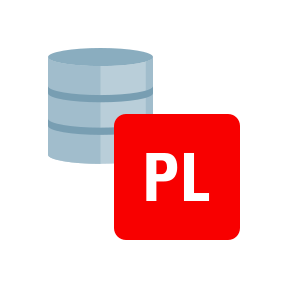
** - BY Ramesh Gangasagare**

* ***PL/SQL is a Procedural Language developed by Oracle and is an extension of Oracle SQL having the functionalities of functions, control structures, and triggers.***

PL/SQL, which stands for Procedural Language/Structured Query Language, is Oracle Corporation's procedural language extension for SQL (Structured Query Language). It combines the data manipulation power of SQL with the processing power of procedural languages.A diagram of a computer process

Description automatically generated

* Oracle Database: The actual relational database management system.
* PL/SQL: The programming language used to interact with and extend the capabilities of Oracle Database.

Developers use PL/SQL to create custom business logic, automate tasks, and implement various database-related functionalities within the Oracle Database environment.

**Here are some key points about PL/SQL:**

1. \*\*Procedural Language: PL/SQL is a procedural language, meaning it allows you to write procedural code (blocks of code that can contain declarations, assignments, loops, conditional statements, etc.).

2. \*\*Extension for SQL: PL/SQL is designed to work seamlessly with SQL, allowing you to embed SQL statements within its procedural code. This integration enables the manipulation of data in the database using SQL and the creation of more sophisticated programs with procedural constructs.

3. \*\*Block Structure: PL/SQL programs are composed of anonymous blocks, procedures, functions, packages, triggers, and types. An anonymous block is a set of PL/SQL statements that can be executed as a unit.

4. \*\*Data Types: PL/SQL supports a variety of data types, including scalar types (like numbers and characters), composite types (like records and arrays), and reference types.

5. \*\*Exception Handling: PL/SQL provides robust error handling mechanisms. You can define and handle exceptions, allowing your code to gracefully handle errors and continue execution or roll back changes.

6. \*\*Modularity: PL/SQL supports modularity through the use of procedures and functions. Procedures are blocks of code that perform a specific task, while functions return a value.

7. \*\*Triggers: PL/SQL can be used to define triggers, which are special kinds of stored procedures that are automatically executed in response to certain events on a particular table or view in the database.

PL/SQL is commonly used in Oracle Database environments for developing stored procedures, functions, triggers, and other database-centric applications. It provides a powerful and flexible environment for building and maintaining database applications.

* **Anonymous Block:**

*In PL/SQL (Procedural Language/Structured Query Language), an anonymous block is a block of code that is not stored in the database as a named PL/SQL object. It is a way to execute PL/SQL code directly without the need for creating a stored procedure, function, or trigger.*

**What is an Anonymous Block?**

-Its is a block without name.

* Basic unit of PLSQL
* It allows us to **combine** together a bunch of **SQL statements**, and run them as a single **block** of code.
* Think of the anonymous block is like the piece of paper with custom instructions. Unlike the prebuilt modes, which we can call by name, and anyone can understand, and get to them, these are known only to me, or to whomsoever I pass on to.
* **Structure of Anonymous Block:**

***A close-up of a document

Description automatically generated***

DECLARE

l\_counter NUMBER;

BEGIN

-- Assigning value 1 to l\_counter

l\_counter := 1;

-- Printing the value of l\_counter

dbms\_output.put\_line('l\_counter in the inner block is ' || l\_counter);

EXCEPTION

-- Handling any exceptions (in this case, doing nothing)

WHEN OTHERS THEN

NULL;

END;

A close-up of a list of text

Description automatically generated

**Nested Blocks**: **A block can contain other blocks** and these blocks are termed nested blocks. A nested block may have an optional declare and exception section.A screenshot of a computer

Description automatically generated

**Scope of Variables:** PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block.A diagram of a program

Description automatically generated with medium confidence

**Simple Block Code in PLSQL:**

DECLARE

NAME VARCHAR2 (20);

POSITION CHAR(50);

AGE NUMBER(3);

