What is PL/SQL:

PL/SQL stands for Procedural Language extension of SQL. It was developed by Oracle Corporation in the late 1980s to enhance the capabilities of SQL. It is the procedural extension language for SQL.

## Advantages of PL/SQL:

1. PL/SQL is a procedural language.  
   2. PL/SQL is a block structure language.  
   3. PL/SQL handles the exceptions.  
   4. PL/SQL engine can process the multiple SQL statements simultaneously as a single block hence reduce network traffic and provides better performance.

PL/SQL block structure:

|  |
| --- |
| DECLARE  Declaration statements;  BEGIN  Execution statements;  EXCEPTION  Exception handling statements;  END;  / |

PL/SQL Block sections:

1. Declaration section (optional).  
2. Execution section (mandatory).  
3. Exception handling section (optional).

Declaration section:

It is an optional section and starts with DECLARE keyword. It is used to declare the variables, constants, records and cursors etc.

Execution section:

Execution section starts with BEGIN keyword and ends with END keyword. It is a mandatory section. It is used to write the program logic code.  
Note: Execution section must have one statement.

Exception handling section:

Execution section starts with EXCEPTION keyword. It is an optional section. It is used to handle the exceptions occurred in execution section.

Important points:

1. Every PL/SQL statement will be followed by semicolon (;).
2. 2. PL/SQL blocks can be nested.

Variable:

A variable is a meaningful name which facilitates a programmer to store data temporarily during the execution of code. It helps you to manipulate data in PL/SQL programs.

A variable should not exceed 30 characters.

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

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

## How to declare variable in PL/SQL

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

Following is the syntax for declaring variable:

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

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

**Example:**

Radius Number := 5;

Date\_of\_birth date;

Declaration Restrictions:

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

Forward references are not allowed i.e. you must declare a constant or variable before referencing it in another statement even if it is a declarative statement.

val number := Total - 200;

Total number := 1000;

The first declaration is illegal because the TOTAL variable must be declared before using it in an assignment expression.

Variables belonging to the same datatype cannot be declared in the same statement.

N1, N2, N3 Number;

It is an illegal declaration.

Naming rules for PL/SQL variables

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

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

## Initializing Variables in PL/SQL

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

* The DEFAULT keyword
* The assignment operator

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

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

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

Example of initilizing variable

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

**DECLARE**

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

After the execution, this will produce the following result:

Value of c: 70

Value of f: 33.333333333333333333

PL/SQL procedure successfully completed.

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

Example of Local and Global variables

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

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

Variable Scope in PL/SQL:

As we discussed in earlier tutorial that PL/SQL allows the nesting of blocks i.e. blocks with blocks. Based on the nesting structure PL/SQL variables can be divide into following categories:  
**Local variables** – Those variables which are declared in an inner block and not accessible to outer blocks are known as local variables.  
**Global variables** – Those variables which are declared in the outer block or a package and accessible to itself and inner blocks are known as global variables.

Example:

|  |
| --- |
| DECLARE  -- Global variables  num1 number := 10;  num2 number := 20;  BEGIN  dbms\_output.put\_line('Outer Variable num1: ' || num1);  dbms\_output.put\_line('Outer Variable num2: ' || num2);  DECLARE  -- Local variables  num3 number := 30;  num4 number := 40;  BEGIN  dbms\_output.put\_line('Outer variable in inner block num1: ' || num1);  dbms\_output.put\_line('Outer variable in inner block num2: ' || num2);  dbms\_output.put\_line('Inner Variable num3: ' || num3);  dbms\_output.put\_line('Inner Variable num4: ' || num4);  END;  END;  / |

Output:

|  |
| --- |
| Outer Variable num1: 10  Outer Variable num2: 20  Outer variable in inner block num1: 10  Outer variable in inner block num2: 20  Inner Variable num3: 30  Inner Variable num4: 40 |

## PL/SQL Constants:

A constant holds a value used in a PL/SQL block that does not change throughout the program. It is a user-defined literal value.

Let's take an example to explain it well:

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

## Syntax to declare a constant:

|  |
| --- |
| constant\_name CONSTANT datatype := VALUE; |

## Where:

**constant\_name** – is a valid identifier name.  
**CONSTANT** – is a keyword.  
**VALUE** – is a value which must be assigned to a constant when it is declared. You cannot assign a value later.

## Example:

* NUMBER(8,2)------ 8 digits in total
* 2 of which are after the decimal point

|  |
| --- |
| DECLARE  -- constant declaration  pi constant number := 3.141592654;  -- other declarations  --ex:555.22  radius number(5,2);  dia number(5,2);  circumference number(7, 2);  area number (10, 2);  BEGIN  -- processing  radius := 10.5;  dia := radius \* 2;  circumference := 2.0 \* pi \* radius;  area := pi \* radius \* radius;  -- output  dbms\_output.put\_line('Radius: ' || radius);  dbms\_output.put\_line('Diameter: ' || dia);  dbms\_output.put\_line('Circumference: ' || circumference);  dbms\_output.put\_line('Area: ' || area);  END;  / |

## Output:

|  |
| --- |
| Radius: 10.5  Diameter: 21  Circumference: 65.97  Area: 346.36 |

## PL/SQL Literals:

Literals is an explicit numeric, character, string or Boolean values which are not represented by identifiers i.e. TRUE, NULL, w3spoint etc.  
**Note: PL/SQL literals are case-sensitive.**

## Types of literals in PL/SQL:

1. Numeric Literals (765, 23.56 etc.).  
2. Character Literals (‘A’ ‘%’ ‘9’ ‘ ‘ ‘z’ etc.).  
3. String Literals (tutorialspointexamples.com etc.).  
4. BOOLEAN Literals (TRUE, FALSE and NULL).  
5. Date and Time Literals (‘2016-12-25’ ‘2016-02-03 12:10:01’ etc.).

Hello worldprogram in plsql:

DECLARE

-- variable declaration

message varchar2(20):= 'Hello World!';

BEGIN

--output

dbms\_output.put\_line(message);

END;

/

## PL/SQL If statement:

If statement is used to execute a block of statements if specified condition is true.  
**Commonly used PL/SQL If statement:**

## IF-THEN statement:

***Syntax:***

|  |
| --- |
| **IF** condition  THEN  *//Block of statements1*  END **IF**; |

Block of statements1 executes when the specified condition is true.

## IF-THEN-ELSE statement:

***Syntax:***

|  |
| --- |
| **IF** condition  THEN  *//Block of statements1*  **ELSE**  *//Block of statements2*  END **IF**; |

Block of statements1 executes when the specified condition is true otherwise Block of statements2 executes.

## IF-THEN-ELSIF statement:

***Syntax:***

|  |
| --- |
| **IF** condition1  THEN  *//Block of statements1*  ELSIF condition2  *//Block of statements2*  **ELSE**  *//Block of statements3*  END **IF**; |

Block of statements1 executes when condition1 is true if false codition2 is checked and Block of statements2 executes if condition2 is true and so on. Block of statements in ELSE block executes when no condition is true.

## Example:

|  |
| --- |
| DECLARE  var number(3) := 50;  BEGIN  **IF** (var = 10) THEN  dbms\_output.put\_line('Value of var is 10');  ELSIF (var = 20) THEN  dbms\_output.put\_line('Value of var is 20');  ELSIF (var = 30) THEN  dbms\_output.put\_line('Value of var is 30');  **ELSE**  dbms\_output.put\_line('None of the above condition is true.');  END **IF**;  dbms\_output.put\_line('Exact value of var is: '|| var);  END;  / |

***Output:***

|  |
| --- |
| None of the above condition is **true**.  Exact value of var is: 50 |

## PL/SQL Case Statement:

Switch statement is used to execute a block of statement based on the switch expression value. An expression must be of type int, short, byte or char. A case value should be a constant literal value and cannot be duplicated. Expression value is compared with each case value. If a match found corresponding block of statements will be executed. A break statement is used to terminate the execution of statements. If no case value matches with expression value then default block of statements will be executed. If break statement is not used within case, all matching cases will be executed.  
***Syntax:***

|  |
| --- |
| **CASE** [expression]  WHEN condition1 THEN Block of statements1  WHEN condition2 THEN Block of statements2  ...  WHEN conditionn THEN Block of statementsn  **ELSE** Block of statements  END |

## Example:

|  |
| --- |
| DECLARE  nameChar **char**(1) := 'J';  BEGIN  **CASE** nameChar  when 'B' then dbms\_output.put\_line('Bharat');  when 'R' then dbms\_output.put\_line('Richi');  when 'S' then dbms\_output.put\_line('Sahdev');  when 'V' then dbms\_output.put\_line('Vinod');  when 'H' then dbms\_output.put\_line('Harish');  when 'M' then dbms\_output.put\_line('Mahesh');  when 'V' then dbms\_output.put\_line('Vivek');  when 'A' then dbms\_output.put\_line('Anil');  when 'J' then dbms\_output.put\_line('Jai');  **else** dbms\_output.put\_line('No such name');  END **CASE**;  END;  / |

## Output:

## Pl sql exit loop:

The pl sql loop repeatedly executes a block of statements until it reaches a loop exit. The EXIT and EXIT WHEN statements are used to terminate a loop.

## Where:

**EXIT:** The EXIT statement is used to terminate the loop unconditionally and normally used with IF statement.  
**EXIT WHEN:** The EXIT WHEN statement is used to terminate the loop conditionally. It terminates the loop when the specified condition is true.

## PL/SQL LOOP statement syntax with EXIT:

|  |
| --- |
| LOOP  *//block of statements*  EXIT;  END LOOP; |

## PL/SQL LOOP statement syntax with EXIT WHEN:

|  |
| --- |
| LOOP  *//block of statements*  EXIT WHEN condition;  END LOOP; |

## PL/SQL LOOP statement example with EXIT:

|  |
| --- |
| DECLARE  num NUMBER := 1;  BEGIN  LOOP  DBMS\_OUTPUT.PUT\_LINE(num);  **IF** num = 10 THEN  EXIT;  END **IF**;  num := num+1;  END LOOP;  END; |

## Output:

|  |
| --- |
| 1  2  3  4  5  6  7  8  9  10 |

## PL/SQL LOOP statement example with EXIT WHEN:

|  |
| --- |
| DECLARE  num NUMBER := 1;  BEGIN  LOOP  DBMS\_OUTPUT.PUT\_LINE(num);  EXIT WHEN num = 10;  num := num+1;  END LOOP;  END; |

## Output:

|  |
| --- |
| 1  2  3  4  5  6  7  8  9  10 |

## PL SQL WHILE LOOP:

The pl sql while loop repeatedly executes a block of statements until a particular condition is true. It first check the condition and executes a block of statements if condition is true.

## PL SQL WHILE LOOP syntax:

|  |
| --- |
| **WHILE** condition  LOOP  *//block of statements;*  END LOOP; |

## PL SQL WHILE LOOP example:

|  |
| --- |
| DECLARE  num NUMBER := 1;  BEGIN  **WHILE** num <= 10  LOOP  DBMS\_OUTPUT.PUT\_LINE(num);  num := num+1;  END LOOP;  END; |

## Output:

|  |
| --- |
| 1  2  3  4  5  6  7  8  9  10 |

## Pl sql for in loop:

The pl sql for in loop repeatedly executes a block of statements for a fixed number of times. The loop iteration occurs between the start and end integer values. The counter is always incremented by 1 and loop terminates when the counter reaches the value of the end integer.

