6. PL/SQL Programming

PL/SQL Introduction

PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements. All the statements of a block are passed to oracle engine all at once which increases processing speed and decreases the traffic.

Advantages of PL/SQL

PL/SQL has the following advantages −

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- 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.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.

Disadvantages of SQL:

- SQL doesn't provide the programmers with a technique of condition checking, looping and branching.
- SQL statements are passed to Oracle engine one at a time which increases traffic and decreases speed.
- SQL has no facility of error checking during manipulation of data.

Features of PL/SQL:

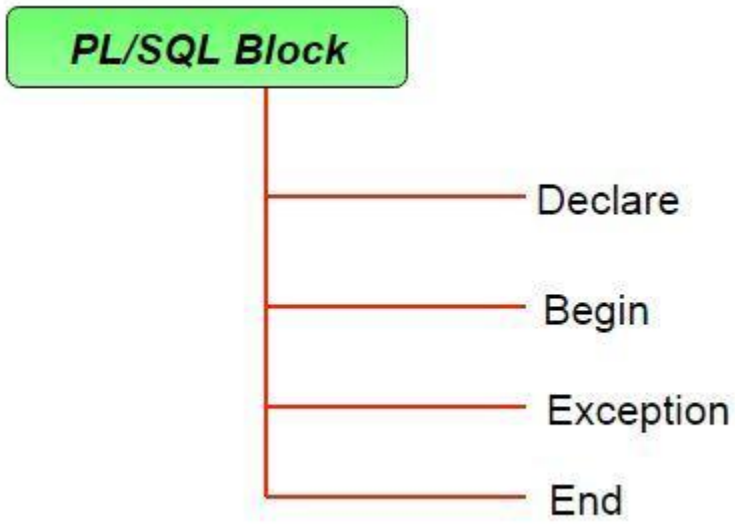
1. PL/SQL is basically a procedural language, which provides the functionality of decision making, iteration and many more features of procedural programming languages.
2. PL/SQL can execute a number of queries in one block using single command.
3. One can create a PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications.
4. PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.
5. Applications written in PL/SQL are portable to computer hardware or operating system where Oracle is operational.
6. PL/SQL Offers extensive error checking.
7. **Differences between SQL and PL/SQL:**

SQL	PL/SQL

SQL is a single query that is used to perform DML and DDL operations.	PL/SQL is a block of codes that used to wri program blocks/ procedure/ function, etc.
It is declarative, that defines what needs to be done, rather than how things need to be done.	PL/SQL is procedural that defines how the to be done.
Execute as a single statement.	Execute as a whole block.
Mainly used to manipulate data.	Mainly used to create an application.
Cannot contain PL/SQL code in it.	It is an extension of SQL, so it can contain it.

Structure of PL/SQL Block:

PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a block. All PL/SQL programs are made up of blocks, which can be nested within each other.



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

```
DECLARE
    declaration statements;
```

```
BEGIN
    executable statements

EXCEPTIONS
    exception handling statements

END;
```

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

- **PL/SQL Execution Environment:**
- The PL/SQL engine resides in the Oracle engine. The Oracle engine can process not only single SQL statement but also block of many statements. The call to Oracle engine needs to be made only once to execute any number of SQL statements if these SQL statements are bundled inside a PL/SQL block.
-

PL/SQL Data types:-

S.No	Category & Description
1	**Scalar** Single values with no internal components, such as a **NUMBER, DATE,** or **BOOLEAN**.
2	**Large Object (LOB)** Pointers to large objects that are stored separately from other data items, such as text, graphic clips, and sound waveforms.
3	**Composite**

	Data items that have internal components that can be accessed individually. For example, c records.
4	**Reference** Pointers to other data items.

The PL/SQL variables, constants and parameters must have a valid data type, which specifies a storage format, constraints, and a valid range of values.

PL/SQL Scalar Data Types and Subtypes

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

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.

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

PL/SQL Numeric Data Types and Subtypes

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

S.No	Data Type & Description
1	**PLS_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
2	**BINARY_INTEGER** Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
3	**BINARY_FLOAT** Single-precision IEEE 754-format floating-point number
4	**BINARY_DOUBLE** Double-precision IEEE 754-format floating-point number
5	**NUMBER(prec, scale)** Fixed-point or floating-point number with absolute value in range 1E-130 to (but not includin NUMBER variable can also represent 0
6	**DEC(prec, scale)** ANSI specific fixed-point type with maximum precision of 38 decimal digits
7	**DECIMAL(prec, scale)** IBM specific fixed-point type with maximum precision of 38 decimal digits
8	**NUMERIC(pre, secale)** Floating type with maximum precision of 38 decimal digits
9	**DOUBLE PRECISION** ANSI specific floating-point type with maximum precision of 126 binary digits (approximate digits)

10	**FLOAT** ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (app decimal digits)
11	**INT** ANSI specific integer type with maximum precision of 38 decimal digits
12	**INTEGER** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
13	**SMALLINT** ANSI and IBM specific integer type with maximum precision of 38 decimal digits
14	**REAL** Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digit

Following is a valid declaration −

```
DECLARE
   num1 INTEGER;
   num2 REAL;
   num3 DOUBLE PRECISION;
BEGIN
   null;
END;
/
```

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

```
PL/SQL procedure successfully completed
```

PL/SQL Character Data Types and Subtypes

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

S.No	Data Type & Description

1	**CHAR** Fixed-length character string with maximum size of 32,767 bytes
2	**VARCHAR2** Variable-length character string with maximum size of 32,767 bytes
3	**RAW** Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
4	**NCHAR** Fixed-length national character string with maximum size of 32,767 bytes
5	**NVARCHAR2** Variable-length national character string with maximum size of 32,767 bytes
6	**LONG** Variable-length character string with maximum size of 32,760 bytes
7	**LONG RAW** Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
8	**ROWID** Physical row identifier, the address of a row in an ordinary table
9	**UROWID** Universal row identifier (physical, logical, or foreign row identifier)

PL/SQL Boolean Data Types

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

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

- SQL statements
- Built-in SQL functions (such as **TO_CHAR**)
- PL/SQL functions invoked from SQL statements

PL/SQL Datetime and Interval Types

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

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

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

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds

TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable

PL/SQL Large Object (LOB) Data Types

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

Data Type	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

PL/SQL User-Defined Subtypes

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

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

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

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

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

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

```
Hello Reader Welcome to the World of PL/SQL

PL/SQL procedure successfully completed.
```

NULLs in PL/SQL

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

PL/SQL Variables

A variable is a reserved memory area for storing the data of a particular datatype. It is an **identifier** which is identifies memory locations where data is stored.

Rules for declaring a Variable in PL/SQL

Following are some important rules to keep in mind while defining and using a variable in PL/SQL:

1. A variable name is user-defined. It should begin with a character and can be followed by maximum of 29 characters.
2. Keywords (i.e, reserved words) of PL/SQL cannot be used as variable name.