BEGIN

NAME:='Ramesh Gangasagare';

POSITION:='Sr.Software Engineer';

AGE:=25;

DBMS\_OUTPUT.PUT\_LINE('MY NAME IS: '||NAME);

DBMS\_OUTPUT.PUT\_LINE('POSITION: '||Position);

DBMS\_OUTPUT.PUT\_LINE('AGE IS: '||AGE);

END;

Statement processed.  
MY NAME IS: Ramesh Gangasagare  
POSITION: Sr.Software Engineer   
AGE IS: 25

* **PLSQL DataTypes:**

In this chapter, we will discuss the Data Types in PL/SQL. The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values. We will focus on the **SCALAR** and the **LOB** data types in this chapter. The other two data types will be covered in other chapters.

| **S.No** | **Category & Description** |
| --- | --- |
| 1 | **Scalar**  Single values with no internal components, such as a **NUMBER, DATE,** or **BOOLEAN**. |
| 2 | **Large Object (LOB)**  Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms. |
| 3 | **Composite**  Data items that have internal components that can be accessed individually. For example, collections and records. |
| 4 | **Reference**  Pointers to other data items. |

## PL/SQL Scalar Data Types and Subtypes

PL/SQL Scalar Data Types and Subtypes come under the following categories −

| **S.No** | **Date Type & Description** |
| --- | --- |
| 1 | **Numeric**  Numeric values on which arithmetic operations are performed. |
| 2 | **Character**  Alphanumeric values that represent single characters or strings of characters. |
| 3 | **Boolean**  Logical values on which logical operations are performed. |
| 4 | **Datetime**  Dates and times. |

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use the subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding the PL/SQL code in another program, such as a Java program.

## PL/SQL Numeric Data Types and Subtypes

Following table lists out the PL/SQL pre-defined numeric data types and their sub-types −

| **S.No** | **Data Type & Description** |
| --- | --- |
| 1 | **PLS\_INTEGER**  Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits |
| 2 | **BINARY\_INTEGER**  Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits |
| 3 | **BINARY\_FLOAT**  Single-precision IEEE 754-format floating-point number |
| 4 | **BINARY\_DOUBLE**  Double-precision IEEE 754-format floating-point number |
| 5 | **NUMBER(prec, scale)**  Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0 |
| 6 | **DEC(prec, scale)**  ANSI specific fixed-point type with maximum precision of 38 decimal digits |
| 7 | **DECIMAL(prec, scale)**  IBM specific fixed-point type with maximum precision of 38 decimal digits |
| 8 | **NUMERIC(pre, secale)**  Floating type with maximum precision of 38 decimal digits |
| 9 | **DOUBLE PRECISION**  ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits) |
| 10 | **FLOAT**  ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits) |
| 11 | **INT**  ANSI specific integer type with maximum precision of 38 decimal digits |
| 12 | **INTEGER**  ANSI and IBM specific integer type with maximum precision of 38 decimal digits |
| 13 | **SMALLINT**  ANSI and IBM specific integer type with maximum precision of 38 decimal digits |
| 14 | **REAL**  Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits) |

Following is a valid declaration −

DECLARE   
   num1 INTEGER;   
   num2 REAL;   
   num3 DOUBLE PRECISION; BEGIN   
   null; END; /

When the above code is compiled and executed, it produces the following result −

PL/SQL procedure successfully completed

* **LOOPS:**

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages.

| **S.No** | **Loop Type & Description** |
| --- | --- |
| 1 | [PL/SQL Basic LOOP](https://www.tutorialspoint.com/plsql/plsql_basic_loop.htm)  In this loop structure, sequence of statements is enclosed between the LOOP and the END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop. |
| 2 | [PL/SQL WHILE LOOP](https://www.tutorialspoint.com/plsql/plsql_while_loop.htm)  Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body. |
| 3 | [PL/SQL FOR LOOP](https://www.tutorialspoint.com/plsql/plsql_for_loop.htm)  Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
| 4 | [Nested loops in PL/SQL](https://www.tutorialspoint.com/plsql/plsql_nested_loops.htm)  You can use one or more loop inside any another basic loop, while, or for loop. |

