UNIT-5 PL/SQL Notes

1. Introduction to PL/SQL

PL/SQL (Procedural Language/SQL) is Oracle's procedural extension of SQL. It combines **SQL** with **programming constructs** (variables, loops, conditions, exceptions, etc.). It allows developers to write powerful programs that combine SQL queries with logic to control how data is processed. With PL/SQL, complex operations, calculations, and error handling can be performed directly within the Oracle database, making data manipulation more efficient and flexible.

PL/SQL allows developers to:

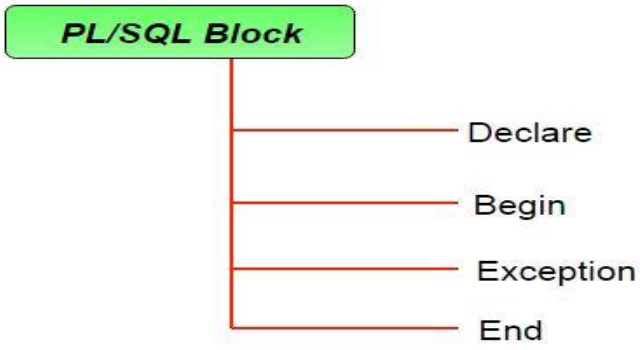
- Execute SQL queries and DML commands inside procedural blocks.
- Define variables and perform complex calculations.
- Create reusable program units, such as procedures, functions, and triggers.
- Handle exceptions, ensuring the program runs smoothly even when errors occur.

✴ Features

- Combines SQL and procedural logic
- Reduces network traffic (runs on server)
- Supports variables, loops, decision-making
- Provides **error handling** (exceptions)
- Allows creation of **functions, procedures, triggers, packages**

2. Structure of a PL/SQL Block

A PL/SQL block is the **basic unit of a program**. Blocks can be **anonymous** or **named** (like procedure/function). PL/SQL extends SQL by adding constructs found in **procedural languages**, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.



Typically, each block performs a logical action in the program. A block has the following structure:

DECLARE
declaration statements;
BEGIN
executable statements
EXCEPTIONS
exception handling statements
END;

PL/SQL code is written in blocks, which consist of three main sections:

- Declare section starts with **DECLARE** keyword in which variables, constants, records as cursors can be declared which stores data temporarily. It basically consists definition of PL/SQL identifiers. This part of the code is optional.
- Execution section starts with **BEGIN** and ends with **END** keyword. This is a mandatory section and here the program logic is written to perform any task like loops and conditional statements. It supports all DML commands, DDL commands and SQL*PLUS built-in functions as well.
- Exception section starts with **EXCEPTION** keyword. This section is optional which contains statements that are executed when a run-time error occurs. Any exceptions can be handled in this section.

Example:

```
DECLARE
  v_name VARCHAR2(20) := 'Dhanasvi';
  v_age  NUMBER := 21;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
  DBMS_OUTPUT.PUT_LINE('Age: ' || v_age);
END;
/
```

3. Elements of PL/SQL

Oracle uses a PL/SQL engine to processes the PL/SQL statements. PL/SQL includes procedural language elements like conditions and loops. It allows declaration of constants and variables, procedures and functions, types and variable of those types and triggers.

Element	Description	Example
Variables	Store data temporarily	v_sal NUMBER := 50000;
Constants	Fixed value	c_bonus CONSTANT NUMBER := 5000;
Data Types	CHAR, VARCHAR2, NUMBER, DATE, BOOLEAN	
Records	Group of related fields	
Cursors	Access query results row by row	
Subprograms	Functions & Procedures	

Example:

```
DECLARE
  salary NUMBER := 20000;
  bonus CONSTANT NUMBER := 3000;
  total NUMBER;
BEGIN
  total := salary + bonus;
  DBMS_OUTPUT.PUT_LINE('Total Salary: ' || total);
END;
/
```

4. Operators in PL/SQL

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators.

Arithmetic Operators

Following table shows all the arithmetic operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 5, then −

Operator	Description	Example
+	Adds two operands	A + B will give 15
-	Subtracts second operand from the first	A - B will give 5
*	Multiplies both operands	A * B will give 50
/	Divides numerator by de-numerator	A / B will give 2
**	Exponentiation operator, raises one operand to the power of other	A ** B will give 100000

Relational Operators

Relational operators compare two expressions or values and return a Boolean result. Following table shows all the relational operators supported by PL/SQL. Let us assume **variable A** holds 10 and **variable B** holds 20, then −

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A = B) is not true.
!= <> ~=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true

Comparison Operators

Comparison operators are used for comparing one expression to another. The result is always either **TRUE, FALSE** or **NULL**.

Operator	Description	Example
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.	If 'Zara Ali' like 'Z% A_i' returns a Boolean true, whereas, 'Nuha Ali' like 'Z% A_i' returns a Boolean false.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b means that x >= a and x <= b.	If x = 10 then, x between 5 and 20 returns true, x between 5 and 10 returns true, but x between 11 and 20 returns false.
IN	The IN operator tests set membership. x IN (set) means that x is equal to any member of set.	If x = 'm' then, x in ('a', 'b', 'c') returns Boolean false but x in ('m', 'n', 'o') returns Boolean true.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. Comparisons involving NULL values always yield NULL.	If x = 'm', then 'x is null' returns Boolean false.

