UNIT-IV : PL / SQL: BASIC CONCEPTS - TABLES AND RECORDS MANIPULATIONS – CONTROL STATEMENTS – ANONYMOUS BLOCKS, STORED PROCEDURES, FUNCTIONS, EXCEPTION HANDLING, CURSORS AND TRIGGERS.

PL / SQL - INTRODUCTION

- PL/SQL is combination of SQL along with the procedural features of programming languages.

FEATURES OF PL/SQL

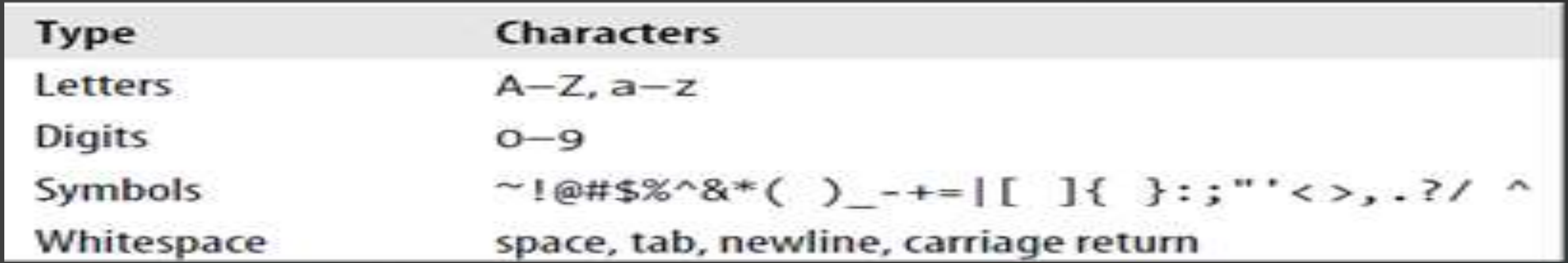
- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking , numerous data types and a variety of programming structures.
- It supports structured programming through functions and procedures, object-oriented programming and the development of web applications and server pages.
- PL/SQL is a completely portable, high-performance transaction-processing language.
- It provides a built-in, interpreted and OS independent programming environment.
- PL/SQL CAN BE DIRECTLY BEING CALLED FROM THE COMMAND-LINE SQL*PLUS INTERFACE AND EXTERNAL PROGRAMMING LANGUAGE CALLS TO DATABASE.

Advantages of PL/SQL

1. PL/SQL is strongly integrated with SQL. It supports both Static and Dynamic SQL.
   - The Static SQL supports DML operations and transaction control from PL/SQL block.
   - The Dynamic SQL, allows embedding DDL statements in PL/SQL blocks.
2. PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
3. PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
4. PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
5. Applications written in PL/SQL are fully portable.
6. PL/SQL provides
   1. high security.
   2. access to predefined SQL packages.

PL/SQL BASICS

PL/SQL Character Set



Type	Characters
Letters	A–Z, a–z
Digits	0–9
Symbols	~!@#$%^&*( )_-+=|[ ]{ }:;"'< >,.?/ ^
Whitespace	space, tab, newline, carriage return

IDENTIFIERS

- Identifiers are names for PL/SQL objects such as constants, variables, exceptions, procedures, cursors.
- Identifiers have the following characteristics:
  - Can be up to 30 characters in length
  - Cannot include whitespace (space, tab, carriage return)
  - Must start with a letter
  - Can include a dollar sign ($), an underscore (_), and a pound sign (#)
  - Are not case-sensitive
- Reserved words must not be used as Identifiers.

PL/SQL BASICS

Comments

- Comments are sections of code that exist to aid readability. The compiler ignores them.
- A single-line comment begins with a double hyphen (--) and ends with a new line.
- A multiline comment begins with slash asterisk (/*) and ends with asterisk slash (*/).
- The /* */ comment delimiters also can be used for a single-line comment.
- We cannot embed multiline comments within a multiline comment

Statements

- A PL/SQL program is composed of one or more logical statements.
- A statement is terminated by a semicolon delimiter.
- The physical end-of-line marker in a PL/SQL program is ignored by the compiler, except to terminate a single-line comment (initiated by the -- symbol).

Delimiters - *Delimiters are symbols with special meaning.*

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
;	Statement terminator
:=	Assignment operator
| |	Concatenation 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

PL/SQL - BASIC SYNTAX

- Every PL/SQL statement ends with a semicolon (;).
- A PL/SQL block can be anonymous or named.
- Named blocks include functions, procedures, packages, and triggers.
- PL/SQL programs are divided and written in logical blocks of code.
- Each PL/SQL block consists of three sub-parts –

Syntax

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling>
END;
```

PL/SQL - BASIC SYNTAX

Syntax

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling>
END;
```

Note :

- The END ; signals the end of the PL/SQL block.
- To run the code from the SQL command line, you may need to type / at the beginning of the first blank line after the last line of the code
- PL/SQL blocks can be nested within other PL/SQL blocks using BEGIN and END.

Declarations

- This **optional section** starts with the keyword **DECLARE**.
- This section **defines all variables, cursors, subprograms, and other elements to be used in the program**.

Executable Commands

- This **mandatory section** is enclosed **between the keywords BEGIN and END.**
- This section **consists of the executable PL/SQL statements of the program.**
- This section **should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed**.

Exception Handling

- This **optional section** starts with the keyword **EXCEPTION**.
- This section **contains exception handling code that handle errors in the program**

EXAMPLES

Example of an anonymous block:

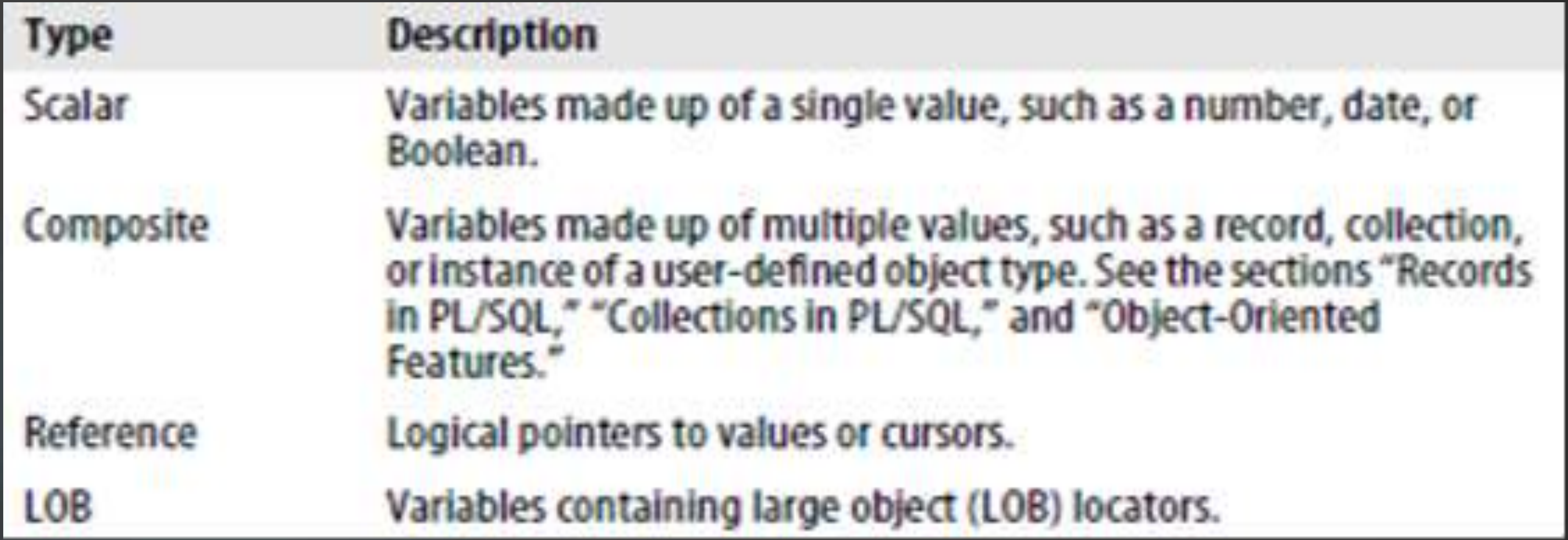
```
DECLARE
            today DATE DEFAULT SYSDATE;
BEGIN
           -- Display the date.
DBMS_OUTPUT.PUT_LINE ('Today is ' || today);
END;
```

Example of a named block

```
CREATE OR REPLACE PROCEDURE show_the_date
IS
            today DATE DEFAULT SYSDATE;

BEGIN
             -- Display the date.
      DBMS_OUTPUT.PUT_LINE ('Today is ' || today);
END show_the_date;
```

PL / SQL DATA TYPES



Type	Description
Scalar	Variables made up of a single value, such as a number, date, or Boolean.
Composite	Variables made up of multiple values, such as a record, collection, or instance of a user-defined object type. See the sections "Records in PL/SQL," "Collections in PL/SQL," and "Object-Oriented Features."
Reference	Logical pointers to values or cursors.
LOB	Variables containing large object (LOB) locators.

Scalar Datatypes

- Scalar datatypes divide into four families: number, character, datetime, and Boolean.
- Subtypes further define a base datatype by restricting the values or size of the base datatype.

