UNIT 4- PL/SQL Programming

Marks-16

What Is PL/SQL?

PL/SQL:

- **Stands for Procedural Language extension to SQL**
- **Is Oracle Corporation's standard data access language for relational databases**
- **Seamlessly integrates procedural constructs with 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.

- PL/SQL is one of three key programming languages embedded in the Oracle Database,along with SQL itself and Java.

Advantages of PL/SQL

- **Block Structure:-**PL/SQL consist of block of code which can be nested within each other . Each block forms a unit of task or a logical module.Block can be stored in the database and reused.
- **Better Performance:**-PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- **Procedural Language Capability:**-PL/SQL consist of procedural language constructs such as conditional statements (if else statements) and loop like (for loops)
- **Error handling:**-PL/SQL handles errors or exception effectively during the execution of pl/Sql program.

PL/SQL Block Structure

- **DECLARE (optional)**
  - Variables, cursors, user-defined exceptions
- **BEGIN (mandatory)**
  - SQL statements
  - PL/SQL statements
- **EXCEPTION (optional)**
  - Actions to perform when errors occur
- **END; (mandatory)**

PL/SQL Character Data Types

Sr.No	Data Type & Description
1	**CHAR**Fixed-length character string with maximum 255 character It contain letter ,Number and Special Character.
2	**VARCHAR/Varchar 2** Variable-length character string with maximum size of 4000 character
3	**Date** Data type is used to represent data and time.
4	**Long** store variable length 2 gb
5	**LONG RAW /RAW** store binary data such as picture and image size 255 bytes to 2 gb

Variable:
A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and the layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.

The syntax for declaring a variable is
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

Example:

Radius Number := 5;

Date_of_birth date;

sales number(10, 2);

pi CONSTANT double precision := 3.1415;

name varchar2(25);

Variable Attributes:

The syntax for declaring a variable with %TYPE is:

<var_name> <tab_name>.<column_name>%TYPE;

Consider a declaration.

SALARY EMP.SAL % TYPE;

This declaration will declare a variable SALARY that has the same data type as column SAL of the EMP table.

Example:

```
DECLARE
SALARY EMP.SAL % TYPE;
ECODE EMP.empno % TYPE;
BEGIN
Ecode :=&Ecode;
Select SAL into SALARY from EMP where EMPNO = ECODE;
dbms_output.put_line('Salary of ' || ECODE || 'is = || salary');
END;
```

```
1.  DECLARE
2.      a integer := 30;
3.      b integer := 40;
4.      c integer;
5.     BEGIN
6.      c := a + b;
7.      dbms_output.put_line('Value of c: ' || c);
8.      END;
```

```
1.  DECLARE
2.     a integer := 30;
3.     b integer := 40;
4.     c integer;
5.     f real;
6.  BEGIN
7.     c := a + b;
8.     dbms_output.put_line('Value of c: ' || c);
9.     f := 100.0/3.0;
10.    dbms_output.put_line('Value of f: ' || f);
11. END;
```

DECLARE

Pi constant number:=3.141592654;

radius number(5,2) ; diameter number(5,2);

circumference number(7,2); Area number(10,2);

Take radius of circle =9.5

```
DECLARE

Pi constant number:=3.141592654;

radius number(5,2) ; diameter number(5,2);

circumference number(7,2);  Area number(10,2);

BEGIN

radius:= 9.5;  diameter:=radius * 9.5;

circumference:= 2.0* pi * radius ; area:=pi * radius * radius;

dbms_output.put_line('Value of  radius: ' || radius);  dbms_output.put_line('Value of
diameter: ' || diameter);  dbms_output.put_line('Value of circumference: ' || circumference);

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

END;
```

Radius: 9.5

Diameter: 19

Circumference: 59.69

Area: 283.53

Control Structures

Decision-making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Conditional Control

1) If- THEN Statement
2) IF-THEN-ELSE Statement
3) IF-THEN-ELSE-IF Statement

Conditional Control

1) If- THEN Statement

The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.