## Pl sql for in loop syntax:

|  |
| --- |
| **FOR** loop\_counter IN [REVERSE] start\_value .. end\_value  LOOP  *//block of statements.*  END LOOP; |

**Note:**  
1. The double dot (..) specifies the range operator.  
2. By default iteration is from start\_value to end\_value but we can reverse the iteration process by using REVERSE keyword.  
3. No need to declare the counter variable explicitly because it is declared implicitly in the declaration section.  
4. The counter variable is incremented by 1 and does not need to be incremented explicitly.  
5. The EXIT and EXIT WHEN statements can be used.

## Pl sql for in loop example:

|  |
| --- |
| DECLARE  BEGIN  **FOR** var IN 1..10  LOOP  DBMS\_OUTPUT.PUT\_LINE(var);  END LOOP;  END; |

## Output:

|  |
| --- |
| 1  2  3  4  5  6  7  8  9  10 |

## Pl sql for in loop reverse example:

|  |
| --- |
| DECLARE  BEGIN  **FOR** var IN REVERSE 1..10  LOOP  DBMS\_OUTPUT.PUT\_LINE(var);  END LOOP;  END;    Output:  10  9  8  7  6  5  4  3  2  1 |

## Pl sql continue statement:

The pl sql continue statement is a control statement which is used to skip the following statement in the body of the loop and continue with the next iteration of the loop.

## Pl sql continue syntax:

|  |
| --- |
| **continue**; |

## Pl sql continue statement example:

|  |
| --- |
| DECLARE  num NUMBER := 0;  BEGIN  **WHILE** num < 10  LOOP  num := num +1;  **IF** num = 5 THEN  **CONTINUE**;  END **IF**;  DBMS\_OUTPUT.PUT\_LINE(num);  END LOOP;  END; |

## Output:

|  |
| --- |
| 1  2  3  4  6  7  8  9  10 |

## Pl sql goto statement:

The pl sql goto statement provides an unconditional jump from the GOTO to a labeled statement in the same subprogram. A label can be declare with the << label >> syntax.

## pl sql goto statement syntax:

|  |
| --- |
| **GOTO** label\_name;  *//Other statements*  <<label\_name>>  Statement; |

## Pl sql goto statement example:

|  |
| --- |
| DECLARE  num number := 1;  BEGIN  <<loop1>>  -- **while** loop execution  **WHILE** num <= 10 LOOP  dbms\_output.put\_line ('Value of num: ' || num);  num := num + 1;  **IF** num = 5 THEN  num := num + 1;  **GOTO** loop1;  END **IF**;  END LOOP;  END;  / |

## Output:

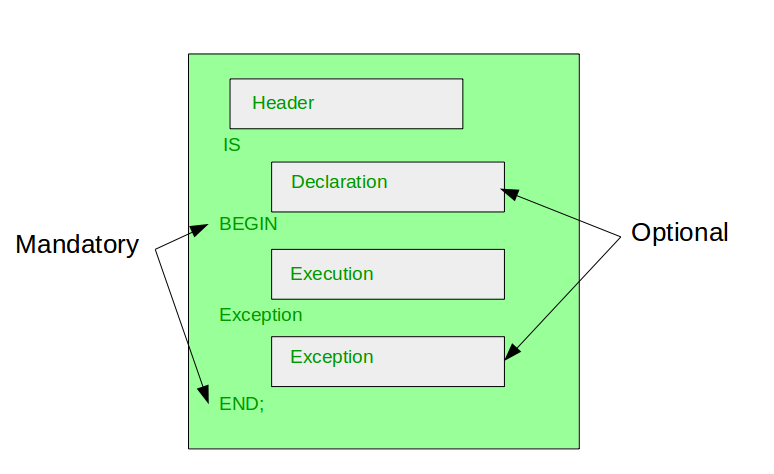
|  |
| --- |
| Value of num: 1  Value of num: 2  Value of num: 3  Value of num: 4  Value of num: 6  Value of num: 7  Value of num: 8  Value of num: 9  Value of num: 10 |

## Pl sql goto statement restrictions:

1. GOTO statement cannot transfer control into an IF statement, CASE statement, LOOP statement or sub-block.  
2. GOTO statement cannot transfer control from one IF statement clause to another or from one CASE statement WHEN clause to another.  
3. GOTO statement cannot transfer control from an outer block into a sub-block.  
4. GOTO statement cannot transfer control out of a subprogram.  
5. GOTO statement cannot transfer control into an exception handler.

# Blocks in PL/SQL:

In PL/SQL, All statements are classified into units that is called Blocks. PL/SQL blocks can include variables, SQL statements, loops, constants, conditional statements and exception handling. Blocks can also build a function or a procedure or a package.



Broadly, PL/SQL blocks are two types: Anonymous blocks and

* 1. **Anonymous blocks:** In PL/SQL, That’s blocks which is not have header are known as anonymous blocks. These blocks do not form the body of a function or triggers or procedure.

1. Example:

DECLARE

-- declare variable a, b and c

-- and these three variables datatype are integer

a number;

b number;

c number;

BEGIN

a:= 10;

b:= 100;

--find largest number

--take it in c variable

IF a > b THEN

c:= a;

ELSE

c:= b;

END IF;

dbms\_output.put\_line(' Maximum number in 10 and 100: ' || c);

END;

/

-- Program End

**2. Named blocks:** That’s PL/SQL blocks which having header or labels are known as Named blocks. These blocks can either be subprograms like functions, procedures, packages or Triggers.

Example: Here a code example of find greatest number with Named blocks means using function.

DECLARE

a number;

b number;

c number;

--Function return largest number of

-- two given number

BEGIN

a:= 10;

b:= 100;

c := findMax(a, b);

dbms\_output.put\_line(' Maximum number in 10 and 100 is: ' || c);

END;

/

Function:

create or replace 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;

/

# Oracle PL/SQL - %TYPE Attribute:

The %TYPE attribute can declare a data type as a previously declared variable or column without knowing what that type is.

The syntax of the declaration is:

your\_variable referenced\_item%TYPE;

The %TYPE attribute is useful when declaring variables to hold values from a table row.

The syntax for declaring a variable of the same type as a column is:

variable\_name **table**\_name.column\_name%TYPE;

SQL> **CREATE** **TABLE** emp(

2 empid **NUMBER**(6),

3 first\_name VARCHAR2(20),

4 last\_name VARCHAR2(25)) ;

SQL>

SQL> **DECLARE**

2 surname emp.last\_name%TYPE;

3 **BEGIN**

4 DBMS\_OUTPUT.PUT\_LINE(**'surname='** || surname);

5 **END**;

6 /

# Oracle / PLSQL: SELECT Statement

This Oracle tutorial explains how to use the Oracle **SELECT statement** with syntax, examples, and practice exercises.

## Description

The Oracle SELECT statement is used to retrieve records from one or more tables in an Oracle database.

## Syntax

The syntax for the SELECT statement in Oracle/PLSQL is:

SELECT expressions

FROM tables

[WHERE conditions];

### Parameters or Arguments

**expressions**

The columns or calculations that you wish to retrieve. Use \* if you wish to select all columns.

**tables**

The tables that you wish to retrieve records from. There must be at least one table listed in the FROM clause.

**WHERE conditions**

Optional. The conditions that must be met for the records to be selected. If no conditions are provided, then all records will be selected.

## Example - Select all fields from one table

Let's look at how to use an Oracle SELECT query to select all fields from a table.

SELECT \*

FROM homes

WHERE bathrooms >= 2

ORDER BY home\_type ASC;

In this Oracle SELECT statement example, we've used \* to signify that we wish to select all fields from the homes table where the number of bathrooms is greater than or equal to 2. The result set is sorted by home\_type in ascending order.

## Example - Select individual fields from one table

You can also use the Oracle SELECT statement to select individual fields from the table, as opposed to all fields from the table.

For example:

SELECT home\_id, home\_type, bathrooms

FROM homes

WHERE home\_id < 500

AND home\_type = 'two-storey'

ORDER BY home\_type ASC, bathrooms DESC;

This Oracle SELECT example would return only the home\_id, home\_type, and bathrooms fields from the homes table where the home\_id is less than 500 and the home\_type is 'two-storey'. The results are sorted by home\_type in ascending order and then bathrooms in descending order.

## Example - Select fields from multiple tables

You can also use the Oracle SELECT statement to retrieve fields from multiple tables by using a join.

SELECT homes.home\_id, customers.customer\_name

FROM customers

INNER JOIN homes

ON customers.customer\_id = homes.customer\_id

ORDER BY home\_id;

This Oracle SELECT example [joins two tables together](https://www.techonthenet.com/oracle/joins.php) to gives us a result set that displays the home\_id and customer\_name fields where the customer\_id value matches in both the customers and homes table. The results are sorted by home\_id in ascending order.

## Practice Exercise #1:

Based on the contacts table below, select all fields from the contacts table whose last\_name is 'Smith', contact\_id is greater than or equal 1000 and contact\_id is less than or equal to 2000 (no sorting is required):

CREATE TABLE contacts

( contact\_id number(10) not null,

last\_name varchar2(50) not null,

first\_name varchar2(50) not null,

address varchar2(50),

city varchar2(50),

state varchar2(2),

zip\_code varchar2(10),

CONSTRAINT contacts\_pk PRIMARY KEY (contact\_id)

);

### Solution for Practice Exercise #1:

The following Oracle SELECT statement would select these records from the employees table:

SELECT \*

FROM contacts

WHERE last\_name = 'Smith'

AND contact\_id >= 1000

AND contact\_id <= 2000;

Or you could write the solution using the [BETWEEN clause](https://www.techonthenet.com/oracle/between.php) as follows:

SELECT \*

FROM contacts

WHERE last\_name = 'Smith'

AND contact\_id BETWEEN 1000 AND 2000;

PLSQL Select statement:

PL/SQL select into statement is used for taking the values of a single row belonging to a single record into a single variable. It becomes easier to handle the data when you have a complete row data present inside the variable in PL/ SQL program. we will study the syntax of the PL/ SQL select into statement, most common exceptions that occur while using this statement, and will also learn how we can implement the select into a statement to retrieve a single column or multiple column values in a variable with the help of certain examples.

**Syntax:**

The PL/ SQL is the easiest, fastest and simplest way of fetching the data of a single row into a variable. The syntax of PL/ SQL SELECT INTO statement used for retrieving the data of a particular row containing single or multiple column values in it into a single variable is as shown below –

SELECT list\_of\_values INTO list\_of\_variables FROM name\_of\_table WHERE required\_condition;

The terminologies used in the above syntax are as explained one by one in the below description –

* List of values – This is the name of the columns or expression values that we are trying to retrieve from the table data.
* List of variables – These are the variable names inside which we will store the retrieved values from the table of the list of values. Note that the number of variables and the datatype of variables used in this list should be the same or corresponding to the datatype of the list of values and the number of values retrieved from it.
* Name of the table – This is the table name from which we want to retrieve the data for the list of values specified in the SELECT statement.
* Required condition – This can be any condition that we are trying to specify in order to filter out the table data and retrieve only one row from the content of the table that we want to store into variables. Use of where clause is optional in nature.

You can make the use of the SELECT INTO statement in PL/ SQL to retrieve the row containing single or multiple column values in the resultant for storing them in variables.

