Sister Nivedita University

Department of Computer Science and Engineering

DATABASE MANAGEMENT SYSTEM LAB

What are PL / SQL?

- PL/SQL is an extension of Structured Query Language (SQL) that is used in Oracle. Unlike SQL, PL/SQL allows the programmer to write code in a procedural format. Full form of PL/SQL is "Procedural Language extensions to SQL".

Features of PL/SQL

PL/SQL has the following features −

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages −

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- 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.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.
- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

PL/SQL - Basic Syntax

PL/SQL is a **block-structured** language. PL/SQL programs are divided and written in logical blocks of code. Each block consists of three sub-parts.

1. **Declarations:**

   This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

2. **Executable Commands:**

   This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.

3. **Exception Handling:**

   This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

   Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**.

Structure of PL / SQL

```
DECLARE
  <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling>
END;
```

Example

```
DECLARE
  message  varchar2(20):= 'Hello, World!';
BEGIN
  dbms_output.put_line(message);
END;
/
```

PL/SQL Identifiers

PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words. The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters.

By default, identifiers are not case-sensitive. So you can use integer or INTEGER to represent a numeric value. You cannot use a reserved keyword as an identifier.

PL/SQL Delimiters

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
||	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
<, >, <=, >=	Relational operators
<>, '=, ~=, ^=	Different versions of NOT EQUAL

PL/SQL Comments

Program comments are explanatory statements that can be included in the PL/SQL code that you write and helps anyone reading its source code. All programming languages allow some form of comments.

The PL/SQL supports single-line and multi-line comments. All characters available inside any comment are ignored by the PL/SQL compiler. The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /* and */.

Example:

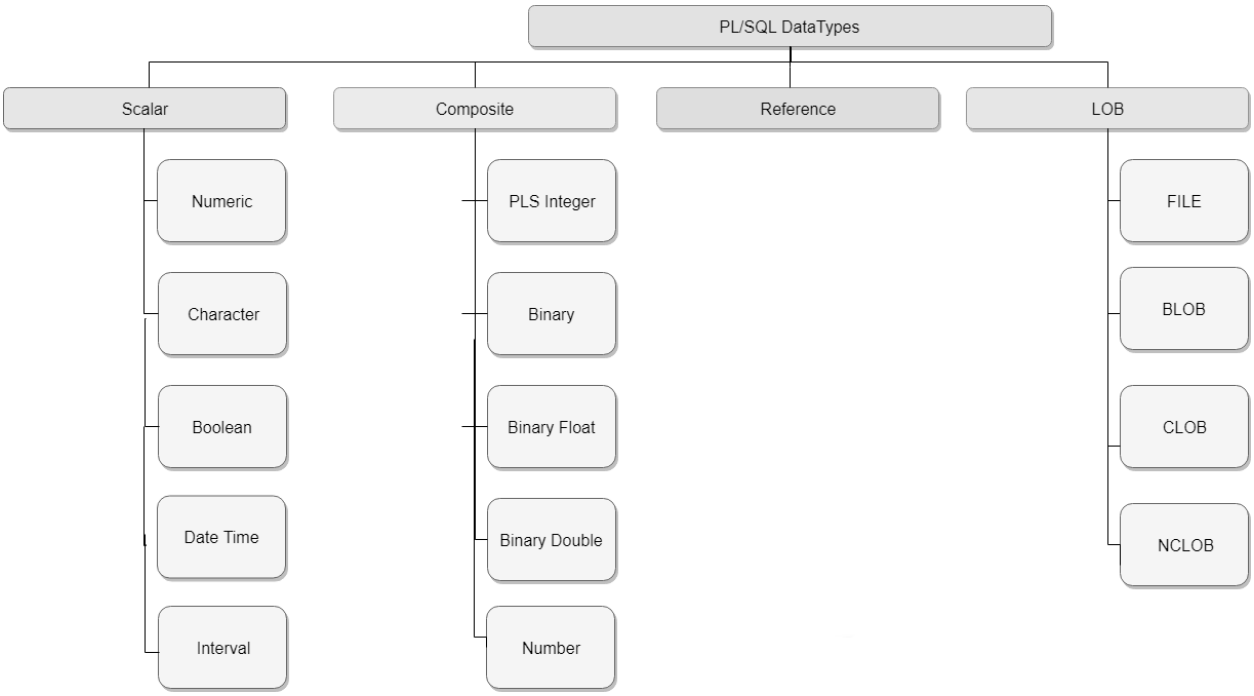
```
DECLARE
  -- variable declaration
  message  varchar2(20):= 'Hello, World!';
BEGIN
  /*
  *  PL/SQL executable statement(s)
  */
  dbms_output.put_line(message);
END;
/
```

PL/SQL Program Units:

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

PL/SQL - Data Types

A data type is associated with the specific storage format and range constraints. In Oracle, each value or constant is assigned with a data type.
Basically, it defines how the data is stored, handled and treated by Oracle during the data storage and processing.
The main difference between PL/SQL and SQL data types is, SQL data type are limited to table column while the PL/SQL data types are used in the PL/SQL blocks.