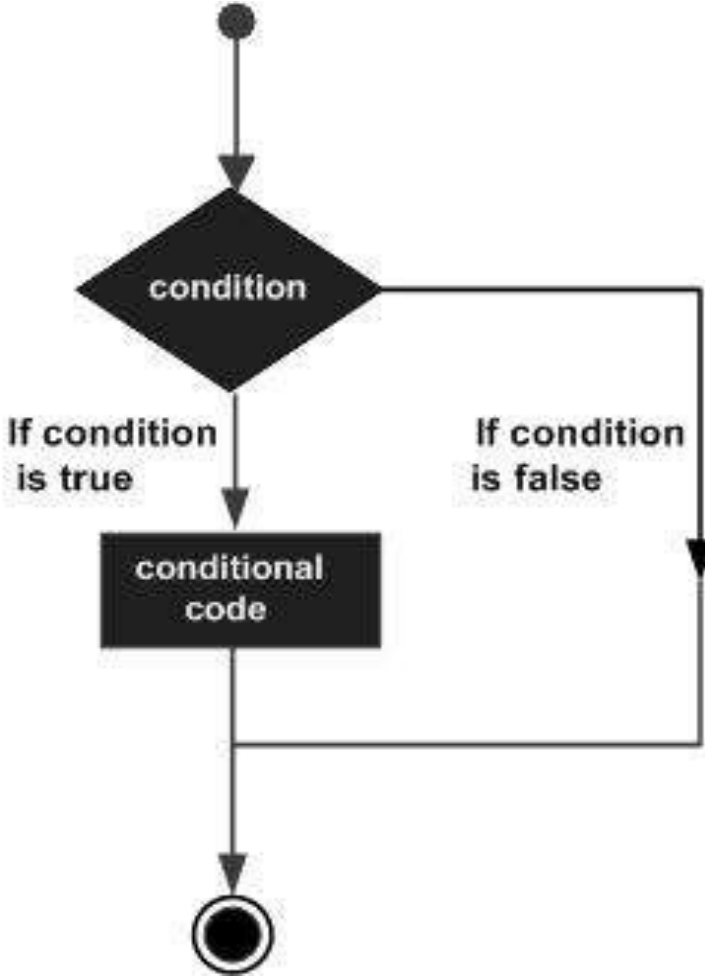
Syntax

IF condition

THEN

Statement: {It is executed when condition is true}

END IF;



Conditional Control

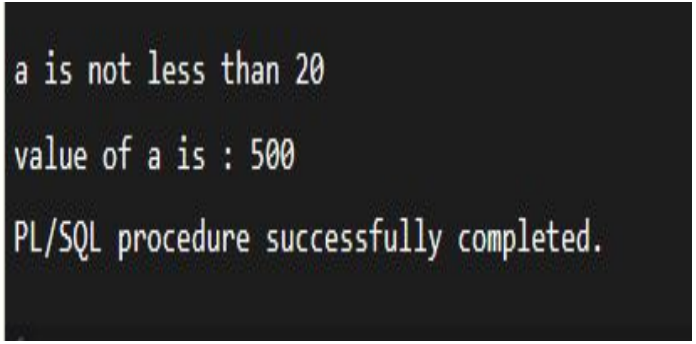
2) If- THEN-ELSE Statement

IF statement adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL, then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.

Syntax

```
IF condition

THEN

  {...statements to execute when condition is TRUE...}

ELSE

  {...statements to execute when condition is FALSE...}

END IF;
```

```
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;
```



Conditional Control

3) If- THEN-ELSEIF Statement

It allows you to choose between several alternatives.

Syntax

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

Program to find largest of two number.

```
DECLARE
a NUMBER; b NUMBER;
BEGIN
dbms_output.Put_line('Enter a:’);dbms_output.Put_line('Enter a:’);
a:=&a; b:=&b;
if(a>b) then
dbms_output.Put_line('max number is '||a);
else
dbms_output.Put_line('max number is '||b);
end if;
END;
```

```
1.  DECLARE
2.      N NUMBER;
3.      M NUMBER;
4.  BEGIN
5.      DBMS_OUTPUT.PUT_LINE('ENTER A NUMBER');
6.      N:=&NUMBER;
7.      DBMS_OUTPUT.PUT_LINE('ENTER A NUMBER');
8.      M:=&NUMBER;
9.  IF N<M THEN
10.     DBMS_OUTPUT.PUT_LINE(''|| N ||' IS GREATER THAN '|| M ||'');
11. ELSE
12.     DBMS_OUTPUT.PUT_LINE(''|| M ||' IS GREATER THAN '|| N ||'');
13. END IF;
14. END;
15. /
```

Program to display whether the salary of johnson is 50000 or not.

```
DECLARE
Jsal emp.salary%type;
BEGIN
Select Salary INTO Jsal
From emp where lastname=’johnson’;
IF(Jsal>50000)then
dbms_output.Put_line('salary of johnson is 50000’);
Else
dbms_output.Put_line('salary of johnson is not 50000’);
end if;
END; /
```

Iterative Control:---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.

There are 3 types of loops in PL/SQL ie

1) Loop
2) For loop
3) While loop.

