**What is PL/SQL?**

**PL/SQL** stands for **Procedural Language** extension of SQL.

PL/SQL is a combination of SQL along with the procedural features of programming languages.

It was developed by Oracle Corporation in the early 90’s to enhance the capabilities of SQL.

**The PL/SQL Engine:**

Oracle uses a **PL/SQL** engine to processes the PL/SQL statements. A PL/SQL language code can be stored in the client system (client-side) or in the database (server-side).

A Simple PL/SQL Block:

Each PL/SQL program consists of SQL and PL/SQL statements which from a PL/SQL block.

**PL/SQL Block consists of three sections:**

* The Declaration section (optional).
* The Execution section (mandatory).
* The [Exception Handling](http://plsql-tutorial.com/plsql-exception-handling.htm) (or Error) section (optional).

**Declaration Section**:

The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE. This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section. Placeholders may be any of Variables, Constants and Records, which stores data temporarily. Cursors are also declared in this section.   
  
**Execution Section:**

The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END. This is a mandatory section and is the section where the program logic is written to perform any task. The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.   
  
**Exception Section:**

The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION. This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully. If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.   
  
Every statement in the above three sections must end with a semicolon **;** . PL/SQL blocks can be nested within other PL/SQL blocks. Comments can be used to document code. 

**How a Sample PL/SQL Block Looks**

|  |
| --- |
| *DECLARE       Variable declaration BEGIN       Program Execution  EXCEPTION       Exception handling END;* |

**SQL Command Categories**

SQL commands are grouped into four major categories depending on their functionality. They are as follows:

**Data Definition Language (DDL)**

These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.

**Data Manipulation Language (DML)**

These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.

**Transaction Control Language (TCL)**

These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.

**Data Control Language (DCL)**

These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.

**Advantages of PL/SQL**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **These are the Advantages of PL/SQL**   * ***Block Structures*:** PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused. * ***Procedural Language Capability*:** PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops). * ***Better Performance*:** PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic. * ***Error Handling*:** PL/SQL handles errors or exceptions effectively during the execution of a PL/SQL program. Once an exception is caught, specific actions can be taken depending upon the type of the exception or it can be displayed to the user with a message.  |  |  | | --- | --- | | **PL/SQL Placeholders**  Placeholders are temporary storage area. PL/SQL Placeholders can be any of Variables, Constants and Records. Oracle defines placeholders to store data temporarily, which are used to manipulate data during the execution of a PL SQL block.  Define PL/SQL Placeholders  Depending on the kind of data you want to store, you can define placeholders with a name and a datatype. Few of the datatypes used to define placeholders are as given below.  Number (n,m) , Char (n) , Varchar2 (n) , Date , Long , Long raw, Raw, Blob, Clob, Nclob, Bfile  PL/SQL Variables  These are placeholders that store the values that can change through the PL/SQL Block.  **General Syntax to declare a variable is**  *variable\_name datatype [NOT NULL := value ];*   * *variable\_name*is the name of the variable. * *datatype*is a valid PL/SQL datatype. * NOT NULL is an optional specification on the variable. * *value*or DEFAULT *value*is also an optional specification, where you can initialize a variable. * Each variable declaration is a separate statement and must be terminated by a semicolon.   For example, if you want to store the current salary of an employee, you can use a variable.  *DECLARE*  *salary number (6);*  \* “salary” is a variable of datatype number and of length 6.  When a variable is specified as NOT NULL, you must initialize the variable when it is declared. |  |   For example: The below example declares two variables, one of which is a not null.  *DECLARE*  *salary number(4);*  *dept varchar2(10) NOT NULL := “HR Dept”;*  The value of a variable can change in the execution or exception section of the PL/SQL Block. We can assign values to variables in the two ways given below.  1) We can directly assign values to variables.      The General Syntax is:  *variable\_name:= value;*  2) We can assign values to variables directly from the database columns by using a SELECT.. INTO statement. The General Syntax is:   |  | | --- | | *SELECT column\_name*  *INTO variable\_name*  *FROM table\_name*  *[WHERE condition];* |   Example: The below program will get the salary of an employee with id '1116' and display it on the screen.  *DECLARE*  *var\_salary number(6);*  *var\_emp\_id number(6) = 1116;*  *BEGIN*  *SELECT salary*  *INTO var\_salary*  *FROM employee*  *WHERE emp\_id = var\_emp\_id;*  *dbms\_output.put\_line(var\_salary);*  *dbms\_output.put\_line('The employee '*  *|| var\_emp\_id || ' has salary ' || var\_salary);*  *END;*  */*  **NOTE: The backward slash '/' in the above program indicates to execute the above PL/SQL Block.**  **Scope of PS/SQL Variables**  PL/SQL allows the nesting of Blocks within Blocks i.e, the Execution section of an outer block can contain inner blocks. Therefore, a variable which is accessible to an outer Block is also accessible to all nested inner Blocks. The variables declared in the inner blocks are not accessible to outer blocks. Based on their declaration we can classify variables into two types.   * *Local* variables - These are declared in a inner block and cannot be referenced by outside Blocks. * *Global* variables - These are declared in a outer block and can be referenced by its itself and by its inner blocks.   For Example: In the below example we are creating two variables in the outer block and assigning thier product to the third variable created in the inner block. The variable 'var\_mult' is declared in the inner block, so cannot be accessed in the outer block i.e. it cannot be accessed after line 11. The variables 'var\_num1' and 'var\_num2' can be accessed anywhere in the block.   |  | | --- | | *1> DECLARE*  *2> var\_num1 number;*  *3>  var\_num2 number;*  *4> BEGIN*  *5>  var\_num1 := 100;*  *6>  var\_num2 := 200;*  *7>  DECLARE*  *8>   var\_mult number;*  *9>   BEGIN*  *10>    var\_mult := var\_num1 \* var\_num2;*  *11>   END;*  *12> END;*  *13> /* | |

**PL/SQL Constants**

|  |  |  |
| --- | --- | --- |
| As the name implies a *constant* is a value used in a PL/SQL Block that remains unchanged throughout the program. A constant is a user-defined literal value. You can declare a constant and use it instead of actual value.  For example: If you want to write a program which will increase the salary of the employees by 25%, you can declare a constant and use it throughout the program. Next time when you want to increase the salary again you can change the value of the constant which will be easier than changing the actual value throughout the program.  **General Syntax to declare a constant is:**   |  | | --- | | *constant\_name CONSTANT datatype := VALUE;* |  * *constant\_name*is the name of the constant i.e. similar to a variable name. * The word *CONSTANT*is a reserved word and ensures that the value does not change. * *VALUE*- It is a value which must be assigned to a constant when it is declared. You cannot assign a value later.   For example, to declare salary\_increase, you can write code as follows:  *DECLARE*  *salary\_increase CONSTANT number (3) := 10;*  You *must*assign a value to a constant at the time you declare it. If you do not assign a value to a constant while declaring it and try to assign a value in the execution section, you will get a error. If you execute the below Pl/SQL block you will get error.   |  | | --- | | *DECLARE*  *salary\_increase CONSTANT number(3);*  *BEGIN*  *salary\_increase := 100;*  *dbms\_output.put\_line (salary\_increase);*  *END;* | |

**PL/SQL Procedure**

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;

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.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.

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.

This chapter is going to cover important aspects of a **PL/SQL procedure**. We will discuss **PL/SQL function** in the next chapter.

Parts of a PL/SQL Subprogram

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

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

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.

Example

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

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

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

END;

/

When the above code is executed using the SQL prompt, it will produce the following result −

Procedure created.

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

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block −

BEGIN

greetings;

END;

/

The above call will display −

Hello World

PL/SQL procedure successfully completed.

