Unit -4

INTRODUCTION TO PL/SQL, ADVANCED PL/SQL

TOPIC

- • SQL v/s PL/SQL
- • PL/SQL Block structure
- • Language construct of PL/SQL (Variable, Basic and Composite Data Type, Conditions, Looping etc.)
- • %Type and %Rowtype
- • Using Cursor (Implicit, Explicit)
- • Exception Handling
- • Creating and Using Procedure
- • Package
- • Trigger
- • Creating Objects
- • Object in Database – Table
- • PL/SQL Tables, Nested Tables, Varrays

SQL V/S PL/SQL

SQL V/S PL/SQL

- **SQL** is a Structured Query Language used to issue a single query or execute a single insert/update/delete.
- **PL/SQL** is a procedural language used to create applications.
- **SQL** is used to write queries, DDL **and** DML statements.
- **PL/SQL** is used to write program blocks, functions, procedures, triggers **and** packages.

SQL V/S PL/SQL

- *SQL may be considered as the source of data for our reports, web pages and screens.*
- *PL/SQL can be considered as the application language similar to Java or PHP.*
- *SQL is a data oriented language used to select and manipulate sets of data.*
- *PL/SQL is a procedural language used to create applications.*

SQL V/S PL/SQL

- *SQL is executed one statement at a time.*
- *PL/SQL is executed as a block of code.*

- *SQL can be embedded within a PL/SQL program.*
- *But PL/SQL can't be embedded within a SQL statement.*

PL/SQL BLOCK STRUCTURE

PL/SQL BLOCK STRUCTURE

- A **PL/SQL block** is defined by the keywords DECLARE, BEGIN, EXCEPTION, and END.
- These keywords divide the **block** into a declarative part, an executable part, and an exception-handling part.
- The declaration section is optional and may be used to define and initialize constants and variables.
- PL/SQL blocks contain three sections
  1. Declare section
  2. Executable section and
  3. Exception-handling section.

DECLARE (Optional)

Declaration of Variable, Constants.

BEGIN

PL/SQL Executable Statements.

EXCEPTION (Optional)

PL/SQL Exception Handler Block.

END;

PL/SQL BLOCK STRUCTURE

- PL/SQL block has the following structure:

```
DECLARE
    Declaration statements
BEGIN
    Executable statements
EXCETION
    Exception-handling statements
END ;
```

EX. 1 SIMAPLE PROGRAM IN PL/SQL

```
Declare

begin
        dbms_output.put_line('kamani college');
        dbms_output.put_line('BCA Department');

end;
```

EX.2 FORMAT TO PRINT MASSAGE

```
DECLARE

BEGIN
   DBMS_OUTPUT.PUT_LINE('Start');   -- Prints with newline
   DBMS_OUTPUT.PUT('This is ');     -- Same line (no newline)
   DBMS_OUTPUT.PUT('one line.');    -- Continues same line
   DBMS_OUTPUT.NEW_LINE;            -- Moves to new line
   DBMS_OUTPUT.PUT_LINE('End');     -- Prints with newline
END;
/
```

- 🔑 Summary:

- PUT_LINE → prints text + newline.
- PUT → prints text only (no newline).
- NEW_LINE → prints only a newline (blank line).

🔑 Detailed Differences

Feature / Method	`PUT_LINE`	`PUT`	`NEW_LINE`
Prints text	✅ Yes	✅ Yes	❌ No
Adds newline automatically	✅ Yes	❌ No	✅ Yes
Output formatting	Each call prints on a **new line**	Calls **append to same line** until newline added	Prints only a **blank line**
Use case	Simple message printing	String concatenation / inline text building	Spacing / formatting output

THE PL/SQL COMMENTS

- The PL/SQL supports ***single-line and multi-line comments***.
- All characters available inside any comment are ignored by the PL/SQL compiler.

- The PL/SQL single-line comments start with the delimiter -- (double hyphen) and multi-line comments are enclosed by /* and */.

```
DECLARE
   -- variable declaration
   message  varchar2(20):= 'Hello, World!';
BEGIN
   /*
   *  PL/SQL executable statement(s)
   */
   dbms_output.put_line(message);
END;
/
```

VARIABLE ,

BASIC DATA TYPE,

CONDITIONS LOOP

THE PL/SQL DELIMITERS

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
||	Concatenation operator
**	Exponentiation operator

**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, !=, ~=, ^=	Different versions of NOT EQUAL

THE PL/SQL IDENTIFIERS

- PL/SQL identifiers are constants, variables, exceptions, procedures, cursors, and reserved words.
- The identifiers consist of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed ***30 characters***.

INITIALIZING VARIABLES IN PL/SQL

- Whenever you declare a variable, PL/SQL assigns it a default value of NULL.

- If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following –

  - The DEFAULT keyword
  - The assignment operator (:=)

Ex. 1 The DEFAULT keyword

```
DECLARE

    greetings varchar2(20) DEFAULT 'Have a Good Day ' ;

BEGIN

    dbms_output.put_line(greetings);

END;
```

```
Have a Good Day

Statement processed.
```

Ex. 2 The assignment operator (:=)

```
DECLARE
    a integer := 10;
    b integer := 20;
    c integer;
    f real;
BEGIN
    c := a + b;
    dbms_output.put_line('Value of c: ' || c);
    f := 70.0/3.0;
    dbms_output.put_line('Value of f: ' || f);
END;
```

```
Value of c: 30
Value of f: 23.333333333333333333333333333333333333

Statement processed.
```

EX.3 FIX VALUES IN VARIABLE

```
-- fix values in variable

declare
   x number(3);
   y number(3);
begin
        x:=10;
        y:=20;
   dbms_output.put_line(x+y);
end;
```

EX.4 USER DEFINE VALUES GET

```
-- user define values get


declare
        x number(3);
        y number(3);
begin
       dbms_output.put_line('Addition of ='||(:x + :y));
       dbms_output.put_line('Subtraction of ='||(:x - :y));
       dbms_output.put_line('Multiplication of ='||(:x * :y));
       dbms_output.put_line('Division of ='||(:x / :y));
end;
```

VARIABLE SCOPE IN PL/SQL

VARIABLE SCOPE IN PL/SQL

- PL/SQL allows the ***nesting of blocks***, i.e., each program block may contain ***another inner block***.
- If a variable is declared within an inner block, it is not accessible to the outer block.
- However, if a variable is declared and accessible to an outer block, it is also accessible to ***all nested inner blocks.***
- There are two types of variable scope
  - ***Local variables*** – Variables declared in an inner block and not accessible to outer blocks.
  - ***Global variables*** – Variables declared in the outer most block or a package.

EX.

```
DECLARE
   -- Global variables
      num1 number := 95;
      num2 number := 85;
BEGIN
      dbms_output.put_line('Outer Variable num1: ' || num1);
      dbms_output.put_line('Outer Variable num2: ' || num2);
      DECLARE

      -- Local variables
         num1 number := 195;
         num2 number := 185;
      BEGIN

         dbms_output.put_line('Inner Variable num1: ' || num1);
         dbms_output.put_line('Inner Variable num2: ' || num2);
      END;
END;
```

```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

Statement processed.
```

PL/SQL-CONSTANTS AND LITERALS

PL/SQL-CONSTANTS AND LITERALS

- A constant holds a value that once declared, does not change in the program.
- A constant declaration specifies its name, data type, and value, and allocates storage for it.
- The declaration can also impose the **NOT NULL constraint.**
- **Declaring a Constant**
  - A constant is declared using the CONSTANT keyword.

EX.-1 PL/SQL-CONSTANTS

```
DECLARE
        college_name constant varchar2(20) := 'SY BCA';

BEGIN
        dbms_output.put_line('I study in '|| college_name);

END;
```

```
I study in SY BCA

Statement processed.
```

EX.-2 PL/SQL-CONSTANTS

```
DECLARE
   -- constant declaration
    pi constant number := 3.141592654;
   -- other declarations
    radius number(5,2);
    dia number(5,2);
    circumference number(7, 2);
    area number (10, 2);
BEGIN
    -- processing
    radius := 9.5;
    dia := radius * 2;
    circumference := 2.0 * pi * radius;
    area := pi * radius * radius;
    -- output
    dbms_output.put_line('Radius: ' || radius);
    dbms_output.put_line('Diameter: ' || dia);
    dbms_output.put_line('Circumference: ' || circumference);
    dbms_output.put_line('Area: ' || area);
END;
```

```
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53

Statement processed.
```

THE PL/SQL LITERALS

THE PL/SQL LITERALS

- A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier.
- For example, TRUE, 786, NULL, 'tutorialspoint' are all literals of type Boolean, number, or string.
-
- PL/SQL,literals are case-sensitive.
- PL/SQL supports the following kinds of literals –
  - Numeric Literals
  - Character Literals
  - String Literals
  - BOOLEAN Literals
  - Date and Time Literals

EX.

```
DECLARE

    message varchar2(30):= 'That''s tutorialspoint.com!';
    str varchar2(30):= 'Welcome to Studytonight.com';
BEGIN

    dbms_output.put_line(message);
    dbms_output.put_line(str);

END;
```