Logical Operators

Following table shows the Logical operators supported by PL/SQL. All these operators work on Boolean operands and produce Boolean results. Let us assume **variable A** holds true and **variable B** holds fals e, then −

Operator	Description	Examples
and	Called the logical AND operator. If both the operands are true then condition becomes true.	(A and B) is false.
or	Called the logical OR Operator. If any of the two operands is true then condition becomes true.	(A or B) is true.
not	Called the logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make it false.	not (A and B) is true.

Example:

```
DECLARE
  a NUMBER := 10;
  b NUMBER := 3;
  result NUMBER;
BEGIN
  result := (a + b) * 2;
  DBMS_OUTPUT.PUT_LINE('Result: ' || result);
END;
/
```

5. Operator Precedence

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example, **x = 7 + 3 * 2**; here, **x** is assigned **13**, not 20 because operator * has higher precedence than +, so it first gets multiplied with **3*2** and then adds into **7**.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

The precedence of operators goes as follows: =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN.

Order	Operator	Description
1	**	Exponentiation
2	NOT	Logical negation
3	*, /	Multiplication, Division
4	+, -	Addition, Subtraction
5	=, <>, >, <	Comparison
6	AND	Logical AND
7	OR	Logical OR

Example:

```
DECLARE
  x NUMBER := 5;
  y NUMBER := 10;
  z NUMBER := 2;
  result NUMBER;
BEGIN
  result := x + y * z; -- multiplication first
  DBMS_OUTPUT.PUT_LINE('Result: ' || result);
END;
/
```

Output → Result: 25

6. Control Structures

Control structures allow conditional or repeated execution.

PL/SQL Conditional Statements:

PL/SQL (Procedural Language/Structured Query Language) is an extension of SQL used in Oracle databases to write procedural code. It includes various conditional statements that allow developers to execute different blocks of code based on specific conditions. Decision-making statements in programming languages decide the direction of the flow of program execution. Conditional Statements available in PL/SQL are defined below:

IF THEN

IF THEN ELSE

NESTED-IF-THEN

IF THEN ELSIF-THEN-ELSE Ladder

1. IF THEN

if then the statement is the simplest decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if condition then
-- do something
end if;
```

Here, condition after evaluation will be either true or false. if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. if and endif consider as a block here.

Example:

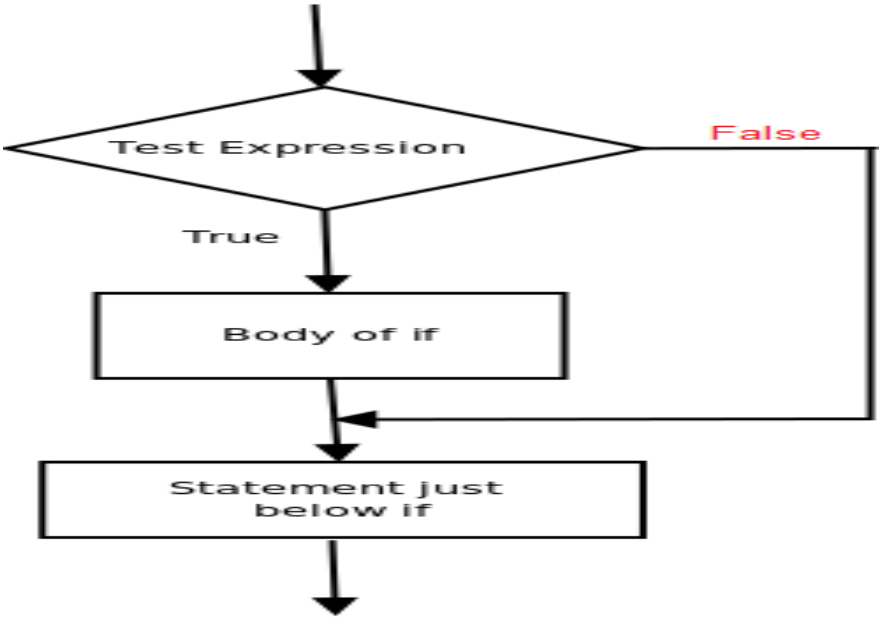
```
declare
-- declare the values here

begin

if condition then
dbms_output.put_line('output');

end if;

dbms_output.put_line('output2');
end;
```



-- pl/sql program to illustrate If statement

```
declare
```

```
num1 number:= 10;

num2 number:= 20;

begin

if num1 > num2 then

dbms_output.put_line('num1 small');

end if;

dbms_output.put_line('I am Not in if');