[**Increment by 2 in oracle for loop**](https://www.orafaq.com/forum/mv/msg/49717/130701/#msg_130701)**:**

DECLARE

curr\_val PLS\_INTEGER;

lower\_bound PLS\_INTEGER := 0;

upper\_bound PLS\_INTEGER := 10;

increment PLS\_INTEGER := 2;

BEGIN

curr\_val := lower\_bound;

LOOP

IF curr\_val <= upper\_bound

THEN

dbms\_output.put\_line('curr\_val : ' || curr\_val);

ELSE

EXIT;

END IF;

curr\_val := curr\_val + increment;

END LOOP;

END;

Statement processed.  
curr\_val : 0  
curr\_val : 2  
curr\_val : 4  
curr\_val : 6  
curr\_val : 8  
curr\_val : 10

* **Simple Loop:**

**Example 1:**

DECLARE

x NUMBER := 10;

BEGIN

LOOP

DBMS\_OUTPUT.PUT\_LINE('x = ' || x);

x := x + 10;

IF x > 50 THEN

EXIT;

END IF;

END LOOP;

END;

**Example 2:**

DECLARE

i int;

BEGIN

i := 1;

LOOP

if i>10 then

exit;

end if;

dbms\_output.put\_line(i);

i := i+1;

END LOOP;

END;

* **For Loop**

DECLARE

A number (2);

BEGIN

FOR i IN 1..10 LOOP

dbms\_output.put\_line('value of i: '||i);

END LOOP;

END;

* **While Loop**

**Example 1:**

DECLARE

a number(2):= 10;

BEGIN

WHILE a < 20 LOOP dbms\_output.put\_line('value of a: ' || a); a :=a+1; END LOOP;

END;

**Example 2:**

 DECLARE

num INT := 1;

BEGIN

WHILE (num <= 10) LOOP

DBMS\_OUTPUT.PUT\_LINE('' || num);

num := num + 2;

END LOOP;

END;

* **CONDITIONAL EXECUTION:**

In PL/SQL, conditional execution is typically achieved using the IF-THEN-ELSE statement. This allows you to conditionally execute a block of code based on a specified condition.

* Executes a set of statements only when given condition is true.
* Can pass multiple conditions.
* Condition can be formed by using comparison, logical operator which returns a Boolean.
* The IF statement checks whether your variable is greater than 0.
* If the condition is true, the code inside the THEN block is executed.
* If the condition is false, it checks the next condition in the ELSIF block.
* If none of the conditions is true, the code inside the ELSE block is executed.

**IF**:

DECLARE

x int:=10;

y int:=80;

BEGIN

if(y>x) then

dbms\_output.put\_line('Result: ' ||y|| ' is greater than ' ||x);

end if;

END;

Statement processed.  
Result: 80 is greater than 10

* **Syntax for the IF-THEN-ELSE statement is -**

IF condition THEN

S1;

ELSE

S2;

END IF;

IF color = red THEN dbms\_output.put\_line('You have chosen a red car')

ELSE dbms\_output.put\_line('Please choose a color for your car');

END IF;

* **The syntax of an IF-THEN-ELSIF Statement in PL/SQL**

DECLARE

a number(3) := 100;

BEGIN

IF (a = 10) THEN

dbms\_output.put\_line('Value of a is 10');

ELSIF (a = 20) THEN

dbms\_output.put\_line('Value of a is 20');

ELSIF (a = 30 ) THEN

dbms\_output.put\_line('Value of a is 30');

ELSE dbms\_output.put\_line('None of the values is matching');

END IF;

dbms\_output.put\_line('Exact value of a is: '|| a);

END;

* **IF-ELSE:**

DECLARE

x INT := 10; -- Declare and initialize a variable x with the value 10

y INT := 5; -- Declare and initialize a variable y with the value 5

BEGIN

-- Check if y is greater than x

IF y > x THEN

-- Print a message if the condition is true

DBMS\_OUTPUT.PUT\_LINE('Result: ' || y || ' is greater than ' || x);

