

constant_name CONSTANT datatype := VALUE;

- ✓ **Constant_name:** it is the name of constant just like variable name. The constant word is a reserved word and its value does not change.
- ✓ **VALUE:** it is a value which is assigned to a constant when it is declared. It can not be assigned later.

Example PL/SQL Constants

DECLARE

-- constant declaration

pi constant number := 3.141592654;

-- other declarations

radius number(5,2);

dia number(5,2);

circumference number(7, 2);

area number (10, 2);

BEGIN

radius := 9.5;

dia := radius * 2;

circumference := 2.0 * pi * radius;

area := pi * radius * radius;

dbms_output.put_line('Radius: ' || radius);

dbms_output.put_line('Diameter: ' || dia);

dbms_output.put_line('Circumference: ' || circumference);

dbms_output.put_line('Area: ' || area);

END;

/

OUTPUT

Radius: 9.5

Diameter: 19

Circumference: 59.69

Area: 283.53

PL/SQL procedure successfully completed.

Control Statements IF

- ❖ PL/SQL supports the programming language features like conditional statements and iterative statements.
- ❖ Its programming constructs are similar to how you use in programming languages like Java and C++.
- ❖ There are different syntaxes for the IF-THEN-ELSE statement.

PL/SQL If Statement

Syntax: (IF-THEN statement):

```
IF condition
THEN
Statement: It is executed when
condition is true
END IF;
```

Syntax: (IF-THEN-ELSE statement):

```
IF condition
THEN
    {statements to execute when condition is TRUE}
ELSE
    {statements to execute when condition is FALSE}
END IF;
```

Syntax: (IF-THEN-ELSIF statement):

```
IF condition1
THEN
    {statements to execute when
condition1 is TRUE...}
ELSIF condition2
THEN
    {statements to execute when
condition2 is TRUE...}
END IF;
```

Syntax: (IF-THEN-ELSIF-ELSE statement):

```
IF condition1
THEN
    {statements to execute when condition1 is TRUE..}
    ELSIF condition2
THEN
    {statements to execute when condition2 is TRUE..}
ELSE
    {statements to execute when both condition1 and
condition2 are FALSE...}
END IF;
```


Example of PL/SQL If Statement

DECLARE

a number(3) := 500;

BEGIN

-- check the boolean condition using if statement

IF(a < 20) **THEN**

-- if condition is true then print the following

dbms_output.put_line('a is less than 20 ');

ELSE

dbms_output.put_line('a is not less than 20 ');

END IF;

dbms_output.put_line('value of a is : ' || a);

END;

/

OUTPUT

a is not less than 20

value of a is : 500

PL/SQL procedure successfully completed.

Example of PL/SQL Case Statement

❖ Syntax for the CASE Statement:

```
CASE [ expression ]  
WHEN condition_1 THEN result_1  
  WHEN condition_2 THEN result_2  
  ...  
  WHEN condition_n THEN result_n  
ELSE result  
END CASE
```

❖ Example for the CASE Statement:

```
DECLARE  
  grade char(1) := 'A';  
BEGIN  
  CASE grade  
  when 'A' then dbms_output.put_line('Excellent');  
  when 'B' then dbms_output.put_line('Good');  
  when 'C' then dbms_output.put_line('Average');  
  else dbms_output.put_line('Failed');  
  END CASE;  
END;  
/
```

OUTPUT

Excellent

PL/SQL procedure successfully completed.

PL/SQL Loop

- ❖ The PL/SQL loops are used to repeat the execution of one or more statements for specified number of times.
- ❖ These are also known as iterative control statements.
- ❖ **Syntax for a basic loop:**

LOOP

Sequence of statements;

END LOOP;

- ❖ **Types of PL/SQL Loops**
 1. Basic Loop / Exit Loop
 2. While Loop
 3. For Loop

PL/SQL Exit Loop

- ❖ PL/SQL exit loop is used when a set of statements is to be executed at least once before the termination of the loop.
- ❖ There must be an EXIT condition specified in the loop, otherwise the loop will get into an infinite number of iterations.