end;
```

As the condition present in the if statement is false. So, the block below the if statement is not executed. Output:

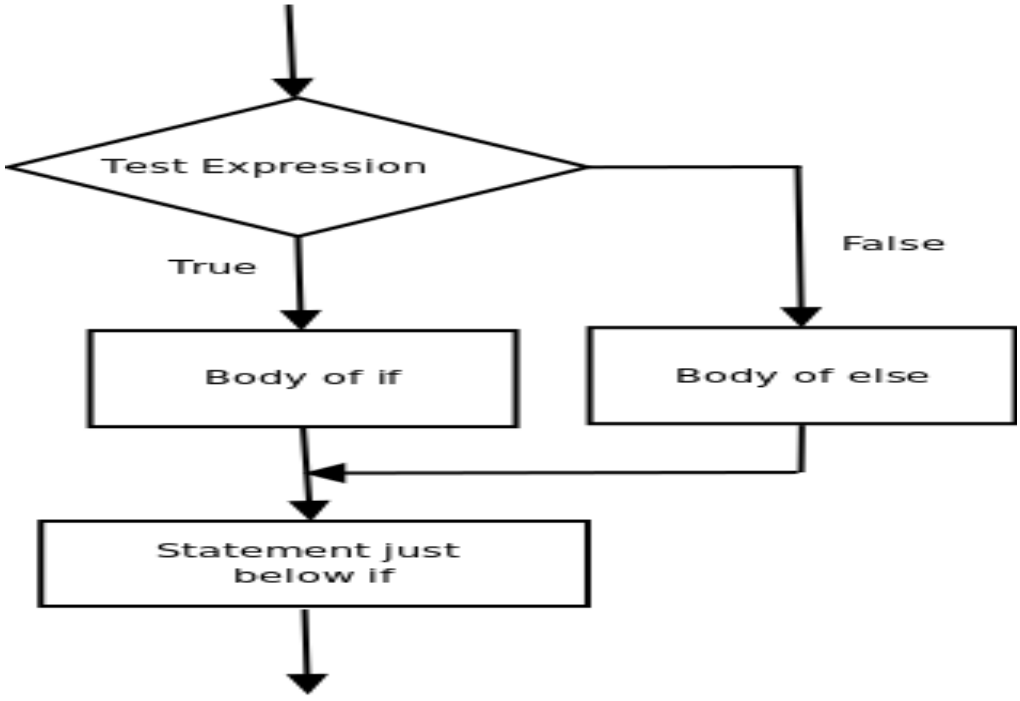
I am Not in if

2. IF THEN ELSE

The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the else statement. We can use the else statement with if statement to execute a block of code when the condition is false.

Syntax:-

```
if (condition) then
   -- Executes this block if
   -- condition is true
else
   -- Executes this block if
   -- condition is false
```



Example:-

```
-- pl/sql program to illustrate If else statement
declare
```

```
num1 number:= 10;
num2 number:= 20;
begin
if num1 < num2 then
dbms_output.put_line('i am in if block');
ELSE
dbms_output.put_line('i am in else Block');
end if;
dbms_output.put_line('i am not in if or else Block');
end;
```

Output:-

i'm in if Block

i'm not in if and not in else Block

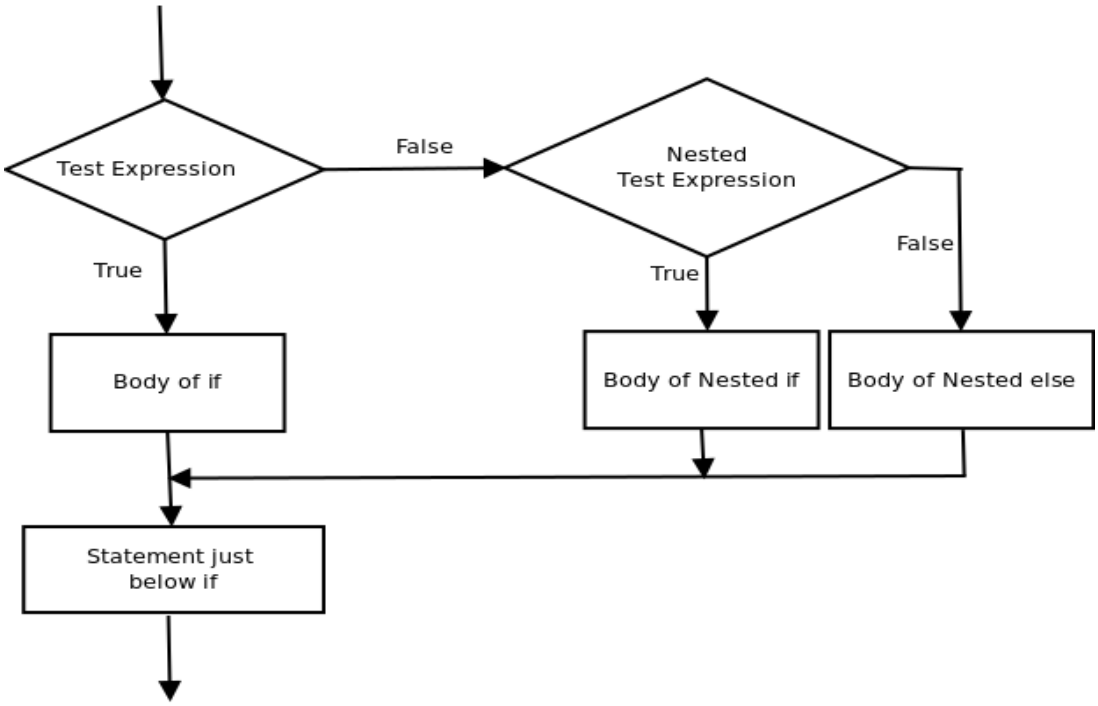
The block of code following the else statement is executed as the condition present in the if statement is false after calling the statement which is not in block(without spaces).

3. NESTED-IF-THEN

A nested if-then is an if statement that is the target of another if statement. Nested if-then statements mean an if statement inside another if statement. Yes, PL/SQL allows us to nest if statements within if-then statements. i.e, we can place an if then statement inside another if then statement.

Syntax:-