3. Multiple variables can be declared in a single line provided they must be separated from each other by at least one space and comma.

   For eg: `a,b,c int;`

4. Variable names containing two words should not contain space between them. It must be covered by underscore instead.

   For eg: `Roll_no`

5. A variable name is case sensitive, which means `a_var` is not same as `A_var`

Variable Declaration in PL/SQL

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

Syntax for declaration of a variable:

```
Variable_name  datatype(size);
```

Let's take a simple example of how we can define a variable in PL/SQL,

```
roll_no NUMBER(2);



a int;
```

And if we want to assign some value to the variable at the time of declaration itself, the syntax would be,

```
Variable_name datatype(size) NOT NULL:=value;
```

Let's take a simple example for this too,

```
eid NUMBER(2) NOT NULL := 5;
```

In the PL/SQL code above, we have defined a variable with name `eid` which is of datatype `NUMBER` and can hold a number of length **2** bytes, which means it can hold a number upto **99**(because 100 onwards we have 3 digits) and the default value for this variable is **5**.

The keyword **NOT NULL** indicates that `eid` cannot be a blank field.

Here, `:=` is an **assignment operator** used to assign a value to a variable.

- Some valid variable declarations along with their definition are shown below −

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

- When you 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. For example −

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

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

For example −

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results. Try the following example which makes use of various types of variables −

```
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;
```

```
/
```

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

```
Value of c: 30
Value of f: 23.333333333333333333

PL/SQL procedure successfully completed.
```

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 outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form −

```
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;
/
```

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

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

PL/SQL procedure successfully completed.
```

Assigning SQL Query Results to PL/SQL Variables

You 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**. The following example illustrates the concept. Let us create a table named CUSTOMERS −

(**For SQL statements, please refer to the** SQL tutorial)

```
CREATE TABLE CUSTOMERS(
   ID   INT NOT NULL,
   NAME VARCHAR (20) NOT NULL,
   AGE INT NOT NULL,
   ADDRESS CHAR (25),
   SALARY   DECIMAL (18, 2),
   PRIMARY KEY (ID)
);

Table Created
```

Let us now insert some values in the table −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );
```

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL −

```
DECLARE
   c_id customers.id%type := 1;
   c_name  customers.name%type;
   c_addr customers.address%type;
   c_sal  customers.salary%type;
```

```
BEGIN
   SELECT name, address, salary INTO c_name, c_addr, c_sal
   FROM customers
   WHERE id = c_id;
   dbms_output.put_line
   ('Customer ' ||c_name || ' from ' || c_addr || ' earns ' ||
c_sal);
END;
/
```

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

```
Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully
```

PL/SQL Constants

Constants are those values which when declared remain fixed throughout the PL/SQL block. For declaring constants, a `constant` keyword is used.

Syntax for declaring constants:

```
Constant_Name  constant  Datatype(size) := value;
```

Let's take a simple code example,

```
school_name constant VARCHAR2(20) := "DPS";
```

In the above code example, constant name is user defined followed by a keyword `constant` and then we have declared its value, which once declared cannot be changed.

Below we have a simple program to demonstrate the use of constants in PL/SQL,

```
set serveroutput on;

DECLARE

        school_name constant varchar2(20) := "DPS";

BEGIN

        dbms_output.put_line('I study in '|| school_name);

END;
```

```
I study in DPS

PL/SQL Procedure successfully completed.
```

Control Structures

PL/SQL Control Structures are used to control flow of execution. PL/SQL provides different kinds of statements to provide such type of procedural capabilities.These statements are almost same as that of provided by other languages.
The flow of control statements can be classified into the following categories:

- Conditional Control
- Iterative Control
- Sequential Control

1.Conditional Control :-

PL/SQL allows the use of an IF statement to control the execution of a block of code.
In PL/SQL, the IF -THEN - ELSIF - ELSE - END IF construct in code blocks allow specifying certain conditions under which a specific block of code should be executed.

Syntax:

```
IF < Condition > THEN
     < Action >
ELSIF <Condition> THEN
     < Action >
ELSE < Action >
END IF;
```

Example:

create file named "condi.sql"

```
DECLARE
        a Number := 30;       b Number;
BEGIN
        IF a > 40 THEN
          b := a - 40;
          DBMS_OUTPUT.PUT_LINE('b=' || b);
       elseif a = 30 then
```

```
          b := a + 40;
          DBMS_OUTPUT.PUT_LINE('b=' || b);
       ELSE
          b := 0;
          DBMS_OUTPUT.PUT_LINE('b=' || b);
       END IF;
END;
/
```

Output:

Run SQL Command Line

```
SQL>set serveroutput on

SQL>start d://condi.sql
b=70

PL/SQL successfully completed.
```

Simple CASE Statement

The simple `CASE` statement has this structure:

```
CASE selector

WHEN selector_value_1 THEN statements_1

WHEN selector_value_2 THEN statements_2

...

WHEN selector_value_n THEN statements_n

[ ELSE

  else_statements ]

END CASE;]
```

The *`selector`* is an expression (typically a single variable). Each *`selector_value`* can be either a literal or an expression.

The simple `CASE` statement runs the first *`statements`* for which *`selector_value`* equals *`selector`*. Remaining conditions are not evaluated. If no *`selector_value`* equals *`selector`*, the `CASE` statement runs *`else_statements`* if they exist and raises the predefined exception `CASE_NOT_FOUND` otherwise.

Simple CASE Statement

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  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('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
```

Result:

```
Very Good
```

Searched CASE Statement

The searched `CASE` statement has this structure:

```
CASE
WHEN condition_1 THEN statements_1
WHEN condition_2 THEN statements_2
...
WHEN condition_n THEN statements_n
[ ELSE
```

```
  else_statements ]
END CASE;]
```

The searched CASE statement runs the first *statements* for which *condition* is true. Remaining conditions are not evaluated. If no *condition* is true, the CASE statement runs *else_statements* if they exist and raises the predefined exception CASE_NOT_FOUND otherwise. (For complete syntax, see "CASE Statement".)

The searched CASE statement is logically equivalent to the simple CASE statement .

Searched CASE Statement

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  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('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
```

Result:

```
Very Good
```

EXCEPTION Instead of ELSE Clause in CASE Statement

```
DECLARE
```

```
  grade CHAR(1);
BEGIN
  grade := 'B';

  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('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
  END CASE;
EXCEPTION
  WHEN CASE_NOT_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No such grade');
END;
/
```

Result:

```
Very Good
```

2.Iterative Control :-

Iterative control indicates the ability to repeat or skip sections of a code block.
A **loop** marks a sequence of statements that has to be repeated. The keyword loop has to be placed before the first statement in the sequence of statements to be repeated, while the keyword end loop is placed immediately after the last statement in the sequence.
Once a loop begins to execute, it will go on forever. Hence a conditional statement that controls the number of times a loop is executed always accompanies loops.
PL/SQL supports the following structures for iterative control:

Simple loop :

In simple loop, the key word loop should be placed before the first statement in the sequence and the keyword end loop should be written at the end of the sequence to end the loop.