1) Loop statements:- let you execute a sequence of statements multiple times.

You place the keyword loop before the first statement in the sequence and the keywords END LOOP after the last statement in the Sequence.

Syntax

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

The Exit-when statement let you complete loop if further processing is impossible or undesirable.

When the exit statement is encountered , the condition in the when clause is evaluated. If the condition is true, the loop complete and control passes to the next statement.

Program to print the number 1 to 10

```
DECLARE
i NUMBER := 1;
BEGIN
LOOP
i := i+1;
DBMS_OUTPUT.PUT_LINE(i);
IF (i>=10) then
EXIT;
END If;
End LOOP;
END; /
```

Program to print the number 1 to 5 using exit when condition

```
DECLARE

i NUMBER := 0;

BEGIN

LOOP

i := i+1;

DBMS_OUTPUT.PUT_LINE(i);

Exit when (i>=5);

End LOOP;

END; /
```

While Loop

PL/SQL while loop is used when a set of statements has to be executed as long as a condition is true, the While loop is used. The condition is decided at the beginning of each iteration and continues until the condition becomes false.

Syntax of while loop:

WHILE <condition>

LOOP statements;

END LOOP;

Program to print the number 1 to 10 using while loop

```
DECLARE
i INTEGER := 1;
BEGIN
WHILE i <= 10 LOOP
DBMS_OUTPUT.PUT_LINE(i);
i := i+1;
END LOOP;
END;
```

Program to print the number using multiplication operator

```
DECLARE
VAR1 NUMBER;
VAR2 NUMBER;
BEGIN
VAR1:=200;
VAR2:=1;
WHILE (VAR2<=10)
LOOP
DBMS_OUTPUT.PUT_LINE (VAR1*VAR2);
VAR2:=VAR2+1;
END LOOP;
END;
```

FOR LOOP

PL/SQL for loop is used when when you want to execute a set of statements for a predetermined number of times. The loop is iterated between the start and end integer values. The counter is always incremented by 1 and once the counter reaches the value of end integer, the loop ends.

Syntax of for loop:

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

initial_value : Start integer value

final_value : End integer value

Program to print number 1 to10

```
DECLARE
VAR1 NUMBER;
BEGIN
VAR1:=10;
FOR VAR2 IN 1..10
LOOP
DBMS_OUTPUT.PUT_LINE (VAR2);
END LOOP;
END;
```

Write PL/Sql Program to calculate factorial of a given number

```
declare
/*it gives the final answer after computation*/
fact number :=1;
n number := &1;
begin
while n > 0 loop
fact:=n* fact;
n:=n-1;
end loop;
dbms_output.put_line(fac);
end;
```

Write PL/Sql Program to print even or odd number from given range (Accept no range from user)

Code for Even Number

```
declare
A number:=&A; B number:=&B;
begin
For I in A..B loop
IF(Mod(I,2)=0)then
dbms_output.put_line(I);
END IF;
End loop;
end;
```

Code for Even Number

```
declare
A number:=&A; B number:=&B;
begin
For I in A..B loop
IF(Mod(I,2)=1)then
dbms_output.put_line(I);
END IF;
End loop;
end;
```

Write PL/Sql Program using while loop to display n even numbers.

```
declare
OP number:=2; I number:=&I;
begin
while(I>0)then
IF(Mod(OP,2)=0)then
dbms_output.put_line(OP);
END IF;
I:=I-1;
OP:=OP+2;
End loop;
end;
```

Output:

Enter Value for i:10

2

4

6

8

10

12

14..

18

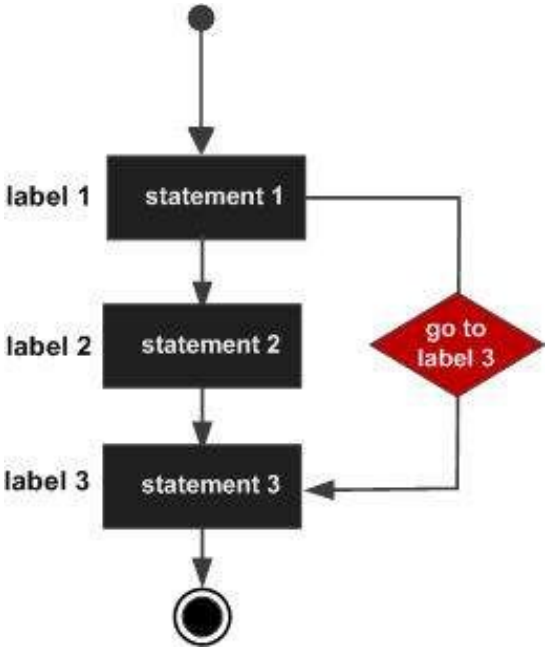
20

Sequential Control

Goto Statement

A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.