❖ Syntax of Exit loop:

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

❖ Example of Exit loop:

```
DECLARE
i integer := 1;
BEGIN
Loop
Exit When i > 10;
dbms_output.put_line(i);
i := i+1;
END Loop;
END;
```

/

OUTPUT

```
1
2
3
4
5
6
7
8
9
10
```

PL/SQL procedure successfully completed.

PL/SQL While Loop

❖ Syntax of While loop:

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

❖ Example of While loop:

```
DECLARE  
i integer := 1;  
BEGIN  
WHILE i <= 10 LOOP  
  dbms_output.put_line(i);  
  i := i+1;  
END LOOP;  
END;  
/
```

OUTPUT

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

PL/SQL procedure successfully completed.

❖ Important steps to follow when executing a while loop:

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

PL/SQL FOR Loop

❖ Syntax of For loop:

```
FOR counter IN initial_value .. final_value  
  LOOP statements;  
END LOOP;
```

- ✓ initial_value : Start integer value
- ✓ final_value : End integer value

❖ Example of For loop:

```
BEGIN  
FOR k IN 1..10  
  LOOP  
    -- note that k was not declared  
    dbms_output.put_line(k);  
  END LOOP;  
END;  
/
```

OUTPUT

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

PL/SQL procedure successfully completed.

PL/SQL Stored Procedure

- ❖ The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.
- ❖ A procedure may or may not return any value
- ❖ The procedure contains a header and a body.
- **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
- **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

Procedures: Passing Parameters

◆ **IN parameters:**

- The IN parameter can be referenced by the procedure or function.
- This parameter is used for giving input to the subprograms.
- It is a read-only variable inside the subprograms, their values cannot be changed inside the subprogram

◆ **OUT parameters:**

- The OUT parameter cannot be referenced by the procedure or function.
- This parameter is used for getting output from the subprograms.
- It is a read-write variable inside the subprograms, their values can be changed inside the subprograms.

◆ **INOUT parameters:**

- The INOUT parameter can be referenced by the procedure or function.
- This parameter is used for both giving input and for getting output from the subprograms.
- It is a read-write variable inside the subprograms, their values can be changed inside the subprograms.

PL/SQL Create Procedure

Syntax for creating procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name  
    [ (parameter_name {IN | OUT | INOUT} datatype , ..... ) ]
```

IS | AS

[Declaration_section]

BEGIN

Executable_section

[EXCEPTION

Exception_section]

END [procedure_name];

/

Syntax for drop procedure

```
DROP PROCEDURE procedure_name
```


Create Procedure: Example

```
CREATE OR REPLACE PROCEDURE Hello(message IN  
VARCHAR2)  
IS  
BEGIN  
    dbms_output.put_line('Hello !!! How are you '||message);  
END;  
/
```

OUTPUT

```
Exec Hello('Shraddha');
```

```
Hello !!! How are you Shraddha
```

Create Procedure: Example

◆ Table creation:

```
create table student(id number(10) Primary key,name varchar2(20));
```

Now write the procedure code to insert record in user table.

◆ Procedure Code:

```
create or replace procedure studentdata(id IN NUMBER,name IN  
    VARCHAR2)
```

```
is
```

```
begin
```

```
insert into student values(id,name);
```

```
end;
```

```
/
```

Procedure created.

Create Procedure Example

◆ PL/SQL program to call procedure

```
SQL> select * from student;
```

```
no rows selected
```

```
BEGIN
```

```
    studentdata(101,'Rahul');
```

```
    dbms_output.put_line('record inserted successfully');
```

```
END;
```

```
/
```

```
record inserted successfully
```

```
PL/SQL procedure successfully completed.
```

```
SQL> select * from student;
```

```
      ID NAME
```

```
-----  
    101 Rahul
```


PL/SQL Function

- ❖ The PL/SQL Function is very similar to PL/SQL Procedure.
- ❖ The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value.
- ❖ Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

PL/SQL Function

❖ Syntax to create a function:

```
CREATE [OR REPLACE] FUNCTION function_name  
[( parameter_name {IN } datatype , ... )]
```

```
RETURN return_datatype
```

```
{IS | AS}
```

```
-- declaration can be done here
```

```
BEGIN
```

```
< function_body >
```

```
END [function_name];
```

```
/
```

Syntax for removing your created function:

```
DROP FUNCTION function_name;
```

PL/SQL Function

❖ Simple example to create a function

```
create or replace function adder(n1 IN number, n2 IN number)
  return number
IS
  n3 number(8);
BEGIN
  n3 :=n1+n2;
  return n3;
END;
/
```

Function created.

PL/SQL Function

❖ Program to call the function.