Scalar Datatypes - Numeric datatypes

- Numeric datatypes represent real numbers, integers, and floating-point numbers.
- Decimal numeric datatypes store fixed and floating-point numbers of just about any size. They include the subtypes

  NUMBER, DEC, DECIMAL, NUMERIC, FLOAT, REAL and DOUBLE PRECISION.
- Variables of type NUMBER can be declared with precision and scale, as follows:

  NUMBER(precision, scale)

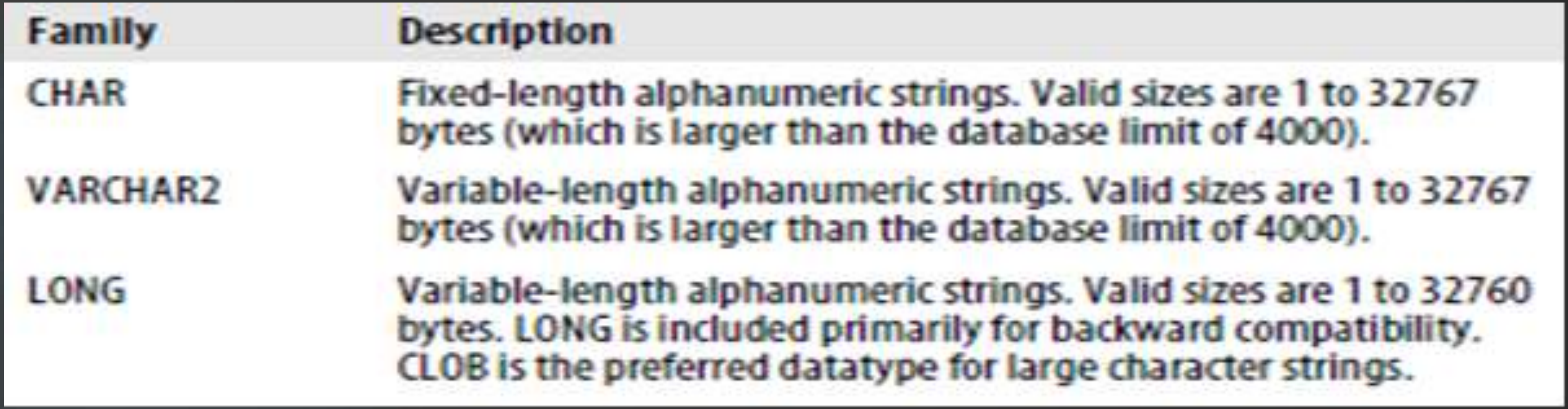
where

precision is the number of digits, and

scale is the number of digits to the right of the decimal point at which rounding occurs.

Scalar Datatypes - Character datatypes

- Character datatypes store alphanumeric text and are manipulated by character functions.
- There main subtypes in the character family, shown in the following table and the other subtypes that deal with Binary strings are RAW, LONG RAW, ROWID, UROWID



Family	Description
CHAR	Fixed-length alphanumeric strings. Valid sizes are 1 to 32767 bytes (which is larger than the database limit of 4000).
VARCHAR2	Variable-length alphanumeric strings. Valid sizes are 1 to 32767 bytes (which is larger than the database limit of 4000).
LONG	Variable-length alphanumeric strings. Valid sizes are 1 to 32760 bytes. LONG is included primarily for backward compatibility. CLOB is the preferred datatype for large character strings.

Scalar Datatypes

Scalar Datatypes - Datetime Types

- The DATE datatype is used to store fixed-length datetimes, which include the time of day in seconds since midnight.
- Valid dates range from January 1, 4712 BC to December 31, 9999 AD.
- The default date format is set by the Oracle initialization parameter NLS_DATE_FORMAT.

  For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year.
  Example, 01-OCT-12.
- Each DATE includes the century, year, month, day, hour, minute, and second.

Scalar Datatypes - Boolean Data Types

- The BOOLEAN data type stores logical values that are used in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL.
- However, SQL has no data type equivalent to BOOLEAN. Therefore, Boolean values cannot be used in –
  - SQL statements
  - Built-in SQL functions (such as TO_CHAR)
  - PL/SQL functions invoked from SQL statements

Large Object (LOB) Data Types

Large Object (LOB) data types **refer to large data items such as text, graphic images, video clips, and sound waveforms.**

LOB data types allow efficient, random, piecewise access to this data.

Following are the predefined PL/SQL LOB data types –

Data Type	Description
BFILE	**Used to store large binary objects in operating system files outside the database.**
BLOB	**Used to store large binary objects in the database.**
CLOB	**Used to store large blocks of character data in the database.**
NCLOB	**Used to store large blocks of NCHAR data in the database. (nchar and nvarchar can store Unicode characters).**

PL/SQL User-Defined Subtypes

- PL/SQL provides subtypes of data types.
- A subtype is a subset of a base type.
- A SUBTYPE HAS THE SAME VALID OPERATIONS AS ITS BASE TYPE, BUT ONLY A SUBSET OF ITS VALID VALUES.
- We can define and use our own subtypes.
- PL/SQL predefines several subtypes in package STANDARD.
  - For example, PL/SQL predefines the subtypes CHARACTER and INTEGER as follows −

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

Note :

You can use the subtypes in your PL/SQL program to make the data types compatible with data types in other programs while embedding the PL/SQL code in another program, such as a Java program.

Example

```
DECLARE
   SUBTYPE name IS char(20);
   SUBTYPE message IS varchar2 (100);
   salutation name;
   greetings message;
BEGIN
   salutation := 'Reader ';
   greetings := 'Welcome to the World of PL/SQL';
   dbms_output.put_line('Hello ' || salutation || greetings);
END;
/
```

NULL in PL/SQL

- **NULL values represent missing or unknown data and they are not an integer, a character, or any other specific data type.**
- **NULL is not the same as an empty data string or the null character value '\0'.**

- **A NULL CAN BE ASSIGNED BUT IT CANNOT BE EQUATED WITH ANYTHING, INCLUDING ITSELF.**
- **We cannot check for *equality or inequality* to NULL; therefore, we must use the IS NULL or IS NOT NULL to check for NULL values.**

Example:

```
BEGIN
IF myvar IS NULL   THEN
 ....
 END IF;
```

PL/SQL - VARIABLES

- A variable is a name given to a storage area that programs can manipulate.
- Each variable in PL/SQL has a specific data type,.
- The name of a PL/SQL variable consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs and should not exceed 30 characters. By default, variable names are not case-sensitive.
- We cannot use a reserved PL/SQL keyword as a variable name.
- PL/SQL allows to define various types of variables, such as date time data types, records, collections, etc..
-

Variable Declaration in PL/SQL

- PL/SQL variables must be declared in the declaration section or in a package as a global variable.
- When we declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is –

variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type .

Example:

sales number(10, 2);

pi CONSTANT double precision := 3.1415;

name varchar2(25);

Constrained Declaration :

- When we provide a size, scale or precision limit with the data type, it is called a **constrained declaration.**
- Constrained declarations **require less memory than unconstrained declarations.**

Example –

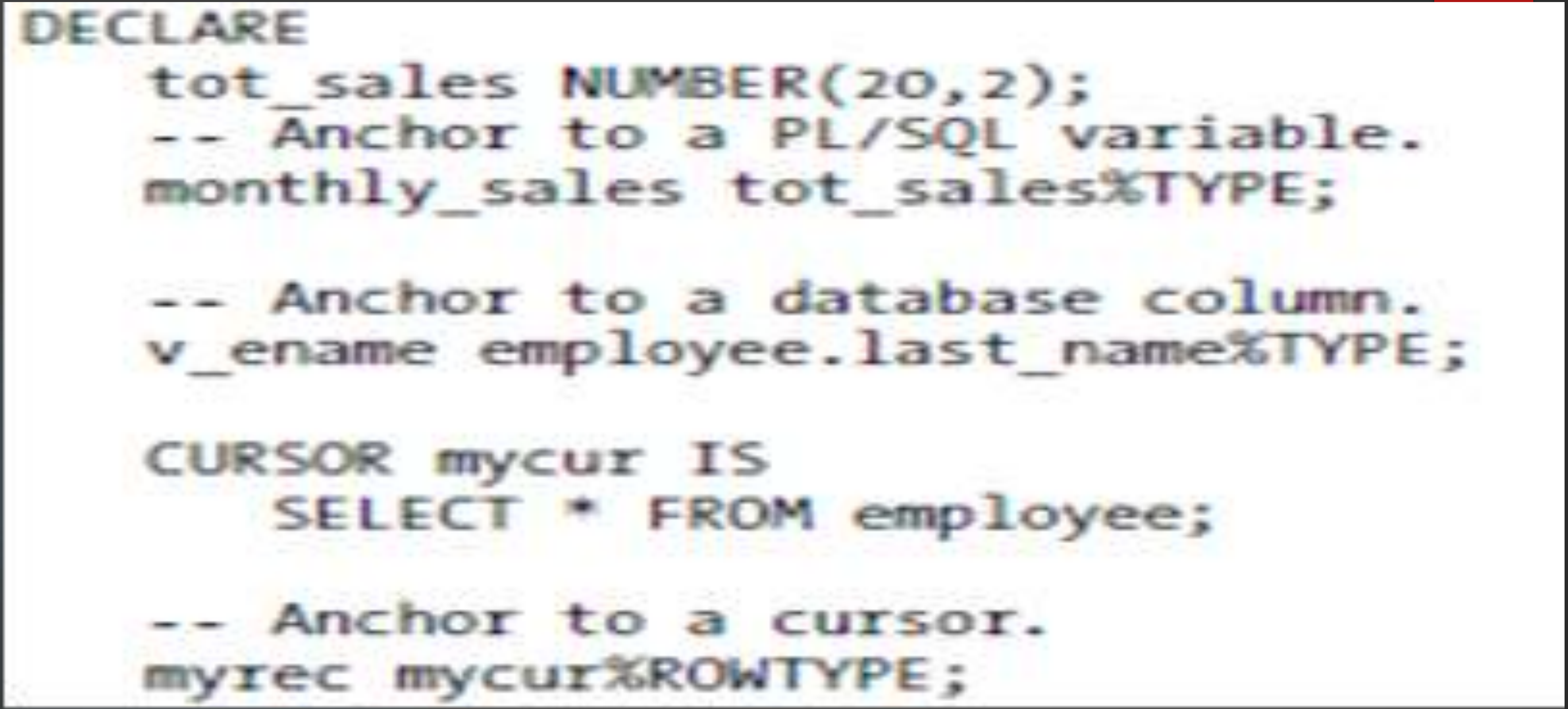
```
sales number(10, 2);
name varchar2(25);
```

Anchored Declarations :

