**CURSOR**

A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.

There are two types of cursors −

* Implicit cursors
* Explicit cursors.

**Implicit cursor**

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

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

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

**1. %FOUND** : Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.

**2. %NOTFOUND :** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

**3. %ISOPEN :** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

**4. %ROWCOUNT :** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

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

> Select \* from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

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

> DECLARE

total\_rows number(2);

BEGIN

UPDATE customers

SET salary = salary + 500;

IF sql%notfound THEN

dbms\_output.put\_line('no customers selected');

ELSIF sql%found THEN

total\_rows := sql%rowcount;

dbms\_output.put\_line( total\_rows || ' customers selected ');

END IF;

END;

/

Output : 6 customers selected

PL/SQL procedure successfully completed.

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

> Select \* from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2500.00 |

| 2 | Khilan | 25 | Delhi | 2000.00 |

| 3 | kaushik | 23 | Kota | 2500.00 |

| 4 | Chaitali | 25 | Mumbai | 7000.00 |

| 5 | Hardik | 27 | Bhopal | 9000.00 |

| 6 | Komal | 22 | MP | 5000.00 |

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

**Emplicit Cursor**

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

The syntax for creating an explicit cursor is −

1. Declaring the cursor for initializing the memory.
2. Opening the cursor for allocating the memory.
3. Fetching the cursor for retrieving the data.
4. Closing the cursor to release the allocated memory.

1. Declaring the Cursor

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

**CURSOR c\_customers IS**

**SELECT id, name, address FROM customers;**

2.Opening the Cursor

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

**OPEN c\_customers;**

3. Fetching the Cursor

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

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

4. Closing the Cursor

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

**CLOSE c\_customers;**

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

> DECLARE

c\_id customers.id%type;

c\_name customerS.No.ame%type;

c\_addr customers.address%type;

**CURSOR c\_customers is**

**SELECT id, name, address FROM customers;**

BEGIN

**OPEN c\_customers;**

LOOP

**FETCH c\_customers into c\_id, c\_name, c\_addr;**

EXIT WHEN c\_customers%notfound;

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

END LOOP;

**CLOSE c\_customers;**

END;

/

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

Output: 1 Ramesh Ahmedabad

2 Khilan Delhi

3 kaushik Kota

4 Chaitali Mumbai

5 Hardik Bhopal

6 Komal MP

PL/SQL procedure successfully completed.

**Transaction**

A database transaction is an atomic unit of work that may consist of one or more related SQL statements. It is called atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.

A successfully executed SQL statement and a committed transaction are not same. Even if an SQL statement is executed successfully, unless the transaction containing the statement is committed, it can be rolled back and all changes made by the statement(s) can be undone.

A transaction has a beginning and an end.

1. The first SQL statement is performed after connecting to the database.

2. At each new SQL statement issued after a transaction is completed.

**1. Committing a Transaction :** A transaction is made permanent by issuing the SQL command COMMIT. The general syntax for the COMMIT command is −

> **COMMIT;**

For example :

> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

> COMMIT;

**2. Rolling Back Transactions :** Changes made to the database without COMMIT could be undone using the ROLLBACK command. The general syntax for the ROLLBACK command is −

> **ROLLBACK [TO SAVEPOINT < savepoint\_name>];**

For Example :

When a transaction is aborted due to some unprecedented situation, like system failure, the entire transaction since a commit is automatically rolled back. If you are not using savepoint, then simply use the following statement to rollback all the changes −

> ROLLBACK;

**3. Savepoints :** Savepoints are sort of markers that help in splitting a long transaction into smaller units by setting some checkpoints. By setting savepoints within a long transaction, you can roll back to a checkpoint if required. This is done by issuing the SAVEPOINT command. The general syntax for the SAVEPOINT command is −

> **SAVEPOINT < savepoint\_name >;**

For example :

> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (7, 'Rajnish', 27, 'HP', 9500.00 );

> INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (8, 'Riddhi', 21, 'WB', 4500.00 );

> SAVEPOINT sav1;

> UPDATE CUSTOMERS

SET SALARY = SALARY + 1000;

ROLLBACK TO sav1;

> UPDATE CUSTOMERS

SET SALARY = SALARY + 1000

WHERE ID = 7;

> UPDATE CUSTOMERS

SET SALARY = SALARY + 1000

WHERE ID = 8;

> COMMIT;

ROLLBACK TO sav1 − This statement rolls back all the changes up to the point, where you had marked savepoint sav1.