PL/SQL-OPERATORS

PL/SQL-OPERATORS

- 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- Addition, Subtraction, Multiplication, Division
  - Relational operators - Less then, Greater Than, etc.
  - Comparison operators- Like ,Between, In, IsNull
  - Logical operators- AND, OR, NOT
  - String operators-

EX. ARITHMETIC OPERATION

```
-- Arithmetic operation user define values get

declare
        x number(3);
        y number(3);
begin
        dbms_output.put_line('Addition of ='||(:x + :y));
        dbms_output.put_line('Subtraction of ='||(:x - :y));
        dbms_output.put_line('Multiplication of ='||(:x * :y));
        dbms_output.put_line('Division of ='||(:x / :y));
end;
```

PL/SQL TABLES

PL/SQL WITH TABLE

- create table emp(id number (3),name varchar2(10), salary number(10));
- insert into emp values(1,'Hetansh',4000)
- select *from emp
- alter table emp ADD (sal_update number(10))

- EX.
- declare
- X number(3):=2;
- begin
- INSERT into emp values(2,'Sagar',3000,'');
- UPDATE emp set sal_update = salary * X where id=1;
- --DELETE from emp where id=1;
- END;

PL/SQL BASIC DATA TYPES

PL/SQL BASIC DATA TYPES

S.No	Date Type & Description
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.

CONTROL STRUCTURE

THREE TYPE OF CONTROL STRUCTURE

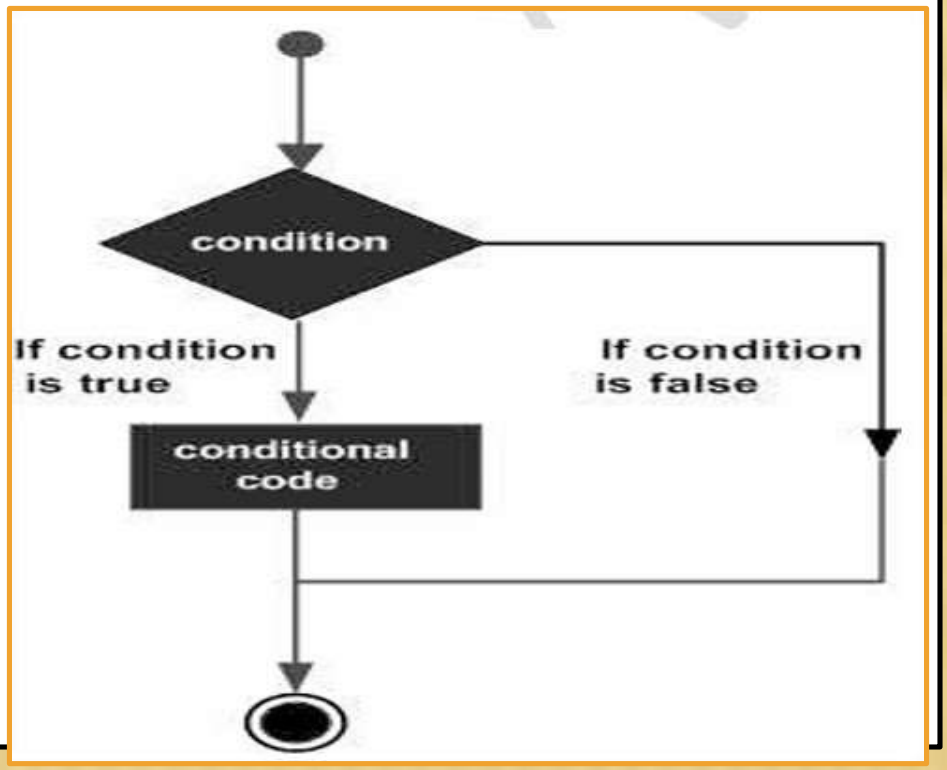
- [1]Conditional Control
  - *1) IF...THEN...END IF*
  - *2) IF...THEN..ELSE...END IF*
  - *3) IF...THEN...ELSIF...END IF*
  - *4) CASE...ENDCASE*
  - 5) Nested IF-THEN-ELSE
- [2] Iterative Control /Looping structure
  - [1]Basic LOOP
  - [2]FOR..LOOP
  - [3]WHILE FOR...LOOP
- [3]Sequential Control
  - [1]GOTO Statement

CONDITIONAL CONTROL

- PL/SQL allows the use of an IF statement to control the execution of a block of code.
- PL/SQL has four conditional or selection statement available for decision making:
  - *1) IF…THEN…END IF*
  - *2) IF…THEN..ELSE…END IF*
  - *3) IF…THEN…ELSIF…END IF*
  - *4) CASE…ENDCASE*

1) IF…THEN…END IF

- A simple IF statement performs action statement if the result of the condition is TRUE.
- If the condition is FALSE no action is performed , and the program continues with the next statement in the block.



- Syntax:
  - If <condition>THEN

    <action>

    END IF

EXAMPLE :

```
DECLARE
      a number(2) := 10;

BEGIN
      a:= 10;

      -- check the boolean condition using if statement
      IF( a < 20 ) THEN

      -- if condition is true then print the following
         dbms_output.put_line('a is less than 20 ' );

       END IF;
      dbms_output.put_line('value of a is : ' || a);

END;
/
```

```
a is less than 20
value of a is : 10

Statement processed.
```

2) IF...THEN..ELSE...END IF

- It is an extension of the simple IF statement .
- It provides action statement for the TRUE outcome as well as for the FALSE outcome.

- Syntax:
  - If <condition>THEN
    <action>
    ELSE
    <some other action>;
    END IF;

EXAMPLE: 1

```
declare
        no number(3);
begin
        no:=20;
        if (no < 70)then
                dbms_output.put_line('smaller');
        else
                dbms_output.put_line('bigest');
        end if;
End;
/
```

EXAMPLE: 2

```
DECLARE
   a number(3) := 100;

BEGIN
   -- check the boolean condition using if statement
   IF( a < 20 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('a is less than 20 ');

   ELSE
      dbms_output.put_line('a is not less than 20 ');

   END IF;
   dbms_output.put_line('value of a is : ' || a);

END;
```

```
a is not less than 20
value of a is : 100

Statement processed.
```

3) IF…THEN…ELSIF…END IF

- It is an extension to the previous statement .
- When you have many alternatives/option, you can use previously explained statement but the ELSIF alternative is more efficient than the other two.
- Syntax:
  - If <condition>THEN

```
        <action>
ELSIF<condition Action>THEN
        <some other action>
ELSE
      <some other action>;
END IF;
```

EXAMPLE : 1

```
declare
        x number(3);
        y number(3);
begin
        x:=200;
        y:=100;
       if (x=y) then
                dbms_output.put_line('equal');
       elsif (x > y)then
                dbms_output.put_line('bigest');
        else
                dbms_output.put_line('smaller');
       end if;
end;
/
```

EXAMPLE : 2

```
DECLARE
    v_number NUMBER := 10;  -- Declare a variable

BEGIN
    -- Conditional statement
    IF v_number > 0 THEN
        DBMS_OUTPUT.PUT_LINE('The number is positive.');
    ELSIF v_number < 0 THEN
        DBMS_OUTPUT.PUT_LINE('The number is negative.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('The number is zero.');
    END IF;

END;
/
```

```
The number is positive.

Statement processed.
```

EXAMPLE : 3

```
DECLARE
   a number(3) := 100;

BEGIN
   IF ( a = 10 ) THEN
      dbms_output.put_line('Value of a is 10' );
   ELSIF ( a = 20 ) THEN
      dbms_output.put_line('Value of a is 20' );
   ELSIF ( a = 30 ) THEN
      dbms_output.put_line('Value of a is 30' );
   ELSE
       dbms_output.put_line('None of the values is matching');
   END IF;
   dbms_output.put_line('Exact value of a is: '|| a );

END;
/
```

```
None of the values is matching
Exact value of a is: 100

Statement processed.
```

EXAMPLE : 4

```
DECLARE
    v_score NUMBER := 85; -- Example score
    v_grade CHAR(1);      -- Variable to hold the grade
BEGIN
    IF v_score >= 90 THEN
        v_grade := 'A';
    ELSIF v_score >= 80 THEN
        v_grade := 'B';
    ELSIF v_score >= 70 THEN
        v_grade := 'C';
    ELSIF v_score >= 60 THEN
        v_grade := 'D';
    ELSE
        v_grade := 'F';
    END IF;

    DBMS_OUTPUT.PUT_LINE('The grade is: ' || v_grade);
END;
```

```
The grade is: B

Statement processed.
```

[4] CASE…ENDCASE

- Like the **IF** statement, the **CASE statement** selects one sequence of statements to execute.
- However, to select the sequence, the **CASE** statement uses a selector rather than multiple Boolean expressions

- Syntax
- CASE selector
  - WHEN 'value1' THEN S1;
  - WHEN 'value2' THEN S2;
  - WHEN 'value3' THEN S3;
  - …
  - ELSE Sn; -- default case

  END CASE;

EX.

```
DECLARE
   grade char(1) := 'B';
BEGIN
   CASE grade
      when 'A' then dbms_output.put_line('Excellent');
      when 'B' then dbms_output.put_line('Very good');
      when 'C' then dbms_output.put_line('Well done');
      when 'D' then dbms_output.put_line('You passed');
      when 'F' then dbms_output.put_line('Better try again');
      else
            dbms_output.put_line('No such grade');
   END CASE;
END;
/
```

[5] Nested IF-THEN-ELSE

- It is always legal in PL/SQL programming to nest the **IF-ELSE** statements, which means you can use one **IF** or **ELSE IF** statement inside another **IF** or **ELSE IF** statement(s).