- **Anchored Declarations** use %TYPE attribute to anchor the datatype of a scalar variable to either another variable or to a column in a database table or view.
- **Anchored Declarations** use %ROWTYPE to anchor a record's declaration to a cursor or table.
- The NOT NULL clause on a variable declaration (but not on a database column definition) follows the %TYPE anchoring and requires anchored declarations to have a default in their declaration.
  - The default value for an anchored declaration can be different from that for the base declaration:

```
tot_sales NUMBER (20,2) NOT NULL DEFAULT 0;
monthly_sales tot_sales%TYPE DEFAULT 10;
```

The following block shows several variations of anchored declarations



```
DECLARE
   tot_sales NUMBER(20,2);
   -- Anchor to a PL/SQL variable.
   monthly_sales tot_sales%TYPE;

   -- Anchor to a database column.
   v_ename employee.last_name%TYPE;

   CURSOR mycur IS
      SELECT * FROM employee;

   -- Anchor to a cursor.
   myrec mycur%ROWTYPE;
```

Initializing Variables in PL/SQL

- **When a variable is declared, PL/SQL assigns it a default value of NULL.**
- **We can specify that a variable should not have a NULL value using the NOT NULL constraint.**

  If we use the NOT NULL constraint, we must explicitly assign an initial value for that variable.
- **We can initialize a variable while declaration either by using DEFAULT keyword or using an assignment operator.**

Example –

```
counter binary_integer := 0;

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

Variable Scope in PL/SQL

- PL/SQL allows the nesting of blocks, i.e., a program block may contain another inner block.
- 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 OUTERMOST BLOCK AND ACCESSIBLE TO INNER BLOCKS.

Example

```
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;
/
```

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. It requires an initial value and does not allow that value to be changed

Example

```
 DECLARE
        pi constant number := 3.141592654;
     radius number(5,2);
     area number (10, 2);
      BEGIN
        radius := 9.5;
        area := pi * radius * radius;
       dbms_output.put_line('Area: ' || area);
      END;
      /
```

The PL/SQL Literals

- A literal is **an explicit numeric, character, string, or Boolean value not represented by an identifier.**
- PL/SQL, literals are *case-sensitive*.
- *To embed single quotes within a string literal, place two single quotes next to each other.*

Example: message varchar2 (30):= 'That''s LIFE!';

S.No.	Literal Type & Example
1	Numeric Literals 0, 5078 , -14.0 , +32767 , 6E5 , 1.0E-8 , 3.14159e0 , -1E38 , -9.5e-3
2	Character Literals 'A' '%' '9' ' ' 'z' '('
3	String Literals 'Hello, world!' , '19-NOV-12'
4	BOOLEAN Literals TRUE, FALSE, and NULL.
5	Date and Time Literals DATE '1978-12-25' , TIMESTAMP '2012-10-29 12:01:01';

PL/SQL – OPERATORS

ARITHMETIC OPERATORS

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation, raises one operand to the power of other

RELATIONAL OPERATORS

Operator	Description
=	Checks for equality
!= , <> , ~=	Checks for Inequality
>	Greater than
<	Smaller than
>=	Greater than or equal to
<=	Less than or equal to

LOGICAL OPERATORS

Operator	Description
AND	Called the logical AND operator. If both the operands are true then condition becomes true.
OR	Called the logical OR Operator. If any of the two operands is true then condition becomes 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.

COMPARISON OPERATORS

Operator	Description
LIKE	The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern else FALSE.
BETWEEN	The BETWEEN operator tests whether a value lies in a specified range. Bounds inclusive.
IN	The IN operator tests **set membership**.
IS NULL	The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL else FALSE

PL/SQL OPERATOR PRECEDENCE

- The operators with the highest precedence appear at the top of the following 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.

Operator	Operation
**	exponentiation
+, -	identity, negation
*, /	multiplication, division
+, -, ||	addition, subtraction, concatenation
comparison	
NOT	logical negation
AND	conjunction
OR	inclusion

ASSIGNING SQL QUERY RESULTS TO PL/SQL VARIABLES

- We can use the SELECT INTO statement of SQL to assign values to PL/SQL variables.
- For each item in the SELECT list, there must be a corresponding, type-compatible variable in the INTO list.

Example :

```
DECLARE
  c_id  customers.id%type := 1;
  c_name  customers.name%type;
  c_addr customers.address%type;
  c_discount  customers.discount%type;
BEGIN
  SELECT name, address, discount INTO c_name, c_addr, c_discount FROM customers  WHERE id = c_id;
  dbms_output.put_line    ('Customer ' ||c_name || ' from ' || c_addr || ' earns ' || c_discount);
END;
 /
```

PL/SQL - CONTROL STATEMENTS (BRANCHING STATEMENTS)

PL/SQL - IF-THEN Statement

- **If the condition is TRUE, the statements inside the if statement get executed, and if the condition is FALSE or NULL, then the IF statement does nothing and the first set of code after the end of the if statement (after the closing end if) will be executed**

Syntax:

```
IF condition THEN
   S;
END IF;
```

- Where condition is a Boolean or relational condition and S is a simple or compound statement.

Example

Consider we have a table and few records in the table as we had created in PL/SQL Variable Types

```
DECLARE
   c_id customers.id%type := 1;
   c_discount  customers.discount%type;
BEGIN
   SELECT discount  INTO  c_discount  FROM customers  WHERE id = c_id;
   IF (c_discount <= 2000) THEN
      UPDATE customers SET discount =  discount + 100  WHERE id = c_id;
      dbms_output.put_line ('Discount updated');
   END IF;
END;
/
```

PL/SQL - IF-THEN-ELSE Statement

- A sequence of IF-THEN statements can be followed by an optional sequence of ELSE statements, which execute when the condition is FALSE.
- If the Boolean expression condition evaluates to true, then the if-then block of code will be executed otherwise the else block of code will be executed.

Syntax:

```
 IF condition THEN
        S1;
     ELSE
        S2;
     END IF;
```

Where, S1 and S2 are different sequence of statements. In the IF-THEN-ELSE statements, when the test condition is TRUE, the statement S1 is executed and S2 is skipped; when the test condition is FALSE, then S1 is bypassed and statement S2 is executed.

Example −

```
DECLARE
   a number(3) := 100;
BEGIN
    IF( a < 20 ) THEN
           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;
/
```

PL/SQL - IF-THEN-ELSIF Statement

- The IF-THEN-ELSIF statement allows you to choose between several alternatives.
- An IF-THEN statement can be followed by an optional ELSIF...ELSE statement.
- The ELSIF clause lets you add additional conditions.
- **It's ELSIF, not ELSEIF.**
- An IF-THEN statement can have zero to many ELSIF's and they must come before the ELSE.
- Once an ELSIF succeeds, none of the remaining ELSIF's or ELSE's will be tested.

PL/SQL - IF-THEN-ELSIF Statement

Syntax

```
IF(boolean_expression 1)THEN
   S1;       -- Executes when the boolean
                  expression 1 is true
ELSIF( boolean_expression 2) THEN
   S2;      -- Executes when the boolean
                  expression 2 is true
ELSE
   S4;    --   Executes when the none of
                the above condition is true
END IF;
```

Example

```
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 matching');
   END IF;
   dbms_output.put_line('Exact value of a is: '|| a );
END;
/
```

PL/SQL - CASE Statement

- The CASE statement selects one sequence of statements to execute.
- To select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions.
- A selector is an expression, the value of which is used to select one of several alternatives.

Syntax

```
CASE selector
   WHEN 'value1' THEN S1;
   WHEN 'value2' THEN S2;
   WHEN 'value3' THEN S3;
   ...
   ELSE Sn;  -- default case
END CASE;
```

Example

```
DECLARE
   grade char(1) := 'A';
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;
/
```

PL/SQL - Searched CASE Statement

- **The searched CASE statement has no selector and the WHEN clauses of the statement contain search conditions that give Boolean values.**

Syntax	Example

```
CASE
   WHEN selector = 'value1' THEN S1;
   WHEN selector = 'value2' THEN S2;
   WHEN selector = 'value3' THEN S3;
   ...
   ELSE    Sn;              -- default case
END CASE;
```
	```
	DECLARE
	   grade char(1) := 'B';
	BEGIN
	   case
	      when grade = 'A' then dbms_output.put_line('Excellent');
	      when grade = 'B' then dbms_output.put_line('Very good');
	      when grade = 'C' then dbms_output.put_line('Well done');
	      else dbms_output.put_line('No such grade');
	   end case;
	END;
	/
	```

PL/SQL - Nested IF-THEN-ELSE Statements

- **It is legal in PL/SQL programming to nest the IF-ELSE statements.**
- **The inner IF-ELSE statement is called the Nested IF statement. It must begin and end either in the if part (true part) or in the else part (false part) of the outer IF .. ELSE statement.**
- **The End If will be associated with the nearest IF statement.**

Syntax

```
IF  ( boolean_expression 1) THEN
          IF (boolean_expression 2)
THEN
                    sequence-of-
statements;
      END IF;
ELSE
    else-statements;
END IF;
```

Example

```
DECLARE
  a number(3) := 100;
  b number(3) := 200;
BEGIN
     IF( a = 100 ) THEN
                IF( b = 200 ) THEN
                     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;
 /
```

Comparison between IF.. ELSIF statement and CASE statement

- Even though both the constructs are used when there are multiple options, IF.. ELSIF statement is considered to be more efficient because :

Any boolean expression or logical expression can be used as condition in, if.. Elsif statement but in the case statement, the left hand side operand which is compared is fixed and only equality of it with the option values can be checked.

PL/SQL – LOOPING STATEMENTS

- A loop statement allows us to execute a statement or group of statements multiple times.
- The loop body or the loop header should have a provision to update the counter variable and make the Boolean test condition evaluate to false. Otherwise, the loop will become an infinite loop.

Labelling a PL/SQL Loop

- PL/SQL loops can be labelled.
- The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement.
- The label name can also appear at the end of the LOOP statement.
- We may use the label in the EXIT statement to exit from the loop.

