# PL/SQL Variables

A variable is a meaningful name which facilitates a programmer to store data temporarily during the execution of code. It helps you to manipulate data in PL/SQL programs. It is nothing except a name given to a storage area. Each variable in the PL/SQL has a specific data type which defines the size and layout of the variable's memory.

A variable should not exceed 30 characters. Its letter optionally followed by more letters, dollar signs, numerals, underscore etc.

#### 1. It needs to declare the variable first in the declaration section of a PL/SQL block before using it.

#### 2. By default, variable names are not case sensitive. A reserved PL/SQL keyword cannot be used as a variable name.

## How to declare variable in PL/SQL

You must declare the PL/SQL variable in the declaration section or in a package as a global variable. After the declaration, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

**Syntax for declaring variable:**

18.8M

281

Triggers in SQL (Hindi)

Following is the syntax for declaring variable:

1. variable\_name [CONSTANT] datatype [NOT NULL] [:= | **DEFAULT** initial\_value]

Here, variable\_name is a valid identifier in PL/SQL and datatype must be valid PL/SQL data type. A data type with size, scale or precision limit is called a constrained declaration. The constrained declaration needs less memory than unconstrained declaration.

**Example:**

Radius Number := 5;

Date\_of\_birth date;

### Declaration Restrictions:

In PL/SQL while declaring the variable some restrictions hold.

* Forward references are not allowed i.e. you must declare a constant or variable before referencing it in another statement even if it is a declarative statement.  
  val number := Total - 200;  
  Total number := 1000;  
  The first declaration is illegal because the TOTAL variable must be declared before using it in an assignment expression.
* Variables belonging to the same datatype cannot be declared in the same statement.  
  N1, N2, N3 Number;  
  It is an illegal declaration.

## Naming rules for PL/SQL variables

The variable in PL/SQL must follow some naming rules like other programming languages.

* The variable\_name should not exceed 30 characters.
* Variable name should not be the same as the table table's column of that block.
* The name of the variable must begin with ASCII letter. The PL/SQL is not case sensitive so it could be either lowercase or uppercase. For example: v\_data and V\_DATA refer to the same variables.
* You should make your variable easy to read and understand, after the first character, it may be any number, underscore (\_) or dollar sign ($).
* NOT NULL is an optional specification on the variable.

## Initializing Variables in PL/SQL

Evertime you declare a variable, PL/SQL defines a default value NULL to it. If you want to initialize a variable with other value than NULL value, you can do so during the declaration, by using any one of the following methods.

* The DEFAULT keyword
* The assignment operator

1. counter binary\_integer := 0;
2. greetings varchar2(20) **DEFAULT** 'Hello JavaTpoint';

You can also specify NOT NULL constraint to avoid NULL value. If you specify the NOT NULL constraint, you must assign an initial value for that variable.

You must have a good programming skill to initialize variable properly otherwise, sometimes program would produce unexpected result.

## Example of initilizing variable

Let's take a simple example to explain it well:

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

After the execution, this will produce the following result:

Value of c: 70

Value of f: 33.333333333333333333

PL/SQL procedure successfully completed.

## Variable Scope in PL/SQL:

PL/SQL allows nesting of blocks. A program block can contain another inner block. If you declare a variable within an inner block, it is not accessible to an outer block. There are two types of variable scope:

* Local Variable: Local variables are the inner block variables which are not accessible to outer blocks.
* Global Variable: Global variables are declared in outermost block.

## Example of Local and Global variables

Let's take an example to show the usage of Local and Global variables in its simple form:

1. **DECLARE**
2. -- Global variables
3. num1 number := 95;
4. num2 number := 85;
5. **BEGIN**
6. dbms\_output.put\_line('Outer Variable num1: ' || num1);
7. dbms\_output.put\_line('Outer Variable num2: ' || num2);
8. **DECLARE**
9. -- Local variables
10. num1 number := 195;
11. num2 number := 185;
12. **BEGIN**
13. dbms\_output.put\_line('Inner Variable num1: ' || num1);
14. dbms\_output.put\_line('Inner Variable num2: ' || num2);
15. **END**;
16. **END**;
17. /

After the execution, this will produce the following result:

Outer Variable num1: 95

Outer Variable num2: 85

Inner Variable num1: 195

Inner Variable num2: 185

PL/SQL procedure successfully completed.

### Variable Attributes:

When you declare a PL/SQL variable to hold the column values, it must be of correct data types and precision, otherwise error will occur on execution. Rather than hard coding the data type and precision of a variable. PL/SQL provides the facility to declare a variable without having to specify a particular data type using %TYPE and %ROWTYPE attributes. These two attributes allow us to specify a variable and have that variable data type be defined by a table/view column or a PL/SQL package variable.

A % sign servers as the attribute indicator. This method of declaring variables has an advantage as the user is not concerned with writing and maintaining code.

**Following are the types of Variable Attributes in PL/SQL.**

* **%TYPE:**

The %TYPE attribute is used to declare variables according to the already declared variable or database column. It is used when you are declaring an individual variable, not a record. The data type and precision of the variable declared using %TYPE attribute is the same as that of the column that is referred from a given table. This is particularly useful when declaring variables that will hold database values. When using the %TYPE keyword, the name of the columns and the table to which the variable will correspond must be known to the user. These are then prefixed with the variable name. If some previously declared variable is referred then prefix that variable name to the %TYPE attribute.