- Syntax
- IF(boolean_expression 1)THEN
- -- executes when the boolean expression 1 is true
- IF(boolean_expression 2) THEN
- -- executes when the boolean expression 2 is true
- sequence-of-statements;
- END IF;
- ELSE
- -- executes when the boolean expression 1 is not true
- else-statements;
- END IF;

EX.

```
DECLARE
   a number(3) := 100;
   b number(3) := 200;
BEGIN
   -- check the boolean condition
   IF( a = 100 ) THEN
   -- if condition is true then check the following
      IF( b = 200 ) THEN
      -- if condition is true then print the following
      dbms_output.put_line('Value of a is 100 and b is 200' );
      END IF;
   END IF;
   dbms_output.put_line('Exact value of a is : ' || a );
   dbms_output.put_line('Exact value of b is : ' || b );
END;
/
```

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200

Statement processed.
```

ITERATIVE CONTROL &

LOOPING STRUCTURE

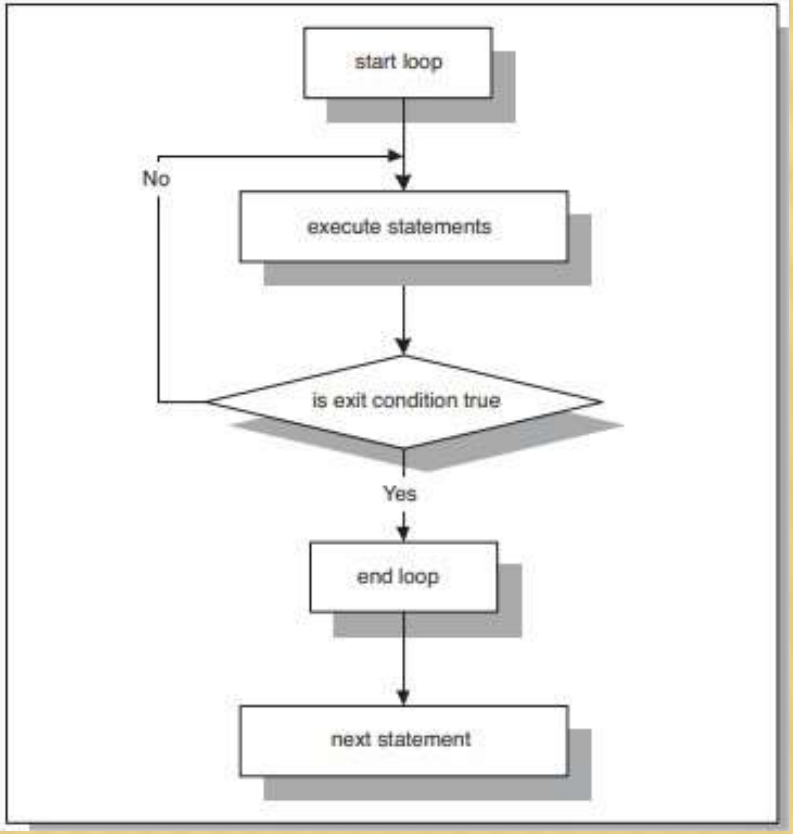
ITERATIVE CONTROL

- Iterative control statement perform one or more statements repeatedly , either a certain number of times or until a condition is meet. There three forms of iterative structures:
  - 1)Basic Loop.
  - 2)While..Loop.
  - 3)For..Loop.

1)Basic Loop.

- A basic loop is a loop that is performed repeatedly. once a loop is entered all statement in the loop are performed.

- Syntax:
  - Loop
    - <statement>
    - Exit [when <condition>];
    - Increment statement;
    - END LOOP



EX.

```
declare
    i number(3):=1;
begin
    loop
      exit when(i>=10);
            dbms_output.put_line(i);
      i:=i+1;
    end loop;
end;
/
```

```
1
2
3
4
5
6
7
8
9

Statement processed.
```

2)While...Loop.

- While loop has a condition associated with the loop.
- The condition is evaluated and if the condition is true the statement inside the loop are executed.
- Ex.
- While<condition>
  - Loop
    - <loop body statement>
    - Increment statement;
  - End loop

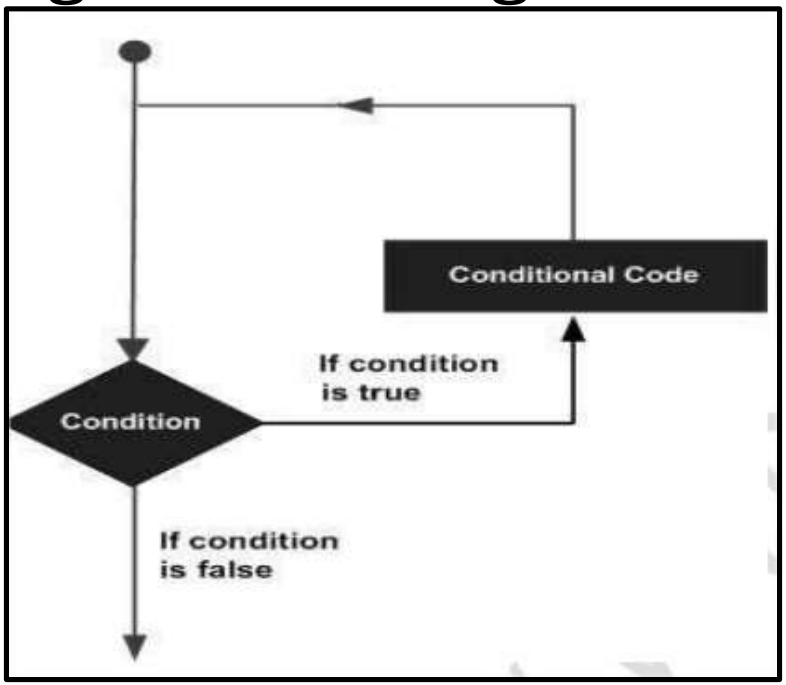
EX.

```
declare
      i number(3):=1;
begin
       while(i<10)
          loop
               dbms_output.put_line(i);
          i:=i+1;
       end loop;
end;
/
```

```
1
2
3
4
5
6
7
8
9

Statement processed.
```

3)For..Loop.

- We use FOR..LOOP if we want the iterations to occur a fixed number of times. The FOR..LOOP is executed for a range values.
- Syntax:
  - For<variable>IN <start range>..<end range>
  - Loop
    - <loop body statement>
  - End loop;



EX...(1)

```
DECLARE
   i number(1);
BEGIN
   -- outer_loop
      FOR i IN 1..3 LOOP
         dbms_output.put_line('i is: '| | i );
      END loop ;
END;
/
```

EX...(2)

```
DECLARE
   i number(1);
   j number(1);
BEGIN
   -- outer_loop
   FOR i IN 1..3 LOOP
      -- inner_loop
      FOR j IN 1..3 LOOP
         dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
      END loop ;
   END loop ;
END;
/
```

EX. (3)

```
declare
      i number(3):=1;
begin
          i:=100;
            for i in reverse 5..10
          loop
            dbms_output.put_line(i);
          end loop;
end;
/
```

SEQUENTIAL CONTROL

SEQUENTIAL CONTROL
GOTO STATEMENT

- A **GOTO statement** with a label may be used to pass control to another part of the program.
- ☞ Note: Using GOTO is not recommended in PL/SQL unless absolutely necessary (it makes the program harder to read and maintain).
- Syntax.
  - GOTO label;
    ...
    ...
    << label >>
    statement;
  - **label:** This is the name of the label to which control will be transferred. Labels are defined within ***double angle brackets*** (<< >>).

EX.

```
declare
      x number(3):=10;
begin
       loop
              x:=x+3;
              if x > 20 then
                  goto stop;
             end if;
        end loop;

        <<stop>>
              dbms_output.put_line('OUTSIDE LOOP...');
end;
/
```

- **Execution Flow:**
- x := 10
- Enter the infinite LOOP
  - Iteration 1 ⟶ x = 13 (not > 20, continue loop)
  - Iteration 2 ⟶ x = 16 (not > 20, continue loop)
  - Iteration 3 ⟶ x = 19 (not > 20, continue loop)
  - Iteration 4 ⟶ x = 22 (**> 20 ⟶ GOTO stop**)
- Control jumps to the label <<stop>>.

- Prints:
- OUTSIDE LOOP...
- Program ends successfully.

PROGRAM IN PL/SQL

PRO-1 NEXT VALUES GENERATE.

