PL/SQL

Procedural Language/Structured Query Language

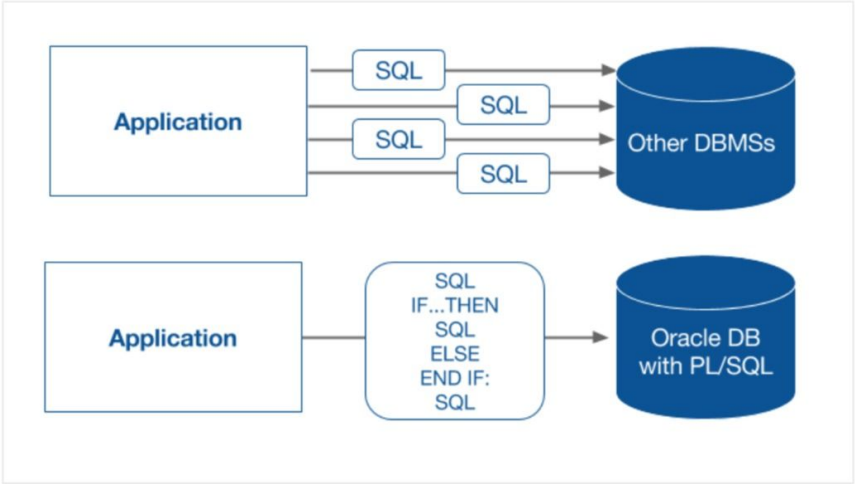
What is PL/SQL?

PL/SQL is an extension of SQL that includes programming constructs like loops, conditional statements, and error handling.

Programming Language used for working

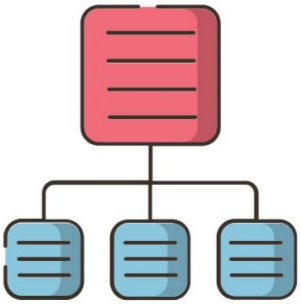
- PL/SQL is used to create custom procedures, functions, and triggers that can manipulate and process data within a database.
- PL/SQL can improve the performance of database operations by reducing the need for multiple round trips between the application and the database server.
- PL/SQL code can be stored as stored procedures and functions, making it reusable across different parts of an application.

WHY?



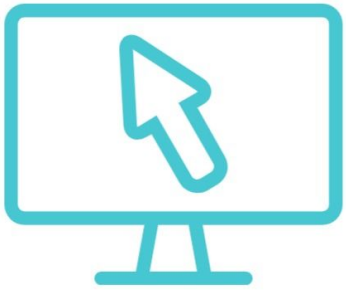
HOW?

Triggers



Stored Procedure

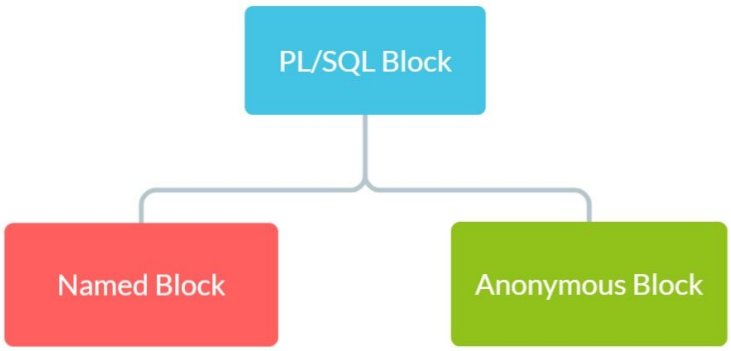
Stored Function



Cursor

Basics Of PL/SQL Block

- PL/SQL code or program is grouped into structure called a Block.
- Named Block is given some name to identify it.
- If PL/SQL block is not given any name then it is called as Anonymous block.