**The syntax for declaring a variable with %TYPE is:**

1. <var\_name> <tab\_name>.<column\_name>%TYPE;

Where <column\_name> is the column defined in the <tab\_name>.

**Consider a declaration.**

SALARY EMP.SAL % TYPE;

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

**Example:**

1. **DECLARE**
2. SALARY EMP.SAL % TYPE;
3. ECODE EMP.empno % TYPE;
4. **BEGIN**
5. Ecode :=&Ecode;
6. **Select** SAL **into** SALARY **from** EMP **where** EMPNO = ECODE;
7. dbms\_output.put\_line('Salary of ' || ECODE || 'is = || salary');
8. **END**;

After the execution, this will produce the following result:

Enter value for ecode: 7499

Salary of 7499 is = 1600

PL/SQL procedure successfully completed.

* **%ROWTYPE:**

The %ROWTYPE attribute is used to declare a record type that represents a row in a table. The record can store an entire row or some specific data selected from the table. A column in a row and corresponding fields in a record have the same name and data types.

**The syntax for declaring a variable with %ROWTYPE is:**

1. <var\_name> <tab\_name>.ROW%TYPE;

Where <variable\_name> is the variable defined in the <tab\_name>.

**Consider a declaration.**

EMPLOYEE EMP. % ROW TYPE;

This declaration will declare a record named EMPLOYEE having fields with the same name and data types as that of columns in the EMP table. You can access the elements of EMPLOYEE record as

EMPLOYEE.SAL := 10000;

EMPLOYEE.ENAME := ‘KIRAN’;

**Example:**

1. **DECLARE**
2. EMPLOYEE EMP. % ROW TYPE;
3. **BEGIN**
4. EMPLOYEE.EMPNO := 2092;
5. 5   EMPLOYEE.ENAME := 'Sanju';
6. **Insert** **into** EMP **where** (EMPNO, ENAME) **Values** (employee.empno, employee.ename);
7. dbms\_output.put\_line('Row Inserted');
8. **END**;

After the execution, this will produce the following result:

Row Inserted

PL/SQL procedure successfully completed.

## Advantages:

* If you don’t know the data type at the time of declaration. The data type assigned to the associated variables will be determined dynamically at run time.
* If the data type of the variable you are referencing changes the %TYPE or %ROWTYPE variable changes at run time without having to rewrite variable declarations. For example: if the ENAME column of an EMP table is changed from a VARCHAR2(10) to VRACHAR2(15) then you don’t need to modify the PL/SQL code.

PL/SQL Constants

A constant is a value used in a PL/SQL block that remains unchanged throughout the program. It is a user-defined literal value. It can be declared and used instead of actual values.

Let's take an example to explain it well:

Suppose, you have to write a program which will increase the salary of the employees upto 30%, you can declare a constant and use it throughout the program. Next time if you want to increase the salary again you can change the value of constant than the actual value throughout the program.

**Syntax to declare a constant:**

47.6M

947

1. constant\_name CONSTANT datatype := VALUE;

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

Example of PL/SQL constant

Let's take an example to explain it well:

1. **DECLARE**
2. -- constant declaration
3. pi constant number := 3.141592654;
4. -- other declarations
5. radius number(5,2);
6. dia number(5,2);
7. circumference number(7, 2);
8. area number (10, 2);
9. **BEGIN**
10. -- processing
11. radius := 9.5;
12. dia := radius \* 2;
13. circumference := 2.0 \* pi \* radius;
14. area := pi \* radius \* radius;
15. -- output
16. dbms\_output.put\_line('Radius: ' || radius);
17. dbms\_output.put\_line('Diameter: ' || dia);
18. dbms\_output.put\_line('Circumference: ' || circumference);
19. dbms\_output.put\_line('Area: ' || area);
20. **END**;
21. /

After the execution of the above code at SQL prompt, it will produce the following result:.

1. Radius: 9.5
2. Diameter: 19
3. Circumference: 59.69
4. Area: 283.53
6. Pl/SQL **procedure** successfully completed.

PL/SQL Literals

Literals are the explicit numeric, character, string or boolean values which are not represented by an identifier. For example: TRUE, NULL, etc. are all literals of type boolean. PL/SQL literals are case-sensitive. There are following kinds of literals in PL/SQL:

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

Example of these different types of Literals:

|  |  |
| --- | --- |
| **Literals** | **Examples** |
| Numeric | 75125, 3568, 33.3333333 etc. |
| Character | 'A' '%' '9' ' ' 'z' '(' |
| String | Hello JavaTpoint! |
| Boolean | TRUE, FALSE, NULL etc. |
| Date and Time | '26-11-2002' , '2012-10-29 12:01:01' |

# PL/SQL If

PL/SQL supports the programming language features like conditional statements and iterative statements. Its programming constructs are similar to how you use in programming languages like Java and C++.