```
if (condition1) then
  -- Executes when condition1 is true
  if (condition2) then
    -- Executes when condition2 is true
  end if;
end if;
```



-- pl/sql program to illustrate nested If statement

```
declare
num1 number:= 10;
```

```
num2 number:= 20;
num3 number:= 20;

begin
if num1 < num2 then
dbms_output.put_line('num1 small num2');

  if num1 < num3 then
  dbms_output.put_line('num1 small num3 also');
  end if;

end if;

dbms_output.put_line('after end if');
end;
```

Output:-

```
num1 small num2
num1 small num3 also
after end if
```

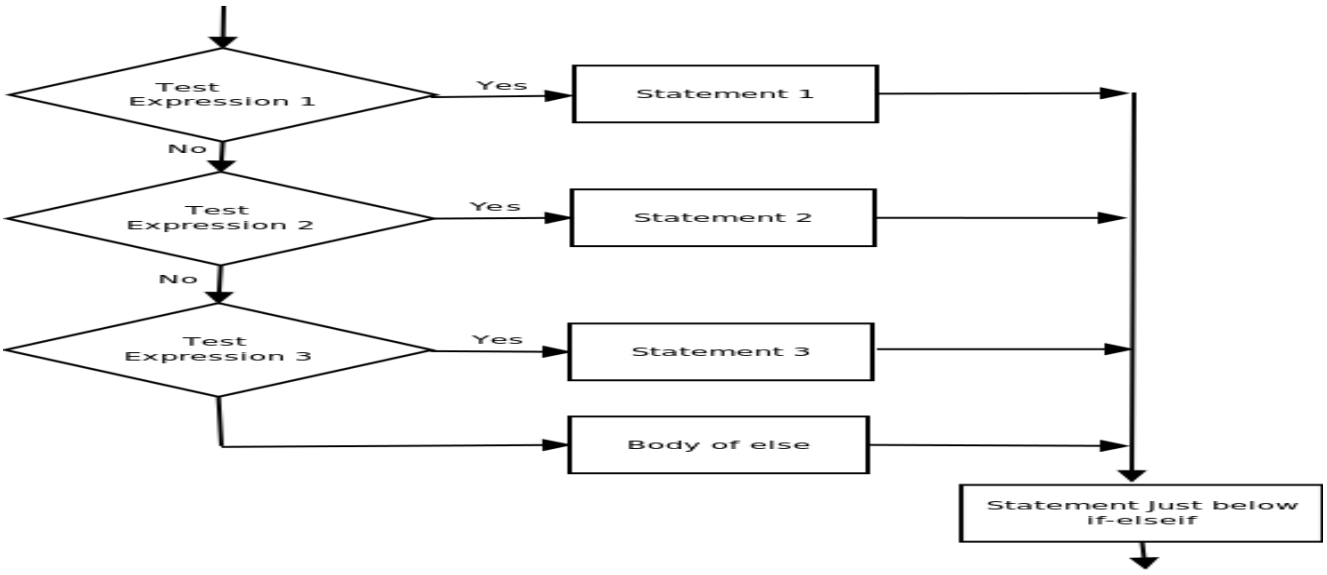
4. IF THEN ELSIF-THEN-ELSE Ladder

Here, a user can decide among multiple options. The if then statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Syntax:-

```
if (condition) then
   --statement
elsif (condition) then
   --statement
.
.
else
   --statement
endif
```

Flow Chart:-



Example:-

```
-- pl/sql program to illustrate if-then-elif-then-else ladder
declare
num1 number:= 10;
num2 number:= 20;
begin
if num1 < num2 then
dbms_output.put_line('num1 small');
ELSEIF num1 = num2 then
dbms_output.put_line('both equal');
ELSE
dbms_output.put_line('num2 greater');
end if;
dbms_output.put_line('after end if');
end;
Output:-
num1 small
after end if
/
```

7. Iterative Control (Loops)

PL/SQL supports **three types of loops**:

(a) Basic LOOP

the **Loop** statement of PL/SQL with all its features like **EXIT**, **EXIT WHEN,** and **Nested Loop** for example.

One of the key features in PL/SQL for controlling program flow is the **LOOP** statement. The **LOOP** statement is a feature of PL/SQL that allows you to repeatedly execute a block of code until a specified condition is satisfied.

Procedural Language/Structured Query Language (PL/SQL) provides a robust environment for database programming, allowing developers to create powerful and efficient code for Oracle databases.

Syntax

```
LOOP
  -- Code block to be executed repeatedly
END LOOP;
```

EXIT Statement

The **EXIT** statement is used to break the loop whether the loop condition has been satisfied or not. This statement is particularly useful when you want to terminate the loop based on certain conditions within the loop block.

Syntax

```
LOOP
  -- Code block
  IF condition THEN
    EXIT;
  END IF;
END LOOP;
```

Example of PL/SQL LOOP with Conditional EXIT

In this example, we showcase the application of a PL/SQL LOOP construct with a conditional EXIT statement. The code demonstrates a scenario where a loop iterates a specific block of code, printing iteration numbers, and breaks out of the loop when a predefined condition is met.