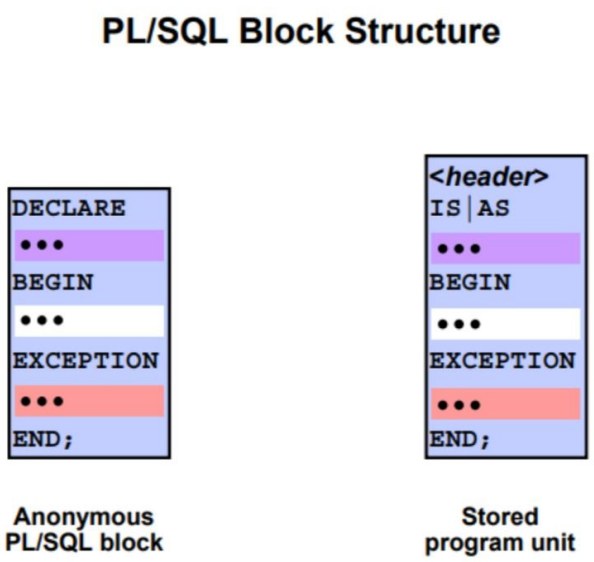


Structure of PL/SQL Block

Declaration:[optional]
This section is used to declare and initialize variables.

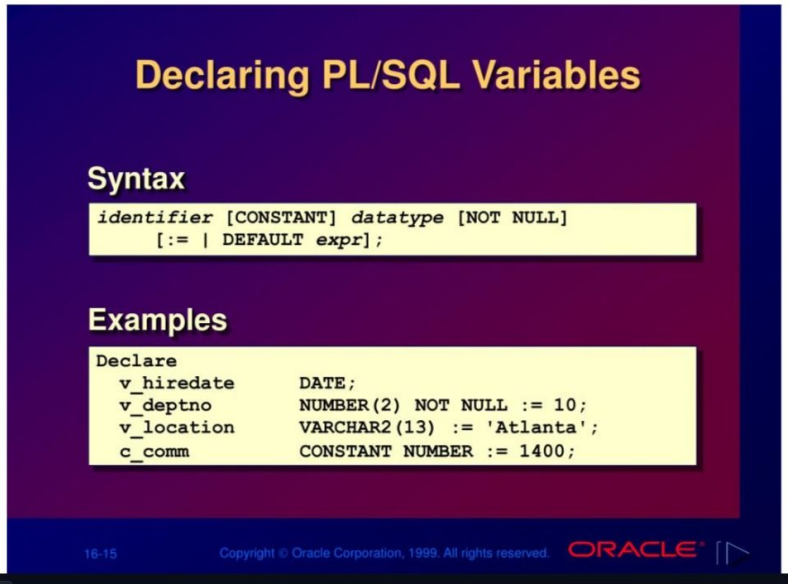
Executable Command:[mandatory]
This section contains various SQL and PL/SQL statements for various functionalities, looping and conditional statements.

Exception Handling:[optional]
This section is used to handles errors that arise during execution of statements.



Variables in PL/SQL Block

- In PL/SQL variables contain values resulting from queries or expression.
- Variables are declared in declaration section of PL/SQL block and must have valid data types.

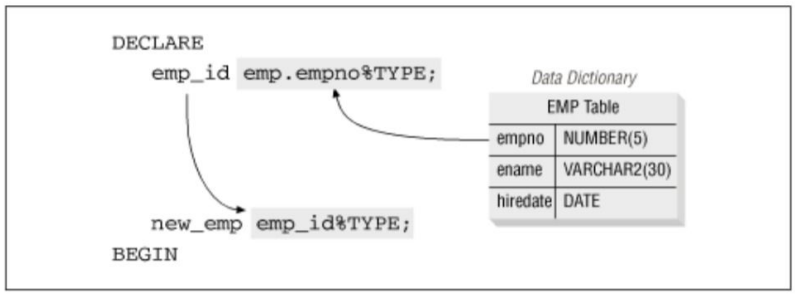


Anchored Data Type

- A variable declared having a anchored data type indicates that a ,variable data type is determined based on the data type of another object.
- This object can be another variable or a column of the table.