Deleting a Standalone Procedure

A standalone procedure is deleted with the **DROP PROCEDURE** statement. Syntax for deleting a procedure is −

DROP PROCEDURE procedure-name;

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

DROP PROCEDURE greetings;

Parameter Modes in PL/SQL Subprograms

The following table lists out the 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 para meter 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.** |

IN & OUT Mode Example 1

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

DECLARE

a number;

b number;

c number;

PROCEDURE findMin(x IN number, y IN number, z OUT number) IS

BEGIN

IF x < 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;

/

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

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

IN & OUT Mode Example 2

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

DECLARE

a number;

PROCEDURE squareNum(x IN OUT number) IS

BEGIN

x := x \* x;

END;

BEGIN

a:= 23;

squareNum(a);

dbms\_output.put\_line(' Square of (23): ' || a);

END;

/

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

Square of (23): 529

PL/SQL procedure successfully completed.

|  |  |
| --- | --- |
| Stored Procedures  What is a Stored Procedure?  A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.  A procedure has a header and a body. The header consists of the name of the procedure and the parameters or variables passed to the procedure. The body consists or declaration section, execution section and exception section similar to a general PL/SQL Block.  A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.  **Procedures: Passing Parameters**  We can pass parameters to procedures in three ways. 1) IN-parameters 2) OUT-parameters 3) IN OUT-parameters  A procedure may or may not return any value. |  |

**General Syntax to create a procedure is:**

*CREATE [OR REPLACE] PROCEDURE proc\_name [list of parameters]*

*IS*

*Declaration section*

*BEGIN*

*Execution section*

*EXCEPTION*

*Exception section*

*END;*

**IS -** marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.

The syntax within the brackets [ ] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.

**Procedures: Example**

The below example creates a procedure ‘employer\_details’ which gives the details of the employee.

*1> CREATE OR REPLACE PROCEDURE employer\_details*

*2> IS*

*3> CURSOR emp\_cur IS*

*4> SELECT first\_name, last\_name, salary FROM emp\_tbl;*

*5> emp\_rec emp\_cur%rowtype;*

*6> BEGIN*

*7> FOR emp\_rec in sales\_cur*

*8> LOOP*

*9> dbms\_output.put\_line(emp\_cur.first\_name || ' ' ||emp\_cur.last\_name*

*10> || ' ' ||emp\_cur.salary);*

*11> END LOOP;*

*12>END;*

*13> /*

How to execute a Stored Procedure?

There are two ways to execute a procedure.

1) From the SQL prompt.

*EXECUTE [or EXEC] procedure\_name;*

2) Within another procedure – simply use the procedure name.

*procedure\_name;*

**NOTE:** In the examples given above, we are using backward slash ‘/’ at the end of the program. This indicates the oracle engine that the PL/SQL program has ended and it can begin processing the statements.

**Subprograms: Procedures and Functions in PL/SQL**

In this section, you are going to see the detailed description on how to create and execute the named blocks (procedures and functions).

Procedures and Functions are the subprograms which can be created and saved in the database as database objects. They can be called or referred inside the other blocks also.

Apart from this we will cover the major differences between these two subprograms. Also, we are going to discuss the oracle built-in functions.

**Terminologies in PL/SQL Subprograms**

Before we learn about PL/SQL subprograms, we will discuss the various terminologies that are the part of these subprograms. Below are the terminologies that we are going to discuss.

**Parameter:**

The parameter is variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code. This parameter allows to give input to the subprograms and to extract from these subprograms.

* These parameters should be defined along with the subprograms at the time of creation.
* These parameters are included in the calling statement of these subprograms to interact the values with the subprograms.
* The datatype of the parameter in the subprogram and in the calling statement should be same.
* The size of the datatype should not mention at the time of parameter declaration, as the size is dynamic for this type.

Based on their purpose parameters are classified as

**IN Parameter:**

* This parameter is used for giving input to the subprograms.
* It is a read-only variable inside the subprograms, their values cannot be changed inside the subprogram.
* In the calling statement these parameters can be a variable or a literal value or an expression, for example, it could be the arithmetic expression like '5\*8' or 'a/b' where 'a' and 'b' are variables.
* By default, the parameters are of IN type.

**OUT Parameter:**

* This parameter is used for getting output from the subprograms.
* It is a read-write variable inside the subprograms, their values can be changed inside the subprograms.
* In the calling statement, these parameters should always be a variable to hold the value from the current subprograms.

**IN OUT Parameter:**

* This parameter is used for both giving input and for getting output from the subprograms.
* It is a read-write variable inside the subprograms, their values can be changed inside the subprograms.
* In the calling statement, these parameters should always be a variable to hold the value from the subprograms.

These parameter type should be mentioned at the time of creating the subprograms.

**RETURN**

RETURN is the keyword that actually instructs the compiler to switch the control from the subprogram to the calling statement. In subprogram RETURN simply means that the control needs to exit from the subprogram. Once the controller finds RETURN keyword in the subprogram, the code after this will be skipped.

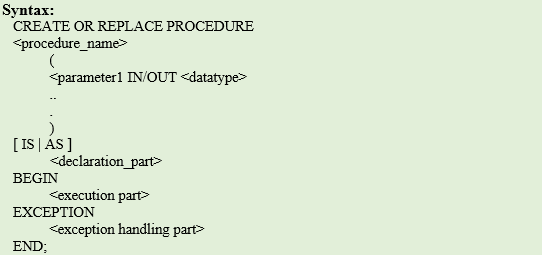
Normally, parent or main block will call the subprograms, and then the control will shift from those parent block to the called subprograms. RETURN in the subprogram will return the control back to their parent block. In the case of functions RETURN statement also returns the value. The datatype of this value is always mentioned at the time of function declaration. The datatype can be of any valid PL/SQL data type.

**Procedure**

Procedure is a subprogram unit that consists of a group of PL/SQL statements. Each procedure in Oracle has its own unique name by which it can be referred. This subprogram unit is stored as a database object. Below are the characteristics of this subprogram unit.

**Note:** Subprogram is nothing but a procedure, and it needs to be created manually as per the requirement. Once created they will be stored as database objects.

* Procedures are standalone blocks of a program that can be stored in the database.
* Call to these procedures can be made by referring to their name, to execute the PL/SQL statements.
* It is mainly used to execute a process in PL/SQL.
* It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
* It contains declaration part (optional), execution part, exception handling part (optional).
* The values can be passed into the procedure or fetched from the procedure through parameters.
* These parameters should be included in the calling statement.
* Procedure can have a RETURN statement to return the control to the calling block, but it cannot return any values through the RETURN statement.
* Procedures cannot be called directly from SELECT statements, they can be called from another block or through EXEC keyword.

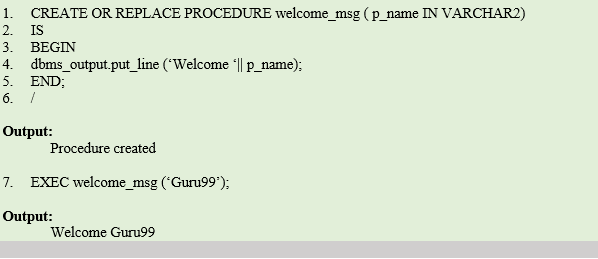
[](https://cdn.guru99.com/images/PL-SQL/110215_0958_Subprograms1.png)

**Syntax Explanation:**

* CREATE PROCEDURE instructs the compiler to create new procedure. Keyword 'OR REPLACE' instructs the compile to replace the existing procedure (if any) with the current one.
* Procedure name should be unique.
* Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning.

**Example1: Creating Procedure and calling it using EXEC**

In this example, we are going to create a procedure that takes the name as input and prints the welcome message as output. We are going to use EXEC command to call procedure.

[](https://cdn.guru99.com/images/PL-SQL/110215_0958_Subprograms2.png)

**Code Explanation:**

* **Code line 1**: Creating the procedure with name 'welcome\_msg' and with one parameter 'p\_name' of 'IN' type.
* **Code line 4**: Printing the welcome message by concatenating the input name.
* Procedure is compiled successfully.
* **Code line 7**: Calling the procedure using EXEC command with the parameter 'Guru99'. Procedure is executed, and the message is printed out as "Welcome Guru99".

PL/SQL procedures create using CREATE PROCEDURE statement. The major difference between PL/SQL function or procedure, function return always value where as procedure may or may not return value.

When you create a function or procedure, you have to define IN/OUT/INOUT parameters parameters.

1. IN : IN parameter referring to the procedure or function and allow to overwritten the value of parameter.
2. OUT : OUT parameter referring to the procedure or function and allow to overwritten the value of parameter.
3. IN OUT : Both IN OUT parameter referring to the procedure or function to pass both IN OUT parameter, modify/update by the function or procedure and also get returned.

IN/OUT/INOUT parameters you define in procedure argument list that get returned back to a result. When you create the procedure default IN parameter is passed in argument list. It's means value is passed but not returned. Explicitly you have define OUT/IN OUT parameter in argument list.