ELSE

-- Print a different message if the condition is false

DBMS\_OUTPUT.PUT\_LINE('Result: ' || y || ' is not greater than ' || x);

END IF; -- End of the IF-ELSE statement

END; -- End of the PL/SQL block

Statement processed.  
Result: 5 is not greater than 10

* **Switch CASE**

DECLARE

grade char(1) := 'A';

BEGIN

CASE grade

when 'A' then dbms\_output.put\_line('Excellent');

when 'B' then dbms\_output.put\_line('Very good');

when 'C' then dbms\_output.put\_line('Well done');

when 'D' then dbms\_output.put\_line('You passed');

when 'F' then dbms\_output.put\_line('Better try again'); else dbms\_output.put\_line('No such grade'); END CASE;

END;

PL/SQL does not have a direct CASE or SWITCH statement as seen in some other programming languages. Instead, it uses the CASE statement within the SELECT or UPDATE statements to provide conditional logic.

Remember that PL/SQL is a procedural language, and conditional logic is often implemented using IF-ELSIF-ELSE constructs or CASE within SQL statements rather than standalone CASE or SWITCH statements.

If you're looking for conditional logic within procedural blocks, here's an example using IF-ELSIF-ELSE:

DECLARE

grade CHAR(1);

student\_name VARCHAR2(50);

BEGIN

SELECT student\_grade

INTO grade

FROM student

WHERE student\_id = 123;

IF grade = 'A' THEN

student\_name := 'Excellent';

ELSIF grade = 'B' THEN

student\_name := 'Good';

ELSIF grade = 'C' THEN

student\_name := 'Satisfactory';

ELSIF grade = 'D' THEN

student\_name := 'Needs Improvement';

ELSE

student\_name := 'Not Graded';

END IF;

DBMS\_OUTPUT.PUT\_LINE('Student Grade: ' || grade);

DBMS\_OUTPUT.PUT\_LINE('Performance: ' || student\_name);

END;

* **CURSORS:**

A cursor is the Private Memory area which is created by an Oracle server for manipulating the data.  
Two Types of CURSORS  
    1.  EXPLICIT: Multiple row SELECT STATEMENTS  
    2.  IMPLICIT:  
        All INSERT statements  
        All UPDATE statements  
        All DELETE statements  
        Single row SELECT…. INTO Statements

**DECLARE**

**OPEN**

**FETCH**

**CLOSE**

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 multiple rows returned by a SQL statement.** The set of rows the cursor holds is referred to as the activeset.

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.

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 SQL cursor has additional attributes, **%BULK\_ROWCOUNT** and **%BULK\_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table.

## **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;

/

DECLARE

c\_empno emp.empno%TYPE;

c\_ename emp.ename%TYPE;

CURSOR c1 IS

SELECT empno, ename FROM emp;

BEGIN

OPEN c1;

LOOP

BEGIN

FETCH c1 INTO c\_empno, c\_ename;

EXIT WHEN c1%NOTFOUND;

-- Process the fetched data

DBMS\_OUTPUT.PUT\_LINE(c\_empno || ' ' || c\_ename);

END; -- Added END statement for the inner block

END LOOP;

CLOSE c1;

END;

Statement processed.  
7839 KING  
7698 BLAKE  
7782 CLARK  
7566 JONES  
7788 SCOTT  
7902 FORD  
7369 SMITH

# **PL/SQL – Exceptions:**

An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions −

* System-defined exceptions
* User-defined exceptions

**Syntax for Exception Handling**

DECLARE

c\_id customers.id%type := 3;

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

dbms\_output.put\_line('No such customer!');

WHEN others THEN

dbms\_output.put\_line('Error!');

END;

* The above program displays the name and address of a customer whose ID is given. Since there is no customer with ID value 8 in our database, the program raises the run-time exception **NO\_DATA\_FOUND**, which is captured in the **EXCEPTION block**.