```
DECLARE
  counter NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('This is iteration number ' || counter);
    IF counter = 3 THEN
      EXIT;
    END IF;
    counter := counter + 1;
  END LOOP;
END;
/
```

Output:

```
Statement processed.
This is iteration number 1
This is iteration number 2
This is iteration number 3
```

Explanation:

- Initially counter variable is set to 1.
- The **LOOP** statement repeatedly executes the code block within it.
- Inside the loop, **DBMS_OUTPUT.PUT_LINE** is used to print Iteration number (value of counter).
- The counter is incremented by 1 in each iteration.
- IF statement is executed when the value of counter will become 3 and The **EXIT** statement is executed and loop stops.

EXIT WHEN Statement

The **EXIT WHEN statement** allows for a more concise way to specify the condition under which a loop should exit. It checks the condition directly within the loop's syntax.

Syntax

```
LOOP
   -- Code block
   EXIT WHEN condition;
END LOOP;
```

Example of PL/SQL LOOP with EXIT WHEN

The purpose of this example is to show how to print "**GeeksForGeeks**" repeatedly using a PL/SQL LOOP construct. With the help of the **EXIT WHEN** statement, the loop can be controlled to end when a counter variable reaches a predetermined threshold.

```
DECLARE
  counter NUMBER := 1;  -- Initialization of the counter variable
BEGIN
  -- Loop that prints "GeeksForGeeks" five times
  LOOP
    DBMS_OUTPUT.PUT_LINE('GeeksForGeeks');
    counter := counter + 1;  -- Increment the counter
    EXIT WHEN counter > 5;  -- Exit the loop when counter exceeds 5
  END LOOP;
```

```
END;
/
```
Output:
```
Statement processed.
GeeksForGeeks
GeeksForGeeks
GeeksForGeeks
GeeksForGeeks
GeeksForGeeks
```

Explanation:

- Initially counter variable is set to 1.
- The **LOOP** statement repeatedly executes the code block within it.
- Inside the loop, **DBMS_OUTPUT.PUT_LINE** is used to print "GeeksForGeeks".
- The counter is incremented by 1 in each iteration.
- The **EXIT WHEN** statement is executed when the loop when the counter exceeds 5.

Nested Loops

Nested Loop is a Loop inside Loop and PL/SQL supports nested loops that allows you to have multiple levels of iteration within a program. This is achieved by placing one or more **LOOP** statements inside another. Each nested loop has its own set of loop control statements.

Syntax

```
-- Outer Loop
LOOP
    -- Code block
    -- Inner Loop
    LOOP
       -- Inner loop code block
       EXIT WHEN inner_condition;
    END LOOP;
  EXIT WHEN outer_condition;
END LOOP;
```

Example of PL/SQL Nested FOR Loop Simultaneous Iteration

In this example, we will create nested FOR loops that iterate over two ranges, demonstrating simultaneous iteration.

```
DECLARE
  outer_counter NUMBER := 1;
  inner_counter NUMBER := 1;
BEGIN
  FOR outer_counter IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE('Outer Loop - Iteration ' || outer_counter);

    FOR inner_counter IN 1..2 LOOP
      DBMS_OUTPUT.PUT_LINE('Inner Loop - Iteration ' || inner_counter);
    END LOOP;
  END LOOP;
END;
/
```
Output:
```
Statement processed.
Outer Loop - Iteration 1
```

Inner Loop - Iteration 1
Inner Loop - Iteration 2
Outer Loop - Iteration 2
Inner Loop - Iteration 1
Inner Loop - Iteration 2
Outer Loop - Iteration 3
Inner Loop - Iteration 1
Inner Loop - Iteration 2

Explanation:

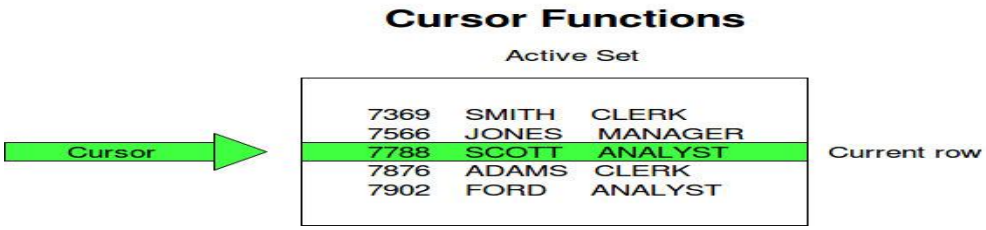
- There are two nested loops
- The outer FOR loop **(FOR outer_counter IN 1..3 LOOP)** runs three times.
- Inside the outer FOR loop, there is an inner FOR loop **(FOR inner_counter IN 1..2 LOOP)** that runs two times for each iteration of the outer loop.
- **DBMS_OUTPUT.PUT_LINE** statements is used to print output.

/

8. Cursors

A **cursor** allows processing query results **row by row**. he cursor is used to retrieve data one row at a time from the results set, unlsike other **SQL commands** that operate on all rows at once. Cursors update table records in a singleton or row-by-row manner.

The Data that is stored in the Cursor is called the **Active Data Set**. Oracle DBMS has another predefined area in the main memory Set, within which the cursors are opened. Hence the size of the cursor is limited by the size of this pre-defined area.



Cursor Actions

Key actions involved in working with cursors in PL/SQL are:

1. **Declare Cursor:** A cursor is declared by defining the SQL statement that returns a result set.
2. **Open:** A Cursor is opened and populated by executing the SQL statement defined by the cursor.
3. **Fetch:** When the cursor is opened, rows can be fetched from the cursor one by one or in a block to perform data manipulation.
4. **Close:** After data manipulation, close the cursor explicitly.
5. **Deallocate:** Finally, delete the cursor definition and release all the system resources associated with the cursor.

Types

Cursors are classified depending on the circumstances in which they are opened.

Implicit cursor: if the oracle engine opened a cursor for its internal processing it is known as an implicit cursor. It is created "automatically" for the user by oracle when a query is executed and is simpler to code.

Explicit cursor: a cursor can also be opened for processing data through a pl/sql block, on demand. Such a user-defined cursor is known as an explicit cursor.

1. **Implicit Cursor** – Automatically created by Oracle for single-row queries.
2. **Explicit Cursor** – Declared by the user for multi-row queries.

Explicit Cursor Example