Here the label declaration which contains the label_name encapsulated within the << >> symbol and must be followed by at least one statement to execute.



Syntax:

GOTO label_name;

..

<<label_name>>

Statement;

Example of PL/SQL GOTO statement

```
1.  DECLARE
2.    a number(2) := 30;
3.  BEGIN
4.    <<loopstart>>
5.    -- while loop execution
6.    WHILE a < 50 LOOP
7.      dbms_output.put_line ('value of a: ' || a);
8.      a := a + 1;
9.      IF a = 35 THEN  `
10.       a := a + 1;
11.       GOTO loopstart;
12.     END IF;
13.   END LOOP;
14. END;
15. /
```

Sequential Control

Null Statement

The Null statement does nothing and passes the control to the next statement. Some language refer to such an instruction as a no-op(no operation).

```
DECLARE
v_emp_id NUMBER(6) := 110; v_job varchar2(20) ;
BEGIN
Select job_id into v_job from employees where employee_id=v_emp_id;
IF  v_job_id=’ Sales_Reprentative’ then
Update employees SET comission_pct=Comission_pct * 1.2;
Else
Null;       - - do nothing if not a sales representative
End if;
End;
/
```

Exception Handling

An error occurs during the program execution is called Exception in PL/SQL.

PL/SQL facilitates programmers to catch such conditions using exception block in the program and an appropriate action is taken against the error condition.

```
1.  System-defined Exceptions
2.  DECLARE  <declarations section>
3.  BEGIN  <executable command(s)>
4.  EXCEPTION
5.    <exception handling goes here >
6.    WHEN exception1 THEN    exception1-handling-statements
7.    WHEN exception2  THEN    exception2-handling-statements
8.       ........
9.    WHEN others THEN    exception3-handling-statements
10.  END;
```

Cursors

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 is used to referred to 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

Cursors

- Implicit Cursors

The implicit cursors are automatically generated by Oracle while an SQL statement is executed.These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.

Orcale provides some attributes known as Implicit cursor's attributes to check the status of DML operations. Some of them are: %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

1) %ROWCOUNT:- The number of rows processed by a sql statement.
2) % FOUND: True if at least one row was processed.
3) %NOTFOUND: True if no rows were processed.
4) %ISOPEN: True if cursor is open or false if cursor has not been opened or has been closed. Only used with explicit cursors.

PL/SQL Implicit Cursor Example

Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE  customers
   SET salary = salary + 5000;
   IF sql%notfound THEN
      dbms_output.put_line('no customers updated');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers updated ');
   END IF;
END;  /
```

```
6 customers updated
```

Cursors

- Explicit Cursors

An explicit cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. A suitable name for the cursor.

General syntax for creating a cursor:

CURSOR cursor_name IS select_statement;

cursor_name – A suitable name for the cursor.

select_statement – A select query which returns multiple rows.

How to use Explicit Cursor?

There are four steps in using an Explicit Cursor.

1. DECLARE the cursor in the Declaration section.
2. OPEN the cursor in the Execution Section.
3. FETCH the data from the cursor into PL/SQL variables or records in the Execution Section.
4. CLOSE the cursor in the Execution Section before you end the PL/SQL Block.

```
DECLARE variables;
 records;
 create a cursor;
 BEGIN
OPEN cursor;
FETCH cursor;
 process the records;
 CLOSE cursor;
END;
```

PL/SQL Implicit Cursor Example

Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Allahabad	20000
2	Suresh	22	Kanpur	22000
3	Mahesh	24	Ghaziabad	24000
4	Chandan	25	Noida	26000
5	Alex	21	Paris	28000
6	Sunita	20	Delhi	30000

Execute the following program to retrieve the customer id ,customer name and address.