Simple example of Exception block:

## DECLARE     c\_empno   emp.empno%TYPE;     c\_ename emp.ename%TYPE;        CURSOR c1 IS        SELECT empno, ename FROM emp;     BEGIN     OPEN c1;     LOOP        BEGIN           FETCH c1 INTO c\_empno, c\_ename;           EXIT WHEN c1%NOTFOUND;              -- Process the fetched data           DBMS\_OUTPUT.PUT\_LINE(c\_empno || ' ' || c\_ename);  EXCEPTION       WHEN No\_Data\_Found Then DBMS\_OUTPUT.PUT\_LINE ('NO DATA FOUND');       END; -- Added END statement for the inner block

   END LOOP;   
   CLOSE c1;   
END;

**ZERO\_DIVIDE** = raises exception WHEN dividing with zero

DECLARE

a int:=10;

b int:=0;

answer int;

BEGIN

answer:=a/b;

dbms\_output.put\_line('the result after division is'||answer);

exception

WHEN zero\_divide THEN

dbms\_output.put\_line('dividing by zero please check the values again');

dbms\_output.put\_line('the value of a is '||a);

dbms\_output.put\_line('the value of b is '||b);

END;

Statement processed.  
dividing by zero please check the values again  
the value of a is 10  
the value of b is 0

**Unnamed system exceptions:** Oracle doesn’t provide name for some system exceptions called unnamed system exceptions. These exceptions *don’t* occur frequently. These exceptions have two parts *code and an associated message*.

**User defined exceptions:**  
This type of users can create their own exceptions according to the need and to raise these exceptions explicitly ***raise*** command is used.

*Example:*

* Divide non-negative integer x by y such that the result is greater than or equal to 1.

From the given question we can conclude that there exist two exceptions

* + Division be zero.
  + If result is greater than or equal to 1 means y is less than or equal to x.

|  |
| --- |
| **DECLARE**     x **int**:=&x; /\*taking value **at** run **time**\*/     y **int**:=&y;     div\_r **float**;     exp1 EXCEPTION;     exp2 EXCEPTION;    **BEGIN**     IF y=0 **then**         raise exp1;       ELSEIF y > x **then**        raise exp2;    **ELSE**        div\_r:= x / y;        dbms\_output.put\_line('the result is '||div\_r);    **END** IF;    EXCEPTION  **WHEN** exp1 **THEN**        dbms\_output.put\_line('Error');        dbms\_output.put\_line('division by zero not allowed');    **WHEN** exp2 **THEN**        dbms\_output.put\_line('Error');        dbms\_output.put\_line('y is greater than x please check the input');    **END**; |

*Input 1:* x = 20

y = 10

*Output:* the result is 2

*Input 2:* x = 20

y = 0

*Output:*

Error

division by zero not allowed

*Input 3:* x=20

y = 30

*Output:<.em>*

*Error*

*y is greater than x please check the input*

* **Functions**:

Functions are like procedures but have a key difference: a function returns a value, whereas a procedure does not.

CREATE [OR REPLACE] FUNCTION function\_name  
    
RETURN return\_datatype  
IS  
   -- Declaration section  
   variable declarations;  
BEGIN  
   -- Execution section  
   -- PL/SQL code  
   RETURN value; -- Return statement  
END function\_name;

CREATE OR REPLACE FUNCTION totalCustomers

RETURN number IS

total number(2) := 0;

BEGIN

SELECT count(\*) into total

FROM customers;

RETURN total;

END;

CREATE OR REPLACE FUNCTION getTotalSTUDENT  
RETURN INT IS  
   v\_count INTEGER := 0;  
BEGIN  
   SELECT COUNT(\*) INTO v\_count FROM STUDENT;  
   RETURN v\_count;  
END;  
/

BEGIN  
   DBMS\_OUTPUT.PUT\_LINE(getTotalSTUDENT());  
END;

### Procedures in PL/SQL:

Procedure is nothing but a named block. They can be reused. Procedure can perform an action but can’t return value.

Procedures in PL/SQL. A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs.