**Syntax for IF Statement:**

There are different syntaxes for the IF-THEN-ELSE statement.

**Syntax: (IF-THEN statement):**

**00:00/04:20**

38.2M

713

Java Try Catch

1. IF condition
2. **THEN**
3. Statement: {It **is** executed **when** condition **is** **true**}
4. **END** IF;

This syntax is used when you want to execute statements only when condition is TRUE.

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

1. IF condition
2. **THEN**
3. {...statements **to** **execute** **when** condition **is** **TRUE**...}
4. **ELSE**
5. {...statements **to** **execute** **when** condition **is** **FALSE**...}
6. **END** IF;

This syntax is used when you want to execute one set of statements when condition is TRUE or a different set of statements when condition is FALSE.

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

1. IF condition1
2. **THEN**
3. {...statements **to** **execute** **when** condition1 **is** **TRUE**...}
4. ELSIF condition2
5. **THEN**
6. {...statements **to** **execute** **when** condition2 **is** **TRUE**...}
7. **END** IF;

This syntax is used when you want to execute one set of statements when condition1 is TRUE or a different set of statements when condition2 is TRUE.

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

1. IF condition1
2. **THEN**
3. {...statements **to** **execute** **when** condition1 **is** **TRUE**...}
4. ELSIF condition2
5. **THEN**
6. {...statements **to** **execute** **when** condition2 **is** **TRUE**...}
7. **ELSE**
8. {...statements **to** **execute** **when** both condition1 and condition2 are **FALSE**...}
9. **END** IF;

It is the most advance syntax and used if you want to execute one set of statements when condition1 is TRUE, a different set of statement when condition2 is TRUE or a different set of statements when both the condition1 and condition2 are FALSE.

#### When a condition is found to be TRUE, the IF-THEN-ELSE statement will execute the corresponding code and not check the conditions any further.

#### If there no condition is met, the ELSE portion of the IF-THEN-ELSE statement will be executed.

#### ELSIF and ELSE portions are optional.

## Example of PL/SQL If Statement

Let's take an example to see the whole concept:

1. **DECLARE**
2. a number(3) := 500;
3. **BEGIN**
4. -- check the boolean condition using if statement
5. IF( a < 20 ) **THEN**
6. -- if condition is true then print the following
7. dbms\_output.put\_line('a is less than 20 ' );
8. **ELSE**
9. dbms\_output.put\_line('a is not less than 20 ' );
10. **END** IF;
11. dbms\_output.put\_line('value of a is : ' || a);
12. **END**;

After the execution of the above code in SQL prompt, you will get the following result:

a is not less than 20

value of a is : 500

PL/SQL procedure successfully completed.

PL/SQL Case Statement

The PL/SQL CASE statement facilitates you to execute a sequence of satatements based on a selector. A selector can be anything such as variable, function or an expression that the CASE statement checks to a boolean value.

The CASE statement works like the IF statement, only using the keyword WHEN. A CASE statement is evaluated from top to bottom. If it get the condition TRUE, then the corresponding THEN calause is executed and the execution goes to the END CASE clause.

**Syntax for the CASE Statement:**

1. CASE [ expression ]
2. **WHEN** condition\_1 **THEN** result\_1
3. **WHEN** condition\_2 **THEN** result\_2
4. ...
5. **WHEN** condition\_n **THEN** result\_n
6. **ELSE** result
7. **END**

Example of PL/SQL case statement

Let's take an example to make it clear:

1. **DECLARE**
2. grade **char**(1) := 'A';
3. **BEGIN**
4. CASE grade
5. **when** 'A' **then** dbms\_output.put\_line('Excellent');
6. **when** 'B' **then** dbms\_output.put\_line('Very good');
7. **when** 'C' **then** dbms\_output.put\_line('Good');
8. **when** 'D' **then** dbms\_output.put\_line('Average');
9. **when** 'F' **then** dbms\_output.put\_line('Passed with Grace');
10. **else** dbms\_output.put\_line('Failed');
11. **END** CASE;
12. **END**;

After the execution of above code, you will get the following result:

Excellent

PL/SQL procedure successfully completed.

PL/SQL Loop

The PL/SQL loops are used to repeat the execution of one or more statements for specified number of times. These are also known as iterative control statements.

**Syntax for a basic loop:**

1. LOOP
2. **Sequence** **of** statements;
3. **END** LOOP;

Types of PL/SQL Loops

There are 4 types of PL/SQL Loops.

1. Basic Loop / Exit Loop
2. While Loop
3. For Loop
4. Cursor For Loop

PL/SQL Exit Loop (Basic Loop)

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

**Syntax of basic loop:**

1. LOOP
2. **Sequence** **of** statements;
3. **END** LOOP;

**Syntax of exit loop:**

1. LOOP
2. statements;
3. EXIT;
4. {or EXIT **WHEN** condition;}
5. **END** LOOP;

Example of PL/SQL EXIT Loop

Let's take a simple example to explain it well:



1. **DECLARE**
2. i NUMBER := 1;
3. **BEGIN**
4. LOOP
5. EXIT **WHEN** i>10;
6. DBMS\_OUTPUT.PUT\_LINE(i);
7. i := i+1;
8. **END** LOOP;
9. **END**;

After the execution of the above code, you will get the following result:

1

2

3

4

5

6

7

8

9

10

Note: You must follow these steps while using PL/SQL Exit Loop.