```
DECLARE
   x number := 10;
BEGIN
   LOOP
      dbms_output.put_line(x);
      x := x + 10;
      IF x > 50 THEN
         exit;
      END IF;
   END LOOP;
   -- after exit, control resumes here
   dbms_output.put_line('After Exit x is: ' || x);
END;
/
```

- O /p :
- 10
- 20
- 30
- 40
- 50
- After Exit x is: 60 Statement processed.

PRO-2 FACTORIAL PROGRAM

```
declare
        i number(4):=1;
        n number(4):=5;
        f number(4):=1;
 begin
    for i in 1..n
    loop
       f:=f*i;
          Dbms_output.put_line('The factorial of '||i||' is:'||f);
     end loop;
end;
/
```

- **Output :**

- The factorial of 1 is:1
- The factorial of 2 is:2
- The factorial of 3 is:6
- The factorial of 4 is:24
- The factorial of 5 is:120

- Statement processed.

PRO.-3 ODD EVEN NUMBER

```
Declare

BEGIN
    for i in 1..10
    loop
      if mod(i,2) = 0 then
         dbms_output.put_line(i||' is an even number');
      else
         dbms_output.put_line(i||' is an odd number');
     end if;
    end loop;
END;
/
```

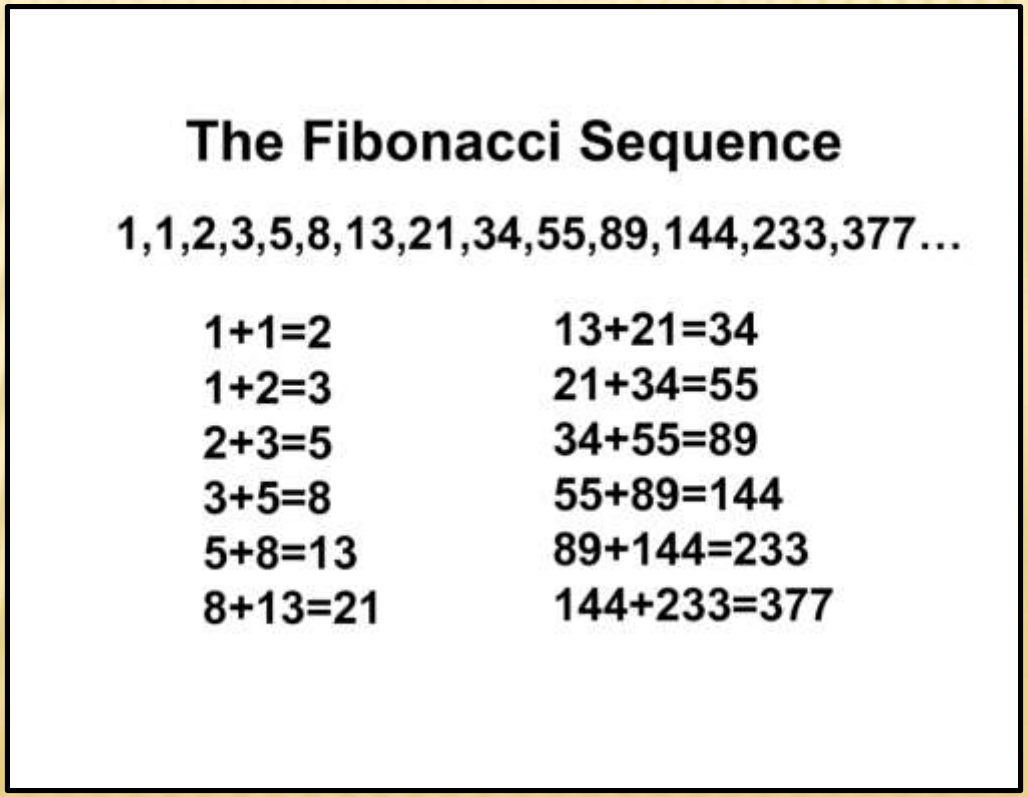
- 1 is an odd number
- 2 is an even number
- 3 is an odd number
- 4 is an even number
- 5 is an odd number
- 6 is an even number
- .....
- .....
- .....
- 10 is an even number

- Statement processed.

PRO-4 BLOCK TO GENERATE FIBONACCI SERIES.

```
declare
      a number:= 0;
      b number:= 1;
      c number;
begin
      dbms_output.put_line(a||' ');
      dbms_output.put_line(b||' ');
      for i in 3..10
      loop
        c := a + b;
        dbms_output.put_line(c||' ');
        a := b;
        b := c;
      end loop;
end;
```

- **Output:**
- 0 1 1 2 3 5 8 13 21 34
- PL/SQL procedure successfully completed



PRO-5 FIND SUM AND AVERAGE OF THREE NUMBERS.

```
declare
        a number:=1;
        b number:=2;
        c number:=3;
        sm number;
        av number;
begin
        sm:=a+b+c;
        av:=sm/3;
             dbms_output.put_line('Sum = '||sm);
             dbms_output.put_line('Average = '||av);
end;
/
```

- **Output :**
- Sum = 6
- Average = 2
- PL/SQL procedure successfully completed.

PRO.6 FIND REVERSE OF A NUMBER

```
declare
   N number;
   S NUMBER := 0;
   R NUMBER;
   K number;
begin
   N := 1234;
   K := N;
   loop
      exit WHEN N = 0;
      S := S * 10;
      R := MOD(N,10);
      S := S + R;
      N := TRUNC(N/10);
   end loop;
   dbms_output.put_line('THE REVERSED DIGITS OF '||K||' = '||S);
End;
```

- **Output :**

- THE REVERSED DIGITS OF 1234 = 4321
- Statement processed.

EX.2 REVERSED NUMBER

```
declare
        i number(3);
begin
         dbms_output.put_line('THE REVERSED DIGITS OF 5 6 7 8 9 10 is  ');
           for i in reverse 5..10
         loop
           dbms_output.put_line(i);
         end loop;
end;
/
```

```
THE REVERSED DIGITS OF 5 6 7 8 9 10 is
10
9
8
7
6
5

Statement processed.
```

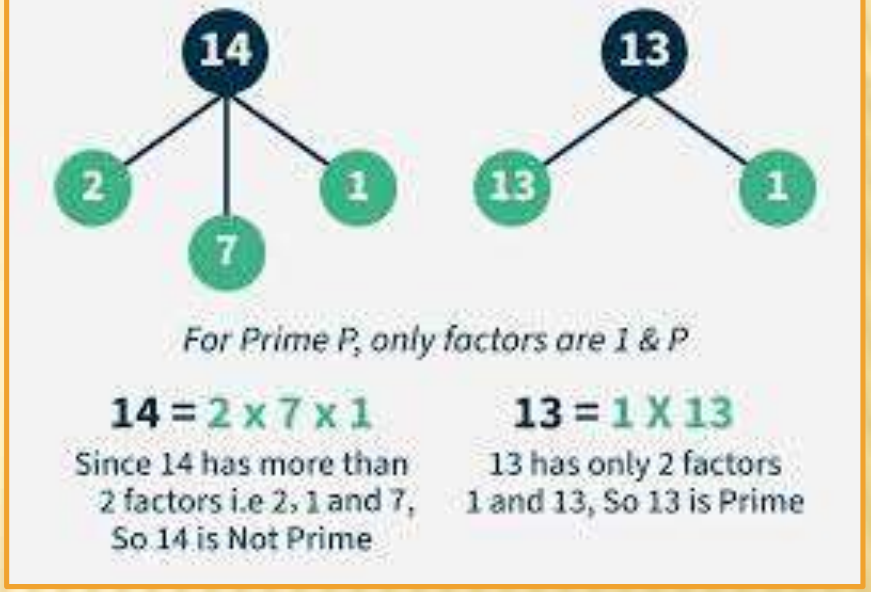
PRIME NUMBER

```
DECLARE
   i NUMBER(3);
   j NUMBER(3);
BEGIN
dbms_output.Put_line('The prime numbers are:');
          dbms_output.new_line;
   i := 2;
   LOOP
      j := 2;
      LOOP
         EXIT WHEN( ( MOD(i, j) = 0 )
                  OR ( j = i ) );
         j := j + 1;
      END LOOP;
      IF( j = i )THEN
        dbms_output.Put(i||'   ');
      END IF;
      i := i + 1;
      exit WHEN i = 50;
   END LOOP;
          dbms_output.new_line;
END;
/
```

```
The prime numbers are:

2   3   5   7   11   13   17   19   23   29   31   37   41   43   47

Statement processed.
```

PRIME NUMBER

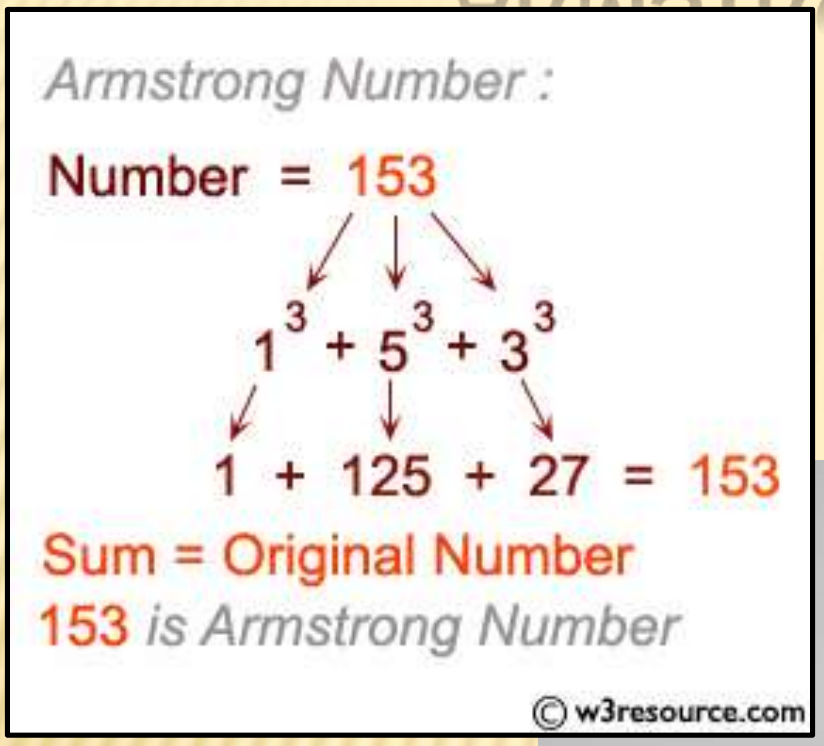
1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100



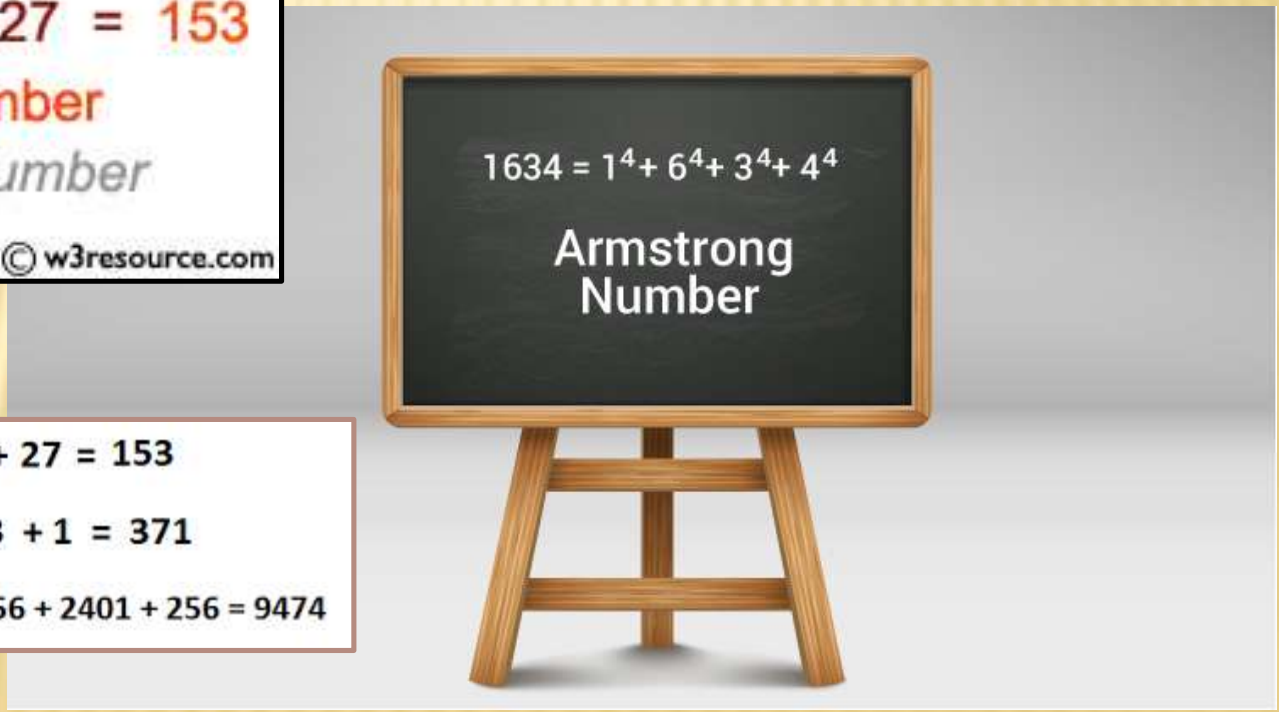
ARMSTRONG NUMBER

```
declare
   n number:=407;
   s number:=0;
   r number;
   len number;
   m number;

begin
   m:=n;

   len:=length(to_char(n));

   while n>0
   loop
      r:=mod(n,10);
      s:=s+power(r,len);
      n:=trunc(n/10);
   end loop;

   if m=s
   then
      dbms_output.put_line('armstrong number');
   else
      dbms_output.put_line('not armstrong number');
   end if;
   end;
```

ARMSTRONG NUMBER




```
Armstrong Numbers between 1 and 1000
1
153
370
371
407
```



$153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$

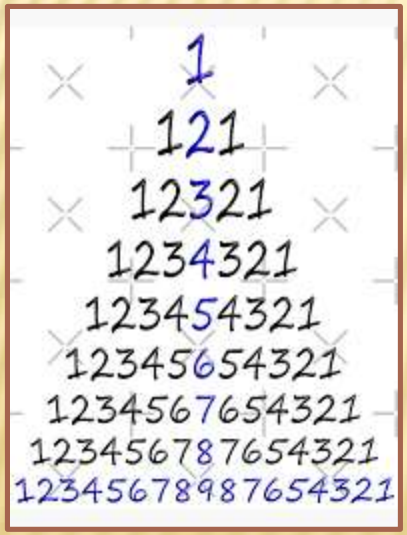
$371 = 3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371$

$9474 = 9^4 + 4^4 + 7 + 4^4 = 6561 + 256 + 2401 + 256 = 9474$

PALINDROME NUMBER IN PL/SQL

```
declare
    num number := 12321;   -- Change this to test
    temp number;
    rev number := 0;
    digit number;
begin
    temp := num;
    while temp > 0 loop
        digit := mod(temp, 10);       -- get last digit
        rev := (rev * 10) + digit;    -- build reverse number
        temp := trunc(temp / 10);     -- remove last digit
    end loop;

    if rev = num then
        dbms_output.put_line(num || ' is a Palindrome Number');
    else
        dbms_output.put_line(num || ' is Not a Palindrome Number');
    end if;
end;
/
```

- 1234321 is a Palindrome Number
- Statement processed.



4-Digit Palindrome	Divided by Eleven	4-Digit Palindrome	Divided by Eleven	4-Digit Palindrome	Divided by Eleven	4-Digit Palindrome	Divided by Eleven	4-Digit Palindrome	Divided by Eleven
1001	91	2882	262	4664	424	6446	586	8228	748
1111	101	2992	272	4774	434	6556	596	8338	758
1221	111	3003	273	4884	444	6666	606	8448	768
1331	121	3113	283	4994	454	6776	616	8558	778
1441	131	3223	293	5005	455	6886	626	8668	788
1551	141	3333	303	5115	465	6996	636	8778	798
1661	151	3443	313	5225	475	7007	637	8888	808
1771	161	3553	323	5335	485	7117	647	8998	818
1881	171	3663	333	5445	495	7227	657	9009	819
1991	181	3773	343	5555	505	7337	667	9119	829
2002	182	3883	353	5665	515	7447	677	9229	839
2112	192	3993	363	5775	525	7557	687	9339	849
2222	202	4004	364	5885	535	7667	697	9449	859
2332	212	4114	374	5995	545	7777	707	9559	869
2442	222	4224	384	6006	546	7887	717	9669	879
2552	232	4334	394	6116	556	7997	727	9779	889
2662	242	4444	404	6226	566	8008	728	9889	899
2772	252	4554	414	6336	576	8118	738	9999	909

REVERSE A STRING IN PL/SQL

```
DECLARE
   original_str VARCHAR2(100) := 'Hello, World!';
   reversed_str VARCHAR2(100) := '';
BEGIN
   FOR i IN REVERSE 1 .. LENGTH(original_str) LOOP
      reversed_str := (reversed_str || SUBSTR(original_str, i, 1));
     DBMS_OUTPUT.PUT_LINE('Reversed String: ' || reversed_str);
   END LOOP;

   DBMS_OUTPUT.PUT_LINE('Original String: ' || original_str);
   DBMS_OUTPUT.PUT_LINE('Reversed String: ' || reversed_str);
END;
/
```

- **Explanation:**
  - original_str is the string you want to reverse.
  - FOR i IN REVERSE loops backward from the end of the string.
  - SUBSTR(original_str, i, 1) gets each character from the end.
  - DBMS_OUTPUT.PUT_LINE prints the result.

- Output :

- Original String: Hello, World!
- Reversed String: !dlroW ,olleH.

- Statement processed.

WRITE A PL/SQL PROGRAM TO FIND THE SUM OF DIGITS OF A GIVEN NUMBER.