- **PL/SQL Scalar Data Types and Subtypes**

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 Numeric Data Types and Subtypes**

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)

- **PL/SQL Character Data Types and Subtypes**

S.No	Data Type & Description
1	**CHAR** Fixed-length character string with maximum size of 32,767 bytes
2	**VARCHAR2** Variable-length character string with maximum size of 32,767 bytes
3	**RAW** Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
4	**NCHAR** Fixed-length national character string with maximum size of 32,767 bytes
5	**NVARCHAR2** Variable-length national character string with maximum size of 32,767 bytes
6	**LONG** Variable-length character string with maximum size of 32,760 bytes

7	**LONG RAW** Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
8	**ROWID** Physical row identifier, the address of a row in an ordinary table
9	**UROWID** Universal row identifier (physical, logical, or foreign row identifier)

- **PL/SQL Boolean Data Types**

The **BOOLEAN** data type stores logical values that are used in logical operations. The logical values are the Boolean values **TRUE** and **FALSE** and the value **NULL**.

However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in –

- SQL statements
- Built-in SQL functions (such as **TO_CHAR**)
- PL/SQL functions invoked from SQL statements

- **PL/SQL Date time and Interval Types**

The **DATE** data type is used to store fixed-length datetimes, which include the time of day in seconds since midnight. Valid dates range from January 1, 4712 BC to December 31, 9999 AD.

The default date format is set by the Oracle initialization parameter NLS_DATE_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year. For example, 01-OCT-12.

Each DATE includes the century, year, month, day, hour, minute, and second. The following table shows the valid values for each field

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer

HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable

- **NULLs in PL/SQL**

PL/SQL NULL values represent **missing** or **unknown data** and they are not an integer, a character, or any other specific data type. Note that **NULL** is not the same as an empty data string or the null character value **'\0'**. A null can be assigned but it cannot be equated with anything, including itself.

PL/SQL – Variables

PL/SQL programming language allows defining various types of variables, such as date time data types, records, collections.

- **Variable Declaration in PL/SQL**

Syntax

variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

variable_name is a valid identifier in PL/SQL, datatype must be a valid PL/SQL data type or any user defined data type which we already have discussed.

Example

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

- **Initializing Variables in PL/SQL**

Example

```
DECLARE
   a integer := 30;
   b integer := 20;
   c integer;
   f real;
BEGIN
   c := a + b;
   dbms_output.put_line('Value of c: ' || c);
   f := 70.0/3.0;
   dbms_output.put_line('Value of f: ' || f);
END;
/
```

Output

```
Value of c: 50

Value of f: 23.333333333333333

PL/SQL procedure successfully completed.
```

- **Assigning SQL Query Results to PL/SQL Variables**

Example 1

```
CREATE TABLE CUSTOMERS(
   ID   INT NOT NULL,
   NAME VARCHAR (20) NOT NULL,
   AGE INT NOT NULL,
   ADDRESS CHAR (25),
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID)
);

Table Created
```

Let us now insert some values in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
```

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

The following program assigns values from the above table to PL/SQL variables

```
DECLARE
  c_id customers.id%type := 1;
  c_name  customers.name%type;
  c_addr customers.address%type;
  c_sal  customers.salary%type;
BEGIN
  SELECT name, address, salary INTO c_name, c_addr, c_sal
  FROM customers
  WHERE id = c_id;
  dbms_output.put_line
  ('Customer ' ||c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

Result

Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully

Example 2

```
DECLARE

TYPE emp_det IS RECORD

(

EMP_NO NUMBER,

EMP_NAME VARCHAR2(150),

MANAGER NUMBER,
```

```
SALARY NUMBER
);
ex2_emp_rec emp_det;
BEGIN
ex2_emp_rec.emp_no:= 1001;
ex2_emp_rec.emp_name:='XYZ';
ex2_emp_rec.manager:= 1000;
ex2_emp_rec.salary:=10000;
dbms_output.put_line('Employee Detail');
dbms_output.put_line ('Employee Number: '|| ex2_emp_rec.emp_no);
dbms_output.put_line ('Employee Name: '|| ex2_emp_rec.emp_name);
dbms_output.put_line ('Employee Salary: ' || ex2_emp_rec.salary);
dbms_output.put_line ('Employee Manager Number: '|| ex2_emp_rec.manager);
END;
/
```

Output:

```
Employee Detail
Employee Number: 1001
Employee Name: XYZ
Employee Salary: 10000
Employee Manager Number: 1000
```

PL/SQL - Constants and Literals

A constant holds a value that once declared, does not change in the program.