Syntax:

```
Loop
   < Sequence of statements >
 End loop;
```

Example:

create file named it.sql

```
DECLARE
    i number := 0;
BEGIN
     LOOP
    dbms_output.put_line ('i = '||i);
    i:=i+1;
                EXIT WHEN i>=11;
     END LOOP;

END;
/
```

Output:

Run SQL Command Line

```
SQL>set serveroutput on

SQL>start d://it.sql
i = 0
i = 1
i = 2
i = 3
i = 4
i = 5
```

```
i = 6
i = 7
i = 8
i = 9
i = 10
PL/SQL successfully completed.
```

WHILE loop

The **while** loop executes commands in its body as long as the condtion remains true

Syntax :

```
WHILE < condition >
LOOP
      < Action >
END LOOP
```

Example :

find reverse of given number using while loop

```
DECLARE
    num Number(3) :=123;
    ans Number(3) :=0;
    i Number(3) :=0;
 BEGIN
    WHILE num != 0
     LOOP
           i:=mod(num,10);
           ans:=(ans * 10 ) + i;
           num:=floor(num/10);
     END LOOP;
    dbms_output.put_line('reverse of given number is: ' || ans);
END;
/
```

Output :

Run SQL Command Line

```
SQL>set serveroutput on

SQL>start d://rev.sql

reverse of given number is: 321
PL/SQL successfully completed.
```

The FOR Loop

The **FOR** loop can be used when the number of iterations to be executed are known.

Syntax :

```
FOR variable IN [REVERSE] start..end
 LOOP
   < Action >
 END LOOP;
```

The variable in the For Loop need not be declared. Also the increment value cannot be specified. The For Loop variable is always incremented by 1.

Example :

```
DECLARE
     i number ;
 BEGIN
    FOR i IN 1 .. 10
     LOOP
         dbms_output.put_line ('i = '||i);
     END LOOP;
 END;
 /
```

Output :

Run SQL Command Line

```
SQL>set serveroutput on
```

```
SQL>start d://it.sql
i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
i = 10
PL/SQL successfully completed.
```

3.Sequential Control :-

The GOTO Statement

The GOTO statement changes the flow of control within a PL/SQL block. This statement allows execution of a section of code, which is not in the normal flow of control. The entry point into such a block of code is marked using the tags «userdefined name». The GOTO statement can then make use of this user-defined name to jump into that block of code for execution.

Syntax :

```
GOTO jump;
....
<<jump>>
```

Example :

```
DECLARE


 BEGIN
     dbms_output.put_line ('code starts');
     dbms_output.put_line ('before GOTO statement');
    GOTO down;
     dbms_output.put_line ('statement will not get executed..');
     <<down>>
     dbms_output.put_line ('flow of execution jumped here.');
 END;
```

```
/
```

Output :

Run SQL Command Line

```
SQL>set serveroutput on

SQL>start d://a.sql
code starts
before gotostatements
flow of execution jumped here.

PL/SQL successfully completed.
```

Exception Handling:-

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

- System-defined exceptions
- User-defined exceptions

Syntax for Exception Handling

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

```
DECLARE
   <declarations section>
BEGIN
   <executable command(s)>
EXCEPTION
   <exception handling goes here >
   WHEN exception1 THEN
      exception1-handling-statements
   WHEN exception2  THEN
      exception2-handling-statements
   WHEN exception3 THEN
      exception3-handling-statements
```

```
   ........
   WHEN others THEN
      exception3-handling-statements
END;
```

Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table −

```
DECLARE
   c_id customers.id%type := 8;
   c_name customerS.Name%type;
   c_addr customers.address%type;
BEGIN
   SELECT  name, address INTO  c_name, c_addr
   FROM customers
   WHERE id = c_id;
   DBMS_OUTPUT.PUT_LINE ('Name: '||  c_name);
   DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
   WHEN no_data_found THEN
      dbms_output.put_line('No such customer!');
   WHEN others THEN
      dbms_output.put_line('Error!');
END;
/
```

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

```
No such customer!

PL/SQL procedure successfully completed.
```

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

Guidelines for Avoiding and Handling PL/SQL Errors and Exceptions

Because reliability is crucial for database programs, use both error checking and exception handling to ensure your program can handle all possibilities:

- Add exception handlers whenever errors can occur.

  Errors are especially likely during arithmetic calculations, string manipulation, and database operations. Errors can also occur at other times, for example if a hardware failure with disk storage or memory causes a problem that has nothing to do with your code; but your code still must take corrective action.

- Add error-checking code whenever bad input data can cause an error.

  Expect that at some time, your code will be passed incorrect or null parameters, that your queries will return no rows or more rows than you expect.

  Test your code with different combinations of bad data to see what potential errors arise.

- Make your programs robust enough to work even if the database is not in the state you expect.

  For example, perhaps a table you query will have columns added or deleted, or their types changed. You can avoid such problems by declaring individual variables with `%TYPE` qualifiers, and declaring records to hold query results with `%ROWTYPE` qualifiers.

- Handle named exceptions whenever possible, instead of using `WHEN OTHERS` in exception handlers.

  Learn the names and causes of the predefined exceptions. If your database operations might cause particular ORA-*n* errors, associate names with these errors so you can write handlers for them. (You will learn how to do that later in this chapter.)

- Write out debugging information in your exception handlers.

  You might store such information in a separate table. If so, do it by invoking a subprogram declared with the `PRAGMA AUTONOMOUS_TRANSACTION`, so that you can commit your debugging information, even if you roll back the work that the main subprogram was doing.

- Carefully consider whether each exception handler should commit the transaction, roll it back, or let it continue.

  No matter how severe the error is, you want to leave the database in a consistent state and avoid storing any bad data.

Raising Exceptions:-

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

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

1.User defined exceptions:

This type of users can create their own exceptions according to the need and to raise these exceptions explicitly ***raise*** command is used.
Example:

- Divide non-negative integer x by y such that the result is greater than or equal to 1. From the given question we can conclude that there exist two exceptions
  - Division be zero.
  - If result is greater than or equal to 1 means y is less than or equal to x.

```
DECLARE
   x int:=&x; /*taking value at run time*/
   y int:=&y;
   div_r float;
   exp1 EXCEPTION;
   exp2 EXCEPTION;

BEGIN
   IF y=0 then
       raise exp1;

   ELSEIF y > x then
      raise exp2;

   ELSE
      div_r:= x / y;
      dbms_output.put_line('the result is '||div_r);

   END IF;

EXCEPTION
   WHEN exp1 THEN
      dbms_output.put_line('Error');
      dbms_output.put_line('division by zero not allowed');

   WHEN exp2 THEN
      dbms_output.put_line('Error');
      dbms_output.put_line('y is greater than x please check the input');

END;
```

Input 1: x = 20

y = 10

Output: the result is 2

Input 2: x = 20

y = 0

Output:

Error

division by zero not allowed

Input 3: x=20

y = 30

Output:

Error

y is greater than x please check the input

***RAISE_APPLICATION_ERROR**:*
It is used to display user-defined error messages with error number whose range is in between -20000 and -20999. When RAISE_APPLICATION_ERROR executes it returns error message and error code which looks ***same as Oracle built-in error.***
Example:

```
DECLARE
    myex EXCEPTION;
    n NUMBER :=10;

BEGIN
    FOR i IN 1..n LOOP
    dbms_output.put_line(i*i);
        IF i*i=36 THEN
        RAISE myex;
        END IF;
    END LOOP;

EXCEPTION
    WHEN myex THEN
        RAISE_APPLICATION_ERROR(-20015, 'Welcome');

END;
```

Output:

```
Error report:
ORA-20015: Welcome
ORA-06512: at line 13


1
4
9
16
```

```
25

36
```

2.System defined exceptions:

These exceptions are predefined in PL/SQL which get raised WHEN certain **database rule is violated.**

System-defined exceptions are further divided into two categories:

1. Named system exceptions.
2. Unnamed system exceptions.

- **Named system exceptions:** They have a predefined name by the system like ACCESS_INTO_NULL, DUP_VAL_ON_INDEX, LOGIN_DENIED etc. the list is quite big.
  So we will discuss some of the most commonly used exceptions:

  Lets create a table student.

```
create table student (s_id int , s_name varchar(20), marks int);

insert into student values(1, 'Suraj',100);

insert into student values(2, 'Praveen',97);

insert into student values(3, 'Jessie', 99);
```

G_id	G_name	marks
1	Suraj	100
2	Praveen	97
3	jessie	99

1. **NO_DATA_FOUND**: It is raised WHEN a SELECT INTO statement returns *no* rows. For eg:

```
DECLARE
   temp varchar(20);

BEGIN
   SELECT s_id into temp from student where s_name='Ravi';

exception
   WHEN no_data_found THEN
      dbms_output.put_line('ERROR');
      dbms_output.put_line('there is no name as');
      dbms_output.put_line('Ravi in student table');
end;
```

Output:

```
ERROR

there is no name as Ravi in student table
```

- **2.TOO_MANY_ROWS**:It is raised WHEN a SELECT INTO statement returns *more* than one row.

```
DECLARE
   temp varchar(20);

BEGIN

-- raises an exception as SELECT
-- into trying to return too many rows
   SELECT s_name into temp from student;
   dbms_output.put_line(temp);

EXCEPTION
   WHEN too_many_rows THEN
      dbms_output.put_line('error trying to SELECT too many rows');

end;
```

Output:

```
error trying to SELECT too many rows
```

3.VALUE_ERROR:This error is raised WHEN a statement is executed that resulted in an arithmetic, numeric, string, conversion, or constraint error. This error mainly results from programmer error or invalid data input.

```
DECLARE
```

```
   temp number;

BEGIN
   SELECT s_name  into temp from student where s_name='Suraj';
   dbms_output.put_line('the s_name is '||temp);

EXCEPTION
   WHEN value_error THEN
   dbms_output.put_line('Error');
   dbms_output.put_line('Change data type of temp to varchar(20)');

END;
```

Output:

```
Error

Change data type of temp to varchar(20)
```

4.ZERO_DIVIDE = raises exception WHEN dividing with zero.

```
DECLARE
   a int:=10;
   b int:=0;
   answer int;

BEGIN
   answer:=a/b;
   dbms_output.put_line('the result after division is'||answer);

exception
   WHEN zero_divide THEN
      dbms_output.put_line('dividing by zero please check the values again');
      dbms_output.put_line('the value of a is '||a);
      dbms_output.put_line('the value of b is '||b);
END;
```

Output:

```
dividing by zero please check the values again

the value of a is 10

the value of b is 0
```

- **Unnamed system exceptions:**Oracle doesn't provide name for some system exceptions called unnamed system exceptions.These exceptions *don't* occur frequently.These exceptions have two parts *code and an associated message*. The way to handle to these exceptions is to *assign name* to them using **Pragma EXCEPTION_INIT**

  Syntax:

- `PRAGMA EXCEPTION_INIT(exception_name, -error_number);`

  error_number are pre-defined and have negative integer range from -20000 to -20999.

Example:

```
DECLARE
   exp exception;
   pragma exception_init (exp, -20015);
   n int:=10;

BEGIN
   FOR i IN 1..n LOOP
      dbms_output.put_line(i*i);
         IF i*i=36 THEN
            RAISE exp;
         END IF;
   END LOOP;

EXCEPTION
   WHEN exp THEN
      dbms_output.put_line('Welcome');

END;
```

Output:

```
1

4

9

16

25

36

Welcome
```

***Scope rules in exception handling**:*

1. *We can't DECLARE an exception twice but we can DECLARE the same exception in **two dIFferent blocks.***
2. *Exceptions DECLAREd inside a block are local to that block and global to all its sub-blocks.*

As a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions DECLAREd in a sub-block.
If we reDECLARE a global exception in a sub-block, the local declaration prevails i.e. the scope of local is more.

Example:

```
DECLARE
   sample EXCEPTION;
```

```
   age NUMBER:=16;
BEGIN

   --  sub-block BEGINs
   DECLARE

      -- this declaration prevails
      sample  EXCEPTION;
      age NUMBER:=22;

   BEGIN
      IF age > 16 THEN
         RAISE sample; /* this is not handled*/
      END IF;

   END;
   -- sub-block ends

EXCEPTION
  -- Does not handle raised exception
  WHEN sample THEN
    DBMS_OUTPUT.PUT_LINE
      ('Handling  sample exception.');

  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE
      ('Could not recognize exception sample  in this scope.');
END;
```

Output:

```
Could not recognize exception sample  in this scope.
```

Advantages:

- *Exception handling is very useful for error handling, without it we have to issue the command at every point to check for execution errors:*
  Example:
- *Select ..*
- *.. check for 'no data found' error*
- *Select ..*
- *.. check for 'no data found' error*
- *Select ..*
- *.. check for 'no data found' error*

  *Here we can see that it is not **robust** as error processing is not separated from normal processing and IF we miss some line in the code than it may lead to some other kind of error.*

- *With exception handling we handle errors without writing statements multiple times and we can even handle **dIFferent** types of errors in one exception block:*
  Example:

```
BEGIN
    SELECT ...
    SELECT ...
    SELECT ...
     .
     .
     .
exception
    WHEN NO_DATA_FOUND THEN  /* catches all 'no data found' errors */
      ...
    WHEN ZERO_DIVIDE THEN    /* different types of */
    WHEN value_error THEN    /* errors handled in same block */
    ...
```

From above code we can conclude that exception handling

1. *Improves **readability** by letting us isolate error-handling routines and thus providing robustness.*
2. *Provides **reliability**, instead of checking for dIFferent types of errors at every point we can simply write them in exception block and IF error exists exception will be raised thus helping the programmer to find out the type of error and eventually resolve it.*

***Uses**: One of the real lIFe use of exception can be found in online train reservation system.*
While filling the station code to book the ticket IF we input wrong code it shows us the exception that the code doesn't exist in database.