### Examples

Let us now try to understand the implementation of PL/ SQL select into a statement with the help of certain examples. Let us talk about the sample data firstly, which we will consider for demonstrating the use of select into a statement. We have one existing table in our PL/ SQL database whose name is customer\_details. To check the existing content of the table, you can fire the following query statement –

SELECT \* FROM customer\_details;

The execution of the above query statement gives out the following output –

#### Example #1 – Retrieving a single column into a variable

Now, we will try to retrieve the value of the first name of the customer stored inside the f\_name column into a variable with the name first\_name. For this, we will make use of the following PL/ SQL program. Along with that, we will also retrieve the value of the variable in the DBMS output and will verify whether the column value is properly retrieved in the variable or not –

DECLARE  
first\_name customer\_details.f\_name%TYPE;  
BEGIN  
-- retrieve the value of the first nam eof customer with id 101 assign it to first\_name  
SELECT f\_name INTO first\_name  
FROM customer\_details  
WHERE customer\_id = 101;  
-- Display the customer first name  
dbms\_output.put\_line( first\_name );  
END;

The execution of the above query statement gives out the following output –

The %ROWTYPE attribute provides a **record type** that represents a row in a database table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. Variables declared using %ROWTYPE are treated like those declared using a datatype name.

#### Example #2 – Retrieving the whole row record into a variable

We can even retrieve the complete row record into a single variable by making the use of the SELECT INTO statement. Let us consider an example where we will try to retrieve the complete row data of the customer having the customer id as 110. We can do this by making use of below PL/ SQL program below –

DECLARE  
cust\_record\_variable customer\_details%ROWTYPE;  
BEGIN  
-- retrieve the value of the row record of customer with id 110 assign it to cust\_record\_variable  
SELECT \* INTO cust\_record\_variable  
FROM customer\_details  
WHERE customer\_id = 110;  
-- Display the customer information  
dbms\_output.put\_line( cust\_record\_variable.f\_name || ', Email Id: ' || cust\_record\_variable.email\_id );  
END;

The execution of the above query statement gives out the following output.

Note that we can access the individual column value stored inside the variable by using the dot notation in the format such as “name of the variable. name of column”.

#### Example #3 – Retrieving the data into multiple variables

We can even retrieve the data of a single row and each and every individual column value of the row of the table or retrieved data into a corresponding variable for each and every field. Consider the same above example where we are trying to retrieve the first name and email id of the customer. Along with that, we will also retrieve the mobile number details but in individual variables as demonstrated in the below example –

DECLARE  
cust\_f\_name customers.f\_name%TYPE;  
cust\_email\_id contacts.email\_id%TYPE;  
cust\_mobile\_number contacts.mobile\_number%TYPE;  
BEGIN  
-- retrieve the value of the row record consisting first name, email id and mobile number of customer with id 110 assign it to respective variables  
SELECT  
f\_name,  
email\_id,  
mobile\_number  
INTO  
cust\_f\_name,  
cust\_email\_id,  
cust\_mobile\_number  
FROM  
customers  
WHERE  
customer\_id = 110;  
-- Display the customer information  
dbms\_output.put\_line(  
cust\_f\_name || ', Contact Details: ' ||  
cust\_email\_id || ' ' || cust\_mobile\_number );  
END;

The execution of the above query statement gives out the following output –

### Conclusion – PL/SQL SELECT INTO

PL/SQL select into statement is used for taking the values of a single row belonging to a single record into a single variable. It becomes easier to handle the data when you have a complete row data present inside the variable in PL/ SQL program.

**DML Transactions in PL/SQL**

DML stands for **Data Manipulation Language**. These statements are mainly used to perform the manipulation activity. It deals with the below operations.

* Data Insertion
* Data Update
* Data Deletion
* Data Selection

In PL/SQL, we can do the data manipulation only by using the SQL commands.

**Data Insertion**

In PL/SQL, we can insert the data into any table using the SQL command INSERT INTO. This command will take the table name, table column and column values as the input and insert the value in the base table.

The INSERT command can also take the values directly from another table using ‘SELECT’ statement rather than giving the values for each column. Through ‘SELECT’ statement, we can insert as many rows as the base table contains.

**Syntax:**

Learn Java Programming with Beginners Tutorial

BEGIN

INSERT INTO <table\_name>(<column1 >,<column2>,...<column\_n>)

VALUES(<valuel><value2>,...:<value\_n>);

END;

* The above syntax shows the INSERT INTO command. The table name and values are a mandatory fields, whereas column names are not mandatory if the insert statements have values for all the column of the table.
* The keyword ‘VALUES’ is mandatory if the values are given separately as shown above.

**Syntax:**

BEGIN

INSERT INTO <table\_name>(<columnl>,<column2>,...,<column\_n>)

SELECT <columnl>,<column2>,.. <column\_n> FROM <table\_name2>;

END;

* The above syntax shows the INSERT INTO command that takes the values directly from the <table\_name2> using the SELECT command.
* The keyword ‘VALUES’ should not be present in this case as the values are not given separately.

**Data Update**

Data update simply means an update of the value of any column in the table. This can be done using ‘UPDATE’ statement. This statement takes the table name, column name and value as the input and updates the data.

**Syntax:**

BEGIN

UPDATE <table\_name>

SET <columnl>=<VALUE1>,<column2>=<value2>,<column\_n>=<value\_n>

WHERE <condition that uniquely identifies the record that needs to be update>;

END;

* The above syntax shows the UPDATE. The keyword ‘SET’ instruct that PL/SQL engine to update the value of the column with the value given.
* ‘WHERE’ clause is optional. If this clause is not given, then the value of the mentioned column in the entire table will be updated.

**Data Deletion**

Data deletion means to delete one full record from the database table. The ‘DELETE’ command is used for this purpose.

**Syntax:**

BEGIN

DELETE

FROM

<table\_name>

WHERE <condition that uniquely identifies the record that needs to be update>;

END;

* The above syntax shows the DELETE command. The keyword ‘FROM’ is optional and with or without ‘FROM’ clause the command behaves in the same way.
* ‘WHERE’ clause is optional. If this clause is not given, then the entire table will be deleted.

**Data Selection**

Data projection/fetching means to retrieve the required data from the database table. This can be achieved by using the command ‘SELECT’ with ‘INTO’ clause. The ‘SELECT’ command will fetch the values from the database, and ‘INTO’ clause will assign these values to the local variable of the PL/SQL block.

Below are the points that need to be considered in ‘SELECT’ statement.

* ‘SELECT’ statement should return only one record while using ‘INTO’ clause as one variable can hold only one value. If the ‘SELECT’ statement returns more than one value than ‘TOO\_MANY\_ROWS’ exception will be raised.
* ‘SELECT’ statement will assign the value to the variable in the ‘INTO’ clause, so it needs to get at least one record from the table to populate the value. If it didn’t get any record, then the exception ‘NO\_DATA\_FOUND’ is raised.
* The number of columns and their datatype in ‘SELECT’ clause should match with the number of variables and their datatypes in the ‘INTO’ clause.
* The values are fetched and populated in the same order as mentioned in the statement.
* ‘WHERE’ clause is optional that allows to having more restriction on the records that are going to be fetched.
* ‘SELECT’ statement can be used in the ‘WHERE’ condition of other DML statements to define the values of the conditions.
* The ‘SELECT’ statement when using ‘INSERT’, ‘UPDATE’, ‘DELETE’ statements should not have ‘INTO’ clause as it will not populate any variable in these cases.

**Syntax:**

BEGIN

SELECT <columnl>,..<column\_n> INTO <vanable 1 >,. .<variable\_n>

FROM <table\_name>

WHERE <condition to fetch the required records>;

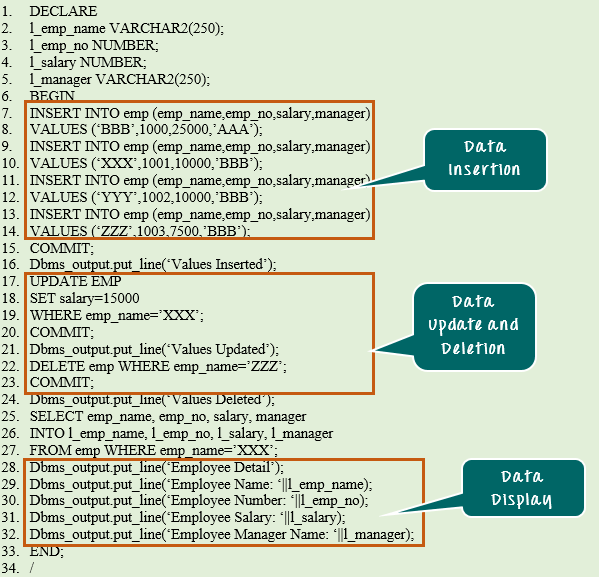
END;

* The above syntax shows the SELECT-INTO command. The keyword ‘FROM’ is mandatory that identifies the table name from which the data needs to be fetched.
* ‘WHERE’ clause is optional. If this clause is not given, then the data from the entire table will be fetched.

**Example 1**: In this example, we are going to see how to perform DML operations in PL/SQL. We are going to insert the below four records into emp table.

|  |  |  |  |
| --- | --- | --- | --- |
| **EMP\_NAME** | **EMP\_NO** | **SALARY** | **MANAGER** |
| BBB | 1000 | 25000 | AAA |
| XXX | 1001 | 10000 | BBB |
| YYY | 1002 | 10000 | BBB |
| ZZZ | 1003 | 7500 | BBB |

Then we are going to update the salary of ‘XXX’ to 15000, and we are going to delete the employee record ‘ZZZ’. Finally, we are going to project the details of the employee ‘XXX’.



DECLARE

l\_emp\_name VARCHAR2(250);

l\_emp\_no NUMBER;

l\_salary NUMBER;

l\_manager VARCHAR2(250);

BEGIN

INSERT INTO emp(emp\_name,emp\_no,salary,manager)

VALUES(‘BBB’,1000,25000,’AAA’);

INSERT INTO emp(emp\_name,emp\_no,salary,manager)

VALUES('XXX',1001,10000,’BBB);

INSERT INTO emp(emp\_name,emp\_no,salary,managed

VALUES(‘YYY',1002,10000,'BBB');

INSERT INTO emp(emp\_name,emp\_no,salary,manager)

VALUES(‘ZZZ',1003,7500,'BBB'):

COMMIT;

Dbms\_output.put\_line(‘Values Inserted');

UPDATE EMP

SET salary=15000

WHERE emp\_name='XXX';

COMMIT;

Dbms\_output.put\_line(‘Values Updated');

DELETE emp WHERE emp\_name='ZZZ';

COMMIT:

Dbms\_output.put\_line('Values Deleted );

SELECT emp\_name,emp\_no,salary,manager INTO l\_emp\_name,l\_emp\_no,l\_salary,l\_manager FROM emp WHERE emp\_name='XXX';

Dbms output.put line(‘Employee Detail’);

Dbms\_output.put\_line(‘Employee Name:‘||l\_emp\_name);

Dbms\_output.put\_line(‘Employee Number:‘||l\_emp\_no);

Dbms\_output.put\_line(‘Employee Salary:‘||l\_salary);

Dbms output.put line(‘Emplovee Manager Name:‘||l\_manager):

END;

/

**Output:**

Values Inserted

Values Updated

Values Deleted

Employee Detail

Employee Name:XXX

Employee Number:1001

Employee Salary:15000

Employee Manager Name:BBB

**Code Explanation:**

* **Code line 2-5**: Declaring the variable.
* **Code line 7-14**: Inserting the records into emp table.
* **Code line 15**: Committing the insert transactions.
* **Code line 17-19**: Updating the salary of the employee ‘XXX’ to 15000
* **Code line 20**: Committing the update transaction.
* **Code line 22**: Deleting the record of ‘ZZZ’
* **Code line 23**: Committing the delete transaction.
* **Code line 25-27**: Selecting the record of ‘XXX’ and populating into the variable l\_emp\_name, l\_emp\_no, l\_salary, l\_manager.
* **Code line 28-32**: Displaying the fetched records value.