```
DECLARE
   v_num  NUMBER := 9875;   -- Input number
   v_sum  NUMBER := 0;      -- To store sum of digits
   v_rem  NUMBER;           -- To hold remainder (digit)
BEGIN
   WHILE v_num > 0 LOOP
      v_rem := MOD(v_num, 10);       -- Get last digit
      v_sum := v_sum + v_rem;        -- Add to sum
      v_num := TRUNC(v_num / 10);    -- Remove last digit
   END LOOP;

   DBMS_OUTPUT.PUT_LINE('Sum of digits = ' || v_sum);
END;
/
```

- For 9875 ⟶ 9 + 8 + 7 + 5 = 29

- Sum of digits = 29

ADVANCED PL/SQL

% TYPE AND % ROWTYPE

% TYPE

- → Used to declare a variable **with the same data type as a table column or another variable.**
- → *%TYPE is used to declare a field with the same type as that of a specified table's column:*
- →**This is particularly useful when you want to ensure that your variable matches the column's data type.**
- Syntax:
- *variable_name table_name.column_name%TYPE;*

EX. 1 MULTIPLE VARIABLES WITH %TYPE

```
declare
        no emp.id %type;
        nm emp.name %type;
        sal emp.salary %type;
begin
        select id,name,salary into no,nm,sal from emp WHERE ROWNUM = 1;
        dbms_output.put_line(no);
        dbms_output.put_line(nm);
        dbms_output.put_line(sal);
end;
/
```

EX. 2

- create table type (emp_name varchar(10),emp_id number(10),TA number(10),DA number(10),total number(10),branch_city varchar(10))

- insert into type values('ABC',10,1200,1500,2700,'DILHI');
- insert into type values('XYZ',20,1000,2000,NULL,'BANGLORE');
- insert into type values('PQR',30,5000,5000,NULL,'RAJKOT');
- select *from type

EMP_NAME	EMP_ID	TA	DA	TOTAL	BRANCH_CITY
ABC	10	1200	1500	2700	DILHI
XYZ	20	1000	2000	-	BANGLORE
PQR	30	5000	5000	-	RAJKOT

EX. 2 CONTI..... %TYPE

```
declare
  a type.TA %type;
  b type.DA %type;
  t type.total %type;
begin
  select TA,DA into a,b from type where emp_id=20;
  t:= a+b;
  update type set total=t where emp_id=20;
end;
```

EMP_NAME	EMP_ID	TA	DA	TOTAL	BRANCH_CITY
ABC	10	1200	1500	2700	DILHI
XYZ	20	1000	2000	3000	BANGLORE
PQR	30	5000	5000	-	RAJKOT

EX.3 VARIABLE BASED ON ANOTHER VARIABLE

```
DECLARE
    v_base_salary NUMBER(8,2);
    v_bonus v_base_salary%TYPE;  -- same type as v_base_salary

BEGIN
    v_base_salary := 50000;
    v_bonus := 5000;

    DBMS_OUTPUT.PUT_LINE('Salary: ' || v_base_salary || ' Bonus: ' || v_bonus);

END;
```

EX 4: INSERT USING %TYPE

```
DECLARE
    v_nm type.emp_name%TYPE := 'red';
    v_id type.emp_id%TYPE := '40';
    v_salary type.total%TYPE := 60000;
BEGIN
    INSERT INTO type(emp_name, emp_id, total)
    VALUES (v_nm, v_id, v_salary);

    DBMS_OUTPUT.PUT_LINE('Record Inserted!');
END;
```

%ROWTYPE

%ROWTYPE

- →%ROWTYPE **has all properties of %TYPE** and one additional what we required **only one variable to access** any number of columns.
- → *%ROWTYPE is used to declare a record with the same types as found in the specified database table, view or cursor:*
- →*T*his is useful when you want to work with multiple columns from a table without declaring each column individually.
- Syntax:
- *variable_name table_name%ROWTYPE;*

EX.1 SIMPLE %ROWTYPE WITH TABLE

```
declare
        my emp %rowtype;

begin
        select * into my from emp WHERE ROWNUM = 1;

        dbms_output.put_line('id is:'||my.id);
        dbms_output.put_line('name is:'||my.name);
        dbms_output.put_line('salary is:'||my.salary);
end;
/
```

EX.2 %ROWTYPE

- declare
- record type%ROWTYPE;

- begin
- select * into record from type where emp_id=30;
- record.total:=record.TA + record.DA;
- update type set total=record.total where emp_id=30;

- end;

EMP_NAME	EMP_ID	TA	DA	TOTAL	BRANCH_CITY
ABC	10	1200	1500	2700	DILHI
XYZ	20	1000	2000	3000	BANGLORE
PQR	30	5000	5000	10000	RAJKOT

EX 3: COPYING ONE ROW TO ANOTHER

```
DECLARE
   v_emp1 emp%ROWTYPE;
   v_emp2 emp%ROWTYPE;

BEGIN
   SELECT * INTO v_emp1 FROM emp WHERE id = 4;

   v_emp2 := v_emp1;  -- copy entire row

   DBMS_OUTPUT.PUT_LINE('Copied Employee: ' || v_emp2.name);

END;
/
```

CURSOR

USING CURSOR (IMPLICIT, EXPLICIT)

WHAT IS CURSOR ?

- The oracle engine uses a work area for its internal processing in order to execute an SQL statement. This work area is call ***CURSOR***.
- The **cursor** is used **to retrieve data one row at a time from the results set .**
- A cursor, either explicit or implicit, is used to handle the result set of a SELECT statement.
- The data stored in the cursor memory is call the 'ACTIVE DATA SET'.

TYPES OF CURSOR

- **(1)implicit cursor** (SQL cursor :open and managed by oracle)
- **(2)Explicit cursor** (user defined cursor : open and managed by user)

(1)IMPLICIT CURSOR

- It is a SQL cursor :open and managed by oracle engine internally.
- Implicit cursor using SELECT statement returning one row of data.
- The SQL cursor/implicit cursor four attributes:
  - SQL%found
  - SQL%notfound
  - SQL%rowcount
  - SQL%ISOPEN

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.**

- Create table emp (id number(3),name char(20),salary number(10))

- insert into emp values(1,'Mahesh',10000)
- insert into emp values(2,'Rajesh',20000)
- insert into emp values(3,'Dipesh',30000)
- insert into emp values(4,'Bhavesh',40000)

- select *from emp

ID	NAME	SALARY
1	Mahesh	10000
2	Rajesh	20000
3	Dipesh	30000
4	Bhavesh	40000

EX.1 SELECT INTO (IMPLICIT CURSOR) %FOUND

```
DECLARE
   v_name emp.name%TYPE;
BEGIN
   -- Implicit cursor created automatically
   SELECT name INTO v_name FROM emp WHERE id = 1;

   DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_name);

   -- Cursor attributes
   IF SQL%FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Row found.');
   END IF;
END;
/
```

EX.2 UPDATE WITH IMPLICIT CURSOR(%FOUND ,%NOTFOUND)

```
--Write a PL/SQL block to display message that whether a record is updated or  not.

Declare

begin
   update emp set name='Rakesh' where id=4;

   if SQL%FOUND then
     dbms_output.put_line('record updated');
   end if;

   if SQL%NOTFOUND then
      dbms_output.put_line('record not updated');
   end if;
end;
```

EX.3 %ROWCOUNT

```
declare
   num number(2);
begin
   update emp set salary=1500 where salary>15000;
   num:=SQL%ROWCOUNT;
   dbms_output.put_line('total rows affected =' ||num);
end;
```

EX.4 SQL%ROWCOUNT

```
DECLARE
   total_rows number(2);

BEGIN
   UPDATE emp SET salary = salary + 500;

   IF sql%notfound THEN
      dbms_output.put_line('no employee selected');

   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' employee selected ');

   END IF;
END;
/
```

(2)EXPLICIT CURSOR

- ◆ **What is an Explicit Cursor?**
- A **cursor** is like a pointer that helps you fetch rows one by one from a query result.
- In **explicit cursor**, you **manually declare, open, fetch, and close** the cursor.
- Useful when query returns **multiple rows.**
- Four action can be perform on explicit cursor:
  - Declare the cursor
  - Open the cursor
  - Fetch the data from cursor
  - Close the cursor

EXAMPLE (EXPLICIT CURSOR)

```
declare
        e_nm emp.name%type;
        e_sl emp.salary%type;
       cursor c1 is select name,salary from emp;
begin
       open c1;
       fetch c1 into e_nm,e_sl;
                dbms_output.put_line('salary of '|| e_nm ||' is '||e_sl);
       fetch c1 into e_nm,e_sl;
                dbms_output.put_line('salary of '|| e_nm ||' is '||e_sl);
       fetch c1 into e_nm,e_sl;
                 dbms_output.put_line('salary of '|| e_nm ||' is '||e_sl);
end;
```

O/p: salary of aaa is 3000
salary of bbb is 2000
salary of bbb is 5000
is Statement processed.

EXAMPLE PROGRAM.