```
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('j is: ' | | j);
      END loop inner_loop;
   END loop outer_loop;
END;
/
```

PL/SQL - Basic Loop Statement

- **Basic loop structure encloses sequence of statements in between the LOOP and END LOOP statements.**
- **With each iteration, the sequence of statements is executed and then control resumes at the top of the loop.**
- **An EXIT statement or an EXIT WHEN statement is required to break the loop when a condition is satisfied.**

Syntax

```
 LOOP
   Sequence of statements;
END LOOP;
```

- ***Here, the sequence of statement(s) may be a single statement or a block of statements.***

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

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

PL/SQL - WHILE LOOP Statement

A WHILE LOOP statement in PL/SQL programming language repeatedly executes a target statement as long as a given condition is true.

```
DECLARE
   a number(2) := 10;
BEGIN
   WHILE a < 20 LOOP
      dbms_output.put_line('value of a: ' || a);
      a := a + 1;
   END LOOP;
END;
/
```

PL/SQL - FOR LOOP Statement

- A FOR LOOP is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

```
DECLARE
  a number(2);
BEGIN
  FOR a in 10 .. 20 LOOP
    dbms_output.put_line('value of a: ' || a);
  END LOOP;
END;
/
```

Flow of control in a For Loop –

1. The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
2. Next, the condition, i.e., initial_value .. final_value is evaluated. If it is TRUE, the body of the loop is executed. If it is FALSE, the body of the loop does not execute and the flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the value of the counter variable is increased or decreased.
4. The condition is now evaluated again. If it is TRUE, the loop executes and the process repeats itself (BODY OF LOOP, THEN INCREMENT STEP, AND THEN AGAIN CONDITION). After the condition becomes FALSE, the FOR-LOOP terminates.

Special characteristics of PL/SQL **for loop**

- **The initial_value and final_value of the loop variable or counter can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.**
- **The initial_value need not be 1; however, *the loop counter increment (or decrement) must be 1.***
- **PL/SQL allows the determination of the loop range dynamically at run time.**

Reverse FOR LOOP Statement

- By default, iteration proceeds from the initial value to the final value, generally upward from the lower bound to the higher bound.

- We can reverse this order by using the REVERSE keyword.
- After each iteration, the loop counter is decremented.

- HOWEVER, WE MUST WRITE THE RANGE BOUNDS IN ASCENDING (NOT DESCENDING) ORDER.

```
DECLARE
   a number(2) ;
BEGIN
   FOR a IN REVERSE 10 .. 20 LOOP
     dbms_output.put_line(‘a: ' || a);
   END LOOP;
END;
/
```

PL/SQL - Nested Loops

- For each iteration of the outer loop, inner loop executes completely (i.e., executes all its iteration.)

Example - TO FIND THE PRIME NUMBERS FROM 2 TO 100 –

```
DECLARE
   i number(3);
   j number(3);
BEGIN
   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_line(i || ' is prime');
         END IF;
       i := i + 1;
      exit WHEN i = 50;
   END LOOP;
END;
/
```

THE LOOP CONTROL STATEMENTS

- Loop control statements **change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed**.
- **Labeling loops also help in taking the control outside a loop.**

PL/SQL - EXIT Statement

- **When the EXIT statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.**
- **If we are using nested loops (i.e., one loop inside another loop), the EXIT statement will stop the execution of the innermost loop and start executing the next line of code after the block.**

```
EXIT;
```

```
WHILE a < 20 LOOP
    dbms_output.put_line ('a: ' || a);
    a := a + 1;
    IF a > 15 THEN
       EXIT;
     END IF;
  END LOOP;
```

THE LOOP CONTROL STATEMENTS - PL/SQL - EXIT WHEN Statement

- **The EXIT-WHEN statement allows the condition in the WHEN clause to be evaluated. If the condition is true, the loop completes and control passes to the statement immediately after the END LOOP.**

Following are the two important aspects for the EXIT WHEN statement –

- **Until the condition is true, the EXIT-WHEN statement acts like a NULL statement, except for evaluating the condition, and does not terminate the loop.**
- **A statement inside the loop must change the value of the condition.**

```
EXIT WHEN condition;
```

- **The EXIT WHEN statement replaces a conditional statement like if-then used with the EXIT statement.**

PL/SQL - EXIT WHEN Statement

```
DECLARE
   a number(2) := 10;
BEGIN
   -- while loop execution
   WHILE a < 20 LOOP
      dbms_output.put_line ('value of a: ' || a);
      a := a + 1;
             -- terminate the loop using the exit when statement
       EXIT WHEN a > 15;
   END LOOP;
END;
```

PL/SQL - GOTO Statement

- A GOTO statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labelled statement in the same subprogram.

RESTRICTIONS WITH GOTO STATEMENT

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
- A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.
- A GOTO statement cannot branch from an outer block into a sub-block (i.e., an inner BEGIN-END block).
- A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.

- A GOTO STATEMENT CANNOT BRANCH FROM AN EXCEPTION HANDLER BACK INTO THE CURRENT BEGIN-END BLOCK. HOWEVER, A GOTO STATEMENT CAN BRANCH FROM AN EXCEPTION HANDLER INTO AN ENCLOSING BLOCK.

PL/SQL - GOTO Statement

```
DECLARE
 a number(2) := 10;
BEGIN
 <<loopstart>>
   WHILE a < 20 LOOP
     dbms_output.put_line ('value of a: ' || a);
    a := a + 1;
    IF a = 15 THEN
      a := a + 1;
      GOTO loopstart;
   END IF;
 END LOOP;
END;
```

PL/SQL - SUBPROGRAM

- A subprogram is a program unit/module that performs a particular task.
- A subprogram can be invoked by another subprogram or program which is called the Calling Program.
- A subprogram can be created –
  1. At the schema level
  2. Inside a package
  3. Inside a PL/SQL block
- At the schema level, subprogram is a Standalone Subprogram.
  1. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database .
  2. It is deleted with the DROP PROCEDURE or DROP FUNCTION statement.

PL/SQL - SUBPROGRAM

- A subprogram created inside a package is a Packaged Subprogram.
  - It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement.
- PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters.
- *PL/SQL provides two kinds of subprograms* –
  1. Functions – These subprograms return a single value; mainly used to compute and return a value.
  2. Procedures – These subprograms do not return a value directly; mainly used to perform an action.

Parts of a PL/SQL Subprogram

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

S.No	Parts & Description
1	Declarative Part ❖ It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. ❖ It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. ❖ These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	Executable Part ❖ This is a mandatory part and contains statements that perform the designated action.
3	Exception-handling ❖ This is an optional part. It contains the code that handles run-time errors.

CREATING A PROCEDURE

Syntax:

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

Where,

- *[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
  - OUT represents the parameter that will be used to return a value outside of the procedure.
- 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 Standalone Procedure

A standalone procedure can be called in two ways

1. Using the EXECUTE keyword
2. Calling the name of the procedure from a PL/SQL block

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

```
EXECUTE greetings;
```

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

```
BEGIN
   greetings;
END;
/
```

DELETING A STANDALONE PROCEDURE

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

```
DROP PROCEDURE procedure-name;
```

Example:

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

```
DROP PROCEDURE greetings;
```

METHODS FOR PASSING PARAMETERS

POSITIONAL NOTATION

- **THE FIRST ACTUAL PARAMETER IS SUBSTITUTED FOR THE FIRST FORMAL PARAMETER; THE SECOND ACTUAL PARAMETER IS SUBSTITUTED FOR THE SECOND FORMAL PARAMETER, AND SO ON.**

In positional notation, we can call the procedure as −

```
findMin(a, b, c);
```

Here , a is substituted for x, b is substituted for y and c is substituted for z .

NAMED NOTATION

- **THE ACTUAL PARAMETER IS ASSOCIATED WITH THE FORMAL PARAMETER USING THE ARROW SYMBOL (=>).**

In named notation, we can call the procedure as −

```
findMin(x => a, y => b, z => c, m => d);
```

MIXED NOTATION

- **WE CAN MIX BOTH NOTATIONS IN PROCEDURE CALL; HOWEVER, THE POSITIONAL NOTATION SHOULD PRECEDE THE NAMED NOTATION.**

In mixed notation, we can call the procedure as −

```
findMin(a, b, c, m => d);
```

PARAMETER MODES IN PL/SQL SUBPROGRAMS

	Parameter Mode & Description
1	<u>IN</u> ❖ An IN parameter lets you pass a value to the subprogram. ❖ It is a read-only parameter. ❖ Inside the subprogram, an IN parameter acts like a constant. ❖ It cannot be assigned a value. ❖ We can pass a constant, literal, initialized variable, or expression as an IN parameter. ❖ We can also initialize it to a default value; however, in that case, it is omitted from the subprogram call. ❖ It is the default mode of parameter passing.
2	<u>OUT</u> ❖ An OUT parameter returns a value to the calling program. ❖ Inside the subprogram, an OUT parameter acts like a variable. We can change its value and reference the value after assigning it.
3	<u>IN OUT</u> ❖ An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. ❖ The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression.

IN & OUT Mode Example 1

This program finds the minimum of two values.

Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
CREATE  PROCEDURE findMin (x IN number,
        y IN  number,     z OUT number) IS
BEGIN
      IF x < y THEN
        z:= x;
      ELSE
        z:= y;
      END IF;
END;
```

```
DECLARE
   a number;
   b number;
   c number;

BEGIN
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

result –

```
Minimum of (23, 45) : 23
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value.

Here, the procedure uses the same parameter to accept a value and then return result.

```
CREATE PROCEDURE squareNum
(x IN OUT number)
IS
BEGIN
    x := x * x;
END;
```

```
DECLARE
   a number;
BEGIN
   a:= 23;
    squareNum(a);
   dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

result –

```
            Square of (23): 529
```

PL/SQL - FUNCTIONS

A standalone function is same as a procedure except that it returns a value.

```
CREATE [OR REPLACE] FUNCTION function_name

[(parameter_name [IN | OUT | IN OUT] type [, ...])]

RETURN return_datatype

{IS | AS}

BEGIN

  < function_body >

END [function_name];
```

Where,

- *function-name specifies the name of the function.*
- *[OR REPLACE] option allows the modification of an existing function.*
- *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.*
- *THE FUNCTION MUST CONTAIN A RETURN STATEMENT.*
  - *The RETURN clause specifies the data type you are going to return from the function.*
- *function-body contains the executable part.*
- *The AS keyword is used instead of the IS keyword for creating a Standalone Function.*

Example

This function returns the total number of CUSTOMERS in the customers table.

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
  total number(2) := 0;
BEGIN
          SELECT count(*) into total     FROM
customers;
           RETURN total;
END;
/
```

result –

Function created.