```
DECLARE
  CURSOR emp_cur IS SELECT empno, ename FROM emp;
  v_empno emp.empno%TYPE;
  v_ename emp.ename%TYPE;
BEGIN
  OPEN emp_cur;
  LOOP
    FETCH emp_cur INTO v_empno, v_ename;
    EXIT WHEN emp_cur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Emp No: ' || v_empno || ', Name: ' || v_ename);
  END LOOP;
  CLOSE emp_cur;
END;
/
```

9. Procedure

A **procedure** is a named block that performs a task. It may accept parameters but **does not return** a value.

Syntax

```
CREATE OR REPLACE PROCEDURE proc_name (param1 IN datatype)
IS
BEGIN
  statements;
END;
/
```

Example

```
CREATE OR REPLACE PROCEDURE greet_user(name VARCHAR2)
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello ' || name || '!');
END;
/
EXEC greet_user('Lavanya');
```

10. Function

A **function** is similar to a procedure but **returns a value**.

Syntax

```
CREATE OR REPLACE FUNCTION func_name (param IN datatype)
RETURN datatype
IS
BEGIN
  RETURN value;
END;
/
```

Example

```
CREATE OR REPLACE FUNCTION square_num(n NUMBER)
RETURN NUMBER
IS
BEGIN
```

```
  RETURN n * n;
END;
/
BEGIN
  DBMS_OUTPUT.PUT_LINE('Square: ' || square_num(6));
END;
/
```

11. Packages

A **package** groups related **procedures, functions, variables, cursors** together.

Two Parts:

1. **Package Specification** – Declaration
2. **Package Body** – Implementation

Example

```
CREATE OR REPLACE PACKAGE math_pack IS
  FUNCTION add_num(a NUMBER, b NUMBER) RETURN NUMBER;
  PROCEDURE show_sum(a NUMBER, b NUMBER);
END math_pack;
/

CREATE OR REPLACE PACKAGE BODY math_pack IS
  FUNCTION add_num(a NUMBER, b NUMBER) RETURN NUMBER IS
  BEGIN
    RETURN a + b;
  END;

  PROCEDURE show_sum(a NUMBER, b NUMBER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Sum: ' || (a + b));
  END;
END math_pack;
/
BEGIN
  DBMS_OUTPUT.PUT_LINE('Addition: ' || math_pack.add_num(10, 5));
  math_pack.show_sum(7, 3);
END;
/
```

12. Exception Handling

Exceptions handle errors gracefully in PL/SQL.

Types

1. **Predefined Exceptions** (e.g., NO_DATA_FOUND, ZERO_DIVIDE)
2. **User-defined Exceptions**

Predefined Example

```
DECLARE
  num NUMBER := 10;
  denom NUMBER := 0;
  result NUMBER;
BEGIN
  result := num / denom;
```

```
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error: Cannot divide by zero');
END;
/
```

User-Defined Exception Example

```
DECLARE
  age NUMBER := 15;
  ex_underage EXCEPTION;
BEGIN
  IF age < 18 THEN
    RAISE ex_underage;
  END IF;
EXCEPTION
  WHEN ex_underage THEN
    DBMS_OUTPUT.PUT_LINE('Access denied: Age must be 18 or above.');
END;
/
```

13. Triggers

A **trigger** executes automatically when a specified database event occurs (INSERT, UPDATE, DELETE).

PL/SQL stands for Procedural Language/ Structured Query Language. It has block structure programming features.PL/SQL supports SQL queries. It also supports the declaration of the variables, **control statements, Functions, Records, Cursor, Procedure, and Triggers.**PL/SQL contains a declaration section, execution section, and exception-handling section. Declare and exception handling sections are optional.

Syntax:

Declaration section
BEGIN
Execution section
EXCEPTION
Exception section
END;

PL/SQL Triggers

PL/SQL triggers are block structures and predefined programs **invoked automatically** when some event occurs. They are stored in the database and **invoked repeatedly** in a particular scenario. There are two states of the triggers, they are **enabled** and **disabled**. When the trigger is created it is enabled. **CREATE TRIGGER** statement creates a trigger.

A triggering event is specified on a table, a view, a schema, or a database.**BEFORE** and **AFTER** are the trigger Timing points.**DML triggers** are created on a table or view, and **triggers.** Crossedition triggers are created on Edition-based redefinition. System Triggers are created on schema or database using DDL or database operation statements.It is applied on new data only ,it don't affect existing data.

They are associated with response-based events such as a

- **Database Definition Language** statements such as CREATE, DROP or ALTER.
- **Database Manipulation Language** statements such as UPDATE, INSERT or DELETE.

- Database operations such as LOGON, LOGOFF, STARTUP, and SHUTDOWN.

Why are Triggers important?
The importance of Triggers is:
- **Automated Action:** It helps to automate actions in response to events on table or views.
- **Data integrity**: Constraint can be applied to the data with the help of trigger.It is used to ensure referential integrity.

- **Consistency**: It helps to maintain the consistency of the database by performing immediate responses to specific events.
- **Error handling**: It helps in error handling by responding to the errors. For example, If specific condition is not met it will provide an error message.

PL/SQL Trigger Structure

Triggers are fired on the tables or views which are in the database. Either table, view ,schema, or a database are the basic requirement to execute a trigger. The trigger is specified first and then the action statement are specified later.

Syntax:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE or AFTER or INSTEAD OF //trigger timings
INSERT or UPDATE or DELETE // Operation to be performed
of column_name
on Table_name
FOR EACH ROW
DECLARE
    Declaration section
BEGIN
    Execution section
EXCEPTION
    Exception section
END;
/
Query operation to be performed i.e INSERT,DELETE,UPDATE.
```