After that, the new changes that you make will start.

**4. Automatic Transaction Control :** To execute a COMMIT automatically whenever an INSERT, UPDATE or DELETE command is executed, you can set the AUTOCOMMIT environment variable as −

> **SET AUTOCOMMIT ON;**

You can turn-off the auto commit mode using the following command −

> **SET AUTOCOMMIT OFF;**

**Collections**

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

PL/SQL provides three collection types −

1. Index-by tables or Associative array.
2. Nested table.
3. Variable-size array or Varray.

**1. Index-By Table**

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

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

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

**table\_name type\_name;**

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

> DECLARE

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

**salary\_list salary;**

name VARCHAR2(20);

BEGIN

-- adding elements to the table

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

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

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

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

-- printing the table

name := salary\_list.FIRST;

WHILE name IS NOT null LOOP

dbms\_output.put\_line('Salary of ' || name || ' is ' || TO\_CHAR(salary\_list(name)));

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

END LOOP;

END;

/

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

Output : Salary of James is 78000

Salary of Martin is 100000

Salary of Minakshi is 75000

Salary of Rajnish is 62000

PL/SQL procedure successfully completed

Example : Elements of an index-by table could also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as −

> Select \* from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

> DECLARE

CURSOR c\_customers is

select name from customers;

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

**name\_list c\_list;**

counter integer :=0;

BEGIN

FOR n IN c\_customers LOOP

counter := counter +1;

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

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

END LOOP;

END;

/

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

Output : Customer(1): Ramesh

Customer(2): Khilan

Customer(3): kaushik

Customer(4): Chaitali

Customer(5): Hardik

Customer(6): Komal

PL/SQL procedure successfully completed.

**2. Nested Tables**

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

An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.

An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A nested table is created using the following syntax −

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

**table\_name type\_name;**

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

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

Example : The following examples illustrate the use of nested table −

> DECLARE

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

TYPE grades IS TABLE OF INTEGER;

names names\_table;

marks grades;

total integer;

BEGIN

names := names\_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');

marks:= grades(98, 97, 78, 87, 92);

total := names.count;

dbms\_output.put\_line('Total '|| total || ' Students');

FOR i IN 1 .. total LOOP

dbms\_output.put\_line('Student:'||names(i)||', Marks:' || marks(i));

end loop;

END;

/

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

Ouput : Total 5 Students

Student:Kavita, Marks:98

Student:Pritam, Marks:97

Student:Ayan, Marks:78

Student:Rishav, Marks:87

Student:Aziz, Marks:92

PL/SQL procedure successfully completed.

Example : Elements of a nested table can also be a %ROWTYPE of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as −

> Select \* from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

> DECLARE

CURSOR c\_customers is

SELECT name FROM customers;

TYPE c\_list IS TABLE of customerS.No.ame%type;

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

counter integer :=0;

BEGIN

FOR n IN c\_customers LOOP

counter := counter +1;

name\_list.extend;

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

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

END LOOP;

END;

/

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

Output : Customer(1): Ramesh

Customer(2): Khilan

Customer(3): kaushik

Customer(4): Chaitali

Customer(5): Hardik

Customer(6): Komal

PL/SQL procedure successfully completed.

**Collection Methods**

PL/SQL provides the built-in collection methods that make collections easier to use. The following table lists the methods and their purpose −

1. EXISTS(n) :

Returns TRUE if the nth element in a collection exists; otherwise returns FALSE.

2. COUNT

Returns the number of elements that a collection currently contains.

3. LIMIT

Checks the maximum size of a collection.

4. FIRST

Returns the first (smallest) index numbers in a collection that uses the integer subscripts.

5. LAST

Returns the last (largest) index numbers in a collection that uses the integer subscripts.

6. PRIOR(n)

Returns the index number that precedes index n in a collection.