Syntax

```
Variable-Name Typed-attribute %Type
```



Declaring Constant

```
2  •  DECLARE
3        var_pi    CONSTANT NUMBER (7,6) := 3.1415;
4     BEGIN
5        DBMS_OUTPUT.put_line (var_pi);
6     END;
```

DBMS Output

Data Grid | DBMS Output

Polling Frequency: 5 seconds

```
1  3.1415
```

A constant must be initialized at declaration time

Anonymous Block

Example : Student Table

```
SQL> select * from student;

NAME
--------------------
Hello
World
Good
Morning
Morning
Morning

6 rows selected.
```

```
SQL> DECLARE
  2  SNAME VARCHAR2(20);
  3  BEGIN
  4  SELECT NAME INTO SNAME FROM STUDENT WHERE NAME LIKE 'H%';
  5  DBMS_OUTPUT.PUT_LINE(SNAME);
  6  END;
  7  /
Hello

PL/SQL procedure successfully completed.
```

NOTE :

1. You can create an SQL script file in specific folder with .sql extension.
2. Run that file with @<path>

PL/SQL Conditional Statements

The IF statement allows you to either execute or skip a sequence of statements, depending on a condition.

```
IF condition THEN
    statements;
END IF;
```

```
IF condition THEN
    statements;
ELSE
    else_statements;
END IF;
```

```
IF condition_1 THEN
  statements_1
ELSIF condition_2 THEN
  statements_2
[ ELSIF condition_3 THEN
    statements_3
]
...
[ ELSE
    else_statements
]
END IF;
```

```
IF condition_1 THEN
    IF condition_2 THEN
        nested_if_statements;
    END IF;
ELSE
    else_statements;
END IF;
```

```
DECLARE
  no1       NUMBER;
BEGIN
  no1 := &no1;
  IF no1>0 THEN
    DBMS_OUTPUT.PUT_LINE(no1||'is positive');
  END IF;
END;
/
```

```
SQL> @D:\PL-SQL\demo_if.sql
Enter value for no1: 1
old   4:   no1 := &no1;
new   4:   no1 := 1;
1is positive

PL/SQL procedure successfully completed.

SQL> @D:\PL-SQL\demo_if.sql
Enter value for no1: -2
old   4:   no1 := &no1;
new   4:   no1 := -2;

PL/SQL procedure successfully completed.
```

```
DECLARE
  no1       NUMBER;
BEGIN
  no1 := &no1;
  IF MOD(no1,2)=0 THEN
    DBMS_OUTPUT.PUT_LINE(no1||'is EVEN');
  ELSE
    DBMS_OUTPUT.PUT_LINE(no1||'is ODD');
  END IF;
END;
/
```

```
SQL> @D:\PL-SQL\demo_if_else.sql
Enter value for no1: 4
old   4:   no1 := &no1;
new   4:   no1 := 4;
4is EVEN

PL/SQL procedure successfully completed.

SQL> @D:\PL-SQL\demo_if_else.sql
Enter value for no1: 5
old   4:   no1 := &no1;
new   4:   no1 := 5;
5is ODD
```

Basic Loop in PL/SQL

- The LOOP statement executes the statements in its body and returns control to the top of the loop.
- Typically, the body of the loop contains at least one EXIT or EXIT WHEN statement for terminating the loop. Otherwise, the loop becomes an **infinite loop**.

```
LOOP
    EXIT;
END LOOP;
```

NOTE : instead of **EXIT** we can also use

```
DECLARE
NUM NUMBER:=1;
BEGIN
    LOOP
        IF MOD(NUM,2)=0 THEN
        DBMS_OUTPUT.PUT_LINE(NUM);
        END IF;
        NUM:=NUM+1;
        EXIT WHEN NUM > 10;
    END LOOP;
END;
/
```

```
SQL> @D:\PL-SQL\LOOP1.SQL
2
4
6
8
10

PL/SQL procedure successfully completed.
```

```
DECLARE
NUM NUMBER:=1;
BEGIN
    LOOP
        IF NUM > 10 THEN
            EXIT;
        END IF;
        DBMS_OUTPUT.PUT_LINE(NUM);
        NUM:=NUM+1;
    END LOOP;
END;
```

```
SQL> set serveroutput on;
SQL> @D:\PL-SQL\LOOP2.SQL
1
2
3
4
5
6
7
8
9
10

PL/SQL procedure successfully completed.
```

For Loop in PL/SQL

PL/SQL `FOR LOOP` executes a sequence of statements a specified number of times.

```
FOR index IN lower_bound .. upper_bound
LOOP
    statements;
END LOOP;
```

The `index` is an implicit variable. It is local to the `FOR`

```
DECLARE
BEGIN
    FOR NUM IN 1..5
    LOOP
        DBMS_OUTPUT.PUT_LINE(NUM);
    END LOOP;
END;
/
```

```
SQL> @D:\PL-SQL\FOR1.SQL
1
2
3
4
5

PL/SQL procedure successfully completed.
```

NOTE : To display numbers in reverse order we need to add just REVERSE keyword in FOR loop syntax after IN keyword.