```
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;
/
```

```
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;
/
```

Procedure

> A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the calling program.

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.

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.

Creating a Procedure

A procedure is created with the CREATE OR REPLACE PROCEDURE statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows −

```
CREATE [OR REPLACE] PROCEDURE procedure_name

[(parameter_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}

BEGIN

  < procedure_body >

END procedure_name;
```

Creating a Procedure

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Passing parameters in procedure:

When you want to create a procedure or function, you have to define parameters .There is three ways to pass parameters in procedure:

1. **IN parameters:** These types of parameter are used to send values to stored procedure.
2. **OUT parameters:** These types of parameter are used to get values from stored procedure.This is similar to a return type in functions.
3. **INOUT parameters:** These types of parameter are used to send values and get values from stored procedure.By default it is an IN Parameter.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

   dbms_output.put_line('Hello World!');

END;

/
```

Output:-

The above procedure named 'greetings' can be called with the EXECUTE keyword as −

```
   EXECUTE greetings;
```

IN & OUT Mode Example 1

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

```
DECLARE
   a number;   b number;  c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
   IF x < yTHEN
      z:= x;
   ELSE
      z:= y;
   END IF; END;
BEGIN
   a:= 23;    b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;/
```

```
Minimum of (23, 45) : 23

PL/SQL procedure successfully
completed.
```

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

```
DECLARE
   a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
  x := x * x;
END;
BEGIN
   a:= 23;
   squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

```
Square of (23): 529

PL/SQL procedure
successfully completed.
```

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement.

Syntax for deleting a procedure is −

```
DROP PROCEDURE procedure-name;
```

You can drop the greetings procedure by using the following statement −

```
DROP PROCEDURE greetings;
```

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.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows −

```
CREATE [OR REPLACE] FUNCTION function_name

[(parameter_name [IN | OUT | IN OUT] type [, ...])]

RETURN return_datatype

{IS | AS}

BEGIN

   < function_body >

END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
DECLARE
   a number;  b number;   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;
   RETURN z;
END;
BEGIN
   a:= 23;  b:= 45;  c := findMax(a, b);
   dbms_output.put_line(' Maximum of
(23,45): ' || c);
END;/
```

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

```
Maximum of (23,45): 45
PL/SQL procedure successfully completed.
```

```
DECLARE
   num number;
   factorial number;
FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
   IF x=0 THEN
      f := 1;
   ELSE
      f := x * fact(x-1);
   END IF;
RETURN f;
END;
BEGIN
   num:= 6;
   factorial := fact(num);
   dbms_output.put_line(' Factorial '|| num || ' is ' ||
factorial);
END;
/
```

Deleting a Function

Function is deleted with the **DROP Function** statement.

Syntax for deleting a procedure is −

DROP Function Function -name;

You can drop the Function by using the following statement −

DROP Function findmax;

Database Triggers

Triggers in PL/SQL. Triggers are stored programs, which are automatically executed or fired when some events occur. .Trigger is stored into database and invoked repeatedly, when specific condition match.

Triggers are, in fact, written to be executed in response to any of the following events −

- A **database manipulation (DML)** statement (DELETE, INSERT, or UPDATE)
- A **database definition (DDL)** statement (CREATE, ALTER, or DROP).
- A **database operation** (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Database Triggers

Benefits/Advantages of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Database Triggers Types

Row Level Triggers	Statement Level Triggers
Row level triggers executes once for each and every row in the transaction.	Statement level triggers executes only once for each single transaction.
Specifically used for data auditing purpose.	Used for enforcing all additional security on the transactions performed on the table.
"FOR EACH ROW" clause is present in CREATE TRIGGER command.	"FOR EACH ROW" clause is omitted in CREATE TRIGGER command.
Example: If 1500 rows are to be inserted into a table, the row level trigger would execute 1500 times.	Example: If 1500 rows are to be inserted into a table, the statement level trigger would execute only once.

Creating Triggers

```
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]  WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
 END;
```

Where,

- CREATE [OR REPLACE] TRIGGER trigger_name − Creates or replaces an existing trigger with the *trigger_name*.
- {BEFORE | AFTER | INSTEAD OF} − This specifies when the trigger will 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 will 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, such as INSERT, UPDATE, and DELETE.
- [FOR EACH ROW] − This specifies a row-level trigger, i.e., the trigger will 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

Creating Triggers Example

```
CREATE OR REPLACE TRIGGER trg1
BEFORE  INSERT ON EMP
For each row
Begin
If :new.sal<=0 then                          // WHEN (condition)
raise_application_error('salary should be greater than 0')
End if
End;
/
```

Trigger Created

```
Insert into emp(Emp_no, Emp_name,Salary)values(1,'Shyam',0);
```

Enabling/Disabling Trigger:

To enable or disable a specific trigger Alter Trigger Command is used.

Syntax:-

Alter Trigger trigger_name Enable/Disable;

When a trigger is created , It is automatically enable.

Deleting a Trigger

To delete a trigger we use drop trigger command.

Syntax for deleting a Trigger is −

DROP Trigger trigger_name;

Example

Drop trigger trg1;

creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR 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);
END;
/
```

creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

Old salary:
New salary:
Salary difference:

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