CALLING A FUNCTION

- We will have to call that function to perform the defined task.
- A called function performs the defined task and when its return statement is executed or when the last end statement is reached, it returns the program control back to the main program.
- To call a function, you need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value..

```
DECLARE
  c number(2);
BEGIN
  c := totalCustomers();
  dbms_output.put_line('Total Customers: ' || c);
END;
/
```

Example

Function that computes and returns the maximum of two values.

```
CREATE FUNCTION findMax(x IN number, y IN
number)
RETURN number    IS       z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;
   RETURN z;
END;
```

To call the function from another block

```
 DECLARE
   a number;
   b number;
   c number;

BEGIN
   a:= 23;
   b:= 45;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum ' || c);
END;
/
```

result –

```
Maximum of (23,45): 45
```

PL/SQL RECURSIVE FUNCTIONS

- When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

```
CREATE FUNCTION fact(x number)
RETURN number
IS    f number;

BEGIN
  IF x=0 THEN
     f := 1;
  ELSE
      f := x * fact(x-1);
  END IF;
RETURN f;
END;
/
```

To call the function

```
 DECLARE
   num number;
   factorial number;


BEGIN
   num:= 6;
    factorial := fact(num);
   dbms_output.put_line(' Factorial  is ' || factorial);
END;
/
```

PL/SQL - EXCEPTIONS

- **An exception is an error condition during a program execution.**
- **PL/SQL helps 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 –***
  1. **System-defined exceptions**

     PL/SQL provides many pre-defined exceptions, which are executed when any database rule is violated by a program.

     For example, the predefined exception NO_DATA_FOUND is raised when a SELECT INTO statement returns no rows.
  2. **User-defined exceptions**

- **The default exception will be handled using *WHEN others THEN***

PRE-DEFINED EXCEPTIONS

Exception	Description
INVALID_CURSOR	It is raised when attempts are made to make a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	It is raised when the conversion of a character string into a number fails because the string does not represent a valid number.
LOGIN_DENIED	It is raised when a program attempts to log on to the database with an invalid username or password.
NO_DATA_FOUND	It is raised when a SELECT INTO statement returns no rows.
NOT_LOGGED_ON	It is raised when a database call is issued without being connected to the database.
PROGRAM_ERROR	It is raised when PL/SQL has an internal problem.
ROWTYPE_MISMATCH	It is raised when a cursor fetches value in a variable having incompatible data type.
STORAGE_ERROR	It is raised when PL/SQL ran out of memory or memory was corrupted.
TOO_MANY_ROWS	It is raised when a SELECT INTO statement returns more than one row.
VALUE_ERROR	It is raised when an arithmetic, conversion, truncation, or sizeconstraint error occurs.
ZERO_DIVIDE	It is raised when an attempt is made to divide a number by zero.

SYNTAX

```
DECLARE
   <declarations section>

BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling goes here >
   WHEN exception1 THEN
      exception1-handling-statements
   WHEN exception2  THEN
      exception2-handling-statements
   ........
   WHEN others THEN
      exception3-handling-statements
END;
```

EXAMPLE

```
DECLARE
   c_id customers.id%type := 8;
   c_name  customers.name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
                              FROM customers  WHERE id = c_id;
   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

RAISING EXCEPTIONS

- Exceptions are raised by the database server automatically whenever there is any internal database error.
- Exceptions can be raised explicitly by the programmer by using the command RAISE.
- *We can use the following syntax to raise an Oracle standard exception or any user-defined exception.*

```
DECLARE
   exception_name EXCEPTION;
BEGIN
   IF condition THEN
      RAISE exception_name;
   END IF;
EXCEPTION
   WHEN exception_name THEN
   statement;
END;
```

USER-DEFINED EXCEPTIONS

- PL/SQL allows us to define our own exceptions according to the need of our program.

- A user-defined exception must be declared and then raised explicitly, using either
  1. a RAISE statement or
  2. the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

- The syntax for declaring an exception is –

```
DECLARE
   my-exception EXCEPTION;
```

Example : The Program below asks for a customer ID, when the user enters an invalid ID, the exception invalid_id is raised.

```
DECLARE
   c_id customers.id%type := &cc_id;
   c_name  customers.name%type;
   c_addr customers.address%type;

   ex_invalid_id  EXCEPTION;

BEGIN
   IF c_id <= 0 THEN
      RAISE ex_invalid_id;
   ELSE
      SELECT  name, address INTO  c_name, c_addr
                FROM customers  WHERE id = c_id;
      DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
      DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
   END IF;
EXCEPTION
   WHEN ex_invalid_id THEN
      dbms_output.put_line(‘ ID must be greater than 0');
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

result –

```
Enter value for cc_id: -6 (let's enter a value -6)
old  2: c_id customers.id%type := &cc_id;
new  2: c_id customers.id%type := -6;
ID must be greater than zero!
```

PL/SQL - CURSORS

- A CURSOR is a pointer to the memory area known as CONTEXT AREA created by Oracle for processing an SQL statement, which contains all the information needed for processing the statement.
- PL/SQL controls the context area through a cursor.
- A cursor holds the rows (one or more) returned by a SQL statement.
- The set of rows the cursor holds is referred to as the ACTIVE SET.
- We can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time.

- *There are two types of cursors* –
  1. IMPLICIT CURSORS
  2. EXPLICIT CURSORS

IMPLICIT CURSORS

- **Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement.**
- **Programmers cannot control the *Implicit Cursors* and the information in it.**
- **Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement.**
    - **For INSERT operations,** **the cursor holds the data that needs to be inserted.**
    - **For UPDATE and DELETE operations**, **the cursor identifies the rows that would be affected.**
- **IN PL/SQL, WE CAN REFER TO THE MOST RECENT IMPLICIT CURSOR AS THE SQL CURSOR, WHICH ALWAYS HAS ATTRIBUTES SUCH AS %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT.**
- **ANY SQL CURSOR ATTRIBUTE WILL BE ACCESSED AS SQL%ATTRIBUTE_NAME**

DESCRIPTION OF THE MOST USED ATTRIBUTES

%FOUND **Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.**
%NOTFOUND **The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.**
%ISOPEN **Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.**
%ROWCOUNT **Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.**

PL/SQL program to update the customer table and increase the discount of each customer by 500

customers (ID, NAME, AGE, ADDRESS, DISCOUNT)

```
DECLARE
  total_rows number(2);
BEGIN
  UPDATE customers    SET discount = discount + 500;
  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;
/
```

EXPLICIT CURSORS

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

CURSOR cursor_name IS select_statement;

Working with an explicit cursor includes the following steps –

1. DECLARING THE CURSOR FOR INITIALIZING THE MEMORY
2. OPENING THE CURSOR FOR ALLOCATING THE MEMORY
3. FETCHING THE CURSOR FOR RETRIEVING THE DATA
4. CLOSING THE CURSOR FOR RELEASING THE ALLOCATED MEMORY

Working with an explicit cursor

DECLARING THE CURSOR

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

Example –

```
CURSOR c_customers IS   SELECT id, name, address FROM customers;
```

OPENING THE CURSOR

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it.

Example –

```
OPEN c_customers;
```

Working with an explicit cursor

FETCHING THE CURSOR

Fetching the cursor involves accessing one row at a time. We will fetch rows from the already opened cursor.

Example –

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

CLOSING THE CURSOR

Closing the already opened cursor means releasing the allocated memory.

Example –

```
CLOSE c_customers;
```

A complete example to illustrate the concepts of explicit cursors

```
DECLARE
   c_id  customers.id%type;
   c_name customers.name%type;
   c_addr customers.address%type;

                      CURSOR c_customers IS  SELECT id, name, address FROM customers;
BEGIN
 OPEN c_customers;
   LOOP
   FETCH c_customers into c_id, c_name, c_addr;
   EXIT WHEN c_customers%notfound;
   dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
            CLOSE c_customers;
END;
/
```

PL/SQL - TABLES AND RECORDS MANIPULATIONS

PL/SQL - RECORDS

1. A record is a data structure that can hold data items of different kinds.
2. Record consists of different fields, similar to a row of a database table.
3. The %ROWTYPE attribute enables a programmer to create Table-Based and Cursor-Based records.
4. A RECORD CONTAINS A FIELD FOR EACH OF THE ATTRIBUTE AND ALLOWS TREATING AN ENTITY AS A LOGICAL UNIT AND ALLOWS US TO ORGANIZE AND REPRESENT ITS INFORMATION IN A BETTER WAY.

PL/SQL - TYPES OF RECORDS

PL/SQL can handle the following types of records −

1. TABLE-BASED RECORDS
2. CURSOR-BASED RECORDS
3. USER-DEFINED RECORDS

Table-Based Records

- Records defined based on an already defined Table are called Table-Based records

```
DECLARE
   customer_rec customers%rowtype;
BEGIN
   SELECT * into customer_rec    FROM customers    WHERE id = 5;
   dbms_output.put_line('Customer ID: ' || customer_rec.id);
   dbms_output.put_line('Customer Name: ' || customer_rec.name);
   dbms_output.put_line('Customer Address: ' || customer_rec.address);
   dbms_output.put_line('Customer Discount: ' || customer_rec.discount);
END;
/
```

result –

```
Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Discount: 9000
```

Cursor-Based Records

- Records defined based on an already defined cursor are called Cursor-Based records

```
DECLARE
  CURSOR customer_cur IS  SELECT id, name, address  FROM customers;

  customer_rec  customer_cur%rowtype;

BEGIN
  OPEN customer_cur;
  LOOP
    FETCH customer_cur into customer_rec;
    EXIT WHEN customer_cur%notfound;
    DBMS_OUTPUT.put_line(customer_rec.id || ' ' || customer_rec.name);
  END LOOP;
END;
/
```
	result –
	1 Ramesh
	2 Khilan
	3 kaushik
	4 Chaitali
	5 Hardik
	6 Komal

User-Defined Records

- PL/SQL provides a user-defined record type that allows us to define the different record structures. These records consist of different fields.