While Loop in PL/SQL

PL/SQL `WHILE` loop is a control structure that repeatedly executes a code block as long as a specific condition remains true.

```
WHILE condition
LOOP
    statements;
END LOOP;
```

```
DECLARE
I NUMBER:=1;
BEGIN
    WHILE I < 5
    LOOP
        DBMS_OUTPUT.PUT_LINE(I);
        I:=I+1;
    END LOOP;
END;
/
```

```
SQL> @D:\PL-SQL\WHILE.SQL
1
2
3
4

PL/SQL procedure successfully completed.
```

Stored Procedure

- Stored Procedure is named PL/SQL block that accept some input as a parameter to perform a specific task and may or may not return a value.
- It is used to perform one or more DML operations on a database

```
CREATE OR REPLACE PROCEDURE
<procedure_name>
        (
        <parameter1 IN/OUT <datatype>
        ..
        .
        )
[ IS | AS ]
        <declaration_part>
BEGIN
        <execution part>
EXCEPTION
        <exception handling part>
END;
```

Stored Procedure Syntax

Stored Procedure Execution Process

Create Stored Procedure

Execute

Display O/P

Example

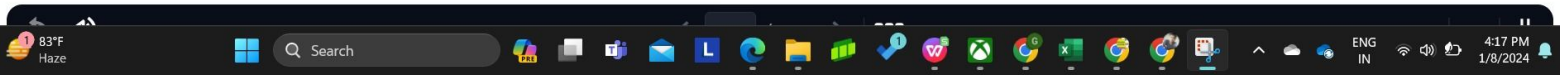
```
CREATE OR REPLACE PROCEDURE
INCT_SALARY (E IN VARCHAR , AMT IN NUMBER, S OUT NUMBER)
IS
BEGIN
    UPDATE EMP SET SALARY=SALARY+AMT WHERE EID=E;
    COMMIT;
    SELECT SALARY INTO S FROM EMP WHERE EID=E;
END;
/
```

Procedure to update salary of Specific employee

EXAMPLE

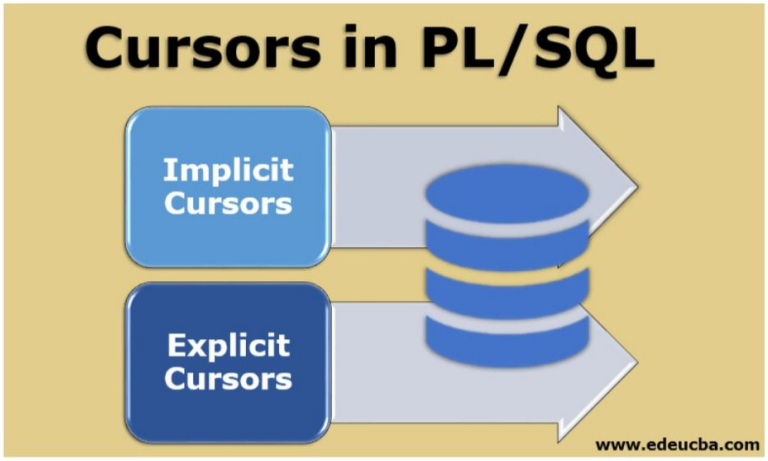
```
CREATE OR REPLACE PROCEDURE
GET_EMP (E IN VARCHAR)
IS
REC EMP%ROWTYPE;
BEGIN
    SELECT * INTO REC FROM EMP WHERE EID=E;
    DBMS_OUTPUT.PUT_LINE(REC.EID||' '||REC.SALARY);
END;
/
```

PROCEDURE THAT RETRIEVES SPECIFIC ROW



What is Cursor?