```
DECLARE
```

```
  n3 number(2);
```

```
BEGIN
```

```
  n3 := adder(11,22);
```

```
  dbms_output.put_line('Addition is: ' || n3);
```

```
END;
```

```
/
```

OUTPUT

Addition is: 33

PL/SQL procedure successfully completed.

PL/SQL Cursor

- ❖ When an SQL statement is processed, Oracle creates a memory area known as **context area**.
- ❖ A cursor is a pointer to this **context area**.
- ❖ It contains all information needed for processing the statement.
- ❖ In PL/SQL, the context area is controlled by **Cursor**.
- ❖ A **cursor** contains information on a select statement and the rows of data accessed by it.
- ❖ 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 data set**.

PL/SQL Implicit Cursors

- ❖ The **implicit cursors** are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.
- ❖ These are created by default to process the statements when DML statements like **INSERT, UPDATE, DELETE** etc. are executed.
- ❖ Oracle provides some attributes known as **Implicit cursor's attributes** to check the status of DML operations.

%FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

Implicit Cursor Attributes

%FOUND - **SQL%FOUND**

Return Value

TRUE - if the DML statements like INSERT, DELETE and UPDATE affect at least one row And if SELECTINTO statement return at least one row.

FALSE - if DML statements like INSERT,DELETE and UPDATE do not affect row and if SELECT....INTO statement do not return a row

Implicit Cursor Attributes

%NOTFOUND - **SQL%NOTFOUND**

Return Value

TRUE - if the DML statements like INSERT, DELETE and UPDATE do not affect at least one row and if SELECT ...INTO statement does not return any row.

FALSE - if the DML statements like INSERT, DELETE and UPDATE affect at least one row And if SELECT ...INTO statement return at least one row.

Implicit Cursor Attributes

%ROWCOUNT - **SQL%ROWCOUNT**

Return Value - Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT.

%ISOPEN - **SQL%ISOPEN**

It always returns **FALSE** for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements.

PL/SQL Implicit Cursors Example

BEGIN

```
UPDATE employee SET branch = 'Pune' where emp_id  
= 101;
```

```
IF sql%found THEN
```

```
    dbms_output.put_line('Branch updated successfully');
```

```
END IF;
```

```
IF sql%notfound THEN
```

```
    dbms_output.put_line('Emp id does not exist ');
```

```
END IF;
```

END;

/

OUTPUT

Branch updated successfully

PL/SQL procedure successfully completed

PL/SQL Implicit Cursors Example

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;

OUTPUT

6 customers selected

PL/SQL procedure successfully completed

PL/SQL Explicit Cursors

- ❖ The Explicit cursors are defined by the **programmers** to gain more control over the **context area**.
- ❖ These cursors 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.
- ❖ General Syntax for creating a cursor:
CURSOR cursor_name **IS** select_statement;
- ❖ **Steps:**
 - Declare the cursor to initialize in the memory.
 - Open the cursor to allocate memory.
 - Fetch the cursor to retrieve data.
 - Close the cursor to release allocated memory.

PL/SQL Explicit Cursors

◆ 1) Declare the cursor:

It defines the cursor with a name and the associated SELECT statement.