* Initialize a variable before the loop body
* Increment the variable in the loop.
* You should use EXIT WHEN statement to exit from the Loop. Otherwise the EXIT statement without WHEN condition, the statements in the Loop is executed only once.

PL/SQL EXIT Loop Example 2

1. **DECLARE**
2. VAR1 NUMBER;
3. VAR2 NUMBER;
4. **BEGIN**
5. VAR1:=100;
6. VAR2:=1;
7. LOOP
8. DBMS\_OUTPUT.PUT\_LINE (VAR1\*VAR2);
9. IF (VAR2=10) **THEN**
10. EXIT;
11. **END** IF;
12. VAR2:=VAR2+1;
13. **END** LOOP;
14. **END**;

Output:

100

200

300

400

500

600

700

800

900

1000

PL/SQL While Loop

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

**Syntax of while loop:**

1. WHILE <condition>
2. LOOP statements;
3. **END** LOOP;

Example of PL/SQL While Loop

Let's see a simple example of PL/SQL WHILE loop.

1. **DECLARE**
2. i **INTEGER** := 1;
3. **BEGIN**
4. WHILE i <= 10 LOOP
5. DBMS\_OUTPUT.PUT\_LINE(i);
6. i := i+1;
7. **END** LOOP;
8. **END**;

After the execution of the above code, you will get the following result:

1

2

3

4

5

6

7

8

9

10

Note: You must follow these steps while using PL/SQL WHILE Loop.

* Initialize a variable before the loop body.
* Increment the variable in the loop.
* You can use EXIT WHEN statements and EXIT statements in While loop but it is not done often.

PL/SQL WHILE Loop Example 2

1. **DECLARE**
2. VAR1 NUMBER;
3. VAR2 NUMBER;
4. **BEGIN**
5. VAR1:=200;
6. VAR2:=1;
7. WHILE (VAR2<=10)
8. LOOP
9. DBMS\_OUTPUT.PUT\_LINE (VAR1\*VAR2);
10. VAR2:=VAR2+1;
11. **END** LOOP;
12. **END**;

Output:

200

400

600

800

1000

1200

1400

1600

1800

2000

PL/SQL FOR Loop

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

**Syntax of for loop:**

1. **FOR** counter IN initial\_value .. final\_value LOOP
2. LOOP statements;
3. **END** LOOP;

* initial\_value : Start integer value
* final\_value : End integer value

PL/SQL For Loop Example 1

Let's see a simple example of PL/SQL FOR loop.

1. **BEGIN**
2. **FOR** k IN 1..10 LOOP
3. -- note that k was not declared
4. DBMS\_OUTPUT.PUT\_LINE(k);
5. **END** LOOP;
6. **END**;

After the execution of the above code, you will get the following result:

1

2

3

4

5

6

7

8

9

10

Note: You must follow these steps while using PL/SQL WHILE Loop.

* You don't need to declare the counter variable explicitly because it is declared implicitly in the declaration section.
* The counter variable is incremented by 1 and does not need to be incremented explicitly.
* You can use EXIT WHEN statements and EXIT statements in FOR Loops but it is not done often.

PL/SQL For Loop Example 2

1. **DECLARE**
2. VAR1 NUMBER;
3. **BEGIN**
4. VAR1:=10;
5. **FOR** VAR2 IN 1..10
6. LOOP
7. DBMS\_OUTPUT.PUT\_LINE (VAR1\*VAR2);
8. **END** LOOP;
9. **END**;

Output:

10

20

30

40

50

60

70

80

90

100

PL/SQL For Loop REVERSE Example 3

Let's see an example of PL/SQL for loop where we are using REVERSE keyword.

1. **DECLARE**
2. VAR1 NUMBER;
3. **BEGIN**
4. VAR1:=10;
5. **FOR** VAR2 IN REVERSE 1..10
6. LOOP
7. DBMS\_OUTPUT.PUT\_LINE (VAR1\*VAR2);
8. **END** LOOP;
9. **END**;

Output:

100

90

80

70

60

50

40

30

20

10

# PL/SQL Continue Statement

The continue statement is used to exit the loop from the reminder if its body either conditionally or unconditionally and forces the next iteration of the loop to take place, skipping any codes in between.

The continue statement is not a keyword in Oracle 10g. It is a new feature encorporated in oracle 11g.

For example: If a continue statement exits a cursor FOR LOOP prematurely then it exits an inner loop and transfer control to the next iteration of an outer loop, the cursor closes (in this context, CONTINUE works like GOTO).

**Syntax:**

1. **continue**;

## Example of PL/SQL continue statement

Let's take an example of PL/SQL continue statement.