Example : *To keep track of books in a library, we might want to track the following attributes about each book – Title, Author, Subject, Book ID*

DEFINING A RECORD

- The record type is defined as −

```
TYPE  type_name IS RECORD
  ( field_name1  datatype1  [NOT NULL]  [:= DEFAULT EXPRESSION],
   field_name2   datatype2   [NOT NULL]   [:= DEFAULT EXPRESSION],
   ...
   field_nameN  datatypeN  [NOT NULL]  [:= DEFAULT EXPRESSION);

      record-name  type_name;
```
	The Book record is declared in the following way −
	```
	DECLARE
	TYPE books IS RECORD
	(title  varchar(50),
	   author  varchar(50),
	   subject varchar(100),
	   book_id   number);

	      book1 books;
	      book2 books;
	```

Accessing Fields

- To access any field of a record, we use the dot (.) operator.
- The member access operator is coded as a period between the record variable name and the field that we wish to access.

```
DECLARE
   type books is record
      (title varchar(50),
      author varchar(50),
      subject varchar(100),
      book_id number);

  book1 books;

BEGIN
   -- Book 1 specification
    book1.title  := 'C Programming';
   book1.author := 'Nuha Ali ';
   book1.subject := 'C Programming guide ';
   book1.book_id := 6495407;
```

```
         -- Print book 1 record
dbms_output.put_line('Book 1 title : '|| book1.title);
dbms_output.put_line('Book 1 author : '|| book1.author);
dbms_output.put_line('Book 1 subject : '|| book1.subject);
dbms_output.put_line('Book 1 book_id : ' || book1.book_id);

END;
/
```

result –

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming guide
Book 1 book_id : 6495407
```

WE CAN PASS A RECORD AS A SUBPROGRAM PARAMETER, AS WE PASS ANY OTHER VARIABLE.

```
DECLARE
   type books is record
      (title  varchar(50),
      author  varchar(50),
      subject varchar(100),
      book_id   number);

   book1 books;

   PROCEDURE printbook (book books) IS

BEGIN


   dbms_output.put_line (' title :  ' || book.title);
   dbms_output.put_line(' author : ' || book.author);
   dbms_output.put_line( 'subject : ' || book.subject);
   dbms_output.put_line( 'book_id : ' || book.book_id);

END;
```

```
BEGIN
      -- Book 1 specification
      book1.title  := 'C Programming';
      book1.author := 'Nuha Ali ';
      book1.subject := 'C Programming Guide’;
      book1.book_id := 6495407;

       -- Use procedure to print book info

      printbook(book1);

   END;
/
```

result –

```
title : C Programming
author : Nuha Ali
subject : C Programming Guide
book_id : 6495407
```

PL/SQL – TRIGGERS

What are Triggers in PL/SQL?

- **A Trigger is a special stored procedure that is run when specific actions occur within a database.**
- **Most triggers are defined to run when changes are made to a table's data.**
- **TRIGGERS CAN BE DEFINED TO RUN AUTOMATICALLY INSTEAD OF OR BEFORE OR AFTER DML COMMANDS SUCH AS INSERT, UPDATE, AND DELETE ISSUED AGAINST THE ASSOCIATED TABLE.**
- **TRIGGERS CAN BE DEFINED TO RUN ON THE OCCURRENCE OF A SPECIFIC EVENT AND AT THE SPECIFIC TIME OF EXECUTION.**
- **A trigger can include SQL and PL/SQL statements to execute as a unit and can invoke stored procedures.**
- **Triggers can be defined ONLY ON TABLES, NOT ON VIEWS.**
  - **However, TRIGGERS ON THE BASE TABLE(S) OF A VIEW ARE FIRED IF AN INSERT, UPDATE, OR DELETE STATEMENT IS ISSUED AGAINST A VIEW.**

BENEFITS OF TRIGGERS

1. Generating derived column values automatically
2. Enforcing referential integrity
3. Event logging and storing information on table access
4. Auditing
5. Synchronous replication of tables
6. Imposing security authorizations
7. Preventing invalid transactions

How Triggers Are Used?

1. Triggers supplement the standard capabilities of DBMS to provide a highly customized DBMS
2. Triggers automatically generate derived column values
3. Triggers prevent invalid transactions
4. TRIGGERS ENFORCE COMPLEX SECURITY AUTHORIZATIONS, REFERENTIAL INTEGRITY ACROSS NODES IN A DISTRIBUTED DATABASE *AND COMPLEX BUSINESS RULES*
5. Triggers provide transparent event logging and sophisticated auditing
6. Triggers maintain synchronous table replicates
7. Triggers gather statistics on table access

CASCADING TRIGGERS

- **WHEN A TRIGGER IS FIRED, A SQL STATEMENT WITHIN ITS TRIGGER ACTION POTENTIALLY CAN FIRE OTHER TRIGGERS.**
- **WHEN A STATEMENT IN A TRIGGER BODY CAUSES ANOTHER TRIGGER TO BE FIRED, THE TRIGGERS ARE SAID TO BE *CASCADING*.**

- **Note** : ***We should use triggers only when necessary. The excessive use of triggers can result in complex interdependences, which may be difficult to maintain in a large application.***

TRIGGERS VS. DECLARATIVE INTEGRITY CONSTRAINTS

- **Triggers and declarative integrity constraints can both be used to constrain data input.**
- ***A declarative integrity constraint is a statement about the database that is never false*** **while the constraint is enabled.**
    - ***A constraint applies to existing data in the table and any statement that manipulates the table.***
- **Triggers constrain what transactions can do.**
    - *A trigger does not apply to data loaded before the definition of the trigger.* Therefore, it does not guarantee all data in a table conforms to its rules.
- ***A trigger enforces transitional constraints;*** **that is,** ***A TRIGGER ONLY ENFORCES A CONSTRAINT AT THE TIME THAT THE DATA CHANGES.***
    - **Therefore, a constraint such as "make sure that the delivery date is at least seven days from today" should be enforced by a trigger, not a declarative integrity constraint.**

PARTS OF A TRIGGER

1. **Triggering Event or Statement**

- **A triggering event or statement is the SQL statement that causes a trigger to be fired.**
- **A triggering event can be an INSERT, UPDATE, or DELETE statement on a table.**
- **When the triggering event is an UPDATE statement, *we can include a column list to identify which columns must be updated to fire the trigger.***
  - ***But, INSERT and DELETE statements affect entire rows of information. So, a column list cannot be specified for these options.***
- ***A SINGLE TRIGGER CAN BE CREATED THAT EXECUTES DIFFERENT CODE BASED ON THE TYPE OF DML STATEMENT THAT FIRED THE TRIGGER.***

i.e., A triggering event can specify multiple DML statements.

  - ***WHEN MULTIPLE TYPES OF DML STATEMENTS CAN FIRE A TRIGGER, CONDITIONAL PREDICATES CAN BE USED TO DETECT THE TYPE OF TRIGGERING STATEMENT.***

PARTS OF A TRIGGER

2. **Trigger Restriction**
   - A trigger restriction specifies a Boolean (logical) expression that must be TRUE for the trigger to fire.
       - THE TRIGGER ACTION IS NOT EXECUTED IF THE TRIGGER RESTRICTION EVALUATES TO FALSE OR UNKNOWN.
   - A trigger restriction is an option available for triggers that are fired for each row.
   - *We specify a trigger restriction using a **WHEN** clause.*

2. **Trigger Action**
   - *A TRIGGER ACTION IS THE PROCEDURE (PL/SQL BLOCK) THAT CONTAINS THE SQL STATEMENTS AND PL/SQL CODE TO BE EXECUTED WHEN A TRIGGERING STATEMENT IS ISSUED AND THE TRIGGER RESTRICTION EVALUATES TO TRUE.*
   - *FOR ROW TRIGGER, THE STATEMENTS IN A TRIGGER ACTION HAVE ACCESS TO COLUMN VALUES (NEW AND OLD) OF THE CURRENT ROW BEING PROCESSED BY THE TRIGGER.*
       - *Two correlation names provide access to the old and new values for each column.*

Types of Triggers

Triggers can be classified based on the following parameters.

1. Classification based on the TIMING
   a) **BEFORE Trigger:** It fires before the specified event has occurred.
   b) **AFTER Trigger:** It fires after the specified event has occurred.
   c) **INSTEAD OF Trigger:** A special type.
2. Classification based on the LEVEL
   a) **STATEMENT level Trigger:** It fires one time for the specified event statement.
   b) **ROW level Trigger:** It fires for each record that got affected in the specified event. (only for DML)
3. Classification based on the EVENT
   a) **DML Trigger:** It fires when the DML event is specified (INSERT/UPDATE/DELETE)
   b) **DDL Trigger:** It fires when the DDL event is specified (CREATE/ALTER)
   c) **DATABASE Trigger:** It fires when the database event is specified (LOGON/LOGOFF/STARTUP/SHUTDOWN)

So each trigger is the combination of above parameters.

BEFORE Triggers

- BEFORE triggers execute the trigger action before the triggering statement executes.
- *BEFORE triggers are used in the following situations:*
  - When the trigger action should determine whether the triggering statement should be allowed to complete or not.
    - By using a BEFORE trigger for this purpose, we can eliminate unnecessary processing of the triggering statement and its eventual rollback in cases where an exception is raised in the trigger action.
  - BEFORE triggers are used to derive specific column values before completing a triggering INSERT or UPDATE statement.

AFTER Triggers

- AFTER triggers execute the trigger action after the triggering statement is executed.
- *AFTER triggers are used in the following situations:*
  - When we want the triggering statement to complete afterexecuting the trigger action.
  - If a BEFORE trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

INSTEAD OF Triggers

- *"INSTEAD OF trigger" is the special type of trigger. It is used only in DML triggers. It is used when any DML event is going to occur on the complex view.*
- **The INSTEAD OF trigger is used to modify the base tables directly instead of modifying the view.**
- These triggers are CALLED INSTEAD OF TRIGGERS BECAUSE, UNLIKE OTHER TYPES OF TRIGGERS, ORACLE FIRES THE TRIGGER INSTEAD OF EXECUTING THE TRIGGERING STATEMENT.
- WE CAN WRITE NORMAL INSERT, UPDATE, AND DELETE STATEMENTS AGAINST THE VIEW AND THE INSTEAD OF TRIGGER IS FIRED TO UPDATE THE UNDERLYING TABLES APPROPRIATELY.
- INSTEAD OF triggers are activated for each row of the view that gets modified.
- Even if the view is inherently modifiable, we might want to perform validations on the values being inserted, updated or deleted. INSTEAD OF triggers can also be used in this case. Here the trigger code performs the validation on the rows being modified and if valid, propagate the changes to the underlying tables.

INSTEAD OF Triggers on Nested Tables

- We The cannot modify the elements of a nested table column in a view directly with the TABLE clause. However, we can do so by defining an INSTEAD OF trigger on the nested table column of the view. triggers on the nested tables fire if a nested table element is updated, inserted, or deleted and handle the actual modifications to the underlying tables.

ROW TRIGGERS

- A row trigger is FIRED EACH TIME THE TABLE IS AFFECTED BY THE TRIGGERING STATEMENT.
  - For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.
- IF A TRIGGERING STATEMENT AFFECTS NO ROWS, A ROW TRIGGER IS NOT EXECUTED AT ALL.
- Row triggers are useful if the code in the trigger action depends on data provided by the triggering statement or rows that are affected.

STATEMENT TRIGGERS

- *A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects* (even if no rows are affected).
  - For example, if a DELETE statement deletes several rows from a table, a statement-level DELETE trigger is fired only once, regardless of how many rows are deleted from the table.
- *Statement triggers are useful if the code in the trigger action does not depend on the data provided by the triggering statement or the rows affected.*
  - For example, if a trigger makes a complex security check on the current time or user, or if a trigger generates a single audit record based on the type of triggering statement, a statement trigger is used.

Different Combinations of Triggers

1. **BEFORE Statement Trigger** Before executing the triggering statement, the trigger action is executed.
2. **BEFORE Row Trigger** Before modifying each row affected by the triggering statement and before checking appropriate integrity constraints, the trigger action is executed provided that the trigger restriction was not violated.
3. **AFTER Statement Trigger** After executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.
4. **AFTER Row Trigger** After modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is executed for the current row provided the trigger restriction was not violated.
   - UNLIKE BEFORE ROW TRIGGERS, AFTER ROW TRIGGERS LOCK ROWS.

Note:

- *WE CAN HAVE MULTIPLE TRIGGERS OF THE SAME TYPE FOR THE SAME STATEMENT FOR ANY GIVEN TABLE.*
- *IF MORE THAN ONE TRIGGER OF THE SAME TYPE FOR A GIVEN STATEMENT EXISTS, ORACLE FIRES EACH OF THOSE TRIGGERS IN AN UNSPECIFIED ORDER.*

:NEW and :OLD Clause

- In a row level trigger, the trigger fires for each related row. And sometimes it is required to know the value before and after the DML statement.
- We can use :NEW and :OLD clauses to refer to the new and old values inside the trigger body.

  :NEW – It holds a new value for the columns of the base table/view during the trigger execution

  :OLD – It holds old value of the columns of the base table/view during the trigger execution

  This clause should be used based on the DML event.

Below table will specify which clause is valid for which DML statement (INSERT/UPDATE/DELETE)

	INSERT	UPDATE	DELETE
:NEW	VALID	VALID	INVALID. There is no new value in delete case.
:OLD	INVALID. There is no old value in insert case	VALID	VALID

EXAMPLE:

```
CREATE OR REPLACE TRIGGER display_discount_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (:NEW.ID > 0)
DECLARE
  disc_diff number;
BEGIN
  disc_diff := :NEW.discount  - :OLD. discount;
  dbms_output.put_line('Old  discount : ' || :OLD. discount);
  dbms_output.put_line('New  discount : ' || :NEW. discount);
  dbms_output.put_line(‘ Discount  difference: ' || disc_diff);
END;
/
```

CREATING A TRIGGER

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

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

TRIGGER EXAMPLES - BEFORE INSERT Trigger

- A BEFORE INSERT Trigger means that Oracle will fire this trigger before the INSERT operation is executed.
- We can not create a BEFORE trigger on a view.
- We can update the :NEW values. But, we cannot update the :OLD values

Syntax	Example : If you had a table orders (order_id, quantity, cost_per_item, total_cost, create_date, created_by)

```
CREATE [ OR REPLACE ]
TRIGGER trigger_name
BEFORE INSERT
ON table_name
   [ FOR EACH ROW ]

DECLARE
   -- variable declarations
BEGIN
   -- trigger code
EXCEPTION
   WHEN ...
   -- exception handling
END;
```
	```
	CREATE OR REPLACE TRIGGER orders_before_insert
	BEFORE INSERT
	ON orders
	FOR EACH ROW

	DECLARE
	   v_username varchar2(10);

	BEGIN
	                  -- Find username of person performing INSERT into table
	   SELECT user INTO v_username    FROM dual;
	               -- Update created_by field to the username of the person performing the INSERT
	   :new.created_by := v_username;
	               -- Update create_date field to current system date
	   :new.create_date := sysdate;
	END;
	```

TRIGGER EXAMPLES - AFTER INSERT Trigger

- **An AFTER INSERT Trigger means that Oracle will fire this trigger after the INSERT is executed.**
- **We cannot create an AFTER trigger on a view.**
- **We cannot update the :NEW values and cannot update the :OLD values.**

Syntax	**Example :** If you had a table orders (<u>order_id</u>, quantity, cost_per_item, total_cost)

```
CREATE  [  OR  REPLACE  ]
TRIGGER trigger_name
AFTER INSERT
   ON table_name
   [ FOR EACH ROW ]

DECLARE
   -- variable declarations
BEGIN
   -- trigger code
EXCEPTION
   WHEN ...
   -- exception handling
END;
```
	```
	CREATE OR REPLACE TRIGGER orders_after_insert
	AFTER INSERT
	   ON orders
	   FOR EACH ROW

	DECLARE
	   v_username varchar2(10);

	BEGIN
	                   -- Find username of person performing INSERT into table
	   SELECT user INTO v_username    FROM dual;
	                  -- Insert record into audit table
	*INSERT INTO orders_audit ( order_id, quantity, cost_per_item, total_cost, username)    VALUES
	( :new.order_id, :new.quantity, :new.cost_per_item, :new.total_cost, v_username );*

	END;
	```

TRIGGER EXAMPLES - BEFORE UPDATE Trigger

- A BEFORE UPDATE Trigger means that Oracle will fire this trigger before the UPDATE is executed.
- We can not create a BEFORE trigger on a view.
- We can update the :NEW values but cannot update the :OLD values.

Syntax	Example : If you had a table orders (order_id, quantity, cost_per_item, total_cost, updated_date, updated_by);

```
CREATE [ OR REPLACE ]
TRIGGER trigger_name
BEFORE UPDATE
   ON table_name
   [ FOR EACH ROW ]

DECLARE
   -- variable declarations
BEGIN
   -- trigger code
EXCEPTION
   WHEN ...
   -- exception handling
END;
```
	```
	CREATE OR REPLACE TRIGGER orders_before_update
	BEFORE UPDATE
	   ON orders
	   FOR EACH ROW

	DECLARE
	   v_username varchar2(10);

	BEGIN
	                  -- Find username of person performing UPDATE  into table
	   SELECT user INTO v_username    FROM dual;
	                 -- Update updated_by field to the username of the person performing the UPDATE
	   :new.updated_by := v_username;
	                    -- Update updated_date field to current system date
	   :new.updated_date := sysdate;
	END;
	```

TRIGGER EXAMPLES - AFTER UPDATE Trigger

- **An AFTER UPDATE Trigger means that Oracle will fire this trigger after the UPDATE is executed.**
- **We can not create an AFTER trigger on a view.**
- **We can not update the :NEW values and also cannot update the :OLD values.**

Syntax	**Example** : *If you had a table orders (order_id number(5), quantity number(4), cost_per_item number(6,2), total_cost number(8,2));*

```
CREATE [ OR REPLACE ]
TRIGGER trigger_name
AFTER UPDATE
   ON table_name
   [ FOR EACH ROW ]

DECLARE
   -- variable declarations
BEGIN
   -- trigger code
EXCEPTION
   WHEN ...
   -- exception handling
END;
```
	```
	CREATE OR REPLACE TRIGGER orders_after_update
	AFTER UPDATE
	   ON orders
	   FOR EACH ROW

	DECLARE
	   v_username varchar2(10);

	BEGIN
	                   -- Find username of person performing UPDATE  into table
	   SELECT user INTO v_username    FROM dual;
	                  -- Insert record into audit table
	   INSERT INTO orders_audit
	   ( order_id, quantity_before, quantity_after, username )
	   VALUES( :new.order_id, :old.quantity, :new.quantity,   v_username );
	END;
	```

TRIGGER EXAMPLES - BEFORE DELETE Trigger

- A BEFORE DELETE Trigger means that Oracle will fire this trigger before the DELETE is executed.
- We cannot create a BEFORE trigger on a view.
- We can update the :NEW values but cannot update the :OLD values.

Syntax	Example : *If you had a table orders (order_id, quantity, cost_per_item, total_cost)*

`CREATE [ OR REPLACE ]`
`TRIGGER trigger_name`
`BEFORE DELETE`
`ON table_name`
`   [ FOR EACH ROW ]`
 	
`DECLARE`
`   -- variable declarations`
`BEGIN`
`   -- trigger code`
`EXCEPTION`
`   WHEN ...`
`   -- exception handling`
`END;`
	`CREATE OR REPLACE TRIGGER orders_before_delete`
	`BEFORE DELETE`
	`ON orders`
	`   FOR EACH ROW`
	
	`DECLARE`
	`   v_username varchar2(10);`
	
	`BEGIN`
	`               -- Find username of person performing DELETE on the  table`
	`   SELECT user INTO v_username    FROM dual;`
	`              -- Insert record into audit table`
	`   INSERT INTO orders_audit`
	`  ( order_id, quantity, cost_per_item, total_cost, delete_date, deleted_by )    VALUES`
	`  ( :old.order_id, :old.quantity, :old.cost_per_item, :old.total_cost, sysdate, v_username );`
	`END;`

TRIGGER EXAMPLES - AFTER DELETE Trigger

- An AFTER DELETE Trigger means that Oracle will fire this trigger after the DELETE is executed.
- We cannot create an AFTER trigger on a view.
- We cannot update the :NEW values and cannot update the :OLD values.

Syntax