```
CURSOR cursor_name IS SELECT statement;
```

◆ 2) Open the cursor:

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

```
OPEN cursor_name;
```

◆ 3) Fetch the cursor:

It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:

```
FETCH cursor_name INTO variable_list;
```

◆ 4) Close the cursor:

It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

```
CLOSE cursor_name;
```

PL/SQL Explicit Cursors

◆ General Form of using an explicit cursor is:

DECLARE

CURSOR <cursor_name> IS <SELECT statement>;
<cursor_variable declaration>;

BEGIN

OPEN <cursor_name>;

FETCH <cursor_name> INTO <cursor_variable>;

•

•

CLOSE <cursor_name>;

END;

PL/SQL Explicit Cursors

- ❖ When a cursor is opened, the first row becomes the current row. When the data is fetched it is copied to the record or variables and the logical pointer moves to the next row and it becomes the current row.
- ❖ **Points to remember while fetching a row:**
 - We can fetch the rows in a cursor into a PL/SQL, record or a list of variables created in the PL/SQL Block.
 - If you are fetching a cursor to a list of variables, the variables should be listed in the same order in the fetch statement as the columns are present in the cursor.

PL/SQL Explicit Cursors

- ◆ **When does an error occur while accessing an explicit cursor?**
 - When we try to open a cursor which is not closed in the previous operation.
 - When we try to fetch a cursor after the last operation.

Attributes	Return Values
%FOUND Cursor_name%FOUND	TRUE, if fetch statement returns at least one row. FALSE, if fetch statement doesn't return a row.
%NOTFOUND Cursor_name %NOTFOUND	TRUE, , if fetch statement doesn't return a row. FALSE, if fetch statement returns at least one row.
%ROWCOUNT Cursor_name %ROWCOUNT	The number of rows fetched by the fetch statement If no row is returned, the PL/SQL statement returns an error.
%ISOPEN Cursor_name%ISOPEN	TRUE, if the cursor is already open in the program FALSE, if the cursor is not opened in the program.

Attributes

Return values

PL/SQL Explicit Cursors Example

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;

/

OUTPUT

```
1 abc pune  
2 pqr mumbai  
3 xyz nasik
```

PL/SQL procedure successfully completed

PL/SQL Explicit Cursors Example

DECLARE

```
CURSOR c_emp IS SELECT emp_name FROM emp;
```

```
c_emp_name emp.emp_name%type;
```

BEGIN

```
OPEN c_emp;
```

LOOP

```
    FETCH c_emp INTO c_emp_name;
```

```
    IF c_emp%NOTFOUND THEN
```

```
        EXIT;
```

```
    END IF;
```

```
    Dbms_output.put_line('Employee
```

OUTPUT

Employee Fetched:BBB

Employee Fetched:XXX

Employee Fetched:YYY

Total rows fetched is 3

Explicit Cursors Example using FOR loop

DECLARE

```
CURSOR c_emp IS SELECT emp_name FROM emp;  
c_emp_name emp.emp_name%type;
```

BEGIN

```
FOR c_emp_name IN c_emp
```

```
LOOP
```

```
Dbms_output.put_line('Employee Fetched:'||c_emp_name);
```

```
END LOOP;
```

END;

/

OUTPUT

Employee Fetched:BBB

Employee Fetched:XXX

Employee Fetched:YYY

Parameterized Cursor

- A cursor that accepts user defined values into its parameters, thus changing the Result extracted, it is called as **Parameterized cursor**.
- PL/SQL Parameterized cursor pass the parameters into a cursor and use them into query.
- PL/SQL Parameterized cursor define only datatype of parameter and not need to define it's length.
- Parameterized cursors are also saying static cursors that can passed parameter value when cursor are opened.

Parameterized Cursor

□ **Syntax for declaring parameterized cursor:**

```
CURSOR cursor_name (variable_name datatype)
IS
select_query;
```

□ **Syntax for opening cursor:**

```
OPEN cursor_name (value_list);
```

Parameterized Cursor Example

DECLARE

```
cursor c(no number) is select * from emp_information
                        where emp_no = no;

tmp emp_information%rowtype;
```

BEGIN

```
OPEN c(4);
```

```
FOR tmp IN c(4)
```

```
LOOP
```

```
  dbms_output.put_line('EMP_No: ' || tmp.emp_no);
```

```
  dbms_output.put_line('EMP_Name: ' || tmp.emp_name);
```

```
  dbms_output.put_line('EMP_Dept: ' || tmp.emp_dept);
```

```
EMP_No: 4
EMP_Name: Zenia Sroll
EMP_Dept: Web Developer
EMP_Salary: 42000

PL/SQL procedure successfully completed.
```


Parameterized Cursor Example

DECLARE