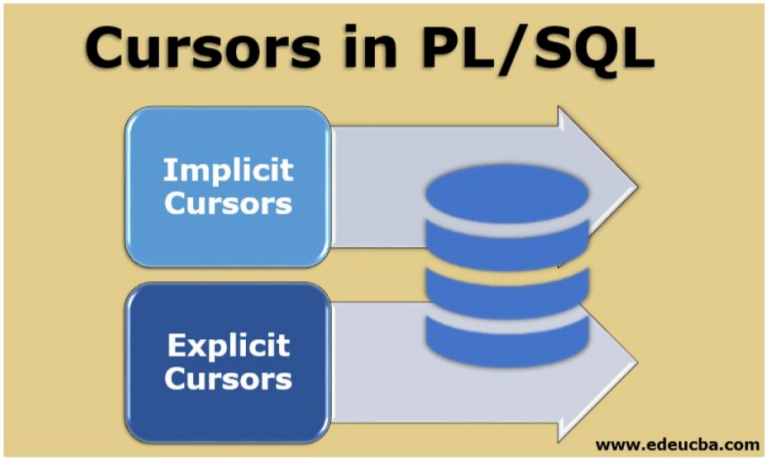
In Oracle, a cursor is a mechanism or a pointer that allows you to fetch and process a set of rows returned by a SQL query.



A cursor is a pointer that points to a result of a query

What is Cursor?

In Oracle, a cursor is a mechanism or a pointer that allows you to fetch and process a set of rows returned by a SQL query.



A cursor is a pointer that points to a result of a query

Implicit Cursor Attributes

Following are implicit cursor attributes,

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

Name of implicit cursor is SQL

```
DECLARE
TOTAL_ROW NUMBER;
BEGIN
DELETE FROM EMP WHERE SALARY=5000;
IF SQL%FOUND THEN
TOTAL_ROW:=SQL%ROWCOUNT;
DBMS_OUTPUT.PUT_LINE(TOTAL_ROW||'NO OF ROWS AFFECTED..');
END IF;
END;
/
```

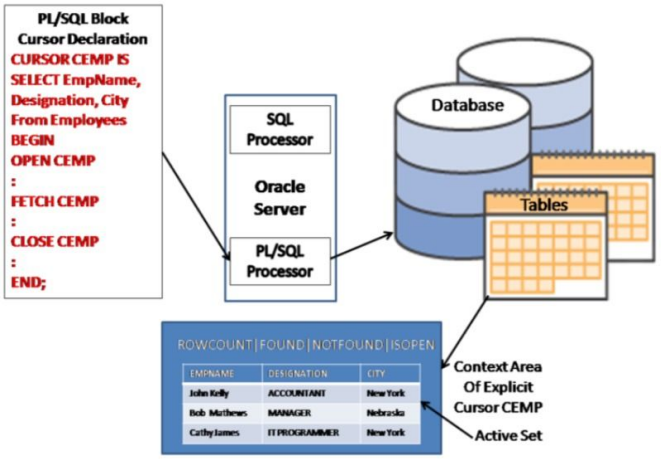
```
SQL> SELECT * FROM EMP;

EID       SALARY
----- ----------
E01         3000
E02         5000
E03         5000

SQL> @D:\PL-SQL\CURSOR1.SQL
2NO OF ROWS AFFECTED..

PL/SQL procedure successfully completed.
```

```
BEGIN
UPDATE EMP SET SALARY = 5000 WHERE EID='E01';
IF SQL%NOTFOUND THEN
DBMS_OUTPUT.PUT_LINE('NO RECORD FOUND');
ELSE
DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || 'UPDATED');
END IF;
END;
/
```

```
SQL> SELECT * FROM EMP;

EID       SALARY
----- ----------
E01         3000

SQL> @D:\PL-SQL\CURSOR2.SQL
1UPDATED

PL/SQL procedure successfully completed.

SQL> SELECT * FROM EMP;

EID       SALARY
----- ----------
E01         5000
```

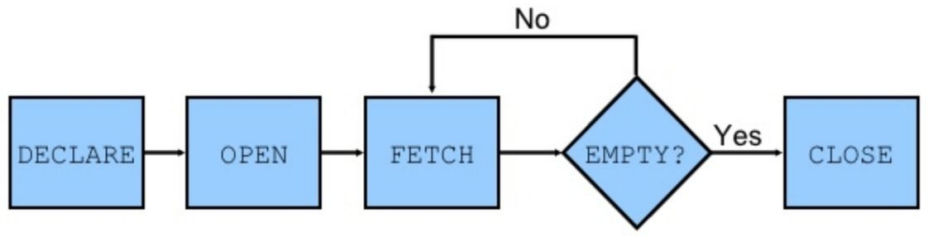
Explicit Cursor

An **explicit cursor** is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.