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

**STRUCTURE OF PROCEDURE:**

CREATE [OR REPLACE] PROCEDURE procedure\_name [(parameter\_name [IN | OUT | IN OUT] type [, ...])] {IS | AS} BEGIN   
  < procedure\_body > END procedure\_name;

**EXAMPLE:**

CREATE OR REPLACE PROCEDURE ABC

IS

VAR NUMBER;

BEGIN

VAR:=2;

DBMS\_OUTPUT.PUT\_LINE('VAR');

END;

Procedure created.

**Parts of a PL/SQL Subprogram**

Each PL/SQL subprogram has a name, and may also have a parameter list. Like anonymous PL/SQL blocks, the named blocks will also have the following three parts −

| **S.No** | **Parts & Description** |
| --- | --- |
| 1 | **Declarative Part**  It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution. |
| 2 | **Executable Part**  This is a mandatory part and contains statements that perform the designated action. |
| 3 | **Exception-handling**  This is again an optional part. It contains the code that handles run-time errors. |

**Parameters Modes:**

**IN:**

* Default mode.
* Read only parameter (cannot be assigned value).

**OUT:**

* It returns a value to the calling program.
* We can change its value.

**IN OUT:**

CREATE OR REPLACE PROCEDURE greetings

AS

--Declare variables

BEGIN

dbms\_output.put\_line('Hello World!'); --Action perf.

END;

### Simple Procedure Program

DECLARE

A NUMBER;

B NUMBER;

C NUMBER;

PROCEDURE FINDMUL(X IN NUMBER, Y IN NUMBER, Z OUT NUMBER)

IS

BEGIN

Z := X \* Y;

END FINDMUL;

BEGIN

A := 50;

B := 20;

FINDMUL(A, B, C);

DBMS\_OUTPUT.PUT\_LINE('MULTIPLICATION IS ' || C);

END;

Statement processed.  
MULTIPLICATION IS 1000

The main difference in Function & Procedures is function must always return a value on the other hand procedure may or may not return a value.

* **Executing Procedure**

EXECUTE ABC;

Inside a program.

* **DELETE a Procedure**

DROP PROCEDURE ABC;

**Transactions in ORACLE PL/SQL:**

* Transaction is a set of operations it used to perform a logical unit of work.
* Transaction generally represent change in database.

A transaction ends when one of the following events take place −

* A **COMMIT** or a **ROLLBACK** statement is issued.
* A **DDL** statement, such as **CREATE TABLE** statement, is issued; because in that case a COMMIT is automatically performed.
* A **DCL** statement, such as a **GRANT** statement, is issued; because in that case a COMMIT is automatically performed.
* User disconnects from the database.
* User exits from **SQL\*PLUS** by issuing the **EXIT** command, a COMMIT is automatically performed.
* SQL\*Plus terminates abnormally, a **ROLLBACK** is automatically performed.
* A **DML** statement fails; in that case a ROLLBACK is automatically performed for undoing that DML statement.

CREATE OR REPLACE PROCEDURE

process\_order (p\_act\_id accounts.act\_id%TYPE, p\_item\_id items.item\_id%TYPE, p\_item\_value items.item\_value%TYPE) IS

BEGIN

SET TRANSACTION NAME 'place\_order';

-- Debit Account

UPDATE accounts

--Place Order

INSERT INTO orders(order\_id,.....

COMMIT;

EXCEPTION

ROLLBACK;

RAISE;

END process\_order;

**Commit Statement:**

* Commit is used to save or stored changes permanently in database.
* Where we must do some DML operations.
* COMMIT statement to end your current transaction and make permanent all changes performed in the transaction.

Following are the practical structure of commit:

SELECT \* FROM DEPT

UPDATE DEPT SET DNAME='ENGINEERING' WHERE DEPTNO=30;

COMMIT;

* Statement processed.

**ROLLBACK Statement:**

Rollback command is used to undo the transaction, but immediately after commit we can not do rollback or undo because commit save changes permanently.

SELECT \* FROM DEPT

UPDATE DEPT SET LOC='PARIS' WHERE DEPTNO=10;

ROLLBACK;

Statement processed.