7. NEXT(n)

Returns the index number that succeeds index n.

8. EXTEND

Appends one null element to a collection.

9. EXTEND(n)

Appends n null elements to a collection.

10. EXTEND(n,i)

Appends n copies of the ith element to a collection.

11. TRIM

Removes one element from the end of a collection.

12. TRIM(n)

Removes n elements from the end of a collection.

13.DELETE

Removes all elements from a collection, setting COUNT to 0.

14. DELETE(n)

Removes the nth element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If n is null, DELETE(n) does nothing.

15. DELETE(m,n)

Removes all elements in the range m..n from an associative array or nested table. If m is larger than n or if m or n is null, DELETE(m,n) does nothing.

**Collection Exceptions**

The following table provides the collection exceptions and when they are raised −

1. COLLECTION\_IS\_NULL : You try to operate on an atomically null collection.

2. NO\_DATA\_FOUND : A subscript designates an element that was deleted, or a nonexistent element of an associative array.

3. SUBSCRIPT\_BEYOND\_COUNT : A subscript exceeds the number of elements in a collection.

4. SUBSCRIPT\_OUTSIDE\_LIMIT : A subscript is outside the allowed range.

5. VALUE\_ERROR : A subscript is null or not convertible to the key type. This exception might occur if the key is defined as a PLS\_INTEGER range, and the subscript is outside this range.

**Composite Data types**

The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values. We will focus on the SCALAR and the LOB data types.

1. Scalar

Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.

2. Large Object (LOB)

Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.

3. Composite

Data items that have internal components that can be accessed individually. For example, collections and records.

4. Reference

Pointers to other data items.

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

PL/SQL Scalar Data Types and Subtypes come under the following categories −

1. Numeric : Numeric values on which arithmetic operations are performed.

2. Character : Alphanumeric values that represent single characters or strings of characters.

3. Boolean : Logical values on which logical operations are performed.

4. Datetime : Dates and times.

PL/SQL provides subtypes of data types. For example, the data type NUMBER has a subtype called INTEGER. You can use the subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding the PL/SQL code in another program, such as a Java program.

**PL/SQL Numeric Data Types and Subtypes**

Following table lists out the PL/SQL pre-defined numeric data types and their sub-types −

1. PLS\_INTEGER

Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits

2. BINARY\_INTEGER

Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits

3. BINARY\_FLOAT

Single-precision IEEE 754-format floating-point number

4. BINARY\_DOUBLE

Double-precision IEEE 754-format floating-point number

5. NUMBER(prec, scale)

Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0

6. DEC(prec, scale)

ANSI specific fixed-point type with maximum precision of 38 decimal digits

7. DECIMAL(prec, scale)

IBM specific fixed-point type with maximum precision of 38 decimal digits

8. NUMERIC(pre, secale)

Floating type with maximum precision of 38 decimal digits

9. DOUBLE PRECISION

ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)

10. FLOAT

ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)

11. INT

ANSI specific integer type with maximum precision of 38 decimal digits

12. INTEGER

ANSI and IBM specific integer type with maximum precision of 38 decimal digits

13. SMALLINT

ANSI and IBM specific integer type with maximum precision of 38 decimal digits

14. REAL

Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Example : Following is a valid declaration −

> DECLARE

num1 INTEGER;

num2 REAL;

num3 DOUBLE PRECISION;

BEGIN

null;

END;

/

When the above code is compiled and executed, it produces the following result −

Output : PL/SQL procedure successfully completed.

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

Following is the detail of PL/SQL pre-defined character data types and their sub-types −

S.No Data Type & Description

1. CHAR

Fixed-length character string with maximum size of 32,767 bytes

2. VARCHAR2

Variable-length character string with maximum size of 32,767 bytes

3. RAW

Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL

4. NCHAR

Fixed-length national character string with maximum size of 32,767 bytes

5. NVARCHAR2

Variable-length national character string with maximum size of 32,767 bytes

6. LONG

Variable-length character string with maximum size of 32,760 bytes

7. LONG RAW

Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL

8. ROWID

Physical row identifier, the address of a row in an ordinary table

9. UROWID

Universal row identifier (physical, logical, or foreign row identifier).