PL/SQL Procedure Syntax

CREATE [OR REPLACE] PROCEDURE [SCHEMA..] procedure\_name

[ (parameter [,parameter]) ]

IS

[declaration\_section

variable declarations;

constant declarations;

]

BEGIN

[executable\_section

PL/SQL execute/subprogram body

]

[EXCEPTION]

[exception\_section

PL/SQL Exception block

]

END [procedure\_name];

/

PL/SQL Procedure Example

In this example we are creating a procedure to pass employee number argument and get that employee information from table. We have emp1 table having employee information,

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_NO | EMP\_NAME | EMP\_DEPT | EMP\_SALARY |
| 1 | Forbs ross | Web Developer | 45k |
| 2 | marks jems | Program Developer | 38k |
| 3 | Saulin | Program Developer | 34k |
| 4 | Zenia Sroll | Web Developer | 42k |

Create PROCEDURE

In this example passing IN parameter (no) and inside procedure SELECT ... INTO statement to get the employee information.

*pro1.sql*

SQL>dit pro1

CREATE or REPLACE PROCEDURE pro1(no in number,temp out emp1%rowtype)

IS

BEGIN

SELECT \* INTO temp FROM emp1 WHERE eno = no;

END;

/

Execute PROCEDURE

After write the PL/SQL Procedure you need to execute the procedure.

**SQL>@pro1**  
Procedure created.  
  
PL/SQL procedure successfully completed.

PL/SQL Program to Calling Procedure

This program (pro) call the above define procedure with pass employee number and get that employee information.

*pro.sql*

SQL>edit pro

DECLARE

temp emp1%rowtype;

no number :=&no;

BEGIN

pro1(no,temp);

dbms\_output.put\_line(temp.eno||' '||

temp.ename||' '||

temp.edept||' '||

temp.esalary||' '||);

END;

/

PL/SQL Program Result

**SQL>@pro**  
no number &n=2  
2    marks jems    Program Developer    38K  
  
PL/SQL procedure successfully completed.

PL/SQL Drop Procedure

You can drop PL/SQL procedure using DROP PROCEDURE statement,

Functions Drop Syntax

DROP PROCEDURE procedure\_name;

Procedure Drop Example

**SQL>DROP PROCEDURE pro1;**  
  
Procedure dropped.

PL/SQL - Functions

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

## **Creating a Function**

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

CREATE [OR REPLACE] FUNCTION function\_name

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

RETURN return\_datatype

{IS | AS}

BEGIN

< function\_body >

END [function\_name];

Where,

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

### **Example**

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

We will use the CUSTOMERS table, which we had created in the [PL/SQL Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter −

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;

/

When the above code is executed using the SQL prompt, it will produce the following 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. Following program calls the function **totalCustomers** from an anonymous block −

DECLARE

c number(2);

BEGIN

c := totalCustomers();

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

END;

/

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

Total no. of Customers: 6

PL/SQL procedure successfully completed.

### **Example**

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

DECLARE

a number;

b number;

c number;

FUNCTION findMax(x IN number, y IN number)

RETURN number

IS

z number;

BEGIN

IF x > y THEN

z:= x;

ELSE

Z:= y;

END IF;

RETURN z;

END;

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

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

END;

/

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

Maximum of (23,45): 45

PL/SQL procedure successfully completed.

## **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;

/

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

Factorial 6 is 720

PL/SQL procedure successfully completed.

# 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**.

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 Functions****What is a Function in PL/SQL?** A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value. General Syntax to create a function is *CREATE [OR REPLACE] FUNCTION function\_name [parameters]*  *RETURN return\_datatype;*  *IS*  *Declaration\_section*  *BEGIN*  *Execution\_section*  *Return return\_variable;*  *EXCEPTION*  *exception section*  *Return return\_variable;*  *END;* |  |

1) **Return Type:** The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.  
2) The execution and exception section both should return a value which is of the datatype defined in the header section.

For example, let’s create a frunction called ''employer\_details\_func' similar to the one created in stored proc

*1> CREATE OR REPLACE FUNCTION employer\_details\_func*

*2> RETURN VARCHAR(20);*

*3> IS*

*5> emp\_name VARCHAR(20);*

*6> BEGIN*

*7> SELECT first\_name INTO emp\_name*

*8> FROM emp\_tbl WHERE empID = '100';*

*9> RETURN emp\_name;*

*10> END;*

*11> /*

In the example we are retrieving the ‘first\_name’ of employee with empID 100 to variable ‘emp\_name’.  
The return type of the function is VARCHAR which is declared in line no 2.   
The function returns the 'emp\_name' which is of type VARCHAR as the return value in line no 9.

### **How to execute a PL/SQL Function?**

A function can be executed in the following ways.

1) Since a function returns a value we can assign it to a variable.

*employee\_name := employer\_details\_func;*

If ‘employee\_name’ is of datatype varchar we can store the name of the employee by assigning the return type of the function to it.

2) As a part of a SELECT statement

*SELECT employer\_details\_func FROM dual;*

3) In a PL/SQL Statements like,

*dbms\_output.put\_line(employer\_details\_func);*

This line displays the value returned by the function.

### **PL/SQL Functions**

PL/SQL functions block create using CREATE FUNCTION statement. The major difference between PL/SQL function or procedure, function return always value where as procedure may or may not return value.

When you create a function or procedure, you have to define IN/OUT/INOUT parameters parameters.

1. IN : IN parameter referring to the procedure or function and allow to overwritten the value of parameter.
2. OUT : OUT parameter referring to the procedure or function and allow to overwritten the value of parameter.
3. IN OUT : Both IN OUT parameter referring to the procedure or function to pass both IN OUT parameter, modify/update by the function or procedure and also get returned.

IN/OUT/INOUT parameters you define in function argument list that get returned back to a result. When you create the function default IN parameter is passed in argument list. It's means value is passed but not returned. Explicitly you have define OUT/IN OUT parameter in argument list.

#### **PL/SQL Functions Syntax**

CREATE [OR REPLACE] FUNCTION [SCHEMA..] function\_name

[ (parameter [,parameter]) ]

RETURN return\_datatype

IS | AS

[declaration\_section

variable declarations;

constant declarations;

]

BEGIN

[executable\_section

PL/SQL execute/subprogram body

]

[EXCEPTION]

[exception\_section

PL/SQL Exception block

]

END [function\_name];

/

### **Function Example**

In this example we are creating a function to pass employee number and get that employee name from table. We have emp1 table having employee information,

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_NO | EMP\_NAME | EMP\_DEPT | EMP\_SALARY |
| 1 | Forbs ross | Web Developer | 45k |
| 2 | marks jems | Program Developer | 38k |
| 3 | Saulin | Program Developer | 34k |
| 4 | Zenia Sroll | Web Developer | 42k |

#### **Create Function**

So lets start passing IN parameter (no). Return datatype set varchar2. Now inside function SELECT ... INTO statement to get the employee name.

*fun1.sql*

SQL>edit fun1

CREATE or REPLACE FUNCTION fun1(no in number)

RETURN varchar2

IS

name varchar2(20);

BEGIN

select ename into name from emp1 where eno = no;

return name;

END;

/

#### **Execute Function**

After write the PL/SQL function you need to execute the function.

**SQL>@fun1**  
Function created.  
  
PL/SQL procedure successfully completed.

#### **PL/SQL Program to Calling Function**

This program call the above define function with pass employee number and get that employee name.

*fun.sql*

SQL>edit fun

DECLARE

no number :=&no;

name varchar2(20);

BEGIN

name := fun1(no);

dbms\_output.put\_line('Name:'||' '||name);

end;

/

#### **PL/SQL Program Result**

**SQL>@fun**  
no number &n=2  
Name: marks jems  
  
PL/SQL procedure successfully completed.

### **PL/SQL Drop Function**

You can drop PL/SQL function using DROP FUNCTION statements.

#### **Functions Drop Syntax**

DROP FUNCTION function\_name;

#### **Functions Drop Example**

**SQL>DROP FUNCTION fun1;**  
  
Function dropped.

**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**.

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.

In PL/SQL, you can refer to the most recent implicit cursor as the **SQL cursor**, which always has attributes such as **%FOUND, %ISOPEN, %NOTFOUND**, and **%ROWCOUNT**. The SQL cursor has additional attributes, **%BULK\_ROWCOUNT** and **%BULK\_EXCEPTIONS**, designed for use with the **FORALL** statement. The following table provides the description of the most used attributes −

|  |  |
| --- | --- |
| **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. |

Any SQL cursor attribute will be accessed as **sql%attribute\_name** as shown below in the example.

### **Example**

We will be using the CUSTOMERS table we had created and used in the previous chapters.