**Save point:**

Save point is a point where we can rollback transaction at certain point.

SELECT \* FROM DEPT

UPDATE DEPT SET LOC='PARIS' WHERE DEPTNO=10;

SAVEPOINT A;

UPDATE DEPT SET LOC='PARIS' WHERE DEPTNO=30;

SAVEPOINT B;

ROLLBACK TO A;

**Multiple Save point:**

BEGIN

INSERT INTO items....

SAVEPOINT first;

UPDATE accounts

SAVEPOINT second;

INSERT INTO orders...

SAVEPOINT third;

IF condition=TRUE THEN ROLLBACK; END IF;

**VIEW’s:**

In PL/SQL View is a virtual table it doesn’t store data itself; instead, its present data from one or more tables or others view.

Logical representation of table data.

Why we created views: Because we don’t want to give someone unauthorized access of data to anyone who is not accessed to authorize it.

Suppose you have a table called employees with columns employee\_id, first\_name, last\_name, and salary. You can create a view to display only specific columns from this table:

 -- Creating a view

CREATE OR REPLACE VIEW employee\_names AS

SELECT employee\_id, first\_name, last\_name

FROM employees;

Now, you can query this view as if it were a table:

-- Querying the view

**SELECT \* FROM employee\_names;**

This query will display the employee\_id, first\_name, and last\_name columns from the employees table as defined in the employee\_names view.

Summary:

Sure, let's go through each lesson briefly and provide an example for each:

1. \*\*Oracle PL/SQL Data Types — Boolean, Number, Date:\*\*  
   - Example:  
     ```sql  
     DECLARE  
         is\_valid BOOLEAN := TRUE;  
         quantity NUMBER := 10;  
         birth\_date DATE := TO\_DATE('1990-05-15', 'YYYY-MM-DD');  
     BEGIN  
         -- Your PL/SQL code here  
     END;  
     ```

2. \*\*PL/SQL Variables Naming Convention — Learn with Example:\*\*  
   - Example:  
     ```sql  
     DECLARE  
         v\_employee\_name VARCHAR2(100);  
         n\_employee\_age NUMBER;  
     BEGIN  
         -- Your PL/SQL code here  
     END;  
     ```

3. \*\*Oracle PL/SQL Collections — Varrays, Nested & Index by Tables:\*\*  
   - Example:  
     ```sql  
     DECLARE  
         TYPE name\_list IS VARRAY(3) OF VARCHAR2(50);  
         employees name\_list := name\_list('John', 'Alice', 'Bob');  
     BEGIN  
         -- Your PL/SQL code here  
     END;  
     ```

4. \*\*Oracle PL/SQL Records Type — Learn with Example:\*\*  
   - Example:  
     ```sql  
     DECLARE  
         TYPE employee\_record IS RECORD (  
             id NUMBER,  
             name VARCHAR2(100)  
         );  
         emp employee\_record;  
     BEGIN  
         -- Your PL/SQL code here  
     END;  
     ```

5. \*\*Oracle PL/SQL IF THEN ELSE Statement — ELSIF, NESTED-IF:\*\*  
   - Example:  
     ```sql  
     DECLARE  
         x INT := 10;  
     BEGIN  
         IF x > 10 THEN  
             DBMS\_OUTPUT.PUT\_LINE('x is greater than 10');  
         ELSIF x = 10 THEN  
             DBMS\_OUTPUT.PUT\_LINE('x is equal to 10');  
         ELSE  
             DBMS\_OUTPUT.PUT\_LINE('x is less than 10');  
         END IF;  
     END;  
     ```

6. \*\*Oracle PL/SQL CASE Statement — Learn with Example:\*\*  
   - Example:  
     ```sql  
     DECLARE  
         x CHAR := 'A';  
     BEGIN  
         CASE x  
             WHEN 'A' THEN DBMS\_OUTPUT.PUT\_LINE('Value is A');  
             WHEN 'B' THEN DBMS\_OUTPUT.PUT\_LINE('Value is B');  
             ELSE DBMS\_OUTPUT.PUT\_LINE('Value is neither A nor B');  
         END CASE;  
     END;  
     ```