PLSQL Cursors:

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

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

* Implicit Cursors
* Explicit Cursors

## 1) PL/SQL Implicit Cursors

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

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

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

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

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

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

PL/SQL Implicit Cursor Example

**Create customers table and have records:**

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

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

**Create procedure:**

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

Output:

6 customers updated

PL/SQL procedure successfully completed.

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

## PL/SQL Explicit Cursors

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

Following is the syntax to create an explicit cursor:

Syntax of explicit cursor

Following is the syntax to create an explicit cursor:

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

Steps:

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

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

## 1) Declare the cursor:

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

**Syntax for explicit cursor decleration**

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

2) Open the cursor:

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

**Syntax for cursor open:**

1. **OPEN** cursor\_name;

## 3) Fetch the cursor:

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

**Syntax for cursor fetch:**

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

4) Close the cursor:

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

**Syntax for cursor close:**

1. **Close** cursor\_name;
2. **DECLARE**
3. c\_id customers.id%type;
4. c\_name customers.**name**%type;
5. c\_addr customers.address%type;
6. -- **Declare the cursor**
7. **CURSOR** c\_customers **is**
8. **SELECT** id, **name**, address **FROM** customers;
9. **BEGIN**
10. –open the cursor
11. **OPEN** c\_customers;
12. LOOP
13. **FETCH** c\_customers **into** c\_id, c\_name, c\_addr;
14. EXIT **WHEN** c\_customers%notfound;
15. dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);
16. **END** LOOP;
17. **CLOSE** c\_customers;
18. **END**;
19. /

**We have created a table with the SQL statement given below:**

|  |
| --- |
| CREATE TABLE TUTOR(  CODE INT NOT NULL,  SUBJECT VARCHAR(15) NOT NULL,  TEACHER VARCHAR(15),  REVIEWS VARCHAR (10) NOT NULL,  PRIMARY KEY (CODE)  ); |

**Inserted values to this table with SQL statements given below:**

|  |
| --- |
| INSERT INTO TUTOR (CODE,SUBJECT,TEACHER,REVIEWS)  VALUES (1, 'Automation', 'Mukul', 'five stars');  INSERT INTO TUTOR (CODE,SUBJECT,TEACHER,REVIEWS)  VALUES (4, 'PLSQL', 'Anand', 'four stars');  INSERT INTO TUTOR (CODE,SUBJECT,TEACHER,REVIEWS)  VALUES (2, 'Performance', 'Arvind', 'four stars'); |

**Code implementation with the implicit cursor:**

|  |
| --- |
| DECLARE      total\_count number(30);  BEGIN        --updating a row      UPDATE TUTOR      SET TEACHER = 'Zen' where CODE = 1;        -- result in boolean, true returned if no rows affected      IF sql%notfound THEN          dbms\_output.put\_line('no subjects fetched');              -- result in boolean, true returned if any rows affected          ELSIF sql%found THEN            -- count the number of rows affected rows affected          total\_count := sql%rowcount;          dbms\_output.put\_line( total\_count || ' teacher name updated ');      END IF;  END;  / |

**Syntax of explicit cursor:**

DECLARE

CURSOR <<cursor name>> IS <<select statement>>

<<Cursor variable>>

BEGIN

OPEN <<cursor name>>;

FETCH <<cursor name>> INTO <Cursor variable>;

.

.

CLOSE <cursor name>;

END;

**Code implementation with explicit cursor:**

|  |  |
| --- | --- |
| DECLARE     -- cursor declaration  CURSOR t\_tutorials is  SELECT code, subject, teacher FROM Tutor;  t\_code Tutor.code%type;  t\_subject Tutor.subject%type;  t\_teacher Tutor.teacher%type;  BEGIN       -- opening a cursor     OPEN t\_tutorials;  LOOP        -- fetching values from cursor      FETCH t\_tutorials into t\_code, t\_subject, t\_teacher;      EXIT WHEN t\_tutorials%notfound;        -- printing in console      dbms\_output.put\_line('Code is: ' || t\_code || ' ' || 'Subject is: ' || t\_subject || ' Teacher is: ' || t\_teacher);  END LOOP;  CLOSE t\_tutorials;  END;  /  PLSQL Strings:  The strings in PL/SQL is a group of characters in a particular order. T  he size of the string may or may not be available.  **Code implementation with strings:**   |  | | --- | | DECLARE      subject varchar2(30);      teacher varchar2(40);      syllabus clob;      options char(1);  BEGIN        -- Initializing values to variables      subject := 'Selenium';      teacher := 'Arun';      syllabus := 'Java, WebDriver Methods, Synchronization, WebTables.';      options := 'S';        -- checking condition and if true      IF options = 'S' THEN           -- printing in console         dbms\_output.put\_line(subject);         dbms\_output.put\_line(teacher);         dbms\_output.put\_line(syllabus);     END IF;  END;  / | |

| **Sl No.** | **Name** | **Purposes** |
| --- | --- | --- |
| **1** | **CONCAT(i, j)** | Appends the strings i and j and returns the new string. |
| **2** | **ASCII(n)** | Returns the equivalent ASCII value of n. |
| **3** | **CHR(n)** | Returns the character along with the equivalent ASCII value of n. |
| **4** | **INSTR(i, x, start, n)** | Finds the substring i in x string and then returns the position of occurrence. The start refers to the beginning position of searching and is an optional parameter. The n is the nth occurrence of the string and is also an optional parameter. |
| **5** | **INSTRB(i)** | Returns the position of a substring in a string in bytes. |
| **6** | **INITCAP(k)** | Converts the initial character of individual words in string k to the uppercase and then gives back the string. |
| **7** | **LENGTH(m)** | Returns the count of the number of characters in string m. |
| **8** | **LENGTHB(n)** | Returns the count of the characters in string m in bytes for single byte character set. |
| **9** | **LTRIM(n, x)** | Removes x characters from left of string n. The x is an optional parameter if not provided , removes all leading spaces of the string n. |
| **10** | **RTRIM(n, x)** | Removes x characters from right of string n. The x is an optional parameter if not provided , removes all trailing spaces of the string n. |
| **11** | **TRIM([trim\_char FROM) x);** | Removes spaces or mentioned characters from the start, end or both ends of the string x. |
| **12** | **LOWER(i)** | Converts the characters of string i to lower case and then returns the string. |
| **13** | **UPPER(i)** | Converts the characters of string i to upper case and then returns the string. |
| **14** | **LPAD(i, l , x)** | Pads string x to the left to make the string i length to l. The parameter x is optional, if omitted spaces are padded to the left of string i. |
| **15** | **RPAD(i, l , x)** | Pads string x to the right to make the string i length to l. The parameter x is optional, if omitted spaces are padded to the right of string i. |
| **16** | **NANVL(n, val)** | Returns val if n is equal to the NaN value, else n is returned. |
| **17** | **NLSSORT(i)** | Modifies the sorting method of characters. It should be mentioned prior to any NLS function, else the default sorting will be done. |
| **18** | **NLS\_INITCAP(i)** | Similar in functionality as function INITCAP but it can take a different sort of technique as mentioned in function NLSSORT. |
| **19** | **NLS\_LOWER(m)** | Similar in functionality as function LOWER but it can take a different sort of technique as mentioned in function NLSSORT. |
| **20** | **NLS\_UPPER(m)** | Similar in functionality as function UPPER but it can take a different sort of technique as mentioned IN function NLSSORT. |
| **21** | **NVL(n, val)** | Returns val if x is equal to the NULL value, else n is returned. |
| **22** | **NVL2(n, val, val2)** | Returns val if x is not equal to the NULL value, else if x is equal to NULL, val2 is returned. |
| **23** | **SOUNDEX(i)** | Returns a string having the vocal representation of i. |
| **24** | **SUBSTR(n, start, l)** | Returns a substring of string n that starts from the position mentioned in start. The parameter l is optional and represents the length of the substring. |
| **25** | **SUBSTRB(n)** | Similar in functionality as function SUBSTR but parameters are in bytes and not in characters for a single byte character system. |
| **26** | **REPLACE(n, s, r)** | Replaces the occurrences of s with the string r with in the string n. |

**Code implementation with some string functions:**

|  |
| --- |
| DECLARE      name varchar2(30) := ' software testing help!';  BEGIN      dbms\_output.put\_line(UPPER(name));      dbms\_output.put\_line(LOWER(name));        dbms\_output.put\_line(LENGTH(name));        dbms\_output.put\_line(INITCAP(name));        /\* get the first word in the string \*/      dbms\_output.put\_line ( SUBSTR (name, 1, 8));        /\* get the location of the first "w" \*/      dbms\_output.put\_line ( INSTR (name, 'w'));        /\* replace a string \*/     dbms\_output.put\_line ( REPLACE( name, 'help', 'solution'));       /\* trim a string from right \*/     dbms\_output.put\_line ( RTRIM(name,'!'));        /\* trim a string \*/      dbms\_output.put\_line ( TRIM(name));  END;  / |

# 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.

Syntax:

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;
2. CREATE OR REPLACE PROCEDURE greetings
3. AS
4. BEGIN
5. dbms\_output.put\_line('Hello World!');
6. END;
7. /

BEGIN

greetings;

END;

/

DROP PROCEDURE greetings;

### 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;

/

### 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;

/

## PL/SQL procedure syntax

A PL/SQL procedure is a reusable unit that encapsulates specific business logic of the application. Technically speaking, a PL/SQL procedure is a named [block](https://www.oracletutorial.com/plsql-tutorial/plsql-anonymous-block/) stored as a schema object in the Oracle Database.

The following illustrates the basic syntax of creating a procedure in PL/SQL:

CREATE [OR REPLACE ] PROCEDURE procedure\_name (parameter\_list)

IS

Code language: SQL (Structured Query Language) (sql)

[declaration statements]

BEGIN

[execution statements]

EXCEPTION

[exception handler]

END [procedure\_name ];

### PL/SQL procedure header

A procedure begins with a header that specifies its name and an optional parameter list.

Each parameter can be in either IN, OUT, or INOUT mode. The parameter mode specifies whether a parameter can be read from or write to.

IN

An IN parameter is read-only. You can reference an IN parameter inside a procedure, but you cannot change its value. Oracle uses IN as the default mode. It means that if you don’t specify the mode for a parameter explicitly, Oracle will use the IN mode.

OUT

An OUT parameter is writable. Typically, you set a returned value for the OUT parameter and return it to the calling program. Note that a procedure ignores the value that you supply for an OUT parameter.

INOUT

An INOUT parameter is both readable and writable. The procedure can read and modify it.

Note that OR REPLACE option allows you to overwrite the current procedure with the new code.

### PL/SQL procedure body

Similar to an [anonymous block](https://www.oracletutorial.com/plsql-tutorial/plsql-anonymous-block/), the procedure body has three parts. The executable part is mandatory whereas the declarative and exception-handling parts are optional. The executable part must contain at least one executable statement.

**1) Declarative part**

In this part, you can declare [variables](https://www.oracletutorial.com/plsql-tutorial/plsql-variables/), [constants](https://www.oracletutorial.com/plsql-tutorial/plsql-constants/), [cursors](https://www.oracletutorial.com/plsql-tutorial/plsql-cursor/), etc. Unlike an [anonymous block](https://www.oracletutorial.com/plsql-tutorial/plsql-anonymous-block/), a declaration part of a procedure does not start with the DECLARE keyword.