1. **DECLARE**
2. x NUMBER := 0;
3. **BEGIN**
4. LOOP -- After CONTINUE statement, control resumes here
5. DBMS\_OUTPUT.PUT\_LINE ('Inside loop:  x = ' || TO\_CHAR(x));
6. x := x + 1;
7. IF x < 3 **THEN**
8. **CONTINUE**;
9. **END** IF;
10. DBMS\_OUTPUT.PUT\_LINE
11. ('Inside loop, after CONTINUE:  x = ' || TO\_CHAR(x));
12. EXIT **WHEN** x = 5;
13. **END** LOOP;
15. DBMS\_OUTPUT.PUT\_LINE (' After loop:  x = ' || TO\_CHAR(x));
16. **END**;
17. /

After the execution of above code, you will get the following result:

Inside loop: x = 0

Inside loop: x = 1

Inside loop: x = 2

Inside loop, after CONTINUE: x = 3

Inside loop: x = 3

Inside loop, after CONTINUE: x = 4

Inside loop: x = 4

Inside loop, after CONTINUE: x = 5

After loop: x = 5

#### Note: The continue statement is not supported in Oracle 10g. Oracle 11g supports this as a new feature.

PL/SQL GOTO Statement

In PL/SQL, GOTO statement makes you able to get an unconditional jump from the GOTO to a specific executable statement label in the same subprogram of the PL/SQL block.

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

**Syntax:**

1. **GOTO** label\_name;

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

1. **GOTO** label\_name;
2. ..
3. ..
4. <<label\_name>>
5. Statement;

Example of PL/SQL GOTO statement

Let's take an example of PL/SQL GOTO statement.

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

After the execution of above code, you will get the following result:

value of a: 30

value of a: 31

value of a: 32

value of a: 33

value of a: 34

value of a: 36

value of a: 37

value of a: 38

value of a: 39

value of a: 40

value of a: 41

value of a: 42

value of a: 43

value of a: 44

value of a: 45

value of a: 46

value of a: 47

value of a: 48

value of a: 49

Statement processed.

Restriction on GOTO statement

Following is a list of some restrictions imposed on GOTO statement.

* Cannot transfer control into an IF statement, CASE statement, LOOP statement or sub-block.
* Cannot transfer control from one IF statement clause to another or from one CASE statement WHEN clause to another.
* Cannot transfer control from an outer block into a sub-block.
* Cannot transfer control out of a subprogram.
* Cannot transfer control into an exception handler.

# PL/SQL Procedure

The PL/SQL stored procedure or simply a procedure is a PL/SQL block which performs one or more specific tasks. It is just like procedures in other programming languages.

The procedure contains a header and a body.

* **Header:** The header contains the name of the procedure and the parameters or variables passed to the procedure.
* **Body:** The body contains a declaration section, execution section and exception section similar to a general PL/SQL block.

## How to pass parameters in procedure:

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

1. **IN parameters:**The IN parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or the function.
2. **OUT parameters:**The OUT parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
3. **INOUT parameters:**The INOUT parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

#### A procedure may or may not return any value.

## PL/SQL Create Procedure

**Syntax for creating procedure:**

1. **CREATE** [OR REPLACE] **PROCEDURE** procedure\_name
2. [ (parameter [,parameter]) ]
3. **IS**
4. [declaration\_section]
5. **BEGIN**
6. executable\_section
7. [EXCEPTION
8. exception\_section]
9. **END** [procedure\_name];

## Create procedure example

In this example, we are going to insert record in user table. So you need to create user table first.

**Table creation:**

1. **create** **table** user(id number(10) **primary** **key**,**name** varchar2(100));

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

**Procedure Code:**

1. **create** or replace **procedure** "INSERTUSER"
2. (id IN NUMBER,
3. **name** IN VARCHAR2)
4. **is**
5. **begin**
6. **insert** **into** user **values**(id,**name**);
7. **end**;
8. /

Output:

Procedure created.

## PL/SQL program to call procedure

Let's see the code to call above created procedure.

1. **BEGIN**
2. insertuser(101,'Rahul');
3. dbms\_output.put\_line('record inserted successfully');
4. **END**;
5. /

Now, see the "USER" table, you will see one record is inserted.

|  |  |
| --- | --- |
| **ID** | **Name** |
| 101 | Rahul |

## PL/SQL Drop Procedure

**Syntax for drop procedure**

1. **DROP** **PROCEDURE** procedure\_name;

## Example of drop procedure

1. **DROP** **PROCEDURE** pro1;

# PL/SQL Function

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

**Syntax to create a function:**

1. **CREATE** [OR REPLACE] **FUNCTION** function\_name [parameters]
2. [(parameter\_name [IN | **OUT** | IN **OUT**] type [, ...])]
3. **RETURN** return\_datatype
4. {**IS** | **AS**}
5. **BEGIN**
6. < function\_body >
7. **END** [function\_name];

**Here:**

* **Function\_name:** specifies the name of the function.
* **[OR REPLACE]** option allows modifying an existing function.
* The **optional parameter list** contains name, mode and types of the parameters.
* **IN** represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.

### The function must contain a return statement.

* RETURN clause specifies that 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.

## PL/SQL Function Example

Let's see a simple example to **create a function**.