Cursor:-

Oracle creates a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.

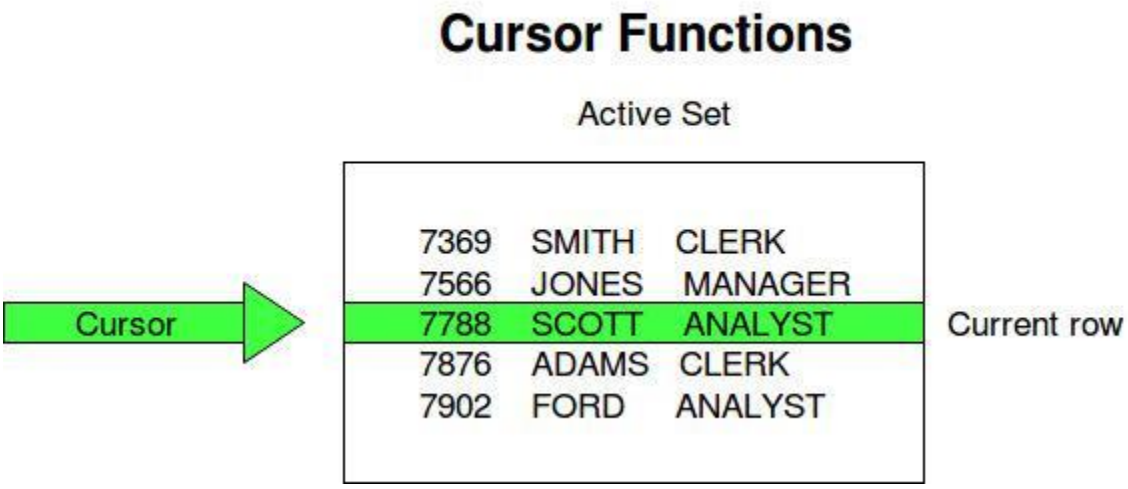
A **cursor** is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the **active set**.

You 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.

Use of Cursor

The major function of a cursor is to retrieve data, one row at a time, from a result set, unlike the SQL commands which operate on all the rows in the result set at one time. Cursors are used when the user needs to update records in a singleton fashion or in a row by row manner, in a database table.

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



Cursor Actions

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

Types of Cursors:-

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

- **Implicit Cursor:** If the Oracle engine opened a cursor for its internal processing it is known as an Implicit Cursor. It is created “automatically” for the user by Oracle when a query is executed and is simpler to code.
- **Explicit Cursor:** A Cursor can also be opened for processing data through a PL/SQL block, on demand. Such a user-defined cursor is known as an Explicit Cursor.

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.

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows o SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND** The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE stateme affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALS
3	**%ISOPEN** Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatica after executing its associated SQL statement.
4	**%ROWCOUNT** Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or return by a SELECT INTO statement.

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

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

Example

We will be using the CUSTOMERS table we had created.

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

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

```
DECLARE
   total_rows number(2);
BEGIN
   UPDATE customers
   SET salary = salary + 500;
   IF sql%notfound THEN
      dbms_output.put_line('no customers selected');
   ELSIF sql%found THEN
      total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

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

```
6 customers selected

PL/SQL procedure successfully completed.
```

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

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
```

```
|  1 | Ramesh   |  32 | Ahmedabad |  2500.00 |
|  2 | Khilan   |  25 | Delhi     |  2000.00 |
|  3 | kaushik  |  23 | Kota      |  2500.00 |
|  4 | Chaitali |  25 | Mumbai    |  7000.00 |
|  5 | Hardik   |  27 | Bhopal    |  9000.00 |
|  6 | Komal    |  22 | MP        |  5000.00 |
+----+----------+-----+-----------+----------+
```

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.

The syntax for creating an explicit cursor is −

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps −

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

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

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

Opening the Cursor

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

```
OPEN c_customers;
```

Fetching the Cursor

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

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

Closing the Cursor

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

```
CLOSE c_customers;
```

Example

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

```
DECLARE
   c_id customers.id%type;
   c_name customer.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;
/
```

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

```
1 Ramesh Ahmedabad
2 Khilan Delhi
3 kaushik Kota
4 Chaitali Mumbai
5 Hardik Bhopal
6 Komal MP

PL/SQL procedure successfully completed.
```

➢ PL/SQL cursor FOR LOOP statement

The cursor `FOR LOOP` statement is an elegant extension of the numeric `FOR LOOP` statement.
The numeric `FOR LOOP` executes the body of a loop once for every integer value in a specified range. Similarly, the cursor `FOR LOOP` executes the body of the loop once for each row returned by the query associated with the cursor.

A nice feature of the cursor `FOR LOOP` statement is that it allows you to fetch every row from a cursor without manually managing the execution cycle i.e., `OPEN`, `FETCH`, and `CLOSE`.
The cursor `FOR LOOP` implicitly creates its loop index as a record variable with the row type in which the cursor returns and then opens the cursor.
In each loop iteration, the cursor `FOR LOOP` statement fetches a row from the result set into its loop index. If there is no row to fetch, the cursor `FOR LOOP` closes the cursor.
The cursor is also closed if a statement inside the loop transfers control outside the loop, e.g., `EXIT` and `GOTO`, or raises an exception.
The following illustrates the syntax of the cursor `FOR LOOP` statement:

```
FOR record IN cursor_name
LOOP
    process_record_statements;
END LOOP;
```

1) record

The `record` is the name of the index that the cursor `FOR LOOP` statement declares implicitly as a `%ROWTYPE` record variable of the type of the cursor.
The `record` variable is local to the cursor `FOR LOOP` statement. It means that you can only reference it inside the loop, not outside. After the cursor `FOR LOOP` statement execution ends, the `record` variable becomes undefined.

2) cursor_name

The `cursor_name` is the name of an explicit cursor that is not opened when the loop starts.
Note that besides the cursor name, you can use a `SELECT` statement as shown below:

```
FOR record IN (select_statement)
LOOP
    process_record_statements;
END LOOP;
```

In this case, the cursor `FOR LOOP` declares, opens, fetches from, and closes an implicit cursor. However, the implicit cursor is internal; therefore, you cannot reference it.
Note that Oracle Database automatically optimizes a cursor `FOR LOOP` to work similarly to a `BULK COLLECT` query. Although your code looks as if it fetched one row at a time, Oracle Database fetches multiple rows at a time and allows you to process each row individually.

PL/SQL cursor FOR LOOP examples

Let's look at some examples of using the cursor `FOR LOOP` statement to see how it works.