**2) Executable part**

This part contains one or more statements that implement specific business logic. It might contain only a [NULL statement](https://www.oracletutorial.com/plsql-tutorial/plsql-null/).

**3) Exception-handling part**

This part contains the code that handles [exceptions](https://www.oracletutorial.com/plsql-tutorial/plsql-exception/).

## Creating a PL/SQL procedure example

The following procedure accepts a customer id and prints out the customer’s contact information including first name, last name, and email:

CREATE OR REPLACE PROCEDURE print\_contact(

in\_customer\_id NUMBER

)

IS

r\_contact contacts%ROWTYPE;

BEGIN

-- get contact based on customer id

SELECT \*

INTO r\_contact

FROM contacts

WHERE customer\_id = p\_customer\_id;

-- print out contact's information

dbms\_output.put\_line( r\_contact.first\_name || ' ' ||

r\_contact.last\_name || '<' || r\_contact.email ||'>' );

EXCEPTION

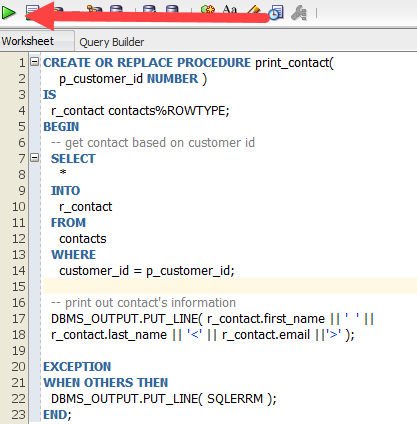
WHEN OTHERS THEN

dbms\_output.put\_line( SQLERRM );

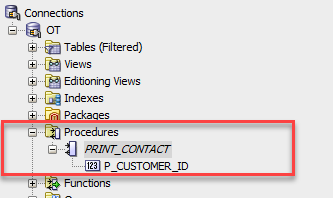
END;

Code language: SQL (Structured Query Language) (sql)

To compile the procedure, you click the **Run Statement** button as shown in the following picture:



If the procedure is compiled successfully, you will find the new procedure under the **Procedures** node as shown below:



## Executing a PL/SQL procedure

The following shows the syntax for executing a procedure:

EXECUTE procedure\_name( arguments);

Code language: SQL (Structured Query Language) (sql)

Or

EXEC procedure\_name( arguments);

Code language: SQL (Structured Query Language) (sql)

For example, to execute the print\_contact procedure that prints the contact information of customer id 100, you use the following statement:

EXEC print\_contact(100);

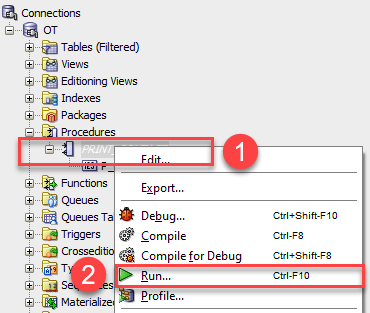
Code language: SQL (Structured Query Language) (sql)

Here is the output:

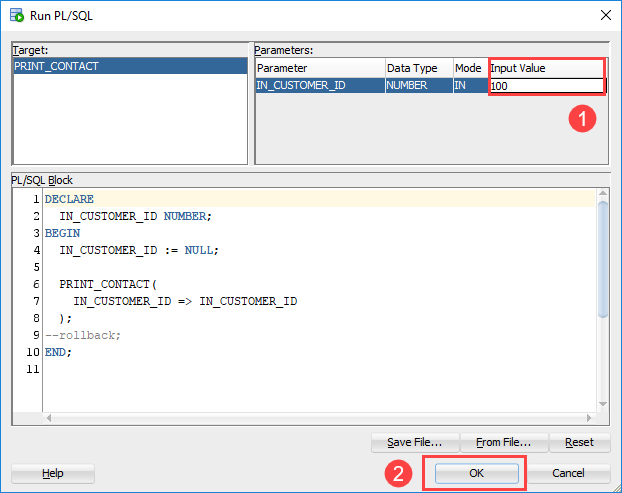
Elisha Lloyd<elisha.lloyd@verizon.com>

You can also execute a procedure from the Oracle SQL Developer using the following steps:

1)  Right-click the procedure name and choose **Run…** menu item



2) Enter a value for the  in\_customer\_id parameter and click **OK** button.



3) The following shows the result

Connecting to the database Local.

Elisha Lloyd<elisha.lloyd@verizon.com>

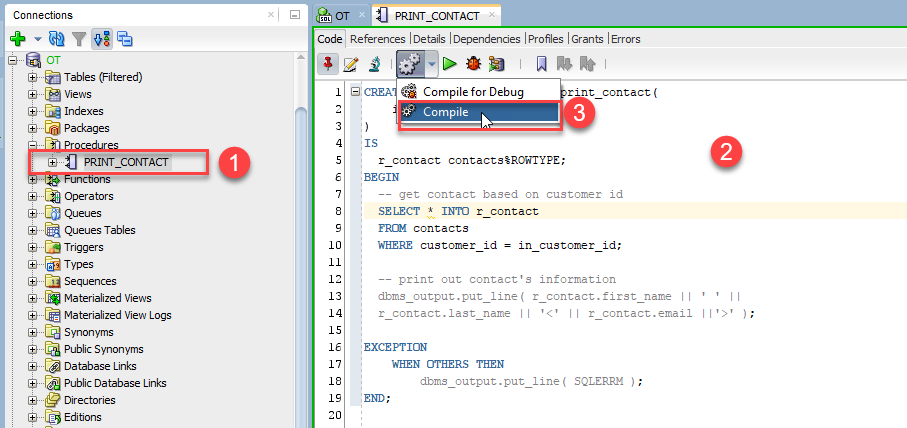
Process exited.

Disconnecting from the database Local.

## Editing a procedure

To change the code of an existing procedure, you can follow these steps:

* Step 1. Click the procedure name under **Procedures** node.
* Step 2. Edit the code of the procedure.
* Step 3. Click Compile menu option to recompile the procedure.



## Removing a procedure

To delete a procedure, you use the DROP PROCEDURE followed by the procedure’s name that you want to drop as shown in the following syntax:

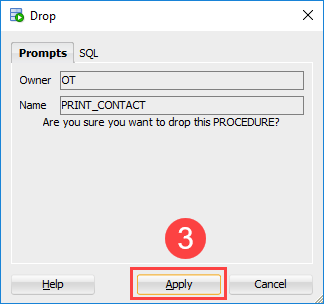
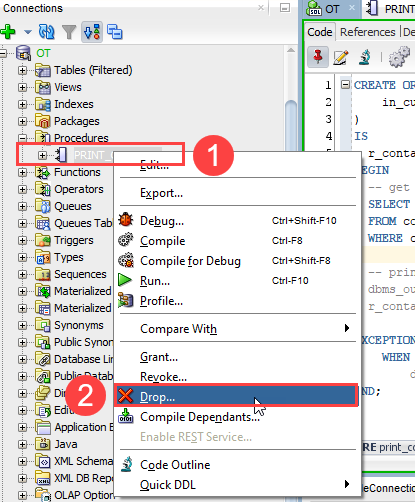
DROP PROCEDURE procedure\_name;

For example, the following statement drops the print\_contact procedure :

DROP PROCEDURE print\_contact;

The following illustrates the steps of dropping a procedure using SQL Developer:

* Step 1. Right click on the procedure name that you want to drop
* Step 2. Choose the **Drop…** menu option
* Step 3. In the Prompts dialog, click the **Apply** button to remove the procedure.



PLSQL Functions:

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:**

**CREATE** [OR REPLACE] **FUNCTION** function\_name [parameters]

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

**RETURN** return\_datatype

{**IS** | **AS**}

**BEGIN**

   < function\_body >

**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

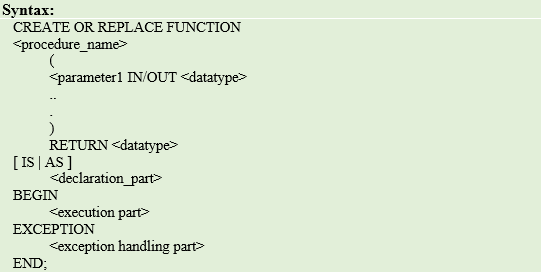
Let's see a simple example to **create a function**.

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. /
10. **DECLARE**
11. n3 number(2);
12. **BEGIN**
13. n3 := adder(11,22);
14. dbms\_output.put\_line('Addition is: ' || n3);
15. **END**;
16. /
17. **DROP** **FUNCTION** function\_name;

## What is Function?

Functions is a standalone PL/SQL subprogram. Like PL/SQL procedure, functions have a unique name by which it can be referred. These are stored as PL/SQL database objects. Below are some of the characteristics of functions.

* Functions are a standalone block that is mainly used for calculation purpose.
* Function use RETURN keyword to return the value, and the datatype of this is defined at the time of creation.
* A Function should either return a value or raise the exception, i.e. return is mandatory in functions.
* Function with no DML statements can be directly called in SELECT query whereas the function with DML operation can only be called from other PL/SQL blocks.
* 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 function or fetched from the procedure through the parameters.
* These parameters should be included in the calling statement.
* A PLSQL function can also return the value through OUT parameters other than using RETURN.
* Since it will always return the value, in calling statement it always accompanies with assignment operator to populate the variables.



#### Syntax

CREATE OR REPLACE FUNCTION

<procedure\_name>

(

<parameterl IN/OUT <datatype>

)

RETURN <datatype>

[ IS | AS ]

<declaration\_part>

BEGIN

<execution part>

EXCEPTION

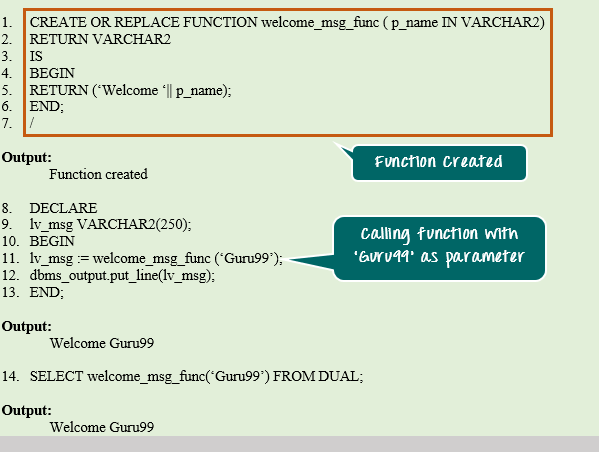
<exception handling part>

END;

* CREATE FUNCTION instructs the compiler to create a new function. Keyword ‘OR REPLACE’ instructs the compiler to replace the existing function (if any) with the current one.
* The Function name should be unique.
* RETURN datatype should be mentioned.
* 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 Function and calling it using Anonymous Block**

In this program, we are going to create a function that takes the name as input and returns the welcome message as output. We are going to use anonymous block and select statement to call the function.