```
-- Suppose we have an EMP table with columns ID, NAME, SALARY
DECLARE
  CURSOR emp_cur IS
    SELECT id, name, salary FROM emp;

  v_id     emp.id%TYPE;
  v_name   emp.name%TYPE;
  v_salary emp.salary%TYPE;
BEGIN
  OPEN emp_cur;

  LOOP
    FETCH emp_cur INTO v_id, v_name, v_salary;
    EXIT WHEN emp_cur%NOTFOUND;

    DBMS_OUTPUT.PUT_LINE('ID: ' || v_id ||
                         ', Name: ' || v_name ||
                         ', Salary: ' || v_salary);
  END LOOP;

  CLOSE emp_cur;
END;
```

FILL IN THE BLANKS

1. A _____ is a pointer to the result set of a SQL query.
   Answer: Cursor

2. _____ cursors are automatically created by Oracle for DML statements.
   Answer: Implicit

3. _____ cursors are created explicitly by the programmer.
   Answer: Explicit

4. The attributes %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN are used with _____.
   Answer: Cursors

5. In explicit cursors, after fetching rows, we must use the _____ statement to release memory.
   Answer: CLOSE

6. In implicit cursors, Oracle automatically handles _____, _____, and _____ operations.
   Answer: Open, Fetch, Close

7. The cursor attribute that gives the total number of rows fetched is _____.
   Answer: %ROWCOUNT

TRUE / FALSE

1. Implicit cursors are created automatically by Oracle for SELECT INTO statements.
   Answer: True

2. Explicit cursors require manual OPEN, FETCH, and CLOSE.
   Answer: True

3. %ISOPEN is always TRUE for implicit cursors.
   Answer: False (implicit cursors are closed automatically)

4. %NOTFOUND returns TRUE when no more rows are available.
   Answer: True

5. Cursors can only be used with SELECT statements.
   Answer: False (also used with INSERT, UPDATE, DELETE for row count checking)

EXCEPTION HANDLING

EXCEPTION HANDLING IN PL/SQL

- 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

```
DECLARE
   -- Declarations (variables, constants, cursors, etc.)
BEGIN
   -- Executable statements
EXCEPTION
   -- Exception handling section
   WHEN exception_name THEN
      -- Statements to handle the error
   WHEN OTHERS THEN
      -- Handle all other exceptions
END;
/
```

TYPES OF EXCEPTIONS IN PL/SQL

- PL/SQL provides **two categories** of exceptions:

- 1. Predefined Exceptions
- Oracle provides many built-in exceptions.
- Example:
  - NO_DATA_FOUND,
  - ZERO_DIVIDE,
  - TOO_MANY_ROWS,
  - INVALID_NUMBER, etc.
- These are automatically raised by Oracle.

- 2. User-Defined Exceptions
- Developers can define their own exceptions and raise them using RAISE.

EX.1 NO_DATA_FOUND

```
DECLARE
  v_name emp.name %type;
BEGIN
  select name into v_name from emp where id=10;
  dbms_output.put_line(v_name);
EXCEPTION
  when NO_DATA_FOUND then
      dbms_output.put_line('Data not found');
  when OTHERS then
      dbms_output.put_line('Error occur');
END;
```

✓ Output: Error: Data not found

EX.2 PREDEFINED EXCEPTION (ZERO_DIVIDE)

```
DECLARE
   v_result NUMBER;
BEGIN
   v_result := 10 / 0;   -- Will cause ZERO_DIVIDE exception
   DBMS_OUTPUT.PUT_LINE('Result: ' || v_result);

EXCEPTION
   WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Error: Division by zero is not allowed.');
END;
/
```

✓ Output: Error: Division by zero is not allowed.

EX.3 PREDEFINED EXCEPTION (TOO_MANY_ROWS)

```
-- Example: TOO_MANY_ROWS Exception
DECLARE
   v_name  emp.name%TYPE;  -- Variable to hold employee name
BEGIN
   -- This query will return more than one row if id = 10 has multiple employees
   SELECT name INTO v_name FROM emp WHERE id = 10;

   DBMS_OUTPUT.PUT_LINE('Employee Name: ' || v_name);

EXCEPTION
   WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE('Error: More than one employee found for
id10.');
   WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Some other error occurred: ' || SQLERRM);
END;`
```

✓ Output: Some other error occurred: ORA-01403: no data found

- *EX.* *EXCEPTION HANDLING*

- *declare*
- *e_name emp.name%type;*
- *e_salary emp.salary%type;*
- *begin*
- *select name into e_name from emp;*
- *exception*
- *when no_data_found then*
- *dbms_output.put_line('record does not exits');*
- *when too_many_rows then*
- *dbms_output.put_line('multiple rows retrieved');*
- *when others then*
- *dbms_output.put_line('errors in retrieval');*
- *end;*

USER-DEFINED EXCEPTION EXAMPLE

```
DECLARE
  insufficient_balance EXCEPTION;
  v_balance NUMBER := 500;
  v_withdraw NUMBER := 1000;
BEGIN
  IF v_withdraw > v_balance THEN
    RAISE insufficient_balance; -- Raise user-defined exception
  ELSE
    v_balance := v_balance - v_withdraw;
  END IF;

  DBMS_OUTPUT.PUT_LINE('Remaining Balance: ' || v_balance);

EXCEPTION
  WHEN insufficient_balance THEN
    DBMS_OUTPUT.PUT_LINE('Error: Withdrawal amount exceeds balance.');
END;
/
```

✓ Output: Error: Withdrawal amount exceeds balance.

EX.2 EXAMPLE: NEGATIVE NUMBER NOT ALLOWED

```
DECLARE
  negative_number EXCEPTION;  -- user-defined exception
  v_num NUMBER := -5;
BEGIN
  IF v_num < 0 THEN
    RAISE negative_number;   -- raise exception
  END IF;

  DBMS_OUTPUT.PUT_LINE('Number is: ' || v_num);

EXCEPTION
  WHEN negative_number THEN
    DBMS_OUTPUT.PUT_LINE('Error: Negative numbers are not allowed.');
END;
```

✓ Output: Error: Negative numbers are not allowed.

CREATING AND USING PROCEDURE

'PL/SQL - PACKAGES'.
ADVANCED PL/SQL

- **'PL/SQL - Packages'.**
- PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms −
- **Procedures** − These subprograms ***do not return a value directly***; mainly used to perform an action.
- **Functions** − These subprograms ***return a single value***; mainly used to compute and return a value.

CREATING A PROCEDURE

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

- CREATE [OR REPLACE] PROCEDURE procedure_name
- [(parameter_name [IN | OUT | IN OUT] type [, ...])]
- {IS | AS}
- BEGIN
- < procedure_body >
- END procedure_name; ;

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

EXAMPLE

- CREATE OR REPLACE PROCEDURE greetings
- AS
- BEGIN
- dbms_output.put_line('Hello World!');
- END;

- Executing a called from another PL/SQL block
- BEGIN
- greetings;
- END;
- O/p:
  - 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;

THERE ARE THREE MODES:

- 1. IN Mode (Default)
- 2. OUT Mode
- 3. IN OUT Mode

1. IN MODE (DEFAULT)

- Passes a value **into** the procedure/function.
- It is **read-only** inside the subprogram (cannot change the caller's variable).

```
CREATE OR REPLACE PROCEDURE show_square (p_num IN NUMBER) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Square is: ' || (p_num * p_num));
END;

BEGIN
  show_square(5);
END;
```

- ☞ Output: Square is: 25

2. OUT MODE

- Used to return a value **from** the subprogram back to the caller.
- The variable **cannot be used before a value is assigned** inside the subprogram.

```
CREATE OR REPLACE PROCEDURE calc_square (p_num IN NUMBER, p_result OUT NUMBER) IS
BEGIN
  p_result := p_num * p_num;
END;

DECLARE
  v_result NUMBER;
BEGIN
  calc_square(6, v_result);
  DBMS_OUTPUT.PUT_LINE('Square is: ' || v_result);
END;
```

- ☞ Output: Square is: 36

3. IN OUT MODE

- Works as both **input and output.**
- Passes a value into the subprogram, and the subprogram can **modify and return** it.