**PL/SQL Boolean Data Types**

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

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

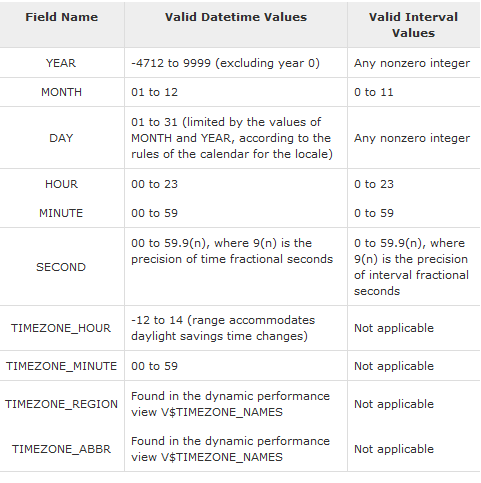
* SQL statements
* Built-in SQL functions (such as TO\_CHAR)
* PL/SQL functions invoked from SQL statements

**PL/SQL Datetime and Interval Types**

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

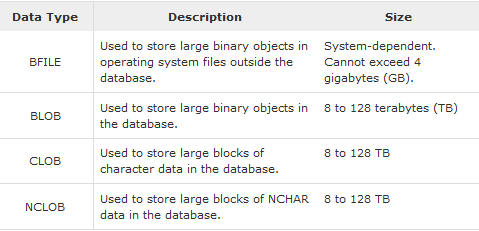
The default date format is set by the Oracle initialization parameter NLS\_DATE\_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year. For example, 01-OCT-12.

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



**PL/SQL Large Object (LOB) Data Types**

Large Object (LOB) data types refer to large data items such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Following are the predefined PL/SQL LOB data types −



**PL/SQL User-Defined Subtypes**

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

PL/SQL predefines several subtypes in package STANDARD. For example, PL/SQL predefines the subtypes CHARACTER and INTEGER as follows −

**SUBTYPE CHARACTER IS CHAR;**

**SUBTYPE INTEGER IS NUMBER(38,0);**

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype −

> DECLARE

SUBTYPE name IS char(20);

SUBTYPE message IS varchar2(100);

salutation name;

greetings message;

BEGIN

salutation := 'Reader ';

greetings := 'Welcome to the World of PL/SQL';

dbms\_output.put\_line('Hello ' || salutation || greetings);

END;

/

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

Output : Hello Reader Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.

**NULLs in PL/SQL**

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

**Procedures** **and Functions**

A PL/SQL subprogram is a named PL/SQL block that can be invoked with a set of parameters. A subprogram can be either a procedure or a function. Typically, you use a procedure to perform an action and a function to compute and return a value.

A subprogram can be created −

1. At the schema level.

At the schema level, subprogram is a standalone subprogram. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

2. Inside a package.

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

3. Inside a PL/SQL block.

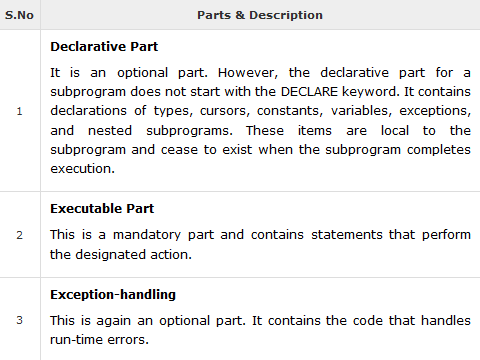
PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −

a) Functions − These subprograms return a single value; mainly used to compute and return a value.

b) Procedures − These subprograms do not return a value directly; mainly used to perform an action.

**Parts of a PL/SQL Subprogram**

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



**Creating a Procedure**

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

**CREATE [OR REPLACE] PROCEDURE procedure\_name**

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

**{IS | AS}**

**BEGIN**

**< procedure\_body >**

**END procedure\_name;**

1. procedure-name specifies the name of the procedure.

2. [OR REPLACE] option allows the modification of an existing procedure.

3. The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

4. procedure-body contains the executable part.