- **Declaring a Constant**

Example

```
PI CONSTANT NUMBER := 3.141592654;
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
   -- processing
   radius := 9.5;
   dia := radius * 2;
   circumference := 2.0 * pi * radius;
   area := pi * radius * radius;
   -- output
   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;
/
```

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

```
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53

Pl/SQL procedure successfully completed.
```

- **PL/SQL Literals**

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier.

- Numeric Literals
- Character Literals
- String Literals
- BOOLEAN Literals
- Date and Time Literals

The following table provides examples from all these categories of literal values.

S.No	Literal Type & Example
1	**Numeric Literals** 050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3
2	**Character Literals** 'A' '%' '9' ' ' 'z' '('
3	**String Literals** 'Hello, world!' '19-NOV-12'
4	**BOOLEAN Literals** TRUE, FALSE, and NULL.
5	**Date and Time Literals** DATE '1978-12-25'; TIMESTAMP '2012-10-29 12:01:01';

To embed single quotes within a string literal, place two single quotes next to each other as shown in the following program –

Example

```
DECLARE
   message  varchar2(30):= 'Sister Nivedita University';
BEGIN
   dbms_output.put_line(message);
END;
/
```

Result
Sister Nivedita University
PL/SQL procedure successfully completed.

PL/SQL – Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators –

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

- **Arithmetic operators**

Following table shows all the arithmetic operators supported by PL/SQL. Let us assume variable A **holds 10 and** variable B **holds 5, then –**

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

- **Relational Operators**

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Let us assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <> ~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.

<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true

- **Comparison Operators**

Comparison operators are used for comparing one expression to another. The result is always either **TRUE, FALSE** or **NULL**.

Show Examples

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b.	If x = 10 then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If x = 'm', then 'x is null' returns Boolean false.

- **Logical Operators**

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produce Boolean results. Let us assume **variable A** holds true and **variable B** holds false, then −

Show Examples

Operator	Description	Examples
and	Called the logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called the logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called the logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.

- **PL/SQL Operator Precedence**

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, **x = 7 + 3 * 2**; here, **x** is assigned **13**, not 20 because operator * has higher precedence than +, so it first gets multiplied with **3*2** and then adds into **7**.

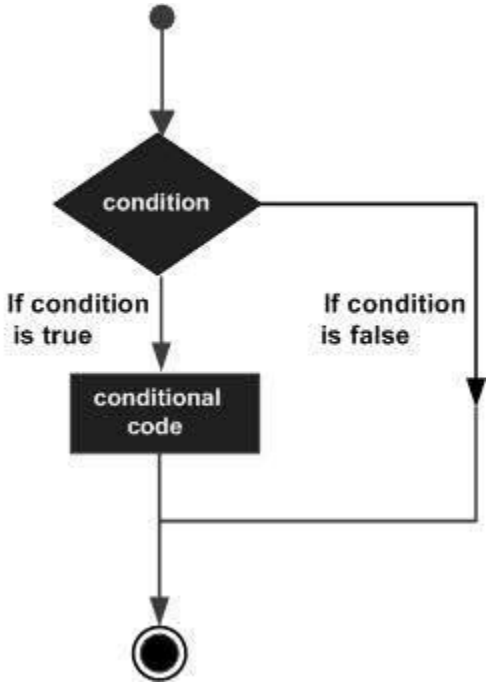
Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

The precedence of operators goes as follows: =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

Show Examples

Operator	Operation
**	exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
comparison	
NOT	logical negation
AND	conjunction
OR	inclusion