Explicit cursors are programmer-defined cursors for gaining more control over the **context area**

Explicit Cursor Life Cycle



- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
- Return to `FETCH` if rows are found.
- Release the active set.

Declare a cursor

Before using an explicit cursor, you must declare it in the declaration section of a block

```
CURSOR cursor_name IS query;
```

Closing a cursor

After fetching all rows, you need to close the cursor with the `CLOSE` statement:

```
CLOSE cursor_name;
```

Closing a cursor instructs Oracle to release allocated memory at an appropriate time.

Open a cursor

Before start fetching rows from the cursor, you must open it. To open a cursor, you use the following syntax:

```
OPEN cursor_name;
```

Fetch from a cursor

The `FETCH` statement places the contents of the current row into variables. The syntax of `FETCH` statement is as follows:

```
FETCH cursor_name INTO variable_list;
```

To retrieve all rows in a result set, you need to fetch each row till the last one.

Example

```
DECLARE
EMP_NO EMP.EID%TYPE;
EMP_SAL EMP.SALARY%TYPE;
CURSOR C1 IS SELECT * FROM EMP;
BEGIN
OPEN C1;
LOOP
FETCH C1 INTO EMP_NO,EMP_SAL;
EXIT WHEN C1%NOTFOUND;
DBMS_OUTPUT.PUT_LINE(EMP_NO ||' '|| EMP_SAL);
END LOOP;
CLOSE C1;
END;
/
```

```
SQL> @D:\PL-SQL\EXP_CURSOR1.SQL
E01 5000
E03 5000

PL/SQL procedure successfully completed.
```

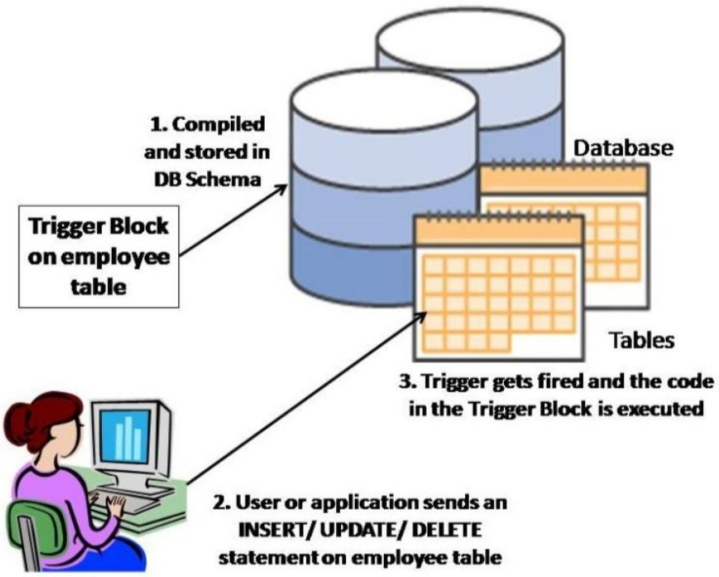
Example

```
DECLARE
    CURSOR C IS SELECT * FROM STUDENT;
BEGIN
    FOR RECORD IN C
    LOOP
        DBMS_OUTPUT.PUT_LINE(RECORD.NAME);
    END LOOP;
END;
/
```

```
SQL> DECLARE
  2     CURSOR C IS SELECT * FROM STUDENT;
  3  BEGIN
  4     FOR RECORD IN C
  5     LOOP
  6             DBMS_OUTPUT.PUT_LINE(RECORD.NAME);
  7     END LOOP;
  8  END;
  9  /
Hello
World
Good
Morning
Morning
```

Trigger

A trigger is a named PL/SQL block stored in the Oracle Database and executed automatically when a triggering event takes place.



Why?

- Enforcing complex business rules that cannot be established using integrity constraint such as `UNIQUE`, `NOT NULL`, and `CHECK`.
- Gathering statistical information on table accesses.
- Preventing invalid transactions.

DML Trigger

- DML trigger executes when a DML statement is executed (INSERT, UPDATE, or DELETE).
- The triggers can run either BEFORE the statement is executed on the database, or AFTER the statement is executed.