5. The AS keyword is used instead of the IS keyword for creating a standalone procedure.

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

CREATE OR REPLACE PROCEDURE greetings

AS

BEGIN

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

END;

/

When the above code is executed using the SQL prompt, it will produce the following result Output : Procedure created.

**Executing a Standalone Procedure**

A standalone procedure can be called in two ways −

1. Using the EXECUTE keyword.

2. Calling the name of the procedure from a PL/SQL block.

The above procedure named 'greetings' can be called with the EXECUTE keyword as −

> EXECUTE greetings;

Output for the above call will display −

Hello World

PL/SQL procedure successfully completed.

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

> BEGIN

greetings;

END;

/

Output for the above call will display −

Hello World

PL/SQL procedure successfully completed.

**Deleting a Standalone Procedure**

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

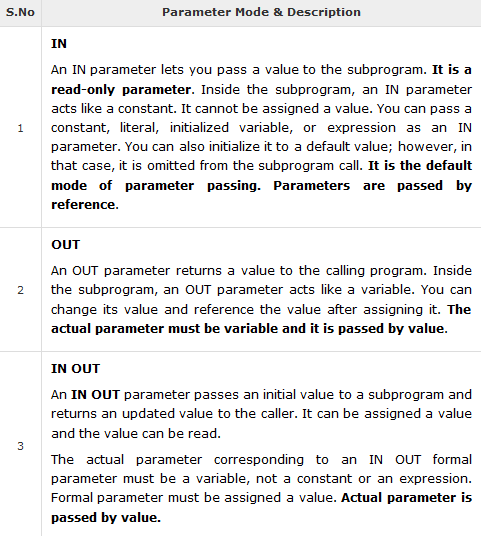
**DROP PROCEDURE procedure-name;**

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

> DROP PROCEDURE greetings;

**Parameter Modes in PL/SQL Subprograms**

The following table lists out the parameter modes in PL/SQL subprograms −



**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;

/

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

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

**IN & OUT Mode Example 2 :** This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

> DECLARE

a number;

PROCEDURE squareNum(x IN OUT number) IS

BEGIN

x := x \* x;

END;

BEGIN

a:= 23;

squareNum(a);

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

END;

/

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

Square of (23): 529

PL/SQL procedure successfully completed.

**Creating a Function**

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

CREATE [OR REPLACE] FUNCTION function\_name

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

RETURN return\_datatype

{IS | AS}

BEGIN

< function\_body >

END [function\_name];

1. function-name specifies the name of the function.

2. [OR REPLACE] option allows the modification of an existing function.

3. The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.

4. The function must contain a return statement.

5. The RETURN clause specifies the data type you are going to return from the function.

6. function-body contains the executable part.

7. The AS keyword is used instead of the IS keyword for creating a standalone function.

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

We will use the CUSTOMERS table, which we had created in the PL/SQL Variables.

> Select \* from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

> CREATE OR REPLACE FUNCTION totalCustomers

RETURN number IS

total number(2) := 0;

BEGIN

SELECT count(\*) into total

FROM customers;

RETURN total;

END;

/

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

Function created.

**Calling a Function**

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

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

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

> DECLARE

c number(2);

BEGIN

c := totalCustomers();

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

END;

/

Outout when the above code is executed at the SQL prompt, it produces the following result − Total no. of Customers: 6

PL/SQL procedure successfully completed.

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

> DECLARE

a number;

b number;

c number;

FUNCTION findMax(x IN number, y IN number)

RETURN number

IS

z number;

BEGIN

IF x > y THEN

z:= x;

ELSE

Z:= y;

END IF;

RETURN z;

END;

BEGIN

a:= 23;

b:= 45;

c := findMax(a, b);

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

END;

/

Output when the above code is executed at the SQL prompt, it produces the following result − Maximum of (23,45): 45

PL/SQL procedure successfully completed.

**PL/SQL Recursive Functions**

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

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

n! = n\*(n-1)!

= n\*(n-1)\*(n-2)!

...