PL/SQL – Conditions



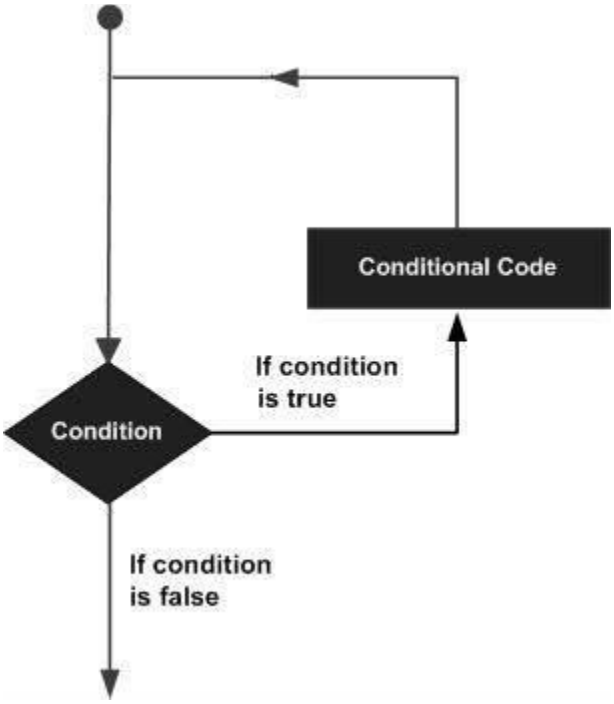
PL/SQL programming language provides following types of decision-making statements.

S.No	Statement & Description
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.
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.
3	IF-THEN-ELSIF statement It allows you to choose between several alternatives.
4	Case statement Like the IF statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.

5	Searched CASE statement The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.
6	nested IF-THEN-ELSE You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s).

PL/SQL – Loops

A loop statement allows us to execute a statement or group of statements multiple times.



S.No	Loop Type & Description
1	PL/SQL Basic LOOP 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 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 Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
4	Nested loops in PL/SQL You can use one or more loop inside any another basic loop, while, or for loop.

1. PL/SQL Basic LOOP

Syntax

The syntax of a basic loop in PL/SQL programming language is −

```
LOOP
  Sequence of statements;
END LOOP;
```

Here, the sequence of statement(s) may be a single statement or a block of statements. An **EXIT statement** or an **EXIT WHEN statement** is required to break the loop.

Example

```
DECLARE
  x number := 10;
BEGIN
  LOOP
    dbms_output.put_line(x);
    x := x + 10;
    IF x > 50 THEN
      exit;
    END IF;
  END LOOP;
  -- after exit, control resumes here
  dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

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

```
10
20
30
```

```
40
50
After Exit x is: 60

PL/SQL procedure successfully completed.
```

You can use the **EXIT WHEN** statement instead of the **EXIT** statement −

```
DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      exit WHEN x > 50;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

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

```
10
20
30
40
50
After Exit x is: 60

PL/SQL procedure successfully completed.
```

2. PL/SQL WHILE LOOP

Syntax

```
WHILE condition LOOP
   sequence_of_statements
END LOOP;
```

Example

```
DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
```

```
    a := a + 1;
  END LOOP;
END;
/
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19

PL/SQL procedure successfully completed.
```

3. PL/SQL FOR LOOP

Syntax

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

Following is the flow of control in a **For Loop** −

- The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- Next, the condition, i.e., *initial_value .. final_value* is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and the flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the value of the counter variable is increased or decreased.
- The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes FALSE, the FOR-LOOP terminates.

Following are some special characteristics of PL/SQL for loop −

- The *initial_value* and *final_value* of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.

- The *initial_value* need not be 1; however, the **loop counter increment (or decrement) must be 1**.
- PL/SQL allows the determination of the loop range dynamically at run time.

Example

```
DECLARE
  a number(2);
BEGIN
  FOR a in 10 .. 20 LOOP
    dbms_output.put_line('value of a: ' || a);
 END LOOP;
END;
/
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20

PL/SQL procedure successfully completed.
```

Reverse FOR LOOP Statement

By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound. You can reverse this order by using the **REVERSE** keyword. In such case, iteration proceeds the other way. After each iteration, the loop counter is decremented.

