**Introduction to PL/SQL**

Whenever we have the requirements for any conditional update or error handling, then we require PL/SQL. It is basically procedural language with sql features, that enables us to do some programming stuffs with sql operations.

**Features:**

* Provides programming constructs like variable declarations, conditional statements and loops
* Supports error handling mechanisms
* Supports object oriented programming
* Supports composite data types

**PL/SQL Block:**

It is the basic programming structure in PL/SQL Program.

It has three sections.

1. Declaration Section
2. Executable Section
3. Exception Section

**Declaration Section (Optional):**

It is the section where we declare variables and cursors etc.

**Executable Section (Mandatory):**

It is the section where we write the programming logics.

**Exception Section (Optional):**

It is the section where we handle error at runtime

**Basic Structure of PL/SQL Block:**

DECLARE

--Declare Variables (Declaration Section)

BEGIN

--Write Programming Logics (Executable Section)

EXCEPTION

--Handle exceptions (Exception Section)

END;

* **Printing a message:**

BEGIN

DBMS\_OUTPUT.PUT\_LINE(‘Welcome to PL/SQL Training’);

END;

Result : Welcome to PL/SQL Training

* **Summation of two numbers**

DECLARE

X NUMBER := 5;

Y NUMBER := 10;

Z NUMBER;

BEGIN

Z := X+Y;

DBMS\_OUTPUT.PUT\_LINE(‘Sum is ’|| Z);

END;

* **Generating Errors**

DECLARE

X NUMBER := 5;

Y VARCHAR2(100) := ‘A’;

Z NUMBER;

BEGIN

Z := X+Y;

DBMS\_OUTPUT.PUT\_LINE(‘Sum is ’|| Z);

END;

Result : ORA-06052 – PL/SQL : numeric or value error : character to number conversion error

* **Handling errors**

DECLARE

X NUMBER := 5;

Y VARCHAR2(100) := ‘A’;

Z NUMBER;

BEGIN

Z := X+Y;

DBMS\_OUTPUT.PUT\_LINE(‘Sum is ’|| Z);

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE(‘Error is encountered’);

END;

Result : Error is encountered.

* **Printing name of employee with PL/SQL**

DECLARE

lv\_name varchar2(100);

BEGIN

SELECT ENAME INTO lv\_name FROM EMP WHERE EMPNO = ‘7499’;

DBMS\_OUTPUT.PUT\_LINE(‘Employee Name is ’ || lv\_name);

END;

Result : Employee Name is ALLEN

* **When we don’t have data into the table and trying to assign in the local variable**

ORA-01403 : No data found

* **Handling those errors**

DECLARE

lv\_name varchar2(100);

BEGIN

SELECT ENAME INTO lv\_name FROM EMP WHERE EMPNO = ‘7499’;

DBMS\_OUTPUT.PUT\_LINE(‘Employee Name is ’ || lv\_name);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE(‘No Data Found’);

WHEN TOO\_MANY\_ROWS THEN

DBMS\_OUTPUT.PUT\_LINE(‘More than 1 data found’);

END;

//In PL/SQL , we have cursor concepts for multiple rows

**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.

**PL/SQL Program Units:**

A PL/SQL unit is any one of the following −

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

**Variable Scope in PL/SQL:**

* **Local variables** − Variables declared in an inner block and not accessible to outer blocks.
* **Global variables** − Variables declared in the outermost block or a package.

DECLARE

-- Global variables

num1 number := 95;

num2 number := 85;

BEGIN

dbms\_output.put\_line('Outer Variable num1: ' || num1);

dbms\_output.put\_line('Outer Variable num2: ' || num2);

DECLARE

-- Local variables

num1 number := 195;

num2 number := 185;

BEGIN

dbms\_output.put\_line('Inner Variable num1: ' || num1);

dbms\_output.put\_line('Inner Variable num2: ' || num2);

END;

END;

/

**Conditions in PL/SQL:**

PL/SQL programming language provides following types of decision-making statements. Click the following links to check their detail.

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;

|  |  |
| --- | --- |
| **S.No** | **Statement & Description** |
| 1 | [IF - THEN statement](https://www.tutorialspoint.com/plsql/plsql_if_then.htm)  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](https://www.tutorialspoint.com/plsql/plsql_if_then_else.htm)  **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](https://www.tutorialspoint.com/plsql/plsql_if_then_elsif.htm)  It allows you to choose between several alternatives. |
| 4 | [Case statement](https://www.tutorialspoint.com/plsql/plsql_case_statement.htm)  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](https://www.tutorialspoint.com/plsql/plsql_searched_case.htm)  The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values. |
| 6 | [nested IF-THEN-ELSE](https://www.tutorialspoint.com/plsql/plsql_nested_if.htm)  You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s). |

**Loops in PL/SQL:**

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

DECLARE

i number(1);

j number(1);

BEGIN

<< outer\_loop >>

FOR i IN 1..3 LOOP

<< inner\_loop >>

FOR j IN 1..3 LOOP

dbms\_output.put\_line('i is: '|| i || ' and j is: ' || j);

END loop inner\_loop;

END loop outer\_loop;

END;

/

**Loop Control Statements:**

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

|  |  |
| --- | --- |
| **S.No** | **Control Statement & Description** |
| 1 | [EXIT statement](https://www.tutorialspoint.com/plsql/plsql_exit_statement.htm)  The Exit statement completes the loop and control passes to the statement immediately after the END LOOP. |
| 2 | [CONTINUE statement](https://www.tutorialspoint.com/plsql/plsql_continue_statement.htm)  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | [GOTO statement](https://www.tutorialspoint.com/plsql/plsql_goto_statement.htm)  Transfers control to the labeled statement. Though it is not advised to use the GOTO statement in your program. |

**Some String Functions:**

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;

/

**Array in PL/SQL:**

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.