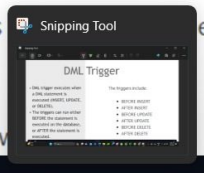
The triggers include:

- BEFORE INSERT
- AFTER INSERT
- BEFORE UPDATE
- AFTER UPDATE
- BEFORE DELETE
- AFTER DELETE

Creating Trigger

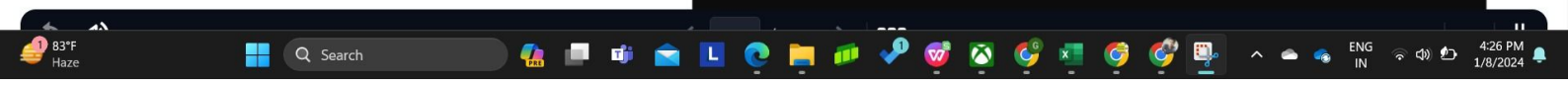
```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} dml_event ON table_name
[FOR EACH ROW]
  [DECLARE variables]
  BEGIN
  pl_sql_code
  [EXCEPTION exception_code]
  END;
```

- **OR REPLACE**: An optional clause. If you specify this, it means the trigger with the same name is replaced with this trigger. It allows you to change the trigger without having to drop the trigger using the DROP TRIGGER statement.
- **trigger_name**: The name of the trigger in the database.
- **BEFORE|AFTER**: Specifies if you want the trigger to run before the statement or after the statement.
- **dml_event**: The DML event that the trigger runs on, which could be INSERT, UPDATE, or DELETE.
- **table_name**: The name of the table that the event runs on.
- **FOR EACH ROW**: This means that the trigger runs for each row affected by the statement. If this is er is run once.
- **DECLARE**: This section allows you to declare variables to use within the trigger.
- **pl_sql_code**: This is the body of your trigger and is where you put the code that defines what you w



```
CREATE OR REPLACE TRIGGER INS_TRIGGER
BEFORE INSERT ON EMP
FOR EACH ROW
BEGIN
DBMS_OUTPUT.PUT_LINE('INSERTING ...');
END;
/
```

```
SQL> @D:\PL-SQL\TRIGGER1.SQL

Trigger created.

SQL> INSERT INTO EMP VALUES('E03',500);
INSERTING ...

1 row created.

SQL>
```



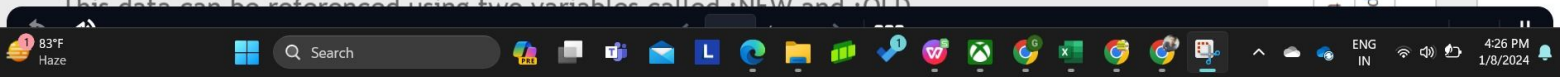
Old and New Values in Trigger

When you write code for DML triggers, you write them for a statement. You often need to reference the data that is included in the statement. For example:

- The data that is being inserted in an INSERT statement
- The data that is being updated (both the old and the new data) in an UPDATE statement
- The data that is being deleted in a DELETE statement

This data can be referenced using two variables called :NEW and :OLD

Statement	:NEW	:OLD
INSERT	The data being inserted into the table	Not applicable
UPDATE	The new data that will be in the table	The old data, or the data that is being rep
DELETE	Not applicable	The data that is being removed from the t



Example of After Insert Trigger

```
CREATE OR REPLACE TRIGGER STUD_AFTER_INS
AFTER INSERT ON STUDENT
FOR EACH ROW
BEGIN
INSERT INTO STUDENT_LOG(NAME,INS_DATE) VALUES (:NEW.NAME,SYSDATE);
END;
/
```

```
SQL> INSERT INTO STUDENT VALUES('PRAGNESH');

1 row created.

SQL> SELECT * FROM STUDENT_LOG;

NAME                 INS_DATE
-------------------- ---------
```