```
rec_product products%ROWTYPE;  
CURSOR cur_product (low_price NUMBER,  
high_price NUMBER)  
IS  SELECT * FROM products  
    WHERE list_price BETWEEN low_price AND  
    high_price;
```

BEGIN

```
OPEN cur_product(50,100);  
LOOP  
    FETCH cur_product INTO rec_product;  
    EXIT WHEN cur_product%NOTFOUND;
```

PL/SQL Trigger

- ❖ A database trigger is a stored procedure that automatically executes whenever an event occurs. The event may be insert-delete-update operations.
- ❖ Trigger is invoked by Oracle engine automatically whenever a specified event occurs.
- ❖ Trigger is stored into database and invoked repeatedly, when specific condition match.
- ❖ Triggers could be defined on the table, view, schema, or database with which the event is associated.

PL/SQL Trigger

- ❖ A **procedure** is executed **explicitly** from another block via a procedure call with passing arguments,
- ❖ While a **trigger** is executed (or fired) **implicitly** whenever the triggering event (DML: INSERT, UPDATE, or DELETE) happens, and a **trigger doesn't accept arguments**.
- ❖ Triggers has three basic parts:
 - **Triggerring Event or Statement** – It is a SQL statement that causes a trigger to be fired.
 - **Trigger Restriction** – A trigger restriction specifies a boolean(logical) expression that must be TRUE for the trigger to fire.
 - **Trigger Action** – Action to be taken when trigger statement is encountered.

Types of Triggers

- ◆ **BEFORE Trigger** : BEFORE trigger execute before the triggering DML statement (INSERT, UPDATE, DELETE) execute. Triggering SQL statement is may or may not execute, depending on the BEFORE trigger conditions block.
- ◆ **AFTER Trigger** : AFTER trigger execute after the triggering DML statement (INSERT, UPDATE, DELETE) executed. Triggering SQL statement is execute as soon as followed by the code of trigger before performing Database operation.

Types of Triggers

- ❖ **ROW Trigger** : ROW trigger fire for each and every record which are performing INSERT, UPDATE, DELETE from the database table. If row deleting is define as trigger event, then trigger is fire, each time row is deleted from the table.
- ❖ **Statement Trigger** : Statement trigger fire only once for each statement. If row deleting is define as trigger event, then trigger is fire, as all five rows deleted from the table.

Types of Triggers

- ❖ **Combination Trigger** : Combination trigger are combination of two trigger type:
 - **Before Statement Trigger** : Trigger fire only once for each statement before the triggering DML statement.
 - **Before Row Trigger** : Trigger fire for each and every record before the triggering DML statement.
 - **After Statement Trigger** : Trigger fire only once for each statement after the triggering DML statement executing.
 - **After Row Trigger** : Trigger fire for each and every record after the triggering DML statement executing.

Syntax of Trigger

CREATE [**OR REPLACE**] **TRIGGER** trigger_name

{**BEFORE** | **AFTER** | **INSTEAD OF**}

{**INSERT** [**OR**] | **UPDATE** [**OR**] | **DELETE**}

[**OF** col_name]

ON table_name

[**REFERENCING OLD AS** o **NEW AS** n]

FOR EACH ROW | **FOR EACH STATEMENT** [**WHEN** Condition]

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

Syntax of Trigger

- ❖ **CREATE [OR REPLACE] TRIGGER trigger_name:** It creates or replaces an existing trigger with the trigger_name.
- ❖ **{BEFORE | AFTER | INSTEAD OF} :** This specifies when the trigger would be executed. The INSTEAD OF clause is used for creating trigger on a view.
- ❖ **{INSERT [OR] | UPDATE [OR] | DELETE}:** This specifies the DML operation.
- ❖ **[OF col_name]:** This specifies the column name that would be updated.
- ❖ **[ON table_name]:** This specifies the name of the table associated with the trigger.
- ❖ **[REFERENCING OLD AS o NEW AS n]:** This allows you to refer new and old values for various DML statements, like INSERT, UPDATE, and DELETE.
- ❖ **[FOR EACH ROW]:** This specifies a row level trigger, i.e., the trigger would be executed for each row being affected. Otherwise the trigger will execute just once when the SQL statement is executed, which is called a table level trigger.
- ❖ **WHEN (condition):** This provides a condition for rows for which the trigger would fire. This clause is valid only for row level triggers

PL/SQL Trigger

- ❖ This trigger execute BEFORE to convert ename field lowercase to uppercase.