7. \*\*Oracle PL/SQL LOOP — Learn with Example:\*\*  
   - Example:  
     ```sql  
     DECLARE  
         counter INT := 1;  
     BEGIN  
         LOOP  
             DBMS\_OUTPUT.PUT\_LINE('Iteration: ' || counter);  
             counter := counter + 1;  
             EXIT WHEN counter > 5;  
         END LOOP;  
     END;  
     ```

8. \*\*Oracle PL/SQL FOR LOOP — Learn with Example:\*\*  
   - Example:  
     ```sql  
     DECLARE  
     BEGIN  
         FOR i IN 1..5 LOOP  
             DBMS\_OUTPUT.PUT\_LINE('Iteration: ' || i);  
         END LOOP;  
     END;  
     ```

9. \*\*Oracle PL/SQL WHILE LOOP — Learn with Example:\*\*  
   - Example:  
     ```sql  
     DECLARE  
         counter INT := 1;  
     BEGIN  
         WHILE counter <= 5 LOOP  
             DBMS\_OUTPUT.PUT\_LINE('Iteration: ' || counter);  
             counter := counter + 1;  
         END LOOP;  
     END;  
     ```

10. \*\*Oracle PL/SQL Stored Procedure & Functions — Learn with Example:\*\*  
    - Example:  
      (Example already provided in the previous response)

11. \*\*Oracle PL/SQL Exception Handling — Examples to Raise User-defined Exception:\*\*  
    - Example:  
      (Example already provided in the previous response)

12. \*\*Oracle PL/SQL Insert, Update, Delete & Select Into — Learn with Example:\*\*  
    - Example:  
      (Example already provided in the previous response)

13. \*\*Oracle PL/SQL Cursor — Implicit, Explicit, Cursor FOR Loop:\*\*  
    - Example:  
      (Example already provided in the previous response)

14. \*\*Oracle PL/SQL BULK COLLECT — FORALL Example:\*\*  
    - Example:  
      ```sql  
      DECLARE  
         TYPE emp\_table IS TABLE OF employees%ROWTYPE;  
         emp\_data emp\_table;  
      BEGIN  
         SELECT \* BULK COLLECT INTO emp\_data FROM employees;  
         -- Process the data  
      END;  
      ```

15. \*\*Autonomous Transaction in Oracle PL/SQL — Commit, Rollback:\*\*  
    - Example:  
      (Example already provided in the previous response)

16. \*\*Oracle PL/SQL Package — Type, Specification, Body:\*\*  
    - Example:  
      (Example already provided in the previous response)

17. \*\*Oracle PL/SQL Trigger Tutorial — Instead of, Compound:\*\*  
    - Example:  
      (Example already provided in the previous response)

18. \*\*Oracle PL/SQL Object Types Tutorial — Learn with Example:\*\*  
    - Example:  
      (Example already provided in the previous response)

19. \*\*Oracle PL/SQL Dynamic SQL Tutorial — Execute Immediate & DBMS\_SQL:\*\*  
    - Example:  
      ```sql  
      DECLARE  
         sql\_stmt VARCHAR2(200);  
      BEGIN  
         sql\_stmt := 'UPDATE employees SET salary = salary \* 1.1';  
         EXECUTE IMMEDIATE sql\_stmt;  
      END;  
      ```

20. \*\*Nested Structure — PL/SQL Variable Scope & Inner Outer Block:\*\*  
    - Example:  
      ```sql  
      DECLARE  
         outer\_variable NUMBER := 10;  
      BEGIN  
         DBMS\_OUTPUT.PUT\_LINE('Outer Variable: ' || outer\_variable);  
         DECLARE  
             inner\_variable NUMBER := 20;  
         BEGIN  
             DBMS\_OUTPUT.PUT\_LINE('Inner Variable: ' || inner\_variable);  
         END;  
      END;  
      ```

These examples should help you understand each lesson better and get you started with PL/SQL programming.