CREATE OR REPLACE FUNCTION welcome\_msgJune ( p\_name IN VARCHAR2) RETURN VAR.CHAR2

IS

BEGIN

RETURN (‘Welcome ‘|| p\_name);

END;

/

DECLARE

lv\_msg VARCHAR2(250);

BEGIN

lv\_msg := welcome\_msg\_func (‘Guru99’);

dbms\_output.put\_line(lv\_msg);

END;

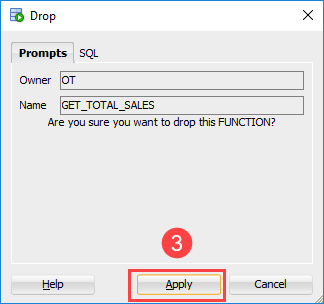
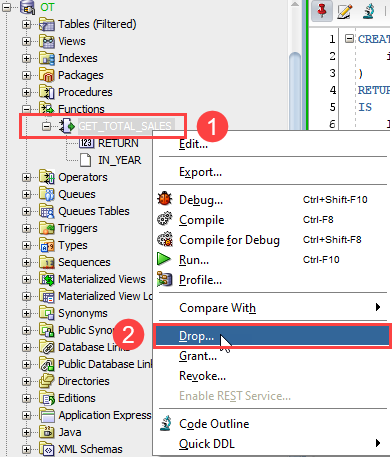
SELECT welcome\_msg\_func(‘Guru99:) FROM DUAL;

**Code Explanation:**

* **Code line 1**: Creating the Oracle function with name ‘welcome\_msg\_func’ and with one parameter ‘p\_name’ of ‘IN’ type.
* **Code line 2**: declaring the return type as VARCHAR2
* **Code line 5**: Returning the concatenated value ‘Welcome’ and the parameter value.
* **Code line 8**: Anonymous block to call the above function.
* **Code line 9**: Declaring the variable with datatype same as the return datatype of the function.
* **Code line 11**: Calling the function and populating the return value to the variable ‘lv\_msg’.
* **Code line 12**: Printing the variable value. The output you will get here is “Welcome Guru99”
* **Code line 14**: Calling the same function through SELECT statement. The return value is directed to the standard output directly.

## Similarities between Procedure and Function

* Both can be called from other PL/SQL blocks.
* If the exception raised in the subprogram is not handled in the subprogram exception handling section, then it will propagate to the calling block.
* Both can have as many parameters as required.
* Both are treated as database objects in PL/SQL.



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. /

# PL/SQL Exception Handling:

## What is Exception

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

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

There are two type of exceptions:

* System-defined Exceptions
* User-defined Exceptions

## PL/SQL Exception Handling

**Syntax for exception handling:**

Following is a general syntax for exception handling:

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

## Example of exception handling

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

SELECT\* FROM COUSTOMERS;

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

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

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

No such customer!

PL/SQL procedure successfully completed.

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

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

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

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

Name: alex

Address: paris

PL/SQL procedure successfully completed.

Raising Exceptions

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

**Syntax for raising an exception:**

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

## PL/SQL Pre-defined Exceptions

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

**For example:**NO\_DATA\_FOUND is a pre-defined exception which is raised when a SELECT INTO statement returns no rows.

Following is a list of some important pre-defined exceptions:

|  |  |  |  |
| --- | --- | --- | --- |
| **Exception** | **Oracle Error** | **SQL Code** | **Description** |
| ACCESS\_INTO\_NULL | 06530 | -6530 | It is raised when a NULL object is automatically assigned a value. |
| CASE\_NOT\_FOUND | 06592 | -6592 | It is raised when none of the choices in the "WHEN" clauses of a CASE statement is selected, and there is no else clause. |
| COLLECTION\_IS\_NULL | 06531 | -6531 | It is raised when a program attempts to apply collection methods other than exists to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| DUP\_VAL\_ON\_INDEX | 00001 | -1 | It is raised when duplicate values are attempted to be stored in a column with unique index. |
| INVALID\_CURSOR | 01001 | -1001 | It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID\_NUMBER | 01722 | -1722 | It is raised when the conversion of a character string into a number fails because the string does not represent a valid number. |
| LOGIN\_DENIED | 01017 | -1017 | It is raised when s program attempts to log on to the database with an invalid username or password. |
| NO\_DATA\_FOUND | 01403 | +100 | It is raised when a select into statement returns no rows. |
| NOT\_LOGGED\_ON | 01012 | -1012 | It is raised when a database call is issued without being connected to the database. |
| PROGRAM\_ERROR | 06501 | -6501 | It is raised when PL/SQL has an internal problem. |
| ROWTYPE\_MISMATCH | 06504 | -6504 | It is raised when a cursor fetches value in a variable having incompatible data type. |
| SELF\_IS\_NULL | 30625 | -30625 | It is raised when a member method is invoked, but the instance of the object type was not initialized. |
| STORAGE\_ERROR | 06500 | -6500 | It is raised when PL/SQL ran out of memory or memory was corrupted. |
| TOO\_MANY\_ROWS | 01422 | -1422 | It is raised when a SELECT INTO statement returns more than one row. |
| VALUE\_ERROR | 06502 | -6502 | It is raised when an arithmetic, conversion, truncation, or size-constraint error occurs. |
| ZERO\_DIVIDE | 01476 | 1476 | It is raised when an attempt is made to divide a number by zero. |

## User-defined Exceptions

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

The syntax for declaring an exception is −

DECLARE

my-exception EXCEPTION;

### Example

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

DECLARE

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

c\_name customerS.Name%type;

c\_addr customers.address%type;

-- user defined exception

ex\_invalid\_id EXCEPTION;

BEGIN

IF c\_id <= 0 THEN

RAISE ex\_invalid\_id;

ELSE

SELECT name, address INTO c\_name, c\_addr

FROM customers

WHERE id = c\_id;

DBMS\_OUTPUT.PUT\_LINE ('Name: '|| c\_name);

DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);

END IF;

EXCEPTION

WHEN ex\_invalid\_id THEN

dbms\_output.put\_line('ID must be greater than zero!');

WHEN no\_data\_found THEN

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

WHEN others THEN

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

END;

/

Syntax to write an exception

**WHEN** exception **THEN**

statement;

*DECLARE  
declarations section;*

*BEGIN  
executable command(s);*

*EXCEPTION  
WHEN exception1 THEN  
statement1;  
WHEN exception2 THEN  
statement2;  
[WHEN others THEN]  
/\* default exception handling code \*/*

*END;*

**Note:**  
**When other** keyword should be used only at the end of the exception handling block as no exception handling part present later will get executed as the control will exit from the block after executing the WHEN OTHERS.

1. **System defined exceptions:**  
   These exceptions are predefined in PL/SQL which get raised WHEN certain **database rule is violated.**  
   System-defined exceptions are further divided into two categories:
   1. Named system exceptions.
   2. Unnamed system exceptions.
   3. **Named system exceptions:** They have a predefined name by the system like ACCESS\_INTO\_NULL, DUP\_VAL\_ON\_INDEX, LOGIN\_DENIED etc. the list is quite big.

So we will discuss some of the most commonly used exceptions:

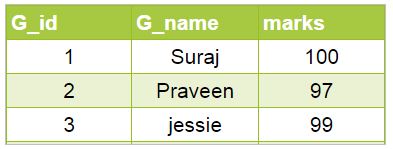
Lets create a table geeks.

create table geeks(g\_id int , g\_name varchar(20), marks int);

insert into geeks values(1, 'Suraj',100);

insert into geeks values(2, 'Praveen',97);

insert into geeks values(3, 'Jessie', 99);



* + 1. **NO\_DATA\_FOUND**: It is raised WHEN a SELECT INTO statement returns *no* rows. For eg:

|  |
| --- |
| DECLARE     temp varchar(20);    BEGIN     SELECT g\_id into temp from geeks where g\_name='GeeksforGeeks';    exception     WHEN no\_data\_found THEN        dbms\_output.put\_line('ERROR');        dbms\_output.put\_line('there is no name as');        dbms\_output.put\_line('GeeksforGeeks in geeks table');  end; |

* + 1. Output:
    2. ERROR
    3. there is no name as GeeksforGeeks in geeks table
    4. **TOO\_MANY\_ROWS**:It is raised WHEN a SELECT INTO statement returns *more* than one row.

|  |
| --- |
| DECLARE     temp varchar(20);    BEGIN    -- raises an exception as SELECT  -- into trying to return too many rows     SELECT g\_name into temp from geeks;     dbms\_output.put\_line(temp);    EXCEPTION     WHEN too\_many\_rows THEN        dbms\_output.put\_line('error trying to SELECT too many rows');    end; |

* + 1. Output:
    2. error trying to SELECT too many rows
    3. **VALUE\_ERROR**:This error is raised WHEN a statement is executed that resulted in an arithmetic, numeric, string, conversion, or constraint error. This error mainly results from programmer error or invalid data input.

|  |
| --- |
| DECLARE     temp number;    BEGIN     SELECT g\_name  into temp from geeks where g\_name='Suraj';     dbms\_output.put\_line('the g\_name is '||temp);    EXCEPTION     WHEN value\_error THEN     dbms\_output.put\_line('Error');     dbms\_output.put\_line('Change data type of temp to varchar(20)');    END; |

* + 1. Output:
    2. Error
    3. Change data type of temp to varchar(20)
    4. **ZERO\_DIVIDE** = raises exception WHEN dividing with zero.

|  |
| --- |
| DECLARE     a int:=10;     b int:=0;     answer int;    BEGIN     answer:=a/b;     dbms\_output.put\_line('the result after division is'||answer);    exception     WHEN zero\_divide THEN        dbms\_output.put\_line('dividing by zero please check the values again');        dbms\_output.put\_line('the value of a is '||a);        dbms\_output.put\_line('the value of b is '||b);  END; |

* + 1. Output:
    2. dividing by zero please check the values again
    3. the value of a is 10
    4. the value of b is 0
  1. **Unnamed system exceptions:**Oracle doesn’t provide name for some system exceptions called unnamed system exceptions.These exceptions *don’t* occur frequently.These exceptions have two parts *code and an associated message*.  
     The way to handle to these exceptions is to *assign name* to them using **Pragma EXCEPTION\_INIT**  
     Syntax:
  2. PRAGMA EXCEPTION\_INIT(exception\_name, -error\_number);

error\_number are pre-defined and have negative integer range from -20000 to -20999.