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;

/

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

6 customers selected

PL/SQL procedure successfully completed.

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 |

+----+----------+-----+-----------+----------+

## **What are Cursors?**

A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it.

This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.

There are two types of cursors in PL/SQL:

|  |  |
| --- | --- |
| **Implicit cursors** These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed. **Explicit cursors** They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.  Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed. |  |

## **Implicit Cursors: Application**

When you execute DML statements like DELETE, INSERT, UPDATE and SELECT statements, implicit statements are created to process these statements.

Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.

For example, When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected.   
When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.

The status of the cursor for each of these attributes are defined in the below table.

|  |  |  |
| --- | --- | --- |
| **Attributes** | **Return Value** | **Example** |
| %FOUND | The return value is TRUE, if the DML statements like INSERT, DELETE and UPDATE affect at least one row and if SELECT ….INTO statement return at least one row. | SQL%FOUND |
| The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE do not affect row and if SELECT….INTO statement do not return a row. |
| %NOTFOUND | The return value is FALSE, if DML statements like INSERT, DELETE and UPDATE at least one row and if SELECT ….INTO statement return at least one row. | SQL%NOTFOUND |
| The return value is TRUE, if a DML statement like INSERT, DELETE and UPDATE do not affect even one row and if SELECT ….INTO statement does not return a row. |
| %ROWCOUNT | Return the number of rows affected by the DML operations INSERT, DELETE, UPDATE, SELECT | SQL%ROWCOUNT |

For Example: Consider the PL/SQL Block that uses implicit cursor attributes as shown below:

*DECLARE var\_rows number(5);*

*BEGIN*

*UPDATE employee*

*SET salary = salary + 1000;*

*IF SQL%NOTFOUND THEN*

*dbms\_output.put\_line('None of the salaries where updated');*

*ELSIF SQL%FOUND THEN*

*var\_rows := SQL%ROWCOUNT;*

*dbms\_output.put\_line('Salaries for ' || var\_rows || 'employees are updated');*

*END IF;*

*END;*

In the above PL/SQL Block, the salaries of all the employees in the ‘employee’ table are updated. If none of the employee’s salary are updated we get a message 'None of the salaries where updated'. Else we get a message like for example, 'Salaries for 1000 employees are updated' if there are 1000 rows in ‘employee’ table.

### **PL/SQL Cursors (Implicit, Explicit, For Loop, Parameterized Cursor)**

#### **What is Cursor?**

Cursor is the work area which Oracle reserves for internal processing of SQL statements. This work area is private for oracles reserved are called cursor.

Cursor area also saying session cursor. because session cursor store information until the session end. Both way you can manage session cursor either implicit cursor or explicit cursor.

Using procedural statement you can get any information using session attribute.

#### **How to Use Cursor**

In PL/SQL block SELECT statement can not return more than one row at a time. So Cursor use to some group of rows (more than one row) for implementing certain logic to get all the group of records.

#### **Classification of CURSORS**

Cursors can be classified as:

1. Implicit Cursor or Internal Cursor : Manage for Oracle itself or internal process itself.
2. Explicit Cursor or User-defined Cursor : Manage for user/programmer or external processing.

### **PL/SQL Implicit Cursor**

* [*« Previous Learn*](http://www.way2tutorial.com/plsql/plsql_cursors.php)
* [*Next Learn »*](http://www.way2tutorial.com/plsql/plsql_explicit_cursor.php)

Oracle uses implicit cursors for its internal processing. Even if we execute a SELECT statement or DML statement Oracle reserves a private SQL area in memory called cursor.

Implicit cursor scope you can get information from cursor by using session attributes until another SELECT statement or DML statement execute.

#### **Implicit Cursor Attributes**

Following are implicit cursor attributes,

|  |  |  |
| --- | --- | --- |
| Cursor Attribute | Cursor Variable | Description |
| %ISOPEN | SQL%ISOPEN | Oracle engine automatically open the cursor If cursor open **return TRUE** otherwise **return FALSE.** |
| %FOUND | SQL%FOUND | If SELECT statement return one or more rows or DML statement (INSERT, UPDATE, DELETE) affect one or more rows If affect **return TRUE** otherwise **return FALSE.** If not execute SELECT or DML statement **return NULL.** |
| %NOTFOUND | SQL%NOTFOUND | If SELECT INTO statement return no rows and fire no\_data\_found PL/SQL exception before you can check SQL%NOTFOUND. If not affect the row **return TRUE** otherwise **return FALSE.** |
| %ROWCOUNT | SQL%ROWCOUNT | Return the number of rows affected by a SELECT statement or DML statement (insert, update, delete). If not execute SELECT or DML statement **return NULL.** |

#### **Syntax**

cursor\_attribute ::=

{

cursor\_name |

cursor\_variable\_name |

:host\_cursor\_variable\_name

}

% {FOUND | ISOPEN | NOTFOUND | ROWCOUNT}

Explanation :

cursor\_name : cursor\_name identifies the current scope which are previously declared.

cursor\_variable\_name : cursor variable or parameter identifies the current scope which are previously declared.

host\_cursor\_variable\_name : host\_cursor\_variable\_name must be prefixed with a colon. Host cursor variable datatype must be compatible with the PL/SQL cursor variable.

#### **Implicit Cursor Example**

Following one emp\_information table:

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_NO | EMP\_NAME | EMP\_DEPT | EMP\_SALARY |
| 1 | Forbs ross | Web Developer | 45k |
| 2 | marks jems | Program Developer | 38k |
| 3 | Saulin | Program Developer | 34k |
| 4 | Zenia Sroll | Web Developer | 42k |

Now above employee information table update the employee name 'Saulin' department 'Program Developer' update to 'Web Developer'.

#### **Example Code**

*implicit\_cursor.sql*

SQL>set serveroutput on

SQL>edit implicit\_cursor

BEGIN

UPDATE emp\_information SET emp\_dept='Web Developer'

WHERE emp\_name='Saulin';

IF SQL%FOUND THEN

dbms\_output.put\_line('Updated - If Found');

END IF;

IF SQL%NOTFOUND THEN

dbms\_output.put\_line('NOT Updated - If NOT Found');

END IF;

IF SQL%ROWCOUNT>0 THEN

dbms\_output.put\_line(SQL%ROWCOUNT||' Rows Updated');

ELSE

dbms\_output.put\_line('NO Rows Updated Found');

END;

/

#### **Example Result**

**SQL>@implicit\_cursor**  
Updated - If Found  
1 Rows Updated  
  
PL/SQL procedure successfully operation.

#### **Explicit Cursor**

Explicit Cursor which are construct/manage by user itself call explicit cursor.

User itself to declare the cursor, open cursor to reserve the memory and populate data, fetch the records from the active data set one at a time, apply logic and last close the cursor.

You can not directly assign value to an explicit cursor variable you have to use expression or create subprogram for assign value to explicit cursor variable.

Step for Using Explicit Cursor :

1. Declare cursor
2. Open cursor
3. Loop
4. Fetch data from cursor
5. Exit loop
6. Close cursor

### **PL/SQL Explicit Cursor**

* [*« Previous Learn*](http://www.way2tutorial.com/plsql/plsql_implicit_cursor.php)
* [*Next Learn »*](http://www.way2tutorial.com/plsql/plsql_cursor_for_loop.php)

Explicit Cursor which are construct/manage by user itself call explicit cursor.

User itself to declare the cursor, open cursor to reserve the memory and populate data, fetch the records from the active data set one at a time, apply logic and last close the cursor.

You can not directly assign value to an explicit cursor variable you have to use expression or create subprogram for assign value to explicit cursor variable.

#### **Step for Using Explicit Cursor**

1. Declare cursor

Declare explicit cursor has this syntax,

CURSOR cursor\_name [ parameter ] RETURN return\_type;

CURSOR cursor\_name [ parameter ] [ RETURN return\_type ]

IS SELECT STATEMENT;

Declaring explicit cursor example,

CURSOR c RETURN EMP\_DEPT%ROWTYPE; -- Declare c

CURSOR c IS -- Define c,

SELECT \* FROM emp\_information; -- all row return type

CURSOR c RETURN EMP\_DEPT%ROWTYPE IS -- Define c,

SELECT \* FROM emp\_information; -- repeating return type

1. Opening Explicit Cursor

DECLARE block you are already declare CURSOR now you can OPEN CURSOR by using following way, and allocate some reserve area for process database query.