However, you must write the range bounds in ascending (not descending) order. The following program illustrates this −

```
DECLARE
  a number(2) ;
BEGIN
  FOR a IN REVERSE 10 .. 20 LOOP
    dbms_output.put_line('value of a: ' || a);
  END LOOP;
END;
/
```

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

```
value of a: 20
value of a: 19
value of a: 18
value of a: 17
value of a: 16
value of a: 15
value of a: 14
value of a: 13
value of a: 12
value of a: 11
value of a: 10

PL/SQL procedure successfully completed.
```

4. Nested loops in PL/SQL

The syntax for a nested basic LOOP statement in PL/SQL is as follows −

```
LOOP
  Sequence of statements1
  LOOP
    Sequence of statements2
  END LOOP;
END LOOP;
```

The syntax for a nested FOR LOOP statement in PL/SQL is as follows −

```
FOR counter1 IN initial_value1 .. final_value1 LOOP
  sequence_of_statements1
  FOR counter2 IN initial_value2 .. final_value2 LOOP
    sequence_of_statements2
  END LOOP;
END LOOP;
```

The syntax for a nested WHILE LOOP statement in Pascal is as follows −

```
WHILE condition1 LOOP
  sequence_of_statements1
  WHILE condition2 LOOP
    sequence_of_statements2
  END LOOP;
END LOOP;
```

Example

The following program uses a nested basic loop to find the prime numbers from 2 to 100 −

```
DECLARE
```

```
  i number(3);
  j number(3);
BEGIN
  i := 2;
  LOOP
    j:= 2;
    LOOP
      exit WHEN ((mod(i, j) = 0) or (j = i));
      j := j +1;
    END LOOP;
  IF (j = i ) THEN
    dbms_output.put_line(i || ' is prime');
  END IF;
  i := i + 1;
  exit WHEN i = 50;
  END LOOP;
END;
/
```

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

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime

PL/SQL procedure successfully completed.
```

PL/SQL – Strings

The string in PL/SQL is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all.

- **Fixed-length strings** − In such strings, programmers specify the length while declaring the string. The string is right-padded with spaces to the length so specified.

- **Variable-length strings** − In such strings, a maximum length up to 32,767, for the string is specified and no padding takes place.
- **Character large objects (CLOBs)** − These are variable-length strings that can be up to 128 terabytes.

- **Declaring String Variables**

Oracle database provides numerous string datatypes, such as CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB.

Example

```
DECLARE
   name varchar2(20);
   company varchar2(30);
   introduction clob;
   choice char(1);
BEGIN
   name := 'John Smith';
   company := 'Infotech';
   introduction := ' Hello! I"m John Smith from Infotech.';
   choice := 'y';
   IF choice = 'y' THEN
      dbms_output.put_line(name);
      dbms_output.put_line(company);
      dbms_output.put_line(introduction);
   END IF;
END;
/
```

Result

```
John Smith
Infotech
Hello! I'm John Smith from Infotech.

PL/SQL procedure successfully completed.
```

PL/SQL String Functions and Operators

S.No	Function & Purpose
1	**ASCII(x);** Returns the ASCII value of the character x.

2	**CHR(x);** Returns the character with the ASCII value of x.
3	**CONCAT(x, y);** Concatenates the strings x and y and returns the appended string.
4	**INITCAP(x);** Converts the initial letter of each word in x to uppercase and returns that string.
5	**INSTR(x, find_string [, start] [, occurrence]);** Searches for **find_string** in x and returns the position at which it occurs.
6	**INSTRB(x);** Returns the location of a string within another string, but returns the value in bytes.
7	**LENGTH(x);** Returns the number of characters in x.
8	**LENGTHB(x);** Returns the length of a character string in bytes for single byte character set.
9	**LOWER(x);** Converts the letters in x to lowercase and returns that string.
10	**LPAD(x, width [, pad_string]) ;** Pads **x** with spaces to the left, to bring the total length of the string up to width characters.
11	**LTRIM(x [, trim_string]);** Trims characters from the left of **x**.
12	**NANVL(x, value);** Returns value if x matches the NaN special value (not a number), otherwise **x** is returned.
13	**NLS_INITCAP(x);** Same as the INITCAP function except that it can use a different sort method as specified by

	NLSSORT.
14	**NLS_LOWER(x) ;** Same as the LOWER function except that it can use a different sort method as specified by NLSSORT.
15	**NLS_UPPER(x);** Same as the UPPER function except that it can use a different sort method as specified by NLSSORT.
16	**NLSSORT(x);** Changes the method of sorting the characters. Must be specified before any NLS function; otherwise, the default sort will be used.
17	**NVL(x, value);** Returns value if **x** is null; otherwise, x is returned.
18	**NVL2(x, value1, value2);** Returns value1 if x is not null; if x is null, value2 is returned.
19	**REPLACE(x, search_string, replace_string);** Searches **x** for search_string and replaces it with replace_string.
20	**RPAD(x, width [, pad_string]);** Pads **x** to the right.
21	**RTRIM(x [, trim_string]);** Trims **x** from the right.
22	**SOUNDEX(x) ;** Returns a string containing the phonetic representation of **x**.
23	**SUBSTR(x, start [, length]);** Returns a substring of **x** that begins at the position specified by start. An optional length for the substring may be supplied.
24	**SUBSTRB(x);**

	Same as SUBSTR except that the parameters are expressed in bytes instead of characters for the single-byte character systems.
25	**TRIM([trim_char FROM) x);** Trims characters from the left and right of **x**.
26	**UPPER(x);** Converts the letters in x to uppercase and returns that string.

Let us now work out on a few examples to understand the concept −

Example 1