Competitive questions on Structures in HindiKeep Watching

1. **create** or replace **function** adder(n1 in number, n2 in number)
2. **return** number
3. **is**
4. n3 number(8);
5. **begin**
6. n3 :=n1+n2;
7. **return** n3;
8. **end**;
9. /

Now write another program to **call the function**.

1. **DECLARE**
2. n3 number(2);
3. **BEGIN**
4. n3 := adder(11,22);
5. dbms\_output.put\_line('Addition is: ' || n3);
6. **END**;
7. /

**Output:**

Addition is: 33

Statement processed.

0.05 seconds

## Another PL/SQL Function Example

Let's take an example to demonstrate Declaring, Defining and Invoking a simple PL/SQL function which will compute and return the maximum of two values.

1. **DECLARE**
2. a number;
3. b number;
4. c number;
5. **FUNCTION** findMax(x IN number, y IN number)
6. **RETURN** number
7. **IS**
8. z number;
9. **BEGIN**
10. IF x > y **THEN**
11. z:= x;
12. **ELSE**
13. Z:= y;
14. **END** IF;
16. **RETURN** z;
17. **END**;
18. **BEGIN**
19. a:= 23;
20. b:= 45;
22. c := findMax(a, b);
23. dbms\_output.put\_line(' Maximum of (23,45): ' || c);
24. **END**;
25. /

**Output:**

Maximum of (23,45): 45

Statement processed.

0.02 seconds

## PL/SQL function example using table

Let's take a customer table. This example illustrates creating and calling a standalone function. This function will return the total number of CUSTOMERS in the customers table.

#### Create customers table and have records in it.

|  |  |  |  |
| --- | --- | --- | --- |
| **Customers** | | | |
| **Id** | **Name** | **Department** | **Salary** |
| 1 | alex | web developer | 35000 |
| 2 | ricky | program developer | 45000 |
| 3 | mohan | web designer | 35000 |
| 4 | dilshad | database manager | 44000 |

**Create Function:**

1. **CREATE** OR REPLACE **FUNCTION** totalCustomers
2. **RETURN** number **IS**
3. total number(2) := 0;
4. **BEGIN**
5. **SELECT** count(\*) **into** total
6. **FROM** customers;
7. **RETURN** total;
8. **END**;
9. /

After the execution of above code, you will get the following result.

Function created.

**Calling PL/SQL Function:**

While creating a function, you have to 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. Once the function is called, the program control is transferred to the called function.

After the successful completion of the defined task, the call function returns program control back to the main program.

To call a function you have to pass the required parameters along with function name and if function returns a value then you can store returned value. Following program calls the function totalCustomers from an anonymous block:

1. **DECLARE**
2. c number(2);
3. **BEGIN**
4. c := totalCustomers();
5. dbms\_output.put\_line('Total no. of Customers: ' || c);
6. **END**;
7. /

After the execution of above code in SQL prompt, you will get the following result.

Total no. of Customers: 4

PL/SQL procedure successfully completed.

## PL/SQL Recursive Function

You already know that a program or a subprogram can call another subprogram. When a subprogram calls itself, it is called recursive call and the process is known as recursion.

## Example to calculate the factorial of a number

Let's take an example to calculate the factorial of a number. This example calculates the factorial of a given number by calling itself recursively.

1. **DECLARE**
2. num number;
3. factorial number;
5. **FUNCTION** fact(x number)
6. **RETURN** number
7. **IS**
8. f number;
9. **BEGIN**
10. IF x=0 **THEN**
11. f := 1;
12. **ELSE**
13. f := x \* fact(x-1);
14. **END** IF;
15. **RETURN** f;
16. **END**;
18. **BEGIN**
19. num:= 6;
20. factorial := fact(num);
21. dbms\_output.put\_line(' Factorial '|| num || ' is ' || factorial);
22. **END**;
23. /

After the execution of above code at SQL prompt, it produces the following result.

Factorial 6 is 720

PL/SQL procedure successfully completed.

## PL/SQL Drop Function

**Syntax for removing your created function:**

If you want to remove your created function from the database, you should use the following syntax.

1. **DROP** **FUNCTION** function\_name;

PL/SQL Cursor

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

A cursor is used to referred to a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

* Implicit Cursors
* Explicit Cursors

1) PL/SQL Implicit Cursors

The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.

These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.

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

**For example:**When you execute the SQL statements like INSERT, UPDATE, DELETE then the cursor attributes tell whether any rows are affected and how many have been affected. If you run a SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement. It will return an error if there no data is selected.

The following table soecifies the status of the cursor with each of its attribute.

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| %FOUND | Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE. |
| %NOTFOUND | Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND. |
| %ISOPEN | It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements. |
| %ROWCOUNT | It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement. |

PL/SQL Implicit Cursor Example

**Create customers table and have records:**

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

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

**Create procedure:**

1. **DECLARE**
2. total\_rows number(2);
3. **BEGIN**
4. **UPDATE**  customers
5. **SET** salary = salary + 5000;
6. IF sql%notfound **THEN**
7. dbms\_output.put\_line('no customers updated');
8. ELSIF sql%found **THEN**
9. total\_rows := sql%rowcount;
10. dbms\_output.put\_line( total\_rows || ' customers updated ');
11. **END** IF;
12. **END**;
13. /

Output:

6 customers updated

PL/SQL procedure successfully completed.