- CREATE [OR REPLACE] TRIGGER trigger_name is used to create a trigger or replace the existing trigger.|
- BEFORE | AFTER | INSTEAD OF specifies trigger timing.
- INSERT | UPDATE | DELETE are the DML operations performed on table or views.
- OF column_name specifies the column that would be updated.
- ON table_name species the table for the operation.
- FOR EACH ROW specify that trigger is executed on each row.

Types of PL/SQL Triggers

Trigger timing and operations forms different combinations such as BEFORE INSERT OR BEFORE DELETE OR BEFORE UPDATE .BEFORE and AFTER are known as conditional triggers.

Conditional Trigger: Before

Trigger is activated before the operation on the table or view is performed.

Query:

```
-- Create Geeks table
CREATE TABLE Geeks (
   Id INT,
   Name VARCHAR2(20),
   Score INT
```

```
);

-- Insert into Geeks Table
INSERT INTO Geeks (Id, Name, Score) VALUES (1, 'Sam', 800);
INSERT INTO Geeks (Id, Name, Score) VALUES (2, 'Ram', 699);
INSERT INTO Geeks (Id, Name, Score) VALUES (3, 'Tom', 250);
INSERT INTO Geeks (Id, Name, Score) VALUES (4, 'Om', 350);
INSERT INTO Geeks (Id, Name, Score) VALUES (5, 'Jay', 750);
-- insert statement should be written for each entry in Oracle Sql Developer

CREATE TABLE Affect (
   Id INT,
   Name VARCHAR2(20),
   Score INT
);

-- BEFORE INSERT trigger
CREATE OR REPLACE TRIGGER BEFORE_INSERT
BEFORE INSERT ON Geeks
FOR EACH ROW
BEGIN
   INSERT INTO Affect (Id, Name, Score)
   VALUES (:NEW.Id, :NEW.Name, :NEW.Score);
END;
/
INSERT INTO Geeks (Id, Name, Score) VALUES (6, 'Arjun', 500);
```

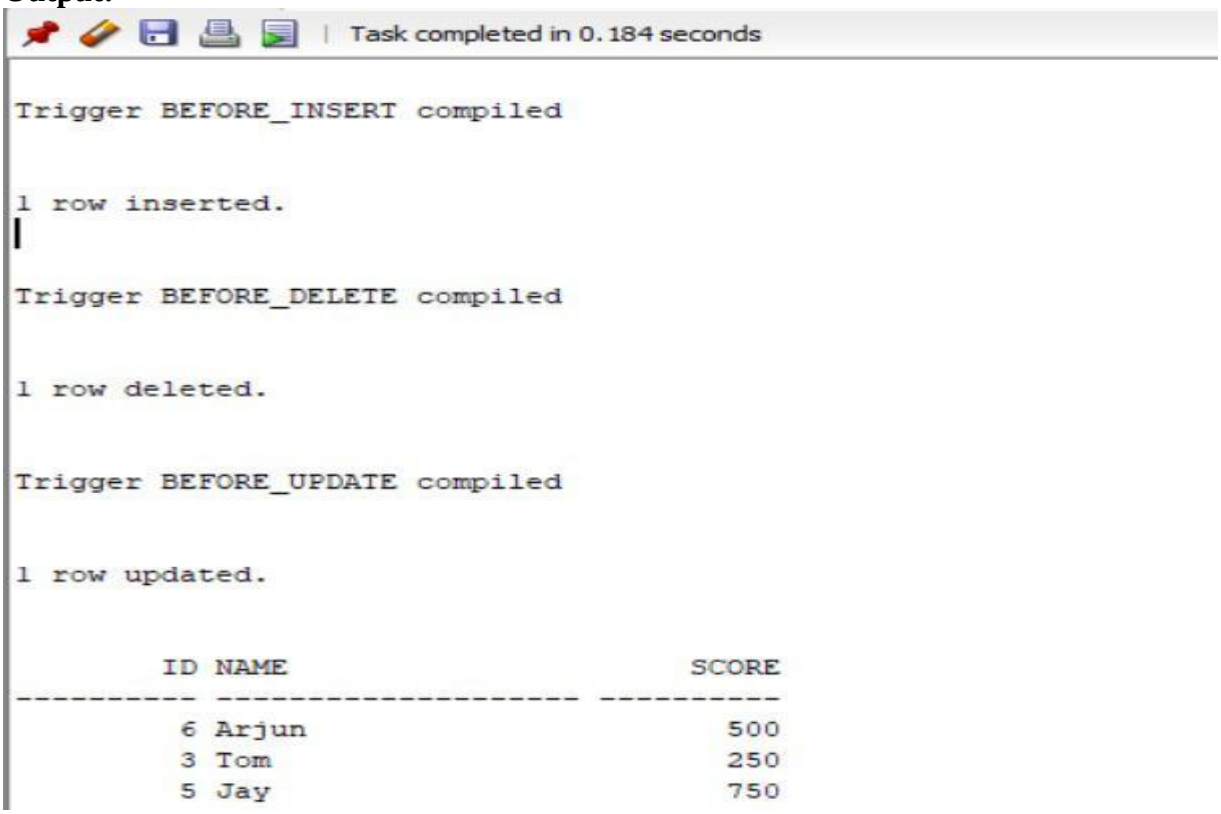
BEFORE DELETE Trigger

```
-- BEFORE DELETE trigger
CREATE OR REPLACE TRIGGER BEFORE_DELETE
BEFORE DELETE ON Geeks
FOR EACH ROW
BEGIN
   INSERT INTO Affect (Id, Name, Score)
   VALUES (:OLD.Id, :OLD.Name, :OLD.Score);
END;
/
DELETE FROM Geeks WHERE Id = 3;
```