OPEN cursor\_name [( cursor\_parameter )];

1. Loop

Loop iterate until ROW not found. Once found loop exit control goes next statement (outside loop).

1. Fetching data from cursor

Using FETCH statement you can fetch CURSOR data into explicit variable.

FETCH cursor\_name INTO variable;

1. Exit loop
2. Closing Explicit Cursor

This way you can close opened CURSOR.

CLOSE cursor\_name [( cursor\_parameter )];

Following emp\_information table having employee information, now we update information using Explicit Cursor,

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_NO | EMP\_NAME | EMP\_DEPT | EMP\_SALARY |
| 1 | Forbs ross | Web Developer | 45k |
| 2 | marks jems | Program Developer | 38k |
| 3 | Saulin | Program Developer | 34k |
| 4 | Zenia Sroll | Web Developer | 42k |

Now above employee information table update the employee name 'Saulin' department 'Program Developer' update to 'Web Developer'.

#### **Example Code**

*explicit\_cursor.sql*

SQL>set serveroutput on

SQL>edit explicit\_cursor

DECLARE

cursor c is select \* from emp\_information

where emp\_name='bhavesh';

tmp emp\_information%rowtype;

BEGIN

OPEN c;

Loop exit when c%NOTFOUND;

FETCH c into tmp;

update emp\_information set tmp.emp\_dept='Web Developer'

where tmp.emp\_name='Saulin';

END Loop;

IF c%ROWCOUNT>0 THEN

dbms\_output.put\_line(SQL%ROWCOUNT||' Rows Updated');

ELSE

dbms\_output.put\_line('NO Rows Updated Found');

END IF;

CLOSE c;

END;

/

#### **Example Result**

**SQL>@explicit\_cursor**  
1 Rows Updated  
  
PL/SQL procedure successfully completed.

## **Explicit Cursors**

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is −

CURSOR cursor\_name IS select\_statement;

Working with an explicit cursor includes the following steps −

* Declaring the cursor for initializing the memory
* Opening the cursor for allocating the memory
* Fetching the cursor for retrieving the data
* Closing the cursor to release the allocated memory

## **Declaring the Cursor**

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example −

CURSOR c\_customers IS

SELECT id, name, address FROM customers;

## **Opening the Cursor**

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows −

OPEN c\_customers;

## **Fetching the Cursor**

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows −

FETCH c\_customers INTO c\_id, c\_name, c\_addr;

## **Closing the Cursor**

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows −

CLOSE c\_customers;

### **Example**

Following is a complete example to illustrate the concepts of explicit cursors &minua;

DECLARE

c\_id customers.id%type;

c\_name customerS.No.ame%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;

/

When the above code is executed at the SQL prompt, it produces the following 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 Cursors For Loop**

PL/SQL cursor FOR loop has one great advantage of loop continued until row not found. In sometime you require to use explicit cursor with FOR loop instead of use OPEN, FETCH, and CLOSE statement.

FOR loop iterate repeatedly and fetches rows of values from database until row not found.

#### **Explicit Cursor FOR LOOP Example**

following one emp\_information table:

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_NO | EMP\_NAME | EMP\_DEPT | EMP\_SALARY |
| 1 | Forbs ross | Web Developer | 45k |
| 2 | marks jems | Program Developer | 38k |
| 3 | Saulin | Program Developer | 34k |
| 4 | Zenia Sroll | Web Developer | 42k |

Display employee number wise first two employee details emp,

#### **Example Code**

*cursor\_for\_loop.sql*

SQL>set serveroutput on

SQL>edit cursor\_for\_loop

DECLARE

cursor c is select \* from emp\_information

where emp\_no <=2;

tmp emp\_information%rowtype;

BEGIN

OPEN c;

FOR tmp IN c LOOP

FETCH c into tmp;

dbms\_output.put\_line('EMP\_No: '||tmp.emp\_no);

dbms\_output.put\_line('EMP\_Name: '||tmp.emp\_name);

dbms\_output.put\_line('EMP\_Dept: '||tmp.emp\_dept);

dbms\_output.put\_line('EMP\_Salary:'||tmp.emp\_salary);

END Loop;

CLOSE c;

END;

/

#### **Example Result**

**SQL>@cursor\_for\_loop**  
EMP\_No:    1  
EMP\_Name:  Forbs ross  
EMP\_Dept:  Web Developer  
EMP\_Salary:45k  
  
EMP\_No:    2  
EMP\_Name:  marks jems  
EMP\_Dept:  Program Developer  
EMP\_Salary:38k  
  
PL/SQL procedure successfully completed.

### **PL/SQL Parameterized Cursor**

* [*« Previous Learn*](http://www.way2tutorial.com/plsql/plsql_cursor_for_loop.php)
* [*Next Learn »*](http://www.way2tutorial.com/plsql/plsql_transaction.php)

PL/SQL Parameterized cursor pass the parameters into a cursor and use them in to query.

PL/SQL Parameterized cursor define only datatype of parameter and not need to define it's length.

Default values is assigned to the Cursor parameters. and scope of the parameters are locally.

Parameterized cursors are also saying static cursors that can passed parameter value when cursor are opened.

Following example introduce the parameterized cursor. following emp\_information table,

|  |  |  |  |
| --- | --- | --- | --- |
| EMP\_NO | EMP\_NAME | EMP\_DEPT | EMP\_SALARY |
| 1 | Forbs ross | Web Developer | 45k |
| 2 | marks jems | Program Developer | 38k |
| 3 | Saulin | Program Developer | 34k |
| 4 | Zenia Sroll | Web Developer | 42k |

#### **Example Code**

Cursor display employee information from emp\_information table whose emp\_no four (4).

*parameter\_cursor\_demo.sql*

SQL>set serveroutput on

SQL>edit parameter\_cursor\_demo

DECLARE

cursor c(no number) is select \* from emp\_information

where emp\_no = no;

tmp emp\_information%rowtype;

BEGIN

OPEN c(4);

FOR tmp IN c(4) LOOP

dbms\_output.put\_line('EMP\_No: '||tmp.emp\_no);

dbms\_output.put\_line('EMP\_Name: '||tmp.emp\_name);

dbms\_output.put\_line('EMP\_Dept: '||tmp.emp\_dept);

dbms\_output.put\_line('EMP\_Salary:'||tmp.emp\_salary);

END Loop;

CLOSE c;

END;

/

#### **Example Result**

**SQL>@parameter\_cursor\_demo**  
EMP\_No:    4  
EMP\_Name:  Zenia Sroll  
EMP\_Dept:  Web Developer  
EMP\_Salary:  42k  
  
PL/SQL procedure successfully completed.

Important key point you must remember

1. Scope of the parameters are locally
2. You can assign default value to a cursor parameter.

# 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.

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

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;

/

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

Customer ID: 5

Customer Name: Hardik

Customer Address: Bhopal

Customer Salary: 9000

PL/SQL procedure successfully completed.

## **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

PL/SQL procedure successfully completed.

## **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 record type is defined as −

TYPE

type\_name IS RECORD

( field\_name1 datatype1 [NOT NULL] [:= DEFAULT EXPRESSION],

field\_name2 datatype2 [NOT NULL] [:= DEFAULT EXPRESSION],

...

field\_nameN datatypeN [NOT NULL] [:= DEFAULT EXPRESSION);

record-name type\_name;

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.

### **Records as Subprogram Parameters**

You can pass a record as a subprogram parameter just as you pass any other variable. You can also access the record fields in the same way as you accessed in the above example −

DECLARE

type books is record

(title varchar(50),

author varchar(50),

subject varchar(100),

book\_id number);

book1 books;

book2 books;

PROCEDURE printbook (book books) IS

BEGIN

dbms\_output.put\_line ('Book title : ' || book.title);

dbms\_output.put\_line('Book author : ' || book.author);

dbms\_output.put\_line( 'Book subject : ' || book.subject);

dbms\_output.put\_line( 'Book book\_id : ' || book.book\_id);

END;

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;

-- Use procedure to print book info

printbook(book1);

printbook(book2);

END;

/

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

Book title : C Programming

Book author : Nuha Ali

Book subject : C Programming Tutorial

Book book\_id : 6495407

Book title : Telecom Billing

Book author : Zara Ali

Book subject : Telecom Billing Tutorial

Book book\_id : 6495700

PL/SQL procedure successfully completed.