**Example:**

|  |
| --- |
| DECLARE     exp exception;     pragma exception\_init (exp, -20015);     n int:=10;    BEGIN     FOR i IN 1..n LOOP        dbms\_output.put\_line(i\*i);           IF i\*i=36 THEN              RAISE exp;           END IF;     END LOOP;    EXCEPTION     WHEN exp THEN        dbms\_output.put\_line('Welcome to GeeksforGeeks');    END; |

Output:

1

4

9

16

25

36

Welcome to GeeksforGeeks

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

*Example:*

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

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

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

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

*Input 1:* x = 20

y = 10

*Output:* the result is 2

*Input 2:* x = 20

y = 0

*Output:*

Error

division by zero not allowed

*Input 3:* x=20

y = 30

*Output:<.em>*

*Error*

*y is greater than x please check the input*

***RAISE\_APPLICATION\_ERROR****:  
It is used to display user-defined error messages with error number whose range is in between -20000 and -20999. When RAISE\_APPLICATION\_ERROR executes it returns error message and error code which looks****same as Oracle built-in error.***

***Example:***

|  |
| --- |
| DECLARE      myex EXCEPTION;      n NUMBER :=10;    BEGIN      FOR i IN 1..n LOOP      dbms\_output.put\_line(i\*i);          IF i\*i=36 THEN          RAISE myex;          END IF;      END LOOP;    EXCEPTION      WHEN myex THEN          RAISE\_APPLICATION\_ERROR(-20015, 'Welcome to GeeksForGeeks');    END; |

*Output:*

*Error report:*

*ORA-20015: Welcome to GeeksForGeeks*

*ORA-06512: at line 13*

*1*

*4*

*9*

*16*

*25*

*36*

***Note:****The output is based on Oracle Sql developer, the output order might change IF you’re running this code somewhere else.*

***Scope rules in exception handling****:*

* 1. *We can’t DECLARE an exception twice but we can DECLARE the same exception in****two dIFferent blocks.***
  2. *Exceptions DECLAREd inside a block are local to that block and global to all its sub-blocks.*

*As a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions DECLAREd in a sub-block.  
If we reDECLARE a global exception in a sub-block, the local declaration prevails i.e. the scope of local is more.*

***Example:***

|  |
| --- |
| DECLARE     GeeksforGeeks EXCEPTION;     age NUMBER:=16;  BEGIN       --  sub-block BEGINs     DECLARE          -- this declaration prevails        GeeksforGeeks  EXCEPTION;        age NUMBER:=22;       BEGIN        IF age > 16 THEN           RAISE GeeksforGeeks; /\* this is not handled\*/        END IF;       END;     -- sub-block ends    EXCEPTION    -- Does not handle raised exception    WHEN GeeksforGeeks THEN      DBMS\_OUTPUT.PUT\_LINE        ('Handling  GeeksforGeeks exception.');      WHEN OTHERS THEN      DBMS\_OUTPUT.PUT\_LINE        ('Could not recognize exception GeeksforGeeks  in this scope.');  END; |

*Output:*

*Could not recognize exception GeeksforGeeks in this scope.*

***Advantages:***

* *Exception handling is very useful for error handling, without it we have to issue the command at every point to check for execution errors:****Example:***
* *Select ..*
* *.. check for 'no data found' error*
* *Select ..*
* *.. check for 'no data found' error*
* *Select ..*
* *.. check for 'no data found' error*

*Here we can see that it is not****robust****as error processing is not separated from normal processing and IF we miss some line in the code than it may lead to some other kind of error.*

* *With exception handling we handle errors without writing statements multiple times and we can even handle****dIFferent****types of errors in one exception block:****Example:***
* *BEGIN*
* *SELECT ...*
* *SELECT ...*
* *SELECT ...*
* *.*
* *.*
* *.*
* *exception*
* *WHEN NO\_DATA\_FOUND THEN /\* catches all 'no data found' errors \*/*
* *...*
* *WHEN ZERO\_DIVIDE THEN /\* different types of \*/*
* *WHEN value\_error THEN /\* errors handled in same block \*/*
* *...*

*From above code we can conclude that exception handling*

1. *Improves****readability****by letting us isolate error-handling routines and thus providing robustness.*
2. *Provides****reliability****, instead of checking for dIFferent types of errors at every point we can simply write them in exception block and IF error exists exception will be raised thus helping the programmer to find out the type of error and eventually resolve it.*

***Uses****: One of the real lIFe use of exception can be found in online train reservation system.  
While filling the station code to book the ticket IF we input wrong code it shows us the exception that the code doesn’t exist in database.*

# PL/SQL Trigger

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

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

Triggers are written to be executed in response to any of the following events.

* A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
* A database definition (DDL) statement (CREATE, ALTER, or DROP).
* A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Creating a trigger:

**Syntax for creating trigger:**

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

**Here,**

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

PL/SQL Trigger Example

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

**Create table and have records:**

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

**Create trigger:**

Let's take a program to create a row level trigger for the CUSTOMERS table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

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

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

Trigger created.

**Check the salary difference by procedure:**

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

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

Output:

Old salary: 20000

New salary: 25000

Salary difference: 5000

Old salary: 22000

New salary: 27000

Salary difference: 5000

Old salary: 24000

New salary: 29000

Salary difference: 5000

Old salary: 26000

New salary: 31000

Salary difference: 5000

Old salary: 28000

New salary: 33000

Salary difference: 5000

Old salary: 30000

New salary: 35000

Salary difference: 5000

6 customers updated

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

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

Old salary: 25000

New salary: 30000

Salary difference: 5000

Old salary: 27000

New salary: 32000

Salary difference: 5000

Old salary: 29000

New salary: 34000

Salary difference: 5000

Old salary: 31000

New salary: 36000

Salary difference: 5000

Old salary: 33000

New salary: 38000

Salary difference: 5000

Old salary: 35000

New salary: 40000

Salary difference: 5000

6 customers updated

Important Points

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

* OLD and NEW references are used for record level triggers these are not avialable for table level triggers.
* If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.

# Oracle / PLSQL: BEFORE INSERT Trigger

This Oracle tutorial explains how to **create a BEFORE INSERT Trigger** in Oracle with syntax and examples.

## Description

A **BEFORE INSERT Trigger** means that Oracle will fire this trigger before the INSERT operation is executed.

## Syntax

The syntax to create a **BEFORE INSERT Trigger** in Oracle/PLSQL is:

CREATE [ OR REPLACE ] TRIGGER trigger\_name

BEFORE INSERT

ON table\_name

[ FOR EACH ROW ]

DECLARE

-- variable declarations

BEGIN

-- trigger code

EXCEPTION

WHEN ...

-- exception handling

END;

### Parameters or Arguments

**OR REPLACE**

Optional. If specified, it allows you to re-create the trigger is it already exists so that you can change the trigger definition without issuing a [DROP TRIGGER statement](https://www.techonthenet.com/oracle/triggers/drop.php).

**trigger\_name**

The name of the trigger to create.

**BEFORE INSERT**

It indicates that the trigger will fire before the INSERT operation is executed.

**table\_name**

The name of the table that the trigger is created on.

## Restrictions

* You can not create a BEFORE trigger on a view.
* You can update the :NEW values.
* You can not update the :OLD values.

## Note

* See also how to create [AFTER DELETE](https://www.techonthenet.com/oracle/triggers/after_delete.php), [AFTER INSERT](https://www.techonthenet.com/oracle/triggers/after_insert.php), [AFTER UPDATE](https://www.techonthenet.com/oracle/triggers/after_update.php), [BEFORE DELETE](https://www.techonthenet.com/oracle/triggers/before_delete.php), and [BEFORE UPDATE](https://www.techonthenet.com/oracle/triggers/before_update.php) triggers.
* See also how to [drop a trigger](https://www.techonthenet.com/oracle/triggers/drop.php).

## Example

Let's look at an example of how to create an BEFORE INSERT trigger using the CREATE TRIGGER statement.

If you had a table created as follows:

CREATE TABLE orders

( order\_id number(5),

quantity number(4),

cost\_per\_item number(6,2),

total\_cost number(8,2),

create\_date date,

created\_by varchar2(10)

);

We could then use the CREATE TRIGGER statement to create an BEFORE INSERT trigger as follows:

**TIP:** When using SQLPlus, you need to enter slash on a new line after the trigger. Otherwise, the script won't execute.

CREATE OR REPLACE TRIGGER orders\_before\_insert

BEFORE INSERT

ON orders

FOR EACH ROW

DECLARE

v\_username varchar2(10);

BEGIN

-- Find username of person performing INSERT into table

SELECT user INTO v\_username

FROM dual;

-- Update create\_date field to current system date

:new.create\_date := sysdate;

-- Update created\_by field to the username of the person performing the INSERT

:new.created\_by := v\_username;

END;

/

# Oracle / PLSQL: AFTER INSERT Trigger

This Oracle tutorial explains how to **create an AFTER INSERT Trigger** in Oracle with syntax and examples.

## Description

An **AFTER INSERT Trigger** means that Oracle will fire this trigger after the INSERT operation is executed.

## Syntax

The syntax to create an **AFTER INSERT Trigger** in Oracle/PLSQL is:

CREATE [ OR REPLACE ] TRIGGER trigger\_name

AFTER INSERT

ON table\_name

[ FOR EACH ROW ]

DECLARE

-- variable declarations

BEGIN

-- trigger code

EXCEPTION

WHEN ...

-- exception handling

END;

### Parameters or Arguments

**OR REPLACE**

Optional. If specified, it allows you to re-create the trigger is it already exists so that you can change the trigger definition without issuing a [DROP TRIGGER statement](https://www.techonthenet.com/oracle/triggers/drop.php).

**trigger\_name**

The name of the trigger to create.

**AFTER INSERT**

It indicates that the trigger will fire after the INSERT operation is executed.

**table\_name**

The name of the table that the trigger is created on.

## Restrictions

* You can not create an AFTER trigger on a view.
* You can not update the :NEW values.
* You can not update the :OLD values.

## Note

* See also how to create [AFTER DELETE](https://www.techonthenet.com/oracle/triggers/after_delete.php), [AFTER UPDATE](https://www.techonthenet.com/oracle/triggers/after_update.php), [BEFORE DELETE](https://www.techonthenet.com/oracle/triggers/before_delete.php), [BEFORE INSERT](https://www.techonthenet.com/oracle/triggers/before_insert.php), and [BEFORE UPDATE](https://www.techonthenet.com/oracle/triggers/before_update.php) triggers.
* See also how to [drop a trigger](https://www.techonthenet.com/oracle/triggers/drop.php).

## Example

Let's look at an example of how to create an AFTER INSERT trigger using the CREATE TRIGGER statement.

If you had a table created as follows:

CREATE TABLE orders

( order\_id number(5),

quantity number(4),

cost\_per\_item number(6,2),

total\_cost number(8,2)

);

We could then use the CREATE TRIGGER statement to create an AFTER INSERT trigger as follows:

**TIP:** When using SQLPlus, you need to enter slash on a new line after the trigger. Otherwise, the script won't execute.

CREATE OR REPLACE TRIGGER orders\_after\_insert

AFTER INSERT

ON orders

FOR EACH ROW

DECLARE

v\_username varchar2(10);

BEGIN

-- Find username of person performing the INSERT into the table

SELECT user INTO v\_username

FROM dual;

-- Insert record into audit table

INSERT INTO orders\_audit

( order\_id,

quantity,

cost\_per\_item,

total\_cost,

username )

VALUES

( :new.order\_id,

:new.quantity,

:new.cost\_per\_item,

:new.total\_cost,

v\_username );

END;

/

# Oracle / PLSQL: BEFORE UPDATE Trigger

This Oracle tutorial explains how to **create a BEFORE UPDATE Trigger** in Oracle with syntax and examples.

## Description

A **BEFORE UPDATE Trigger** means that Oracle will fire this trigger before the UPDATE operation is executed.

## Syntax

The syntax to create a **BEFORE UPDATE Trigger** in Oracle/PLSQL is:

CREATE [ OR REPLACE ] TRIGGER trigger\_name

BEFORE UPDATE

ON table\_name

[ FOR EACH ROW ]

DECLARE

-- variable declarations

BEGIN

-- trigger code

EXCEPTION

WHEN ...

-- exception handling

END;

### Parameters or Arguments

**OR REPLACE**

Optional. If specified, it allows you to re-create the trigger is it already exists so that you can change the trigger definition without issuing a [DROP TRIGGER statement](https://www.techonthenet.com/oracle/triggers/drop.php).

**trigger\_name**

The name of the trigger to create.

**BEFORE UPDATE**

It indicates that the trigger will fire before the UPDATE operation is executed.

**table\_name**

The name of the table that the trigger is created on.

## Restrictions

* You can not create a BEFORE trigger on a view.
* You can update the :NEW values.
* You can not update the :OLD values.