= n\*(n-1)\*(n-2)\*(n-3)... 1

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

> DECLARE

num number;

factorial number;

FUNCTION fact(x number)

RETURN number

IS

f number;

BEGIN

IF x=0 THEN

f := 1;

ELSE

f := x \* fact(x-1);

END IF;

RETURN f;

END;

BEGIN

num:= 6;

factorial := fact(num);

dbms\_output.put\_line(' Factorial '|| num || ' is ' || factorial);

END;

/

Output when the above code is executed at the SQL prompt, it produces the following result − Factorial 6 is 720

PL/SQL procedure successfully completed.

**Exceptions Handling**

An exception is an error condition during a program execution. PL/SQL supports programmers to catch such conditions using **EXCEPTION** block in the program and an appropriate action is taken against the error condition. There are two types of exceptions –

* System-defined exceptions.
* User-defined exceptions.

Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** –

DECLARE

<declarations section>

BEGIN

<executable command(s)>

EXCEPTION

<exception handling goes here >

WHEN exception1 THEN

exception1-handling-statements

WHEN exception2 THEN

exception2-handling-statements

WHEN exception3 THEN

exception3-handling-statements

........

WHEN others THEN

exception3-handling-statements

END;

### Example : Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters −

DECLARE

c\_id customers.id%type := 8;

c\_name customerS.Name%type;

c\_addr customers.address%type;

BEGIN

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

FROM customers

WHERE id = c\_id;

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

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

EXCEPTION

WHEN no\_data\_found THEN

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

WHEN others THEN

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

END;

/

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

No such customer!

PL/SQL procedure successfully completed.

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

## Raising Exceptions

Exceptions are raised by the database server automatically whenever there is any internal database error, but exceptions can be raised explicitly by the programmer by using the command **RAISE**. Following is the simple syntax for raising an exception −

DECLARE

exception\_name EXCEPTION;

BEGIN

IF condition THEN

RAISE exception\_name;

END IF;

EXCEPTION

WHEN exception\_name THEN

statement;

END;

You can use the above syntax in raising the Oracle standard exception or any user-defined exception. In the next section, we will give you an example on raising a user-defined exception. You can raise the Oracle standard exceptions in a similar way.

## User-defined Exceptions

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

The syntax for declaring an exception is −

DECLARE

my-exception EXCEPTION;

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

DECLARE

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

c\_name customerS.Name%type;

c\_addr customers.address%type;

-- user defined exception

ex\_invalid\_id EXCEPTION;

BEGIN

IF c\_id <= 0 THEN

RAISE ex\_invalid\_id;

ELSE

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

FROM customers

WHERE id = c\_id;

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

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

END IF;

EXCEPTION

WHEN ex\_invalid\_id THEN

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

WHEN no\_data\_found THEN

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

WHEN others THEN

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

END;

/

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

Enter value for cc\_id: -6 (let's enter a value -6)

old 2: c\_id customers.id%type := &cc\_id;

new 2: c\_id customers.id%type := -6;

ID must be greater than zero!

PL/SQL procedure successfully completed.

## Pre-defined Exceptions

PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program. For example, the predefined exception NO\_DATA\_FOUND is raised when a SELECT INTO statement returns no rows. The following table lists few of the important pre-defined exceptions –

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

**Package**

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

A package will have two mandatory parts –

* Package specification
* Package body or definition

## Package Specification

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

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

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

CREATE PACKAGE cust\_sal AS

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

END cust\_sal;

/

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

Package created.

## Package Body

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

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

CREATE OR REPLACE PACKAGE BODY cust\_sal AS

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

c\_sal customers.salary%TYPE;

BEGIN

SELECT salary INTO c\_sal

FROM customers

WHERE id = c\_id;

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

END find\_sal;

END cust\_sal;

/

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

Package body created.

## Using the Package Elements

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

package\_name.element\_name;

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

DECLARE

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

BEGIN

cust\_sal.find\_sal(code);

END;

/

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

Enter value for cc\_id: 1

Salary: 3000

PL/SQL procedure successfully completed.