```
DECLARE
  greetings varchar2(11) := 'hello world';
BEGIN
  dbms_output.put_line(UPPER(greetings));

  dbms_output.put_line(LOWER(greetings));

  dbms_output.put_line(INITCAP(greetings));

  /* retrieve the first character in the string */
  dbms_output.put_line ( SUBSTR (greetings, 1, 1));

  /* retrieve the last character in the string */
  dbms_output.put_line ( SUBSTR (greetings, -1, 1));

  /* retrieve five characters,
     starting from the seventh position. */
  dbms_output.put_line ( SUBSTR (greetings, 7, 5));

  /* retrieve the remainder of the string,
     starting from the second position. */
  dbms_output.put_line ( SUBSTR (greetings, 2));

  /* find the location of the first "e" */
  dbms_output.put_line ( INSTR (greetings, 'e'));
END;
/
```

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

```
HELLO WORLD
hello world
Hello World
```

h
d
World
ello World
2

PL/SQL procedure successfully completed.

Example 2

```
DECLARE
   greetings varchar2(30) := '......Hello World.....';
BEGIN
   dbms_output.put_line(RTRIM(greetings,'.'));
   dbms_output.put_line(LTRIM(greetings, '.'));
   dbms_output.put_line(TRIM( '.' from greetings));
END;
/
```

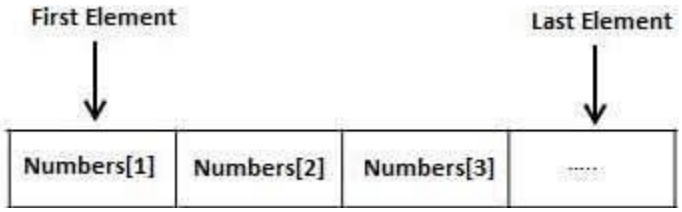
When the above code is executed at the SQL prompt, it produces the following result −

......Hello World
Hello World.....
Hello World

PL/SQL procedure successfully completed.

PL/SQL – Arrays

The PL/SQL programming language provides a data structure called the **VARRAY**, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, however it is often better to think of an array as a collection of variables of the same type. All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Creating a Varray Type

A varray type is created with the **CREATE TYPE** statement. You must specify the maximum size and the type of elements stored in the varray.

Syntax:

CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>

Where

- *varray_type_name* is a valid attribute name,
- *n* is the number of elements (maximum) in the varray,
- *element_type* is the data type of the elements of the array.

Example:

```
CREATE Or REPLACE TYPE namearray AS VARRAY(3) OF VARCHAR2(10);
/

Type created.
```

Example to illustrate the use of varrays:

```
DECLARE
  type namesarray IS VARRAY(5) OF VARCHAR2(10);
  type grades IS VARRAY(5) OF INTEGER;
  names namesarray;
  marks grades;
  total integer;
BEGIN
  names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
  marks:= grades(98, 97, 78, 87, 92);
  total := names.count;
  dbms_output.put_line('Total '|| total || ' Students');
  FOR i in 1 .. total LOOP
    dbms_output.put_line('Student: ' || names(i) || '
    Marks: ' || marks(i));
  END LOOP;
END;
/
```

Result:

```
Total 5 Students
Student: Kavita  Marks: 98
Student: Pritam  Marks: 97
Student: Ayan  Marks: 78
Student: Rishav  Marks: 87
Student: Aziz  Marks: 92
```

Note:

- In Oracle environment, the starting index for varrays is always 1.
- You can initialize the varray elements using the constructor method of the varray type, which has the same name as the varray.
- Varrays are one-dimensional arrays.
- A varray is automatically NULL when it is declared and must be initialized before its elements can be referenced.

PL/SQL – Procedures

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.
A subprogram can be created −

- At the schema level
- Inside a package
- Inside a PL/SQL block

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.
PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters.

PL/SQL provides two kinds of subprograms −

- **Functions** − these subprograms return a single value; mainly used to compute and return a value.
- **Procedures** − these subprograms do not return a value directly; mainly used to perform an action.

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 −

Sl.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.

Creating a Procedure
A procedure is created with the CREATE OR REPLACE PROCEDURE statement.
Syntax:-
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}
BEGIN
< procedure_body >
END procedure_name;
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.

Example:

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
  dbms_output.put_line('Hello World!');
END;
/
```