```
CREATE [ OR REPLACE ]
TRIGGER trigger_name
AFTER  DELETE
ON table_name
   [ FOR EACH ROW ]

DECLARE
   -- variable declarations
BEGIN
   -- trigger code
EXCEPTION
   WHEN ...
   -- exception handling
END;
```

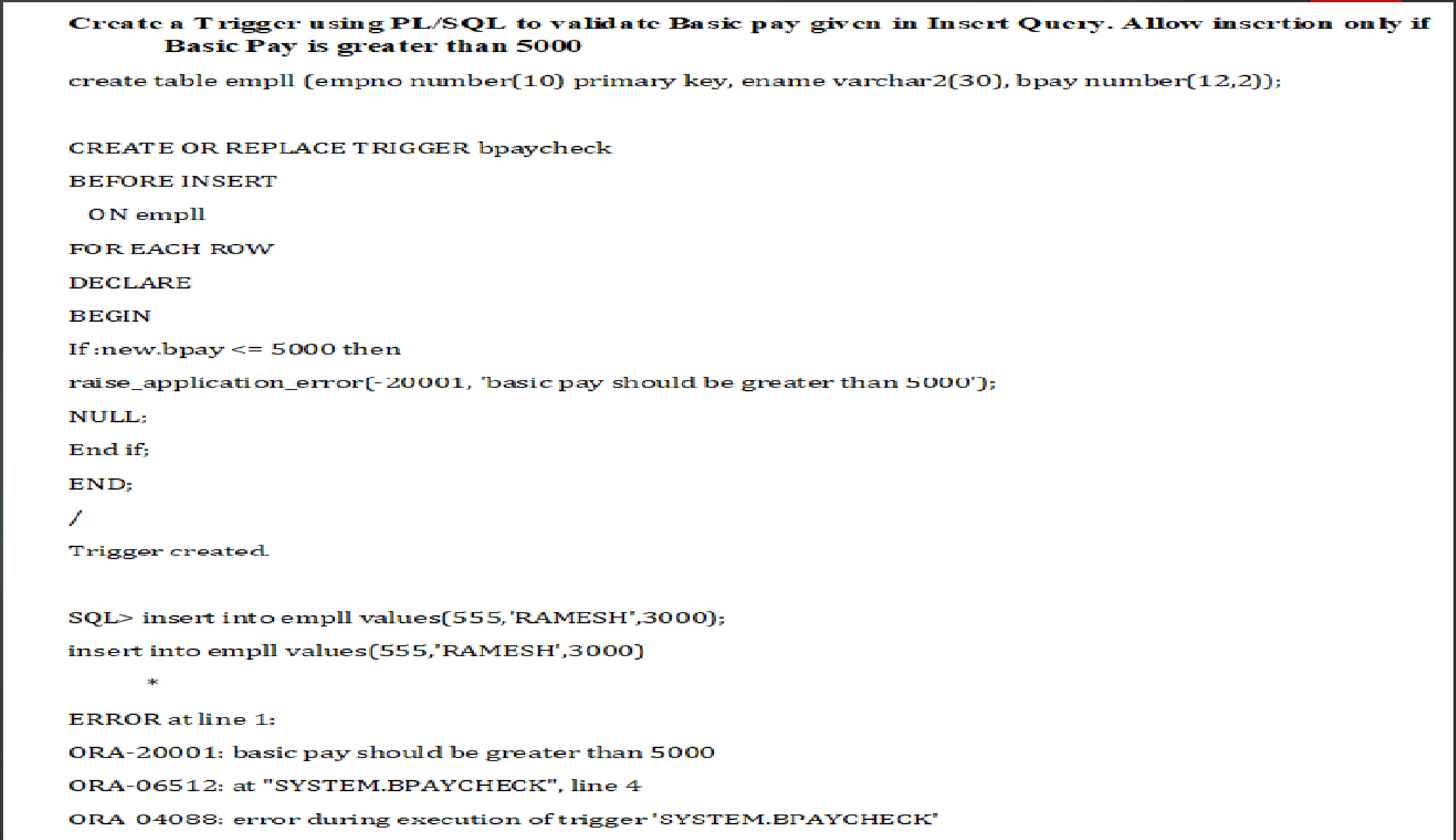
Example : *If you had a table orders (order_id, quantity , cost_per_item, total_cost)*

```
CREATE OR REPLACE TRIGGER orders_after_delete
AFTER DELETE
   ON orders
   FOR EACH ROW

DECLARE
   v_username varchar2(10);

BEGIN
                   -- Find username of person performing DELETE on the  table
   SELECT user INTO v_username    FROM dual;
                    -- Insert record into audit table
   INSERT INTO orders_audit
    ( order_id, quantity, cost_per_item, total_cost, delete_date, deleted_by)  VALUES
     ( :old.order_id, :old.quantity, :old.cost_per_item, :old.total_cost,  sysdate,
        v_username );
END;
```

IMPORTANT EXAMPLE : TRIGGER THAT RESTRICTS DML ACTION



Create a Trigger using PL/SQL to validate Basic pay given in Insert Query. Allow insertion only if Basic Pay is greater than 5000

```
create table empll (empno number(10) primary key, ename varchar2(30), bpay number(12,2));


CREATE OR REPLACE TRIGGER bpaycheck
BEFORE INSERT
  ON empll
FOR EACH ROW
DECLARE
BEGIN
If :new.bpay <= 5000 then
raise_application_error(-20001, 'basic pay should be greater than 5000');
NULL;
End if;
END;
/
Trigger created.


SQL> insert into empll values(555,'RAMESH',3000);
insert into empll values(555,'RAMESH',3000)
            *
ERROR at line 1:
ORA-20001: basic pay should be greater than 5000
ORA-06512: at "SYSTEM.BPAYCHECK", line 4
ORA 04088: error during execution of trigger 'SYSTEM.BPAYCHECK'
```

INSTEAD OF Triggers

- *INSTEAD OF trigger" is the special type of trigger. It is used only in DML triggers. It is used when any DML event is going to occur on the complex view.*
- **The INSTEAD OF trigger is used to modify the base tables directly instead of modifying the view.**

Example : **Consider an example in which a view is made from 2 base tables. When any DML event is issued over this view, that will become invalid because the data is taken from 3 different tables. So in this INSTEAD OF trigger is used.**

```
CREATE VIEW emp_view AS
SELECT emp.emp_name, dept.dept_name, dept.location FROM emp, dept
                                        WHERE emp.dept_no=dept.dept_no;
```

To avoid the error encounter during updating a complex view we create an "instead of trigger."

```
CREATE TRIGGER   view_modify_trg   INSTEAD OF UPDATE ON  emp_view  FOR EACH ROW
BEGIN
UPDATE dept SET location= :new.location WHERE dept_name= :old.dept_name;
END;
```

- **Here Update statement uses ': OLD' and ':NEW' to find the value of columns before and after the update.**

Statement-Level Trigger - Before

The Statement level Before Trigger, irrespective of number of rows affected will execute once.

Example : To display a message prior to insert operation on the emp table.

```
CREATE OR REPLACE TRIGGER emp_alert_trig
BEFORE INSERT ON emp
BEGIN
DBMS_OUTPUT.PUT_LINE('New employees are about to be added');
END;
```

The message, New employees are about to be added, is displayed once by the firing of the trigger irrespective of number of rows inserted into the emp table.

Statement-Level Trigger - After

The Statement level After Trigger, irrespective of number of rows affected will execute once.

Example : To display a message after insert operation on the emp table.

```
CREATE OR REPLACE TRIGGER emp_alert_trig
AFTER INSERT ON emp
BEGIN
DBMS_OUTPUT.PUT_LINE('New employees were added');
END;
```

The message, New employees were added, is displayed once by the firing of the trigger irrespective of number of rows inserted into the emp table.

DROP TRIGGER Statement

- **To remove a trigger from the database use DROP TRIGGER statement.**

Syntax

```
DROP TRIGGER trigger_name;
```

Example

```
DROP TRIGGER orders_before_insert;
```

MODES OF TRIGGER

1. ***ENABLED*** : An *enabled* trigger executes its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to TRUE.

   For enabled triggers, Oracle automatically

   1. executes triggers of each type in a planned firing sequence when more than one trigger is fired by a single SQL statement
   2. performs integrity constraint checking at a set point in time with respect to the different types of triggers and guarantees that triggers cannot compromise integrity constraints
   3. provides read-consistent views for queries and constraints
   4. manages the dependencies among triggers and objects referenced in the code of the trigger action
   5. uses two-phase commit if a trigger updates remote tables in a distributed database
2. **DISABLED** : A *disabled* trigger does not execute its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to TRUE

MODES OF TRIGGER

1. Enable a Trigger

   - If a trigger on a table was disabled and to enable the trigger again

Syntax	Example
ALTER TRIGGER trigger_name ENABLE;	ALTER TRIGGER orders_before_insert ENABLE;

   - If all triggers on a table were disabled and to enable the triggers again

Syntax	Example
ALTER TABLE table_name ENABLE ALL TRIGGERS;	ALTER TABLE orders ENABLE ALL TRIGGERS;

1. Disable a Trigger

   - If a trigger on a table was enabled and to disable the trigger

Syntax	Example
ALTER TRIGGER trigger_name DISABLE;	ALTER TRIGGER orders_before_insert DISABLE;

   - If all triggers on a table were enabled and to disable the triggers again

Syntax	Example
ALTER TABLE table_name DISABLE ALL TRIGGERS;	ALTER TABLE orders DISABLE ALL TRIGGERS;