Example After Update

```
CREATE OR REPLACE TRIGGER AFT_UP_EMP
AFTER UPDATE ON EMP
FOR EACH ROW
BEGIN
DBMS_OUTPUT.PUT_LINE('NAME'||'-----'||'OLD SALARY'||'-----'||'NEW SALARY');
DBMS_OUTPUT.PUT_LINE(:OLD.EID||'-----'||:OLD.SALARY||'-----'||:NEW.SALARY);
END;
/
```

```
EID       SALARY
----- ----------
E01         4000
E02         2000
E03         2000
E05         6000
E06         1700
E07         1500
E08         1900
E04         1500
```

```
SQL> update emp set salary=salary+1000 where salary!=2000;
NAME-----OLD SALARY-----NEW SALARY
E01-----4000-----5000
NAME-----OLD SALARY-----NEW SALARY
E05-----6000-----7000
NAME-----OLD SALARY-----NEW SALARY
E06-----1700-----2700
NAME-----OLD SALARY-----NEW SALARY
E07-----1500-----2500
NAME-----OLD SALARY-----NEW SALARY
E08-----1900-----2900
NAME-----OLD SALARY-----NEW SALARY
E04-----1500-----2500
```

After Delete Example

```
CREATE OR REPLACE TRIGGER AFT_DEL_EMP
AFTER DELETE ON EMP
FOR EACH ROW
BEGIN
DBMS_OUTPUT.PUT_LINE(:OLD.EID||'...'||'RECORD IS DELETED');
END;
/
```

```
SQL> select * from emp;

EID        SALARY
----- ----------
E01          5000
E02          2000
E03          2000
E05          7000
E06          2700
E07          2500
E08          2900
E04          2500
```

```
SQL> DELETE FROM EMP WHERE SALARY < 3000;
E02   RECORD IS DELETED
E03   RECORD IS DELETED
E06   RECORD IS DELETED
E07   RECORD IS DELETED
E08   RECORD IS DELETED
E04   RECORD IS DELETED

6 rows deleted.
```

Stored Function

- The function header has the function name and a `RETURN` clause that specifies the datatype of the returned value.
- Each parameter of the function can be either in the `IN`, `OUT`, or `INOUT` mode.
- The declarative section is between the `IS` and `BEGIN` keywords.
- The executable section is between the `BEGIN` and `END` keywords. Unlike a procedure, you must have at least one `RETURN` statement in the executable statement.

```
CREATE [OR REPLACE] FUNCTION function_name (parameter_list)
    RETURN return_type
IS
    [declarative section]

BEGIN

    [executable section]

[EXCEPTION]

    [exception-handling section]

END;
```

PL/SQL function is a reusable program unit stored as

Stored Function Example

```
CREATE OR REPLACE FUNCTION
WEL_MSG(NAME IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
RETURN ('WELLCOME'|| NAME);
END;
/
```

```
SQL> CREATE OR REPLACE FUNCTION
  2  WEL_MSG(NAME IN VARCHAR2)
  3  RETURN VARCHAR2
  4  IS
  5  BEGIN
  6  RETURN ('WELLCOME'|| NAME);
  7  END;
  8  /

Function created.

SQL> select WEL_MSG('India') from dual;

WEL_MSG('INDIA')
----------------------------------------
WELLCOMEIndia

SQL>
```

```
SQL> Declare
  2  n varchar2(20);
  3  begin
  4  n:=WEL_MSG('pragnesh');
  5  dbms_output.put_line(n);
  6  end;
  7  /
WELLCOMEpragnesh

PL/SQL procedure successfully completed.
```

Example Stored Function

```
CREATE OR REPLACE FUNCTION
CALCULATOR(NUM1 IN NUMBER,NUM2 IN NUMBER,OP IN CHAR)
RETURN NUMBER
IS
BEGIN
    IF OP='+' THEN
    RETURN (NUM1+NUM2);
    ELSIF OP='-'THEN
    RETURN (NUM1-NUM2);
    ELSIF OP='*'THEN
    RETURN (NUM1*NUM2);
    ELSE
    RETURN (NUM1/NUM2);
    END IF;
END;
```

```
SQL> SELECT CALCULATOR(1,2,'*') FROM DUAL;

CALCULATOR(1,2,'*')
-------------------
                  2

SQL> SELECT CALCULATOR(2,2,'*') FROM DUAL;

CALCULATOR(2,2,'*')
-------------------
                  4

SQL> SELECT CALCULATOR(2,2,'-') FROM DUAL;

CALCULATOR(2,2,'-')
-------------------
                  0
```