Executing a Standalone Procedure
A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named '**greetings**' can be called with the EXECUTE keyword as −
EXECUTE greetings;
The above call will display −
Hello World
The procedure can also be called from another PL/SQL block −

```
BEGIN
  greetings;
END;
/
```

The above call will display −
Hello World

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;

Parameter Modes in PL/SQL Subprograms

S.No	Parameter Mode & Description
1	IN

	An IN parameter lets you pass a value to the subprogram. It is a read-only parameter. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. It is the default mode of parameter passing. Parameters are passed by reference.
2	OUT An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. The actual parameter must be variable and it is passed by value.
3	IN OUT An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. Actual parameter is passed by value.

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 < y THEN
    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;
/
```

Result:

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

Example 2

```
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;
/
```

Result:

Square of (23): 529

Methods for Passing Parameters

Actual parameters can be passed in three ways −

- Positional notation
- Named notation
- Mixed notation

- **Positional Notation**

In positional notation, you can call the procedure as −

findMin(a, b, c, d);

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

- **Named Notation**

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol (=>)**. The procedure call will be like the following −

findMin(x => a, y => b, z => c, m => d);

- **Mixed Notation**

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal −

findMin(a, b, c, m => d);

However, this is not legal:

findMin(x => a, b, c, d);

PL/SQL – Functions

A function is same as a procedure except that it returns a value.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement.

Syntax:

```
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 illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

Select * from customers;

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

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
  total number(2) := 0;
BEGIN
```

```
  SELECT count(*) into total
  FROM customers;

  RETURN total;
END;
/
```

Result:

Function created.

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value.

Example

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
  dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

Result

Total no. of Customers: 6

Example

```
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;
/
```

Result

Maximum of (23,45): 45

PL/SQL Recursive Functions

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as **recursion**.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as −

```
n! = n*(n-1)!
  = n*(n-1)*(n-2)!
     ...
  = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively −

```
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;
/
```

Result −

Factorial 6 is 720

PL/SQL – Cursors

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc. A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.
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. Programmers cannot control the implicit cursors and the information in it. Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected. In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has attributes such as %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT.

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Example
We will be using the CUSTOMERS table.

```
Select * from customers;
```

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

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

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

Result

6 customers selected

If you check the records in customers table, you will find that the rows have been updated −
Select * from customers;

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

```
|  6 | Komal    |  22 | MP         |  5000.00 |
+----+----------+-----+------------+----------+
```

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block.
Syntax:
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

```
DECLARE
  c_id customers.id%type;
  c_name customer.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;
/
```

Result

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP
PL/SQL procedure successfully completed.
```

PL/SQL – Records

A record is a data structure that can hold data items of different kinds. Records consist of different fields, similar to a row of a database table.

For example, you want to keep track of your books in a library. You might want to track the following attributes about each book, such as Title, Author, Subject, Book ID. A record containing a field for each of these items allows treating a BOOK as a logical unit and allows you to organize and represent its information in a better way.

PL/SQL can handle the following types of records −

- Table-based
- Cursor-based records
- User-defined records

- **Table-Based Records**

The %ROWTYPE attribute enables a programmer to create **table-based** and **cursorbased** records.

We will be using the CUSTOMERS table.

```
DECLARE
  customer_rec customers%rowtype;
BEGIN
  SELECT * into customer_rec
  FROM customers
  WHERE id = 5;
  dbms_output.put_line('Customer ID: ' || customer_rec.id);
  dbms_output.put_line('Customer Name: ' || customer_rec.name);
  dbms_output.put_line('Customer Address: ' || customer_rec.address);
  dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
/
```

Result

Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000

- **Cursor-Based Records**

The following example illustrates the concept of **cursor-based** records. We will be using the CUSTOMERS table we had created and used in the previous chapters −

```
DECLARE
  CURSOR customer_cur is
    SELECT id, name, address
    FROM customers;
  customer_rec customer_cur%rowtype;
BEGIN
  OPEN customer_cur;
  LOOP
    FETCH customer_cur into customer_rec;
    EXIT WHEN customer_cur%notfound;
    DBMS_OUTPUT.put_line(customer_rec.id || ' ' || customer_rec.name);
  END LOOP;
END;
/
```

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

1 Ramesh
2 Khilan
3 kaushik
4 Chaitali
5 Hardik
6 Komal

- User-Defined Records

PL/SQL provides a user-defined record type that allows you to define the different record structures. These records consist of different fields. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

- Title
- Author
- Subject
- Book ID

Defining a Record

The Book record is declared in the following way −