Now, if you check the records in customer table, you will find that the rows are updated.

1. **select** \* **from** customers;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **NAME** | **AGE** | **ADDRESS** | **SALARY** |
| 1 | Ramesh | 23 | Allahabad | 25000 |
| 2 | Suresh | 22 | Kanpur | 27000 |
| 3 | Mahesh | 24 | Ghaziabad | 29000 |
| 4 | Chandan | 25 | Noida | 31000 |
| 5 | Alex | 21 | Paris | 33000 |
| 6 | Sunita | 20 | Delhi | 35000 |

2) PL/SQL Explicit Cursors

The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors should be defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Following is the syntax to create an explicit cursor:

Syntax of explicit cursor

Following is the syntax to create an explicit cursor:

1. **CURSOR** cursor\_name **IS** select\_statement;;

Steps:

You must follow these steps while working with an explicit cursor.

1. Declare the cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

1) Declare the cursor:

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

**Syntax for explicit cursor decleration**

1. **CURSOR** **name** **IS**
2. **SELECT** statement;

2) Open the cursor:

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

**Syntax for cursor open:**

1. **OPEN** cursor\_name;

3) Fetch the cursor:

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

**Syntax for cursor fetch:**

1. **FETCH** cursor\_name **INTO** variable\_list;

4) Close the cursor:

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

**Syntax for cursor close:**

1. **Close** cursor\_name;

PL/SQL Explicit Cursor Example

Explicit cursors are defined by programmers to gain more control over the context area. It is defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns more than one row.

Let's take an example to demonstrate the use of explicit cursor. In this example, we are using the already created CUSTOMERS table.

**Create customers table and have records:**

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

**Create procedure:**

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

1. **DECLARE**
2. c\_id customers.id%type;
3. c\_name customers.**name**%type;
4. c\_addr customers.address%type;
5. **CURSOR** c\_customers **is**
6. **SELECT** id, **name**, address **FROM** customers;
7. **BEGIN**
8. **OPEN** c\_customers;
9. LOOP
10. **FETCH** c\_customers **into** c\_id, c\_name, c\_addr;
11. EXIT **WHEN** c\_customers%notfound;
12. dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);
13. **END** LOOP;
14. **CLOSE** c\_customers;
15. **END**;
16. /

Output:

1 Ramesh Allahabad

2 Suresh Kanpur

3 Mahesh Ghaziabad

4 Chandan Noida

5 Alex Paris

6 Sunita Delhi

PL/SQL procedure successfully completed.

# PL/SQL Exception Handling

## What is Exception

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

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

There are two type of exceptions:

* System-defined Exceptions
* User-defined Exceptions

## PL/SQL Exception Handling

**Syntax for exception handling:**

Difference between structure and union in HindiKeep Watching

Following is a general syntax for exception handling:

1. **DECLARE**
2. <declarations **section**>
3. **BEGIN**
4. <executable command(s)>
5. EXCEPTION
6. <exception handling goes here >
7. **WHEN** exception1 **THEN**
8. exception1-handling-statements
9. **WHEN** exception2  **THEN**
10. exception2-handling-statements
11. **WHEN** exception3 **THEN**
12. exception3-handling-statements
13. ........
14. **WHEN** others **THEN**
15. exception3-handling-statements
16. **END**;

## Example of exception handling

Let's take a simple example to demonstrate the concept of exception handling. Here we are using the already created CUSTOMERS table.

SELECT\* FROM COUSTOMERS;

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

1. **DECLARE**
2. c\_id customers.id%type := 8;
3. c\_name  customers.**name**%type;
4. c\_addr customers.address%type;
5. **BEGIN**
6. **SELECT**  **name**, address **INTO**  c\_name, c\_addr
7. **FROM** customers
8. **WHERE** id = c\_id;
9. DBMS\_OUTPUT.PUT\_LINE ('Name: '||  c\_name);
10. DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);
11. EXCEPTION
12. **WHEN** no\_data\_found **THEN**
13. dbms\_output.put\_line('No such customer!');
14. **WHEN** others **THEN**
15. dbms\_output.put\_line('Error!');
16. **END**;
17. /

After the execution of above code at SQL Prompt, it produces the following result:

No such customer!

PL/SQL procedure successfully completed.

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

#### Note: You get the result "No such customer" because the customer\_id used in the above example is 8 and there is no cutomer having id value 8 in that table.

If you use the id defined in the above table (i.e. 1 to 6), you will get a certain result. For a demo example: here, we are using the id 5.

1. **DECLARE**
2. c\_id customers.id%type := 5;
3. c\_name  customers.**name**%type;
4. c\_addr customers.address%type;
5. **BEGIN**
6. **SELECT**  **name**, address **INTO**  c\_name, c\_addr
7. **FROM** customers
8. **WHERE** id = c\_id;
9. DBMS\_OUTPUT.PUT\_LINE ('Name: '||  c\_name);
10. DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);
11. EXCEPTION
12. **WHEN** no\_data\_found **THEN**
13. dbms\_output.put\_line('No such customer!');
14. **WHEN** others **THEN**
15. dbms\_output.put\_line('Error!');
16. **END**;
17. /

After the execution of above code at SQL prompt, you will get the following result:

Name: alex

Address: paris

PL/SQL procedure successfully completed.