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.

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;

/

**Please 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 SubPrograms:**

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.

**Creating a Procedure**

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

CREATE [OR REPLACE] PROCEDURE procedure\_name

[(parameter\_name [IN | OUT | IN OUT] type [, ...])]

{IS | AS}

BEGIN

< procedure\_body >

END procedure\_name;

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.

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

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

END;

/

EXECUTE greetings;

BEGIN

greetings;

END;

/

For dropping procedures : DROP PROCEDURE greetings;

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;

/

**Functions in PL/SQL:**

## Creating a Function

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

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.

CREATE OR REPLACE FUNCTION totalCustomers

RETURN number IS

total number(2) := 0;

BEGIN

SELECT count(\*) into total

FROM customers;

RETURN total;

END;

/

DECLARE

c number(2);

BEGIN

c := totalCustomers();

dbms\_output.put\_line('Total no. of Customers: ' || c);

END;

/

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

DECLARE

a number;

b number;

c number;

FUNCTION findMax(x IN number, y IN number)

RETURN number

IS

z number;

BEGIN

IF x > y THEN

z:= x;

ELSE

Z:= y;

END IF;

RETURN z;

END;

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

dbms\_output.put\_line(' Maximum of (23,45): ' || c);

END;

/

**Cursor in PL/SQL:**

A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors −

* Implicit cursors
* Explicit cursors

**Implicit Cursors:**

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

Any SQL cursor attribute will be accessed as **sql%attribute\_name.**

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;

/

**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

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; --Declaring cursor

BEGIN

OPEN c\_customers; --Opening cursor

LOOP

FETCH c\_customers into c\_id, c\_name, c\_addr; --Fetching cursor

EXIT WHEN c\_customers%notfound;

dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);

END LOOP;

CLOSE c\_customers; --Closing cursor

END;

/

**Exception Handling:**

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

WHEN exception3 THEN

exception3-handling-statements

........

WHEN others THEN

exception3-handling-statements

END;

**Raising exceptions:**

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;

/

**Collections in PL/SQL:**

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types −

* Index-by tables or Associative array
* Nested table
* Variable-size array or Varray

## Index-By Table

An **index-by** table (also called an **associative array**) is a set of **key-value** pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an **index-by** table named **table\_name**, the keys of which will be of the subscript\_type and associated values will be of the *element\_type*

TYPE type\_name IS TABLE OF element\_type [NOT NULL] INDEX BY subscript\_type;

table\_name type\_name;

### **Example**

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

DECLARE

TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);

salary\_list salary;

name VARCHAR2(20);

BEGIN

-- adding elements to the table

salary\_list('Rajnish') := 62000;

salary\_list('Minakshi') := 75000;

salary\_list('Martin') := 100000;

salary\_list('James') := 78000;

-- printing the table

name := salary\_list.FIRST;

WHILE name IS NOT null LOOP

dbms\_output.put\_line

('Salary of ' || name || ' is ' || TO\_CHAR(salary\_list(name)));

name := salary\_list.NEXT(name);

END LOOP;

END;

/

DECLARE

CURSOR c\_customers is

select name from customers;

TYPE c\_list IS TABLE of customers.Name%type INDEX BY binary\_integer;

name\_list c\_list;

counter integer :=0;

BEGIN

FOR n IN c\_customers LOOP

counter := counter +1;

name\_list(counter) := n.name;

dbms\_output.put\_line('Customer('||counter||'):'||name\_lis t(counter));

END LOOP;

END;

/

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

Customer(1): Ramesh

Customer(2): Khilan

Customer(3): kaushik

Customer(4): Chaitali

Customer(5): Hardik

Customer(6): Komal

## Nested Tables

A **nested table** is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects −

* An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
* An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A nested table is created using the following syntax −

TYPE type\_name IS TABLE OF element\_type [NOT NULL];

table\_name type\_name;

This declaration is similar to the declaration of an **index-by** table, but there is no **INDEX BY** clause.

A nested table can be stored in a database column. It can further be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

### **Example**

The following examples illustrate the use of nested table −

DECLARE

TYPE names\_table IS TABLE OF VARCHAR2(10);

TYPE grades IS TABLE OF INTEGER;

names names\_table;

marks grades;

total integer;

BEGIN

names := names\_table('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;

/

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

Total 5 Students

Student:Kavita, Marks:98

Student:Pritam, Marks:97

Student:Ayan, Marks:78

Student:Rishav, Marks:87

Student:Aziz, Marks:92

PL/SQL procedure successfully completed.