## **PL/SQL Records**

|  |  |
| --- | --- |
| **What are records?** Records are another type of datatypes which oracle allows to be defined as a placeholder. Records are composite datatypes, which means it is a combination of different scalar datatypes like char, varchar, number etc.  Each scalar data types in the record holds a value. A record can be visualized as a row of data. It can contain all the contents of a row. **Declaring a record:** To declare a record, you must first define a composite datatype; then declare a record for that type. | |
| The General Syntax to define a composite datatype is:   |  | | --- | | TYPE record\_type\_name IS RECORD  (first\_col\_name column\_datatype,  second\_col\_name column\_datatype, ...); |  * record\_type\_name – it is the name of the composite type you want to define. * first\_col\_name, second\_col\_name, etc.,- it is the names the fields/columns within the record. * column\_datatype defines the scalar datatype of the fields.   There are different ways you can declare the datatype of the fields.  1) You can declare the field in the same way as you declare the fieds while creating the table.  2) If a field is based on a column from database table, you can define the field\_type as follows:   |  | | --- | | col\_name table\_name.column\_name%type; | |  |

By declaring the field datatype in the above method, the datatype of the column is dynamically applied to the field.  This method is useful when you are altering the column specification of the table, because you do not need to change the code again.

**NOTE:** You can use also %type to declare variables and constants.   
  
The General Syntax to declare a record of a uer-defined datatype is:

|  |
| --- |
| record\_name record\_type\_name; |

The following code shows how to declare a record called employee\_rec based on a user-defined type.

|  |
| --- |
| DECLARE  TYPE employee\_type IS RECORD  (employee\_id number(5),   employee\_first\_name varchar2(25),   employee\_last\_name employee.last\_name%type,   employee\_dept employee.dept%type);   employee\_salary employee.salary%type;  employee\_rec employee\_type; |

If all the fields of a record are based on the columns of a table, we can declare the record as follows:

|  |
| --- |
| record\_name table\_name%ROWTYPE; |

For example, the above declaration of employee\_rec can as follows:

|  |
| --- |
| DECLARE   employee\_rec employee%ROWTYPE; |

The advantages of declaring the record as a ROWTYPE are:  
1)  You do not need to explicitly declare variables for all the columns in a table.   
2) If you alter the column specification in the database table, you do not need to update the code.

The disadvantage of declaring the record as a ROWTYPE is:  
1) When u create a record as a ROWTYPE, fields will be created for all the columns in the table and memory will be used to create the datatype for all the fields. So use ROWTYPE only when you are using all the columns of the table in the program.

**NOTE:** When you are creating a record, you are just creating a datatype, similar to creating a variable. You need to assign values to the record to use them.  
  
The following table consolidates the different ways in which you can define and declare a pl/sql record.

|  |  |
| --- | --- |
| **Syntax** | **Usage** |
| TYPE record\_type\_name IS RECORD (column\_name1 datatype, column\_name2 datatype, ...); | Define a composite datatype, where each field is scalar. |
| col\_name table\_name.column\_name%type; | Dynamically define the datatype of a column based on a database column. |
| record\_name record\_type\_name; | Declare a record based on a user-defined type. |
| record\_name table\_name%ROWTYPE; | Dynamically declare a record based on an entire row of a table. Each column in the table corresponds to a field in the record. |

### Passing Values To and From a Record

When you assign values to a record, you actually assign values to the fields within it.   
The General Syntax to assign a value to a column within a record direclty is:

|  |
| --- |
| record\_name.col\_name := value; |

If you used %ROWTYPE to declare a record, you can assign values as shown:

|  |
| --- |
| record\_name.column\_name := value; |

We can assign values to records using SELECT Statements as shown:

|  |
| --- |
| SELECT col1, col2  INTO record\_name.col\_name1, record\_name.col\_name2  FROM table\_name  [WHERE clause]; |

If %ROWTYPE is used to declare a record then you can directly assign values to the whole record instead of each columns separately. In this case, you must SELECT all the columns from the table into the record as shown:

|  |
| --- |
| SELECT \* INTO record\_name  FROM table\_name  [WHERE clause]; |

Lets see how we can get values from a record.   
The General Syntax to retrieve a value from a specific field into another variable is:

|  |
| --- |
| var\_name := record\_name.col\_name; |

The following table consolidates the different ways you can assign values to and from a record:

|  |  |
| --- | --- |
| **Syntax** | **Usage** |
| record\_name.col\_name := value; | To directly assign a value to a specific column of a record. |
| record\_name.column\_name := value; | To directly assign a value to a specific column of a record, if the record is declared using %ROWTYPE. |
| SELECT col1, col2 INTO record\_name.col\_name1, record\_name.col\_name2 FROM table\_name [WHERE clause]; | To assign values to each field of a record from the database table. |
| SELECT \* INTO record\_name FROM table\_name [WHERE clause]; | To assign a value to all fields in the record from a database table. |
| variable\_name := record\_name.col\_name; | To get a value from a record column and assigning it to a variable. |

the **PL/SQL record**that is a composite data structure, which allows you to manage your data in program more efficiently.

## **What is a PL/SQL Record**

A PL/SQL record is a composite data structure that is a group of related data stored in fields. Each field in the PL/SQL record has its own name and data type.

## **Declaring a PL/SQL Record**

PL/SQL provides three ways to declare a record: table-based record, cursor-based record andprogrammer-defined records.

### **Declaring Table-based Record**

To declare a table-based record you use a table name with %ROWTYPE attribute. The fields of the PL/SQL record has the same name and data type as the column of the table.

The following illustrates table-based record declaration:



|  |  |
| --- | --- |
| 1  2 | DECLARE     table\_based\_record table\_name%ROWTYPE; |

After having the table-based record, you can use it in various ways, for example in SQL SELECT statement as follows:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | SET SERVEROUTPUT ON SIZE 1000000;  DECLARE    r\_emp employees%ROWTYPE;    n\_emp\_id  employees.employee\_id%TYPE := 200;  BEGIN    SELECT \*    INTO r\_emp    FROM employees    WHERE employee\_id = n\_emp\_id;    -- print out the employee's first name    DBMS\_OUTPUT.PUT\_LINE(r\_emp.first\_name);  END;  / |

In the above example:

* First, we defined a record based on employees table in HR sample database.
* Second, we used the SELECT statement to retrieve the employee information of the employee id 200 and populate the data into the r\_emp record .
* Third, we print out the first name of the selected employee from the r\_emp employee record.

### **Declaring Programmer-defined Record**

To declare programmer-defined record, first you have to define a record type by using TYPE statement with the fields of record explicitly. Then, you can declare a record based on record type that you’ve defined.

The following illustrates the syntax of the defining programmer-defined record with TYPE statement:



|  |  |
| --- | --- |
| 1  2  3  4  5  6 | TYPE type\_name IS RECORD     (field1 data\_type1 [NOT NULL] := [DEFAULT VALUE],      field2 data\_type2 [NOT NULL] := [DEFAULT VALUE],      ...      fieldn data\_type3 [NOT NULL] := [DEFAULT VALUE]      ); |

The data type of field can be any of the following:

* Scalar type ( VARCHAR2, NUMBER…).
* Anchor declaration %TYPE.
* %ROW type, in this case we have a nested record.
* SUBTYPE
* PL/SQL collection types.
* Cursor variable REF CURSOR.

Once you define the record type, you can declare a record based on the record type as follows:



|  |  |
| --- | --- |
| 1 | record\_name type\_name; |

The following example demonstrates how to declare programmer-defined record:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | SET SERVEROUTPUT ON SIZE 1000000;  DECLARE    TYPE t\_name IS RECORD(       first\_name employees.first\_name%TYPE,       last\_name  employees.last\_name%TYPE    );    r\_name   t\_name; -- name record    n\_emp\_id employees.employee\_id%TYPE := 200;  BEGIN    SELECT first\_name,           last\_name    INTO r\_name    FROM employees    WHERE employee\_id = n\_emp\_id;    -- print out the employee's name    DBMS\_OUTPUT.PUT\_LINE(r\_name.first\_name || ',' || r\_name.last\_name );  END;  / |

### **Declaring Cursor-based Record**

You can define a record based on a [cursor](http://www.plsqltutorial.com/plsql-cursor/). First, you must define a cursor. And then you use %ROWTYPE with the cursor variable to declare a record. The fields of the record correspond to the columns in the cursor SELECT statement.

The following is an example of declaring a record based on a cursor.



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | SET SERVEROUTPUT ON SIZE 1000000;  DECLARE    CURSOR cur\_emp IS      SELECT \*      FROM employees      WHERE employee\_id = 200;      emp\_rec cur\_emp%ROWTYPE;  BEGIN    NULL;  END;  / |

## **Working with PL/SQL Record**

After having a PL/SQL record, you can work with a record as a whole or you can work with individual field of the record.