A) PL/SQL cursor FOR LOOP example

The following example declares an explicit cursor and uses it in the cursor `FOR LOOP` statement.

```
DECLARE
  CURSOR c_product IS SELECT product_name,list_price FROM
products ORDER BY list_price DESC;
BEGIN
  FOR r_product IN c_product
  LOOP
    dbms_output.put_line( r_product.product_name || ': $' ||
r_product.list_price );
  END LOOP;
END;
```

In this example, the `SELECT` statement of the cursor retrieves data from the `products` table. The `FOR LOOP` statement opened, fetched each row in the result set, displayed the product information, and closed the cursor.

B) Cursor FOR LOOP with a SELECT statement example

The following example is equivalent to the example above but uses a query in a cursor `FOR LOOP` statement.

```
BEGIN
  FOR r_product IN (SELECT product_name, list_price FROM
products ORDER BY list_price DESC)
  LOOP
     dbms_output.put_line( r_product.product_name ||
        ': $' ||
        r_product.list_price );
  END LOOP;
END;
```

➢ Parameterized cursor:-

An explicit cursor may accept a list of parameters. Each time you open the cursor, you can pass different arguments to the cursor, which results in different result sets.

The following shows the syntax of a declaring a cursor with parameters:

```
CURSOR cursor_name (parameter_list)
IS
cursor_query;
```

In the cursor query, each parameter in the parameter list can be used anywhere which a constant is used. The cursor parameters cannot be referenced outside of the cursor query.

To open a cursor with parameters, you use the following syntax:

```
OPEN cursor_name (value_list);
```

In this syntax, you passed arguments corresponding to the parameters of the cursor.

Cursors with parameters are also known as parameterized cursors.

PL/SQL cursor with parameters example

The following example illustrates how to use a cursor with parameters:

```
DECLARE
    r_product products%rowtype;
    CURSOR c_product (low_price NUMBER, high_price NUMBER)
    IS
        SELECT *
        FROM products
        WHERE list_price BETWEEN low_price AND high_price;
BEGIN
    -- show mass products
    dbms_output.put_line('Mass products: ');
    OPEN c_product(50,100);
    LOOP
        FETCH c_product INTO r_product;
        EXIT WHEN c_product%notfound;
        dbms_output.put_line(r_product.product_name || ': '
||r_product.list_price);
    END LOOP;
    CLOSE c_product;

    -- show luxury products
    dbms_output.put_line('Luxury products: ');
    OPEN c_product(800,1000);
    LOOP
        FETCH c_product INTO r_product;
        EXIT WHEN c_product%notfound;
        dbms_output.put_line(r_product.product_name || ': '
||r_product.list_price);
    END LOOP;
    CLOSE c_product;

END;
/
```

In this example:

- First, declare a cursor that accepts two parameters low price and high price. The cursor retrieves products whose prices are between the low and high prices.
- Second, open the cursor and pass the low and high prices as 50 and 100 respectively. Then fetch each row in the cursor and show the product's information, and close the cursor.
- Third, open the cursor for the second time but with different arguments, 800 for the low price and 100 for the high price. Then the rest is fetching data, printing out product's information, and closing the cursor.

PL/SQL parameterized cursor with default values

A parameterized cursor can have default values for its parameters as shown below:

```
CURSOR cursor_name (
    parameter_name datatype := default_value,
    parameter_name datatype := default_value,
    ...
) IS
    cursor_query;
```

If you open the parameterized cursor without passing any argument, the cursor will use the default values for its parameters.

The following example shows how to use a parameterized cursor with default values.

```
DECLARE
    CURSOR c_revenue (in_year NUMBER :=2017 , in_customer_id
NUMBER := 1)
    IS
        SELECT SUM(quantity * unit_price) revenue
        FROM order_items
        INNER JOIN orders USING (order_id)
        WHERE status = 'Shipped' AND EXTRACT( YEAR FROM
order_date) = in_year
        GROUP BY customer_id
        HAVING customer_id = in_customer_id;

    r_revenue c_revenue%rowtype;
BEGIN
    OPEN c_revenue;
    LOOP
        FETCH c_revenue INTO r_revenue;
        EXIT    WHEN c_revenue%notfound;
```

```
        -- show the revenue
        dbms_output.put_line(r_revenue.revenue);
    END LOOP;
    CLOSE c_revenue;
END;
```

In this example, we declared a parameterized cursor with default values. When we opened the cursor, we did not pass any arguments; therefore, the cursor used the default values, 2017 for `in_year` and 1 for `in_customer_id`.

PL/SQL **Procedures and Functions:-**

Procedures and Functions are the subprograms which can be created and saved in the database as database objects. They can be called or referred inside the other blocks also.

Terminologies in PL/SQL Subprograms

Before we learn about PL/SQL subprograms, we will discuss the various terminologies that are the part of these subprograms. Below are the terminologies that we are going to discuss.

Parameter:

The parameter is variable or placeholder of any valid PL/SQL datatype through which the PL/SQL subprogram exchange the values with the main code. This parameter allows to give input to the subprograms and to extract from these subprograms.

- These parameters should be defined along with the subprograms at the time of creation.
- These parameters are included n the calling statement of these subprograms to interact the values with the subprograms.
- The datatype of the parameter in the subprogram and the calling statement should be same.
- The size of the datatype should not mention at the time of parameter declaration, as the size is dynamic for this type.

Based on their purpose parameters are classified as

1. IN Parameter
2. OUT Parameter
3. IN OUT Parameter

IN Parameter:

- This parameter is used for giving input to the subprograms.

- It is a read-only variable inside the subprograms. Their values cannot be changed inside the subprogram.
- In the calling statement, these parameters can be a variable or a literal value or an expression, for example, it could be the arithmetic expression like '5*8' or 'a/b' where 'a' and 'b' are variables.
- By default, the parameters are of IN type.

OUT Parameter:

- This parameter is used for getting output from the subprograms.
- It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the current subprograms.

IN OUT Parameter:

- This parameter is used for both giving input and for getting output from the subprograms.
- It is a read-write variable inside the subprograms. Their values can be changed inside the subprograms.
- In the calling statement, these parameters should always be a variable to hold the value from the subprograms.

These parameter type should be mentioned at the time of creating the subprograms.

RETURN

RETURN is the keyword that instructs the compiler to switch the control from the subprogram to the calling statement. In subprogram RETURN simply means that the control needs to exit from the subprogram. Once the controller finds RETURN keyword in the subprogram, the code after this will be skipped.

Normally, parent or main block will call the subprograms, and then the control will shift from those parent block to the called subprograms. RETURN in the subprogram will return the control back to their parent block. In the case of functions RETURN statement also returns the value. The datatype of this value is always mentioned at the time of function declaration. The datatype can be of any valid PL/SQL data type.

What is Procedure in PL/SQL?

A Procedure is a subprogram unit that consists of a group of PL/SQL statements. Each procedure in Oracle has its own unique name by which it can be referred. This subprogram unit is stored as a database object. Below are the characteristics of this subprogram unit.

Note: Subprogram is nothing but a procedure, and it needs to be created manually as per the requirement. Once created they will be stored as database objects.