```
CREATE OR REPLACE PROCEDURE double_value (p_num IN OUT NUMBER)
IS
BEGIN
  p_num := p_num * 2;
END;

DECLARE
  v_num NUMBER := 10;
BEGIN
  double_value(v_num);
  DBMS_OUTPUT.PUT_LINE('Doubled value: ' || v_num);
END;
```

- ☞ Output: Doubled value: 20

FUNCTIONS

WHAT IS A **FUNCTION** ?

- In PL/SQL, a **function** is a named block of code that **performs a specific task** and **returns a single value.**
- Unlike a **procedure** (which does not return a value), a **function must return a value** using the RETURN keyword.

SYNTAX OF A FUNCTION IN PL/SQL

```
CREATE OR REPLACE FUNCTION function_name (
    param1 IN datatype,
    ...
) RETURN return_datatype
IS
    -- variable declarations
BEGIN
    -- logic
    RETURN result_value;
END;
```

EXAMPLE 1: A SIMPLE FUNCTION

- -- Task: Create a function that calculates the **SQUARE** given number.

- CREATE OR REPLACE FUNCTION get_square (
- p_num IN NUMBER)
- RETURN NUMBER
- IS
- BEGIN
- RETURN p_num * p_num;
- END;

USING (CALLING) THE FUNCTION

- You can call a function in:
  - A **PL/SQL block**
  - A **SQL query** (if the function is **deterministic** and doesn't access tables)

- ✓ **Example in PL/SQL:**
- SELECT get_square(5) FROM dual; -- returns 25

- Output : GET_SQUARE(80)
- 6400

EXAMPLE 2:

```
-- Task: Create a function that calculates the annual salary given a monthly salary.
CREATE OR REPLACE FUNCTION get_annual_salary (
    p_monthly_salary IN NUMBER
) RETURN NUMBER
IS
    v_annual_salary NUMBER;
BEGIN
    v_annual_salary := p_monthly_salary * 12;
    RETURN v_annual_salary;
END;
```

USING (CALLING) THE FUNCTION

- You can call a function in:
  - A **PL/SQL block**
  - A **SQL query** (if the function is **deterministic** and doesn't access tables)

- ✓ **Example in PL/SQL:**

```
DECLARE
    v_salary NUMBER := 3000;
    v_annual NUMBER;
BEGIN
    v_annual := get_annual_salary(v_salary);
    DBMS_OUTPUT.PUT_LINE('Annual Salary: ' || v_annual);
END;
```

- Output : Annual Salary: 36000

Package

WHAT IS A PL/SQL PACKAGE?

- A **PL/SQL package** is a group of related procedures, functions, variables, and other PL/SQL constructs grouped together in a single unit. It has two parts:

- **Package Specification** – The interface (what is visible to users).
- **Package Body** – The implementation (actual code).

- **Key Points:**
- **Modularity:** Packages let you organize your code into logical units.
- **Two parts:**
  - **Package Specification:** The interface — declares the public elements (procedures, functions, variables) that other programs can use.
  - **Package Body:** The implementation — contains the actual code for those procedures/functions.
- **Encapsulation:** You can hide internal details (private code) in the package body, exposing only what's necessary.
- **Reusability:** Once created, packages can be reused by multiple programs.
- **Performance:** When you call a packaged procedure or function for the first time, the entire package is loaded into memory, making subsequent calls faster.

TRIGGER

- ✓ What is a Trigger in PL/SQL?
- A **trigger** in PL/SQL is a **stored program** that is **automatically executed (or "fired")** in response to certain events on a table or view.

- ✓ In Simple Terms:
- A **trigger** is like a **watchdog** on your table.
- Whenever something happens (like an insert, update, or delete), the trigger **automatically runs** some code **without you having to call it manually.**

- WRITE A PL/SQL CODE WHICH WILL INSERT THE DATA AUTOMATICALLY INTO ‘BACKUP’ TABLE WHENEVER DATA IS DELETED FROM ‘MAIN’ TABLE.

- ☐ STEP 1: Create MAIN Table
- CREATE TABLE MAIN (ID INT,SALARY INT);

- INSERT INTO MAIN VALUES (1,10000);
- INSERT INTO MAIN VALUES (2,20000);
- SELECT *FROM MAIN;

- ☐ STEP 2: Create BACKUP Table
- CREATE TABLE BACKUP (ID INT,SALARY INT);

- 🛠STEP 3: Create the Trigger

- CREATE OR REPLACE TRIGGER T1
- BEFORE DELETE ON MAIN
- FOR EACH ROW
- BEGIN
- INSERT INTO BACKUP VALUES(:OLD.ID,:OLD.SALARY);
- END;

- SELECT *FROM BACKUP;

- 🚀 STEP 4: Test the Trigger

- DELETE FROM MAIN WHERE ID=1;

- SELECT *FROM MAIN;

- 🔍 STEP 5: Check the BACKUP Table
- SELECT *FROM BACKUP;

EX.2

- --Create a trigger to display salary changes in the customer table.
- --First create customer table and insert records.
- create table customers (id number(3), name varchar2(20), age number(2), address varchar2(20), salary number(10));
- insert into customers values(1,‘aarva',25,‘ahmedabad',55000);
- insert into customers values(2,'hetansh',4,'pune',80000);
- insert into customers values(3,'riyan',20,'surat',90000);
- select *From customers

ID	NAME	AGE	ADDRESS	SALARY
1	aarva	25	ahmedabad	55000
2	hetansh	4	pune	80000
3	riyan	20	surat	90000

CREATE A TRIGGER

```
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;
```

Output :- Trigger created.

PL/SQL BLOCK TO UPDATE CUSTOMER TABLE.

```
DECLARE
    total_rows number(2);
BEGIN
    UPDATE customers SET salary = salary + 5000;
    IF sql%notfound THEN
      dbms_output.put_line('no customers updated');
    ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers updated ');
    END IF;
END;
```

Output : -

Old salary: 50000 New salary: 55000 Salary difference: 5000
Old salary: 75000 New salary: 80000 Salary difference: 5000
Old salary: 85000 New salary: 90000 Salary difference: 5000
3 customers updated

PL/SQL- ARRAYS

PL/SQL– VARRAYS

- The PL/SQL programming language provides a data structure called the VARRAY.
- which can store a fixed-size sequential collection of elements of the same type.
- A varray is used to store an ordered collection of data, however it is often better to think of an array as a collection of variables of the same type.
- All varrays consist of contiguous memory locations.

- ✓ PL/SQL VARRAY Syntax

- TYPE type_name IS VARRAY(size_limit) OF element_type;
- variable_name type_name;

EXAMPLE 1:

```
DECLARE
    -- Step 1: Define a VARRAY type
    TYPE NumberArray IS VARRAY(5) OF NUMBER;

    -- Step 2: Declare a variable of that type
    my_numbers NumberArray := NumberArray(10, 20, 30);

BEGIN
    -- Step 3: Use the VARRAY
    FOR i IN 1 .. my_numbers.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE('Element ' || i || ': ' || my_numbers(i));
    END LOOP;
END;
```

☐ Output:

```
Element 1: 10
Element 2: 20
Element 3: 30
```

EXAMPLE 2:

```
DECLARE
   type namesarray IS VARRAY(5) OF VARCHAR2(10);
   type grades IS VARRAY(5) OF INTEGER;
   names namesarray;
   marks grades;
   total integer;
BEGIN
   names := namesarray('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;
```

```
Total 5 Students
Student: Kavita
Marks: 98
Student: Pritam
Marks: 97
Student: Ayan
Marks: 78
Student: Rishav
Marks: 87
Student: Aziz
Marks: 92

Statement processed.
```

PL/SQL Table

WHAT IS A PL/SQL TABLE?

- A **PL/SQL Table** (now called **Associative Array**) is a one-dimensional collection that can store an **unbounded number of elements.**
- It is indexed using **numbers** (like an array) or sometimes **strings** (from Oracle 9i onwards).
- Data is stored in **memory only** (cannot be stored directly in a DB column).

- Syntax

- TYPE type_name IS TABLE OF element_type
- INDEX BY BINARY_INTEGER; -- or PLS_INTEGER

EXAMPLE 1: SIMPLE PL/SQL TABLE OF NUMBERS

```
DECLARE
  -- Step 1: Define PL/SQL table type
  TYPE num_table IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;

  -- Step 2: Declare variable of this type
  my_numbers num_table;

BEGIN
  -- Step 3: Assign values
  my_numbers(1) := 100;
  my_numbers(2) := 200;
  my_numbers(3) := 300;

  -- Step 4: Display values
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE('Value at index ' || i || ' = ' || my_numbers(i));
  END LOOP;
END;
```

```
Value at index 1 = 100
Value at index 2 = 200
Value at index 3 = 300
```

Nested Table

WHAT IS A NESTED TABLE IN PL/SQL?

- A **Nested Table** in PL/SQL is a collection type (like an array) that can store **an unbounded number of elements** of the same data type.
- Unlike **VARRAY**, it is not limited to a fixed size.
- Unlike **Associative Arrays**, nested tables can be stored in the database columns and manipulated with SQL.

- Syntax for Declaring Nested Table

- TYPE type_name IS TABLE OF element_type;

EXAMPLE 1: SIMPLE NESTED TABLE OF NUMBERS

```
DECLARE
  -- Step 1: Define Nested Table Type
  TYPE num_table IS TABLE OF NUMBER;

  -- Step 2: Declare a variable of nested table
  my_numbers num_table := num_table();

BEGIN
  -- Step 3: Extend and Add Elements
  my_numbers.EXTEND(3);
  my_numbers(1) := 10;
  my_numbers(2) := 20;
  my_numbers(3) := 30;

  -- Step 4: Loop through table
  FOR i IN 1..my_numbers.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE('Value: ' || my_numbers(i));
  END LOOP;
END;
```

Value: 10
Value: 20
Value: 30


The end
?