### **Working with PL/SQL record at record level**

At record level, you can do the following:

* You can assign a PL/SQL record to another PL/SQL record. The pair of PL/SQL records must have the same number of fields and the data type of each field has to be convertible.
* You can assign a PL/SQL record NULL value by assigning an uninitialized record.
* A PL/SQL record can be used as an argument of parameter in a [function](http://www.plsqltutorial.com/plsql-function/)
* You can return a PL/SQL record from a [function](http://www.plsqltutorial.com/plsql-function/)
* To check if the record is NULL, you have to check each individual field of the record.
* To compare two records, you have to compare each individual field of each record.

Here is an example of working with PL/SQL record at record level:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | SET serveroutput ON SIZE 1000000;  DECLARE    TYPE t\_name IS RECORD(      first\_name employees.first\_name%TYPE,      last\_name employees.last\_name%TYPE    );    r\_name      t\_name;    r\_name2     t\_name;    r\_name\_null t\_name;    n\_emp\_id employees.employee\_id%TYPE := 200;  BEGIN    -- assign employee's infomation to record    SELECT first\_name,           last\_name    INTO r\_name    FROM employees    WHERE employee\_id = n\_emp\_id;      -- assign record to another record    r\_name2 := r\_name;    -- print out the employee's name    DBMS\_OUTPUT.PUT\_LINE(r\_name2.first\_name || ',' || r\_name2.last\_name);      -- assign record to NULL    r\_name2 := r\_name\_null;      -- check NULL for each individual field    IF r\_name2.first\_name IS NULL AND       r\_name2.last\_name IS NULL THEN      DBMS\_OUTPUT.PUT\_LINE('Record r\_name2 is NULL');    END IF;    END;  / |

### **Working with PL/SQL record at field level**

As you see in the above example, we can reference to a field of a record by using dot notation (.) as follows:



|  |  |
| --- | --- |
| 1 | record\_name.field |

If you reference to a record variable in different package or schema you need to explicitly specify those information as shown below:



|  |  |
| --- | --- |
| 1 | [schema\_name.][package\_name.]record\_name.field |

You can use the assignment operator ( :=) to change the value of field of a record that you reference to.

For the nested record you need to use extra dot notation ( .)

The following example demonstrates how to use PL/SQL record a field level:



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | DECLARE    TYPE t\_address IS RECORD(      house\_number VARCHAR2(6),      street       VARCHAR2(50),      phone        VARCHAR2(15),      region       VARCHAR2(10),      postal\_code  VARCHAR2(10),      country      VARCHAR2(25)    );      TYPE t\_contact IS RECORD(      home     t\_address,      business t\_address    );    r\_contact t\_contact;  BEGIN    r\_contact.business.house\_number := '500';    r\_contact.business.street       := 'Oracle Parkway';    r\_contact.business.region       := 'CA';    r\_contact.business.postal\_code  := '94065';    r\_contact.business.country      := 'USA';    r\_contact.business.phone        := '+1.800.223.1711';  END; |

In this tutorial, you’ve learned how to use PL/SQL record to manipulate data more efficiently, and to make your code cleaner and easier to maintain.

# PL/SQL - Packages

the Packages in PL/SQL. Packages are schema objects that groups logically related PL/SQL types, variables, and subprograms.

A package will have two mandatory parts −

* Package specification
* Package body or definition

## **Package Specification**

The specification is the interface to the package. It just **DECLARES** the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. In other words, it contains all information about the content of the package, but excludes the code for the subprograms.

All objects placed in the specification are called **public** objects. Any subprogram not in the package specification but coded in the package body is called a **private** object.

The following code snippet shows a package specification having a single procedure. You can have many global variables defined and multiple procedures or functions inside a package.

CREATE PACKAGE cust\_sal AS

PROCEDURE find\_sal(c\_id customers.id%type);

END cust\_sal;

/

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

Package created.

## **Package Body**

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from the code outside the package.

The **CREATE PACKAGE BODY** Statement is used for creating the package body. The following code snippet shows the package body declaration for the ***cust\_sal*** package created above. I assumed that we already have CUSTOMERS table created in our database as mentioned in the [PL/SQL - Variables](https://www.tutorialspoint.com/plsql/plsql_variable_types.htm) chapter.

CREATE OR REPLACE PACKAGE BODY cust\_sal AS

PROCEDURE find\_sal(c\_id customers.id%TYPE) IS

c\_sal customers.salary%TYPE;

BEGIN

SELECT salary INTO c\_sal

FROM customers

WHERE id = c\_id;

dbms\_output.put\_line('Salary: '|| c\_sal);

END find\_sal;

END cust\_sal;

/

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

Package body created.

## **Using the Package Elements**

The package elements (variables, procedures or functions) are accessed with the following syntax −

package\_name.element\_name;

Consider, we already have created the above package in our database schema, the following program uses the ***find\_sal*** method of the ***cust\_sal***package −

DECLARE

code customers.id%type := &cc\_id;

BEGIN

cust\_sal.find\_sal(code);

END;

/

When the above code is executed at the SQL prompt, it prompts to enter the customer ID and when you enter an ID, it displays the corresponding salary as follows −

Enter value for cc\_id: 1

Salary: 3000

PL/SQL procedure successfully completed.

### **Example**

The following program provides a more complete package. We will use the CUSTOMERS table stored in our database with the following records −

Select \* from customers;

+----+----------+-----+-----------+----------+

| ID | NAME | AGE | ADDRESS | SALARY |

+----+----------+-----+-----------+----------+

| 1 | Ramesh | 32 | Ahmedabad | 3000.00 |

| 2 | Khilan | 25 | Delhi | 3000.00 |

| 3 | kaushik | 23 | Kota | 3000.00 |

| 4 | Chaitali | 25 | Mumbai | 7500.00 |

| 5 | Hardik | 27 | Bhopal | 9500.00 |

| 6 | Komal | 22 | MP | 5500.00 |

+----+----------+-----+-----------+----------+

### **The Package Specification**

CREATE OR REPLACE PACKAGE c\_package AS

-- Adds a customer

PROCEDURE addCustomer(c\_id customers.id%type,

c\_name customerS.No.ame%type,

c\_age customers.age%type,

c\_addr customers.address%type,

c\_sal customers.salary%type);

-- Removes a customer

PROCEDURE delCustomer(c\_id customers.id%TYPE);

--Lists all customers

PROCEDURE listCustomer;

END c\_package;

/

When the above code is executed at the SQL prompt, it creates the above package and displays the following result −

Package created.

### **Creating the Package Body**

CREATE OR REPLACE PACKAGE BODY c\_package AS

PROCEDURE addCustomer(c\_id customers.id%type,

c\_name customerS.No.ame%type,

c\_age customers.age%type,

c\_addr customers.address%type,

c\_sal customers.salary%type)

IS

BEGIN

INSERT INTO customers (id,name,age,address,salary)

VALUES(c\_id, c\_name, c\_age, c\_addr, c\_sal);

END addCustomer;

PROCEDURE delCustomer(c\_id customers.id%type) IS

BEGIN

DELETE FROM customers

WHERE id = c\_id;

END delCustomer;

PROCEDURE listCustomer IS

CURSOR c\_customers is

SELECT name FROM customers;

TYPE c\_list is TABLE OF customerS.No.ame%type;

name\_list c\_list := c\_list();

counter integer :=0;

BEGIN

FOR n IN c\_customers LOOP

counter := counter +1;

name\_list.extend;

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

dbms\_output.put\_line('Customer(' ||counter|| ')'||name\_list(counter));

END LOOP;

END listCustomer;

END c\_package;

/

The above example makes use of the **nested table**. We will discuss the concept of nested table in the next chapter.

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

Package body created.

### **Using The Package**

The following program uses the methods declared and defined in the package *c\_package*.

DECLARE

code customers.id%type:= 8;

BEGIN

c\_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);

c\_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);

c\_package.listcustomer;

c\_package.delcustomer(code);

c\_package.listcustomer;

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

Customer(7): Rajnish

Customer(8): Subham

Customer(1): Ramesh

Customer(2): Khilan

Customer(3): kaushik

Customer(4): Chaitali

Customer(5): Hardik

Customer(6): Komal

Customer(7): Rajnish

PL/SQL procedure successfully completed

PL/SQL package is nothing but a logical grouping of a related subprogram (procedure/function) into a single element. Package is compiled and stored as a database objects that can be used later.