```
DECLARE
TYPE books IS RECORD
(title  varchar(50),
   author  varchar(50),
   subject varchar(100),
   book_id   number);
book1 books;
book2 books;
```

Accessing Fields

To access any field of a record, we use the dot **(.)** operator. The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is an example to explain the usage of record −

```
DECLARE
   type books is record
      (title varchar(50),
      author varchar(50),
      subject varchar(100),
      book_id number);
   book1 books;
   book2 books;
BEGIN
   -- Book 1 specification
   book1.title  := 'C Programming';
   book1.author := 'Nuha Ali ';
   book1.subject := 'C Programming Tutorial';
   book1.book_id := 6495407;
   -- Book 2 specification
   book2.title := 'Telecom Billing';
   book2.author := 'Zara Ali';
   book2.subject := 'Telecom Billing Tutorial';
   book2.book_id := 6495700;
```

```
 -- Print book 1 record
  dbms_output.put_line('Book 1 title : '|| book1.title);
  dbms_output.put_line('Book 1 author : '|| book1.author);
  dbms_output.put_line('Book 1 subject : '|| book1.subject);
  dbms_output.put_line('Book 1 book_id : ' || book1.book_id);

  -- Print book 2 record
  dbms_output.put_line('Book 2 title : '|| book2.title);
  dbms_output.put_line('Book 2 author : '|| book2.author);
  dbms_output.put_line('Book 2 subject : '|| book2.subject);
  dbms_output.put_line('Book 2 book_id : '|| book2.book_id);
END;
/
```

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

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700

PL/SQL procedure successfully completed.
```

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

```
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
    exception3-handling-statements
  ........
  WHEN others THEN
    exception3-handling-statements
END;
```

Example

We will be using the CUSTOMERS table.

```
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
    dbms_output.put_line('No such customer!');
  WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

Result −

No such customer!

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**.

Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**.

Syntax for raising an exception −

```
DECLARE
  exception_name EXCEPTION;
BEGIN
  IF condition THEN
```

```
    RAISE exception_name;
  END IF;
EXCEPTION
  WHEN exception_name THEN
  statement;
END;
```

User-defined Exceptions

PL/SQL allows you to define your own exceptions according to the need of your program. A user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure **DBMS_STANDARD.RAISE_APPLICATION_ERROR**.

The syntax for declaring an exception is −

```
DECLARE
  my-exception EXCEPTION;
```

Example

The following example illustrates the concept. This program asks for a customer ID, when the user enters an invalid ID, the exception **invalid_id** is raised.

```
DECLARE
  c_id customers.id%type := &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 IF;

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

```
/
```

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

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
new  2: c_id customers.id%type := -6;
ID must be greater than zero!
```

Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program.

Exception	Oracle Error	SQLCODE	Description
ACCESS_INTO_NULL	06530	-6530	It is raised when a null object is automatically assigned a value.
CASE_NOT_FOUND	06592	-6592	It is raised when none of the choices in the WHEN clause of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	06531	-6531	It is raised when a program attempts to apply collection methods other than EXISTS to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
DUP_VAL_ON_INDEX	00001	-1	It is raised when duplicate values are attempted to be stored in a column with unique index.
INVALID_CURSOR	01001	-1001	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.

LOGIN_DENIED	01017	-1017	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	01403	+100	It is raised when a SELECT INTO statement returns no rows.
NOT_LOGGED_ON	01012	-1012	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	06501	-6501	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	It is raised when a cursor fetches value in a variable having incompatible data type.
SELF_IS_NULL	30625	-30625	It is raised when a member method is invoked, but the instance of the object type was not initialized.
STORAGE_ERROR	06500	-6500	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	01422	-1422	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	06502	-6502	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	01476	1476	It is raised when an attempt is made to divide a number by zero.

PL/SQL - Triggers

Triggers are stored programs, which are automatically executed or fired when some events occur. 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).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits 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.

Creating Triggers

The syntax for creating a trigger is −

```
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.

Example

we will be using the CUSTOMERS table.

```
Select * from customers;
```

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

The following program 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;
/
```

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

Trigger created.

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.

- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

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

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary:
New salary: 7500
Salary difference:
```

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

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

Result

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```