BEFORE UPDATE Trigger

```
-- BEFORE UPDATE trigger
CREATE OR REPLACE TRIGGER BEFORE_UPDATE
BEFORE UPDATE ON Geeks
FOR EACH ROW
BEGIN
   INSERT INTO Affect (Id, Name, Score)
   VALUES (:OLD.Id, :OLD.Name, :OLD.Score);
END;
/
UPDATE Geeks SET Score = 900 WHERE Id = 5;
SELECT * FROM Affect;
SELECT * FROM Geeks;
```

Output:



Conditional Trigger Before

Explanation:

- BEFORE_INSERT Trigger is fired before adding a row in Geeks Table,and row is inserted in the Affect table.
- BEFORE_DELETE Trigger is activated before the row is delete from the Geeks table and row which satisfy the condition is added to Affect table.
- BEFORE_UPDATE TRIGGER is activated before the row with Id=5 is updated and row with old values is added to Affect table

Geeks table after trigger events

	ID	NAME	SCORE
1	1	Sam	800
2	2	Ram	699
3	4	Om	350
4	5	Jay	900
5	6	Arjun	500

Conditional Trigger Before

Conditional Trigger: After

Trigger is activated after the operation on the table or view is performed.

Query:

```
SET SERVEROUTPUT ON;


CREATE TABLE Geeks (
   Id INT,
   Name VARCHAR2(20),
```

```
    Score INT
);

-- Insert into Geeks Table
INSERT INTO Geeks (Id, Name, Score) VALUES (1, 'Sam', 800);
INSERT INTO Geeks (Id, Name, Score) VALUES (2, 'Ram', 699);
INSERT INTO Geeks (Id, Name, Score) VALUES (3, 'Tom', 250);
INSERT INTO Geeks (Id, Name, Score) VALUES (4, 'Om', 350);
INSERT INTO Geeks (Id, Name, Score) VALUES (5, 'Jay', 750);
-- insert statement should be written for each entry in Oracle Sql Developer
CREATE TABLE Affect (
    Id INT,
    Name VARCHAR2(20),
    Score INT
);
SELECT * FROM Geeks;
-- AFTER DELETE trigger
CREATE OR REPLACE TRIGGER AFTER_DELETE
AFTER DELETE ON Geeks
FOR EACH ROW
BEGIN
    INSERT INTO Affect (Id, Name, Score)
    VALUES (:OLD.Id, :OLD.Name, :OLD.Score);
END;
/
DELETE FROM Geeks WHERE Id = 4;
-- AFTER UPDATE trigger
CREATE OR REPLACE TRIGGER AFTER_UPDATE
AFTER UPDATE ON Geeks
FOR EACH ROW
BEGIN
    INSERT INTO Affect (Id, Name, Score)
    VALUES (:NEW.Id, :NEW.Name, :NEW.Score);
END;
/
UPDATE Geeks SET Score = 1050 WHERE Id = 5;
SELECT * FROM Affect;
SELECT * FROM Geeks;
```

Output:

Conditional Trigger After

Explanation: After the deletion of the row from the Geek table trigger is fired and the row which is deleted is added to the Affect Table.In second trigger i.e After_update trigger is fired after performing update on Geeks table and the row is added to Affect Table.Output contains the Affect table and the Geek table after the trigger events.

Common Use Cases of PL/SQL Triggers

- To automate the actions in response to the events and reducing manual task.
- To apply constraint to ensure referential integrity and to prevent invalid data in table or database.
- In error handling to response to errors.

Syntax

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE | AFTER
INSERT | UPDATE | DELETE
ON table_name
FOR EACH ROW
BEGIN
  statements;
END;
/
```

Example

```
CREATE OR REPLACE TRIGGER emp_audit
AFTER INSERT ON emp
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('New Employee Added: ' || :NEW.ename);
END;
/
```