## Raising Exceptions

In the case of any internal database error, exceptions are raised by the database server automatically. But it can also be raised explicitly by programmer by using command RAISE.

**Syntax for raising an exception:**

1. **DECLARE**
2. exception\_name EXCEPTION;
3. **BEGIN**
4. IF condition **THEN**
5. RAISE exception\_name;
6. **END** IF;
7. EXCEPTION
8. **WHEN** exception\_name **THEN**
9. statement;
10. **END**;

## PL/SQL User-defined Exceptions

PL/SQL facilitates their users to define their own exceptions according to the need of the program. A user-defined exception can be raised explicitly, using either a RAISE statement or the procedure DBMS\_STANDARD.RAISE\_APPLICATION\_ERROR.

**Syntax for user define exceptions**

1. **DECLARE**
2. my-exception EXCEPTION;

## PL/SQL Pre-defined Exceptions

There are many pre-defined exception in PL/SQL which are executed when any database rule is violated by the programs.

PL/SQL Trigger

Trigger is invoked by Oracle engine automatically whenever a specified event occurs.Trigger is stored into database and invoked repeatedly, when specific condition match.

Triggers are stored programs, which are automatically executed or fired when some event occurs.

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

Advantages of Triggers

These are the following advantages of Triggers:

* Trigger generates some derived column values automatically
* Enforces referential integrity
* Event logging and storing information on table access
* Auditing
* Synchronous replication of tables
* Imposing security authorizations
* Preventing invalid transactions

Creating a trigger:

**Syntax for creating trigger:**

1. **CREATE** [OR REPLACE ] **TRIGGER** trigger\_name
2. {BEFORE | **AFTER** | **INSTEAD** **OF** }
3. {**INSERT** [OR] | **UPDATE** [OR] | **DELETE**}
4. [**OF** col\_name]
5. **ON** table\_name
6. [REFERENCING OLD **AS** o NEW **AS** n]
7. [**FOR** EACH ROW]
8. **WHEN** (condition)
9. **DECLARE**
10. Declaration-statements
11. **BEGIN**
12. Executable-statements
13. EXCEPTION
14. Exception-handling-statements
15. **END**;

**Here,**

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

PL/SQL Trigger Example

Let's take a simple example to demonstrate the trigger. In this example, we are using the following CUSTOMERS table:

**Create table and have records:**

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

**Create trigger:**

Let's take a program to create 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:

1. **CREATE** OR REPLACE **TRIGGER** display\_salary\_changes
2. BEFORE **DELETE** OR **INSERT** OR **UPDATE** **ON** customers
3. **FOR** EACH ROW
4. **WHEN** (NEW.ID > 0)
5. **DECLARE**
6. sal\_diff number;
7. **BEGIN**
8. sal\_diff := :NEW.salary  - :OLD.salary;
9. dbms\_output.put\_line('Old salary: ' || :OLD.salary);
10. dbms\_output.put\_line('New salary: ' || :NEW.salary);
11. dbms\_output.put\_line('Salary difference: ' || sal\_diff);
12. **END**;
13. /

After the execution of the above code at SQL Prompt, it produces the following result.

Trigger created.

**Check the salary difference by procedure:**

Use the following code to get the old salary, new salary and salary difference after the trigger created.

1. **DECLARE**
2. total\_rows number(2);
3. **BEGIN**
4. **UPDATE**  customers
5. **SET** salary = salary + 5000;
6. IF sql%notfound **THEN**
7. dbms\_output.put\_line('no customers updated');
8. ELSIF sql%found **THEN**
9. total\_rows := sql%rowcount;
10. dbms\_output.put\_line( total\_rows || ' customers updated ');
11. **END** IF;
12. **END**;
13. /

Output:

Old salary: 20000

New salary: 25000

Salary difference: 5000

Old salary: 22000

New salary: 27000

Salary difference: 5000

Old salary: 24000

New salary: 29000

Salary difference: 5000

Old salary: 26000

New salary: 31000

Salary difference: 5000

Old salary: 28000

New salary: 33000

Salary difference: 5000

Old salary: 30000

New salary: 35000

Salary difference: 5000

6 customers updated

**Note:** As many times you executed this code, the old and new both salary is incremented by 5000 and hence the salary difference is always 5000.

After the execution of above code again, you will get the following result.

Old salary: 25000

New salary: 30000

Salary difference: 5000

Old salary: 27000

New salary: 32000

Salary difference: 5000

Old salary: 29000

New salary: 34000

Salary difference: 5000

Old salary: 31000

New salary: 36000

Salary difference: 5000

Old salary: 33000

New salary: 38000

Salary difference: 5000

Old salary: 35000

New salary: 40000

Salary difference: 5000

6 customers updated

Important Points

Following are the two very important point and should be noted carefully.

* OLD and NEW references are used for record level triggers these are not avialable for table 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.