```
CREATE or REPLACE TRIGGER trg1
```

```
BEFORE
```

```
INSERT ON emp1
```

```
FOR EACH ROW
```

```
BEGIN
```

```
    :new.ename := upper(:new.ename);
```

```
END;
```


PL/SQL Trigger

This trigger is preventing to deleting row having eno as 1.

```
CREATE or REPLACE TRIGGER trg1
```

```
  BEFORE
```

```
  DELETE ON emp1
```

```
  FOR EACH ROW
```

```
BEGIN
```

```
  IF :old.eno = 1 THEN
```

```
    raise_application_error(20015, 'You can't delete this row');
```

```
  END IF;
```

```
END;
```

```
SQL>delete from emp1 where eno = 1;  
Error Code: 20015  
Error Name: You can't delete this row
```

PL/SQL Trigger

```
CREATE OR REPLACE TRIGGER display_salary_changes
```

```
BEFORE UPDATE ON customers
```

```
FOR EACH ROW
```

```
WHEN (NEW.ID > 0)
```

```
DECLARE
```

```
    sal_diff number;
```

```
BEGIN
```

```
    sal_diff := :NEW.salary - :OLD.salary;
```

```
    dbms_output.put_line('Old salary: ' || :OLD.salary);
```

```
    dbms_output.put_line('New salary: ' || :NEW.salary);
```

```
    dbms_output.put_line('Salary difference: ' || sal_diff);
```

PL/SQL Trigger

```
SQL>UPDATE customers SET salary = salary + 500  
WHERE id = 2;
```

- ❖ When a record is updated in the CUSTOMERS table, the trigger, display_salary_changes will be fired and it will display the following result –

Old salary: 1500

New salary: 2000

Salary difference: 500

Exception

Syntax for Exception Handling

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception₁ THEN

 exception₁-handling-statements

WHEN exception₂ THEN

 exception₂-handling-statements

WHEN exception₃ THEN

Exception

```
DECLARE
```

```
  c_id customers.id%type := 8;
```

```
  c_name customers.Name%type;
```

```
  c_addr customers.address%type;
```

```
BEGIN
```

```
  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);
```

```
EXCEPTION
```

```
  WHEN no_data_found THEN
```

Pre-defined Exception

Exception	Description
ACCESS_INTO_NULL	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
INVALID_CURSOR	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
LOGIN_DENIED	It is raised when a program attempts to log on to the database with an invalid username or password.
ROWTYPE_MISMATCH	It is raised when a cursor fetches value in a variable having incompatible data type.
NOT_LOGGED_ON	It is raised when a database call is issued without being connected to the database.

User Defined Exception

DECLARE

exp_name EXCEPTION;

BEGIN

If condition then

RAISE exp_name;

End IF;

EXCEPTION

When exp_name then

Statements;

END.

DECLARE

c_id customers.id%type := &cc_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;

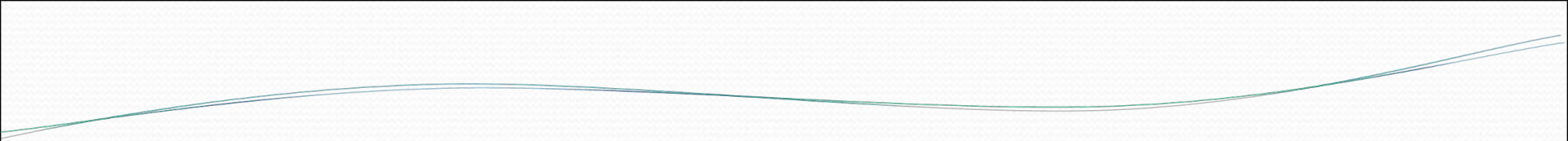
ELSE

 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
of
PL/SQL