## Note

* See also how to create [AFTER DELETE](https://www.techonthenet.com/oracle/triggers/after_delete.php), [AFTER INSERT](https://www.techonthenet.com/oracle/triggers/after_insert.php), [AFTER UPDATE](https://www.techonthenet.com/oracle/triggers/after_update.php), [BEFORE DELETE](https://www.techonthenet.com/oracle/triggers/before_delete.php), and [BEFORE INSERT](https://www.techonthenet.com/oracle/triggers/before_insert.php) triggers.
* See also how to [drop a trigger](https://www.techonthenet.com/oracle/triggers/drop.php).

## Example

Let's look at an example of how to create an BEFORE UPDATE trigger using the CREATE TRIGGER statement.

If you had a table created as follows:

CREATE TABLE orders

( order\_id number(5),

quantity number(4),

cost\_per\_item number(6,2),

total\_cost number(8,2),

updated\_date date,

updated\_by varchar2(10)

);

We could then use the CREATE TRIGGER statement to create an BEFORE UPDATE trigger as follows:

**TIP:** When using SQLPlus, you need to enter slash on a new line after the trigger. Otherwise, the script won't execute.

CREATE OR REPLACE TRIGGER orders\_before\_update

BEFORE UPDATE

ON orders

FOR EACH ROW

DECLARE

v\_username varchar2(10);

BEGIN

-- Find username of person performing UPDATE on the table

SELECT user INTO v\_username

FROM dual;

-- Update updated\_date field to current system date

:new.updated\_date := sysdate;

-- Update updated\_by field to the username of the person performing the UPDATE

:new.updated\_by := v\_username;

END;

/

# Oracle / PLSQL: AFTER UPDATE Trigger

This Oracle tutorial explains how to **create an AFTER UPDATE Trigger** in Oracle with syntax and examples.

## Description

An **AFTER UPDATE Trigger** means that Oracle will fire this trigger after the UPDATE operation is executed.

## Syntax

The syntax to create an **AFTER UPDATE Trigger** in Oracle/PLSQL is:

CREATE [ OR REPLACE ] TRIGGER trigger\_name

AFTER UPDATE

ON table\_name

[ FOR EACH ROW ]

DECLARE

-- variable declarations

BEGIN

-- trigger code

EXCEPTION

WHEN ...

-- exception handling

END;

### Parameters or Arguments

**OR REPLACE**

Optional. If specified, it allows you to re-create the trigger is it already exists so that you can change the trigger definition without issuing a [DROP TRIGGER statement](https://www.techonthenet.com/oracle/triggers/drop.php).

**trigger\_name**

The name of the trigger to create.

**AFTER UPDATE**

It indicates that the trigger will fire after the UPDATE operation is executed.

**table\_name**

The name of the table that the trigger is created on.

## Restrictions

* You can not create an AFTER trigger on a view.
* You can not update the :NEW values.
* You can not update the :OLD values.

## Note

* See also how to create [AFTER DELETE](https://www.techonthenet.com/oracle/triggers/after_delete.php), [AFTER INSERT](https://www.techonthenet.com/oracle/triggers/after_insert.php), [BEFORE DELETE](https://www.techonthenet.com/oracle/triggers/before_delete.php), [BEFORE INSERT](https://www.techonthenet.com/oracle/triggers/before_insert.php), and [BEFORE UPDATE](https://www.techonthenet.com/oracle/triggers/before_update.php) triggers.
* See also how to [drop a trigger](https://www.techonthenet.com/oracle/triggers/drop.php).

## Example

Let's look at an example of how to create an AFTER UPDATE trigger using the CREATE TRIGGER statement.

If you had a table created as follows:

CREATE TABLE orders

( order\_id number(5),

quantity number(4),

cost\_per\_item number(6,2),

total\_cost number(8,2)

);

We could then use the CREATE TRIGGER statement to create an AFTER UPDATE trigger as follows:

**TIP:** When using SQLPlus, you need to enter slash on a new line after the trigger. Otherwise, the script won't execute.

CREATE OR REPLACE TRIGGER orders\_after\_update

AFTER UPDATE

ON orders

FOR EACH ROW

DECLARE

v\_username varchar2(10);

BEGIN

-- Find username of person performing UPDATE into table

SELECT user INTO v\_username

FROM dual;

-- Insert record into audit table

INSERT INTO orders\_audit

( order\_id,

quantity\_before,

quantity\_after,

username )

VALUES

( :new.order\_id,

:old.quantity,

:new.quantity,

v\_username );

END;

/

# Oracle / PLSQL: BEFORE DELETE Trigger

This Oracle tutorial explains how to **create a BEFORE DELETE Trigger** in Oracle with syntax and examples.

## Description

A **BEFORE DELETE Trigger** means that Oracle will fire this trigger before the DELETE operation is executed.

## Syntax

The syntax to create a **BEFORE DELETE Trigger** in Oracle/PLSQL is:

CREATE [ OR REPLACE ] TRIGGER trigger\_name

BEFORE DELETE

ON table\_name

[ FOR EACH ROW ]

DECLARE

-- variable declarations

BEGIN

-- trigger code

EXCEPTION

WHEN ...

-- exception handling

END;

### Parameters or Arguments

**OR REPLACE**

Optional. If specified, it allows you to re-create the trigger is it already exists so that you can change the trigger definition without issuing a [DROP TRIGGER statement](https://www.techonthenet.com/oracle/triggers/drop.php).

**trigger\_name**

The name of the trigger to create.

**BEFORE DELETE**

It indicates that the trigger will fire before the DELETE operation is executed.

**table\_name**

The name of the table that the trigger is created on.

## Restrictions

* You can not create a BEFORE trigger on a view.
* You can update the :NEW values.
* You can not update the :OLD values.

## Note

* See also how to create [AFTER DELETE](https://www.techonthenet.com/oracle/triggers/after_delete.php), [AFTER INSERT](https://www.techonthenet.com/oracle/triggers/after_insert.php), [AFTER UPDATE](https://www.techonthenet.com/oracle/triggers/after_update.php), [BEFORE INSERT](https://www.techonthenet.com/oracle/triggers/before_insert.php), and [BEFORE UPDATE](https://www.techonthenet.com/oracle/triggers/before_update.php) triggers.
* See also how to [drop a trigger](https://www.techonthenet.com/oracle/triggers/drop.php).

## Example

Let's look at an example of how to create an BEFORE DELETE trigger using the CREATE TRIGGER statement.

If you had a table created as follows:

CREATE TABLE orders

( order\_id number(5),

quantity number(4),

cost\_per\_item number(6,2),

total\_cost number(8,2)

);

We could then use the CREATE TRIGGER statement to create an BEFORE DELETE trigger as follows:

**TIP:** When using SQLPlus, you need to enter slash on a new line after the trigger. Otherwise, the script won't execute.

CREATE OR REPLACE TRIGGER orders\_before\_delete

BEFORE DELETE

ON orders

FOR EACH ROW

DECLARE

v\_username varchar2(10);

BEGIN

-- Find username of person performing the DELETE on the table

SELECT user INTO v\_username

FROM dual;

-- Insert record into audit table

INSERT INTO orders\_audit

( order\_id,

quantity,

cost\_per\_item,

total\_cost,

delete\_date,

deleted\_by )

VALUES

( :old.order\_id,

:old.quantity,

:old.cost\_per\_item,

:old.total\_cost,

sysdate,

v\_username );

END;

/

# Oracle / PLSQL: AFTER DELETE Trigger

This Oracle tutorial explains how to **create an AFTER DELETE Trigger** in Oracle with syntax and examples.

## Description

An **AFTER DELETE Trigger** means that Oracle will fire this trigger after the DELETE operation is executed.

## Syntax

The syntax to create an **AFTER DELETE Trigger** in Oracle/PLSQL is:

CREATE [ OR REPLACE ] TRIGGER trigger\_name

AFTER DELETE

ON table\_name

[ FOR EACH ROW ]

DECLARE

-- variable declarations

BEGIN

-- trigger code

EXCEPTION

WHEN ...

-- exception handling

END;

### Parameters or Arguments

**OR REPLACE**

Optional. If specified, it allows you to re-create the trigger is it already exists so that you can change the trigger definition without issuing a [DROP TRIGGER statement](https://www.techonthenet.com/oracle/triggers/drop.php).

**trigger\_name**

The name of the trigger to create.

**AFTER DELETE**

It indicates that the trigger will fire after the DELETE operation is executed.

**table\_name**

The name of the table that the trigger is created on.

## Restrictions

* You can not create an AFTER trigger on a view.
* You can not update the :NEW values.
* You can not update the :OLD values.

## Note

* See also how to create [AFTER INSERT](https://www.techonthenet.com/oracle/triggers/after_insert.php), [AFTER UPDATE](https://www.techonthenet.com/oracle/triggers/after_update.php), [BEFORE DELETE](https://www.techonthenet.com/oracle/triggers/before_delete.php), [BEFORE INSERT](https://www.techonthenet.com/oracle/triggers/before_insert.php), and [BEFORE UPDATE](https://www.techonthenet.com/oracle/triggers/before_update.php) triggers.
* See also how to [drop a trigger](https://www.techonthenet.com/oracle/triggers/drop.php).

## Example

Let's look at an example of how to create an AFTER DELETE trigger using the CREATE TRIGGER statement.

If you had a table created as follows:

CREATE TABLE orders

( order\_id number(5),

quantity number(4),

cost\_per\_item number(6,2),

total\_cost number(8,2)

);

We could then use the CREATE TRIGGER statement to create an AFTER DELETE trigger as follows:

**TIP:** When using SQLPlus, you need to enter slash on a new line after the trigger. Otherwise, the script won't execute.

CREATE OR REPLACE TRIGGER orders\_after\_delete

AFTER DELETE

ON orders

FOR EACH ROW

DECLARE

v\_username varchar2(10);

BEGIN

-- Find username of person performing the DELETE on the table

SELECT user INTO v\_username

FROM dual;

-- Insert record into audit table

INSERT INTO orders\_audit

( order\_id,

quantity,

cost\_per\_item,

total\_cost,

delete\_date,

deleted\_by)

VALUES

( :old.order\_id,

:old.quantity,

:old.cost\_per\_item,

:old.total\_cost,

sysdate,

v\_username );

END;

/

# Oracle / PLSQL: DROP TRIGGER Statement

This Oracle tutorial explains how to use the **DROP TRIGGER statement** to drop a trigger in Oracle with syntax and examples.

## Description

Once you have created a trigger in Oracle, you might find that you need to remove it from the database. You can do this with the DROP TRIGGER statement.

## Syntax

The syntax to a **drop a trigger** in Oracle in Oracle/PLSQL is:

DROP TRIGGER trigger\_name;

### Parameters or Arguments

**trigger\_name**

The name of the trigger that you wish to drop.

## Note

* See also how to create [AFTER DELETE](https://www.techonthenet.com/oracle/triggers/after_delete.php), [AFTER INSERT](https://www.techonthenet.com/oracle/triggers/after_insert.php), [AFTER UPDATE](https://www.techonthenet.com/oracle/triggers/after_update.php), [BEFORE DELETE](https://www.techonthenet.com/oracle/triggers/before_delete.php), [BEFORE INSERT](https://www.techonthenet.com/oracle/triggers/before_insert.php), and [BEFORE UPDATE](https://www.techonthenet.com/oracle/triggers/before_update.php) triggers.

## Example

Let's look at an example of how to drop a trigger in Oracle.

For example:

DROP TRIGGER orders\_before\_insert;

This example uses the ALTER TRIGGER statement to drop the trigger called orders\_before\_insert.