- Procedures are standalone blocks of a program that can be stored in the database.
- Call to these procedures can be made by referring to their name, to execute the PL/SQL statements.
- It is mainly used to execute a process in PL/SQL.
- It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
- It contains declaration part (optional), execution part, exception handling part (optional).
- The values can be passed into the procedure or fetched from the procedure through parameters.
- These parameters should be included in the calling statement.
- Procedure can have a RETURN statement to return the control to the calling block, but it cannot return any values through the RETURN statement.
- Procedures cannot be called directly from SELECT statements. They can be called from another block or through EXEC keyword.

Syntax:

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

- CREATE PROCEDURE instructs the compiler to create new procedure. Keyword 'OR REPLACE' instructs the compile to replace the existing procedure (if any) with the current one.
- Procedure name should be unique.
- Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning.

Example1: Creating Procedure and calling it using EXEC

In this example, we are going to create a procedure that takes the name as input and prints the welcome message as output. We are going to use EXEC command to call procedure.

```
CREATE OR REPLACE PROCEDURE welcome_msg (p_name IN VARCHAR2)
IS
BEGIN
dbms_output.put_line (‘Welcome '|| p_name);
END;
/
EXEC welcome_msg (‘STUDENTS’);
```

Code Explanation:

- **Code line 1**: Creating the procedure with name 'welcome_msg' and with one parameter 'p_name' of 'IN' type.
- **Code line 4**: Printing the welcome message by concatenating the input name.
- Procedure is compiled successfully.
- **Code line 7**: Calling the procedure using EXEC command with the parameter 'STUDENTS'. Procedure is executed, and the message is printed out as "Welcome STUDENTS".

What is Function?

Functions is a standalone PL/SQL subprogram. Like PL/SQL procedure, functions have a unique name by which it can be referred. These are stored as PL/SQL database objects. Below are some of the characteristics of functions.

- Functions are a standalone block that is mainly used for calculation purpose.
- Function use RETURN keyword to return the value, and the datatype of this is defined at the time of creation.
- A Function should either return a value or raise the exception, i.e. return is mandatory in functions.

- Function with no DML statements can be directly called in SELECT query whereas the function with DML operation can only be called from other PL/SQL blocks.
- It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
- It contains declaration part (optional), execution part, exception handling part (optional).
- The values can be passed into the function or fetched from the procedure through the parameters.
- These parameters should be included in the calling statement.
- Function can also return the value through OUT parameters other than using RETURN.
- Since it will always return the value, in calling statement it always accompanies with assignment operator to populate the variables.

Syntax

```
CREATE OR REPLACE FUNCTION
<Function_name>
(
<parameterl IN/OUT <datatype>
)
RETURN <datatype>
[ IS | AS ]
<declaration_part>
BEGIN
<execution part>
EXCEPTION
<exception handling part>
END;
```

- CREATE FUNCTION instructs the compiler to create a new function. Keyword 'OR REPLACE' instructs the compiler to replace the existing function (if any) with the current one.
- The Function name should be unique.
- RETURN datatype should be mentioned.
- Keyword 'IS' will be used, when the procedure is nested into some other blocks. If the procedure is standalone then 'AS' will be used. Other than this coding standard, both have the same meaning.

Example1: Creating Function and calling it using Anonymous Block

In this program, we are going to create a function that takes the name as input and returns the welcome message as output. We are going to use anonymous block and select statement to call the function.

```
CREATE OR REPLACE FUNCTION welcome_msgJune ( p_name IN VARCHAR2)
```

```
RETURN VARCHAR2
IS
BEGIN
RETURN (‘Welcome ‘|| p_name);
END;
/
DECLARE
lv_msg VARCHAR2(250);
BEGIN
lv_msg := welcome_msg_func (‘STUDENTS’);
dbms_output.put_line(lv_msg);
END;
SELECT welcome_msg_func(‘STUDENTS’) FROM DUAL;
```

Code Explanation:

- **Code line 1**: Creating the function with name 'welcome_msg_func' and with one parameter 'p_name' of 'IN' type.
- **Code line 2**: declaring the return type as VARCHAR2
- **Code line 5**: Returning the concatenated value 'Welcome' and the parameter value.
- **Code line 8**: Anonymous block to call the above function.
- **Code line 9**: Declaring the variable with datatype same as the return datatype of the function.
- **Code line 11**: Calling the function and populating the return value to the variable 'lv_msg'.
- **Code line 12**: Printing the variable value. The output you will get here is "Welcome STUDENTS"
- **Code line 14**: Calling the same function through SELECT statement. The return value is directed to the standard output directly.

Similarities between Procedure and Function

- Both can be called from other PL/SQL blocks.
- If the exception raised in the subprogram is not handled in the subprogram exception handling section, then it will propagate to the calling block.
- Both can have as many parameters as required.
- Both are treated as database objects in PL/SQL.

Procedure Vs. Function: Key Differences

Procedure	Function

• Used mainly to a execute certain process	• Used mainly to perform some calculation
• Cannot call in SELECT statement	• A Function that contains no DML statements can be called in SELECT statement
• Use OUT parameter to return the value	• Use RETURN to return the value
• It is not mandatory to return the value	• It is mandatory to return the value
• RETURN will simply exit the control from subprogram.	• RETURN will exit the control from subprogram and also returns the value
• Return datatype will not be specified at the time of creation	• Return datatype is mandatory at the time creation

Creating a Function

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

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END [function_name];
```

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

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

We will use the CUSTOMERS table

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

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

   RETURN total;
END;
/
```

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

```
Total no. of Customers: 6
```

```
PL/SQL procedure successfully completed.
```

Example

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

```
DECLARE
   a number;
   b number;
   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;
   RETURN z;
END;
BEGIN
   a:= 23;
   b:= 45;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

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

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

PL/SQL procedure successfully completed.
```

PL/SQL Recursive Functions

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

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

```
n! = n*(n-1)!
   = n*(n-1)*(n-2)!
      ...
   = n*(n-1)*(n-2)*(n-3)... 1
```

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

```
DECLARE
   num number;
   factorial number;

FUNCTION fact(x number)
RETURN number
IS
   f number;
BEGIN
   IF x=0 THEN
      f := 1;
   ELSE
      f := x * fact(x-1);
   END IF;
RETURN f;
END;

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

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

```
Factorial 6 is 720

PL/SQL procedure successfully completed.
```

Removing a function:-

The `DROP FUNCTION` deletes a function from the Oracle Database. The syntax for removing a function is straightforward:

```
DROP FUNCTION function_name;
```

Followed by the `DROP FUNCTION` keywords is the function name that you want to drop.

For example, the following statement drops the `GET_TOTAL_SALES` function:

```
DROP FUNCTION GET_TOTAL_SALES;
```

Oracle issued a message indicating that the function `GET_TOTAL_SALES` has been dropped:

```
Function GET_TOTAL_SALES dropped.
```

Advantages of function:-

1. We can make a single call to the database to run a block of statements thus it improves the performance against running SQL multiple times. This will reduce the number of calls between the database and the application.
2. We can divide the overall work into small modules which becomes quite manageable also enhancing the readability of the code.
3. It promotes reusability.
4. It is secure since the code stays inside the database thus hiding internal database details from the application (user). The user only makes a call to the PL/SQL functions. Hence security and data hiding is ensured.

Advantages of stored procedures:

- Since stored procedures are compiled and stored, whenever you call a procedure the response is quick.
- you can group all the required SQL statements in a procedure and execute them at once.
- Since procedures are stored on the database server which is faster than client. You can execute all the complicated quires using it, which will be faster.
- Using procedures, you can avoid repetition of code moreover with these you can use additional SQL functionalities like calling stored functions.
- Once you compile a stored procedure you can use it in any number of applications. If any changes are needed you can just change the procedures without touching the application code.

Triggers in PL/SQL:-

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

These events can be:

- DDL statements (CREATE, ALTER, DROP, TRUNCATE)
  These triggers are often used for auditing purposes to record changes of the schema.
- DML statements (INSERT, SELECT, UPDATE, DELETE) For example, if you define a trigger that fires before an `INSERT` statement on the `customers` table, the trigger will fire once before a new row is inserted into the `customers` table.
- Database operation like connecting or disconnecting to oracle (LOGON, LOGOFF, SHUTDOWN)

- The act of executing a trigger is also known as firing a trigger. We say that the trigger is fired.
- Triggers are automatically and repeatedly called upon by oracle engine on satisfying certain condition.
- Triggers can be activated or deactivated depending on the requirements.
- If triggers are activated then they are executed implicitly by oracle engine and if triggers are deactivated then they are executed explicitly by oracle engine.

Oracle trigger usages

Oracle triggers are useful in many cases such as the following:

- Enforcing complex business rules that cannot be established using integrity constraint such as `UNIQUE`, `NOT NULL`, and `CHECK`.
- Preventing invalid transactions.
- Gathering statistical information on table accesses.
- Generating value automatically for derived columns.
- Auditing sensitive data.

PL/SQL: Parts of a Trigger

Whenever a trigger is created, it contains the following three sequential parts:

- **Triggering Event or Statement:** The statements due to which a trigger occurs is called triggering event or statement. Such statements can be DDL statements, DML statements or any database operation, executing which gives rise to a trigger.
- **Trigger Restriction:** The condition or any limitation applied on the trigger is called trigger restriction. Thus, if such a condition is **TRUE** then trigger occurs otherwise it does not occur.
- **Trigger Action:** The body containing the executable statements that is to be executed when trigger occurs that is with the execution of Triggering statement and upon evaluation of Trigger restriction as **True** is called Trigger Action.

PL/SQL: Types of Triggers

The above diagram clearly indicated that Triggers can be classified into three categories:

1. Level Triggers
2. Event Triggers
3. Timing Triggers

which are further divided into different parts.

Level Triggers

There are 2 different types of level triggers, they are:

1. **ROW LEVEL TRIGGERS**
    - It fires for every record that got affected with the execution of DML statements like INSERT, UPDATE, DELETE etc.
    - It always use a `FOR EACH` ROW clause in a triggering statement.
2. **STATEMENT LEVEL TRIGGERS**
    - It fires once for each statement that is executed.

Event Triggers

There are 3 different types of event triggers, they are:

1. **DDL EVENT TRIGGER**
    - It fires with the execution of every DDL statement(CREATE, ALTER, DROP, TRUNCATE).
2. **DML EVENT TRIGGER**
    - It fires with the execution of every DML statement(INSERT, UPDATE, DELETE).
3. **DATABASE EVENT TRIGGER**
    - It fires with the execution of every database operation which can be LOGON, LOGOFF, SHUTDOWN, SERVERERROR etc.

Timing Triggers

There are 2 different types of timing triggers, they are:

- **BEFORE TRIGGER**
    - It fires before executing DML statement.
    - Triggering statement may or may not executed depending upon the before condition block.
- **AFTER TRIGGER**
    - It fires after executing DML statement.

Syntax for creating Triggers

Following is the syntax for creating a trigger:

```
CREATE OR REPLACE TRIGGER <trigger_name>

BEFORE/AFTER/INSTEAD OF

        INSERT/DELETE/UPDATE ON <table_name>

REFERENCING (OLD AS O, NEW AS N)
```

```
        FOR EACH ROW WHEN (test_condition)
DECLARE
        -- Variable declaration;
BEGIN
        -- Executable statements;
EXCEPTION
        -- Error handling statements;
END <trigger_name>;
END;
```

where,

`CREATE OR REPLACE TRIGGER` is a keyword used to create a trigger and **<trigger_name>** is user-defined where a trigger can be given a name.

`BEFORE/AFTER/INSTEAD OF` specify the timing of the trigger's occurance. `INSTEAD OF` is used when a view is created.

INSERT/UPDATE/DELETE specify the DML statement.

<table_name> specify the name of the table on which DML statement is to be applied.

`REFERENCING` is a keyword used to provide reference to old and new values for DML statements.

`FOR EACH ROW` is the clause used to specify row level tigger.

`WHEN` is a clause used to specify condition to be applied and is only applicable for row-level trigger.

`DECLARE`, `BEGIN`, `EXCEPTION`, `END` are the different sections of PL/SQL code block containing variable declaration, executable statements, error handling statements and marking end of PL/SQL block respectively where DECLARE and EXCEPTION part are optional.

Example!

Below we have a simple program to demonstrate the use of Triggers in PL/SQL code block.

```
CREATE OR REPLACE TRIGGER CheckAge
BEFORE
INSERT OR UPDATE ON student
```

```
FOR EACH ROW

BEGIN

        IF :new.Age>30 THEN

                raise_application_error(-20001, 'Age should not be greater
than 30');

        END IF;

END;
```

```
Trigger created.
```

Following is the STUDENT table,

ROLLNO	SNAME	AGE	COURSE
11	Anu	20	BSC
12	Asha	21	BCOM
13	Arpit	18	BCA
14	Chetan	20	BCA
15	Nihal	19	BBA

After initializing the trigger `CheckAge`, whenever we will insert any new values or update the existing values in the above table STUDENT our trigger will check the **age** before executing `INSERT` or `UPDATE` statements and according to the result of triggering restriction or condition it will execute the statement.

Let's take a few examples and try to understand this,

Example 1:

```
INSERT into STUDENT values(16, 'Saina', 32, 'BCOM');
```

```
Age should not be greater than 30
```

Example 2:

```
INSERT into STUDENT values(17, 'Anna', 22, 'BCOM');
```

```
1 row created
```

Example 3:

```
UPDATE STUDENT set age=31 where ROLLNO=12;
```

```
Age should not be greater than 30
```

Example 4:

```
 UPDATE STUDENT set age=23 where ROLLNO=12;
```

```
1 row updated.
```