In this tutorial, you will learn-

* [Components of Packages](https://www.guru99.com/packages-pl-sql.html#1)
* [Package Specification](https://www.guru99.com/packages-pl-sql.html#2)
* [Package Body](https://www.guru99.com/packages-pl-sql.html#3)
* [Referring Package Elements](https://www.guru99.com/packages-pl-sql.html#4)
* [Package Initialization](https://www.guru99.com/packages-pl-sql.html#5)
* [Forward Declarations](https://www.guru99.com/packages-pl-sql.html#6)
* [Cursors Usage in Package](https://www.guru99.com/packages-pl-sql.html#7)
* [Overloading](https://www.guru99.com/packages-pl-sql.html#8)
* [Dependency in Packages](https://www.guru99.com/packages-pl-sql.html#9)
* [Package Information](https://www.guru99.com/packages-pl-sql.html#10)
* [UTL FILE – An Overview](https://www.guru99.com/packages-pl-sql.html#11)

## Components of Packages

PL/SQL package basically have two components.

* Package Specification
* Package Body

## Package Specification

Package specification consists of a declaration of all the public variables, cursors, objects, procedures, functions, and exception.

Below are few characteristics of Package specification

* The elements which are all declared in the specification can be accessed from outside of the package. Such element are known as public element.
* The package specification is a standalone element that means it can exist alone without package body.
* Whenever a package is referred an instance of the package is created for that particular session.
* After the instance is created for a session, all the package elements that are initiated in that instance are valid till the end of the session.

**Syntax Explanation:**

* The above syntax shows the creation of package specification.

## Package Body

It consists of definition of all the elements that are present in the package specification. It can also have definition of elements that are not declared in the specification, these elements are called private elements and can be called only from inside the package.

Below are characteristics of package body.

* It should contain definitions for all the subprograms/cursors that have been declared in the specification.
* It can also have more subprograms or other elements that are not declared in specification. These are called private elements.
* It is a dependable object, and it depends on package specification.
* The state of the package body becomes 'Invalid' every time when the specification is compiled. Therefore, it needs to be recompiled each time after the compilation of specification.
* The private elements should be defined first before they are used in the package body.
* The first part of the package is the global declaration part. This includes variables, cursors and private elements (forward declaration) that is visible to the entire package.
* The last part of the package is Package initialization part that execute one time whenever a package is referred first time in the session.

**Syntax Explanation:**

* The above syntax shows the creation of package body.

Now we are going to see how to refer package elements in the program.

## Referring Package Elements

Once the elements are declared and defined in the package, we need to refer the elements in order to use them.

All the public elements of the package can be referred by calling the package name followed by the element name separated by period i.e. '<package\_name>.<element\_name>'.

The public variable of the package can also be used in the same way to assign and fetch values from them i.e. '<package\_name>.<variable\_name>'.

## Package Initialization

In PL/SQL whenever a package is referred/called in a session a new instance will be created for that package.

Oracle provides a facility to initialize package elements or to perform any activity at the time of this instance creation through 'Package Initialization'.

This is nothing but an execution block that is written in the package body after defining all the package elements. This block will be executed whenever a package is referred for the first time in the session.

**Syntax Explanation:**

* The above syntax shows the definition of package initialization in the package body.

## Forward Declarations

Forward declaration/reference in the package is nothing but declaring the private elements separately and defining it in the later part of the package body.

Private elements can be referred only if it is already declared in the package body. For this reason forward declaration is used. But it is rather unusual to use because in most of the time private elements are declared and defined in the first part of the package body.

Forward declaration is an option provided by oracle, it is not mandatory and using and not using is up to programmer's requirement.

**Syntax Explanation:**

* The above syntax shows forward declaration. The private elements are declared separately in the forward part of the package, and they have been defined in the later part.

## Cursors Usage in Package

Unlike other elements one needs to be careful in using cursors inside the package.

If the cursor is defined in the package specification or in global part of package body, then the cursor once opened will persist till the end of the session.

So one should always use the cursor attributes '%ISOPEN' to verify the state of the cursor before referring it.

## Overloading

Overloading is the concept of having many subprograms with the same name. These subprograms will be differing from each other by number of parameters or types of parameters or return type i.e. subprogram with same name but with different number of parameters, different type of parameters or different retype are considered as overloading.

This is useful when many subprogram needs to do the same task, but the way of calling each of them should be different. In this case, the subprogram name will be kept same for all and the parameters will be changed as per calling statement.

**Example 1**: In this example, we are going to create a package to get and set the values of employee's information in 'emp' table. The get\_record function will return the record type output for the given employee number, and set\_record procedure will insert the record type record into the emp table.

S**tep 1)** Package Specification Creation

**Code Explanation**

* **Code line 1-5**: Creating the package specification for guru99\_get\_set with one procedure and one function. These two are now public elements of this package.

**Step 2)**Package contains Package body, where all procedures and functions actual definition will be defined. In this step, Package Body is created.

**Code Explanation**

* **Code line 7**: Creating the package body.
* **Code line 9-16**: Defining the element 'set\_record' that is declared in the specification. This is same as defining the standalone procedure in PL/SQL.
* **Code line 17-24:**Defining the element 'get\_record'. It is same as defining the standalone function.
* **Code line 25-26:**Defining the package initialization part.

**Step 3)**Creating an anonymous block to insert and display the records by referring to the above created package.

**Code Explanation:**

* **Code line 34-37:**Populating the data for record type variable in an anonymous block to call 'set\_record' element of the package.
* **Code line 38:**Call has been made to 'set\_record' of guru99\_get\_set package. Now the package is instantiated and it will persists till the end of the session.
* The package initialization part is executed since this is the first call to the package.
* The record in inserted by the 'set\_record' element into the table.
* **Code line 41:**Calling the 'get\_record' element in order to display the details of the inserted employee.
* The package is referred for the second time during the 'get\_record' call to the package. But the initialization part is not executed this time as the package is already initialized in this session.
* **Code line 42-45:** Printing the employee details.

## Dependency in Packages

Since the package is the logical grouping of related things it has some dependencies. Following are the dependency that are to be taken care.

* Specification is a standalone object.
* Package body is dependent on specification.
* Package body can be compiled separately. Whenever specification is compiled, the body needs to be recompiled as it will become invalid.
* The subprogram in package body that are dependent on a private element should be defined only after the private element declaration.
* The database objects that are referred in the specification and body needs to be in valid status at the time of package compilation.

## Package Information

Once the package information is created, the package information such as package source, subprogram details and overload details are available in the oracle data definition tables.

Below table gives the data definition table and the package information that is available in the table.

|  |  |  |
| --- | --- | --- |
| **Table Name** | **Description** | **Query** |
| ALL\_OBJECT | Gives the details of the package like object\_id, creation\_date, last\_ddl\_time, etc. It will contain the objects created by all users. | SELECT \* FROM all\_objects where object\_name ='<package\_name>' |
| USER\_OBJECT | Gives the details of the package like object\_id, creation\_date, last\_ddl\_time, etc. It will contain the objects created by the current user. | SELECT \* FROM user\_objects where object\_name ='<package\_name>' |
| ALL\_SOURCE | Gives the source of the objects created by all users. | SELECT \* FROM all\_source where name='<package\_name>' |
| USER\_SOURCE | Gives the source of the objects created by the current user. | SELECT \* FROM user\_source where name='<package\_name>' |
| ALL\_PROCEDURES | Gives the subprogram details like object\_id, overload details, etc created by all users. | SELECT \* FROM all\_procedures Where object\_name='<package\_name>' |
| USER\_PROCEDURES | Gives the subprogram details like object\_id, overload details, etc. created by the current user. | SELECT \* FROM user\_procedures Where object\_name='<package\_name>' |

## UTL FILE – An Overview

UTL File is the separate utility package provided by oracle to perform special tasks. This is mainly used for read and write the operating system files from PL/SQL packages or from subprograms. It got the separate functions to put the information and to get the information from files. It also allow to read/write in native character set.

Programmer can use this to write operating system files of any type and the file will be written directly in the database server. The name and directory path will be mentioned at the time writing.

## Summary

We have now learnt the packages in PL/SQL, and you should be now able to work in the following.

* PL/SQL packages and its Components
* Characteristics of packages
* Referring and overloading package elements
* Managing dependencies in packages
* Viewing package information
* What is UTL File