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

Select \* from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 3000.00 |

| 2 | Khilan | 25 | Delhi | 3000.00 |

| 3 | kaushik | 23 | Kota | 3000.00 |

| 4 | Chaitali | 25 | Mumbai | 7500.00 |

| 5 | Hardik | 27 | Bhopal | 9500.00 |

| 6 | Komal | 22 | MP | 5500.00 |

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

### The Package Specification

CREATE OR REPLACE PACKAGE c\_package AS

-- Adds a customer

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

c\_name customerS.No.ame%type,

c\_age customers.age%type,

c\_addr customers.address%type,

c\_sal customers.salary%type);

-- Removes a customer

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

--Lists all customers

PROCEDURE listCustomer;

END c\_package;

/

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

Package created.

### Creating the Package Body

CREATE OR REPLACE PACKAGE BODY c\_package AS

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

c\_name customerS.No.ame%type,

c\_age customers.age%type,

c\_addr customers.address%type,

c\_sal customers.salary%type)

IS

BEGIN

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

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

END addCustomer;

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

BEGIN

DELETE FROM customers

WHERE id = c\_id;

END delCustomer;

PROCEDURE listCustomer IS

CURSOR c\_customers is

SELECT name FROM customers;

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

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

counter integer :=0;

BEGIN

FOR n IN c\_customers LOOP

counter := counter +1;

name\_list.extend;

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

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

END LOOP;

END listCustomer;

END c\_package;

/

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

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

Package body created.

### Using The Package

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

DECLARE

code customers.id%type:= 8;

BEGIN

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

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

c\_package.listcustomer;

c\_package.delcustomer(code);

c\_package.listcustomer;

END;

/

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

Customer(1): Ramesh

Customer(2): Khilan

Customer(3): kaushik

Customer(4): Chaitali

Customer(5): Hardik

Customer(6): Komal

Customer(7): Rajnish

Customer(8): Subham

Customer(1): Ramesh

Customer(2): Khilan

Customer(3): kaushik

Customer(4): Chaitali

Customer(5): Hardik

Customer(6): Komal

Customer(7): Rajnish

PL/SQL procedure successfully completed

**Trigger**

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −

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

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

### Benefits of Triggers

Triggers can be written for the following purposes −

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

## Creating Triggers

The syntax for creating a trigger is −

CREATE [OR REPLACE ] TRIGGER trigger\_name

{BEFORE | AFTER | INSTEAD OF }

{INSERT [OR] | UPDATE [OR] | DELETE}

[OF col\_name]

ON table\_name

[REFERENCING OLD AS o NEW AS n]

[FOR EACH ROW]

WHEN (condition)

DECLARE

Declaration-statements

BEGIN

Executable-statements

EXCEPTION

Exception-handling-statements

END;

Where,

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

### Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

Select \* from customers;

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

| ID | NAME | AGE | ADDRESS | SALARY |

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

| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |

| 2 | Khilan | 25 | Delhi | 1500.00 |

| 3 | kaushik | 23 | Kota | 2000.00 |

| 4 | Chaitali | 25 | Mumbai | 6500.00 |

| 5 | Hardik | 27 | Bhopal | 8500.00 |

| 6 | Komal | 22 | MP | 4500.00 |

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

The following program creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

CREATE OR REPLACE TRIGGER display\_salary\_changes

BEFORE DELETE OR INSERT OR UPDATE ON customers

FOR EACH ROW

WHEN (NEW.ID > 0)

DECLARE

sal\_diff number;

BEGIN

sal\_diff := :NEW.salary - :OLD.salary;

dbms\_output.put\_line('Old salary: ' || :OLD.salary);

dbms\_output.put\_line('New salary: ' || :NEW.salary);

dbms\_output.put\_line('Salary difference: ' || sal\_diff);

END;

/

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

Trigger created.

The following points need to be considered here −

* OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
* If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
* The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

## Triggering a Trigger

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

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)

VALUES (7, 'Kriti', 22, 'HP', 7500.00 );

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

Old salary:

New salary: 7500

Salary difference:

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

UPDATE customers

SET salary = salary + 500

WHERE id = 2;

When a record is updated in the CUSTOMERS table, the above create trigger, **display\_salary\_changes** will be fired and it will display the following result −

Old salary: 1500

New salary: 2000

Salary difference: 500