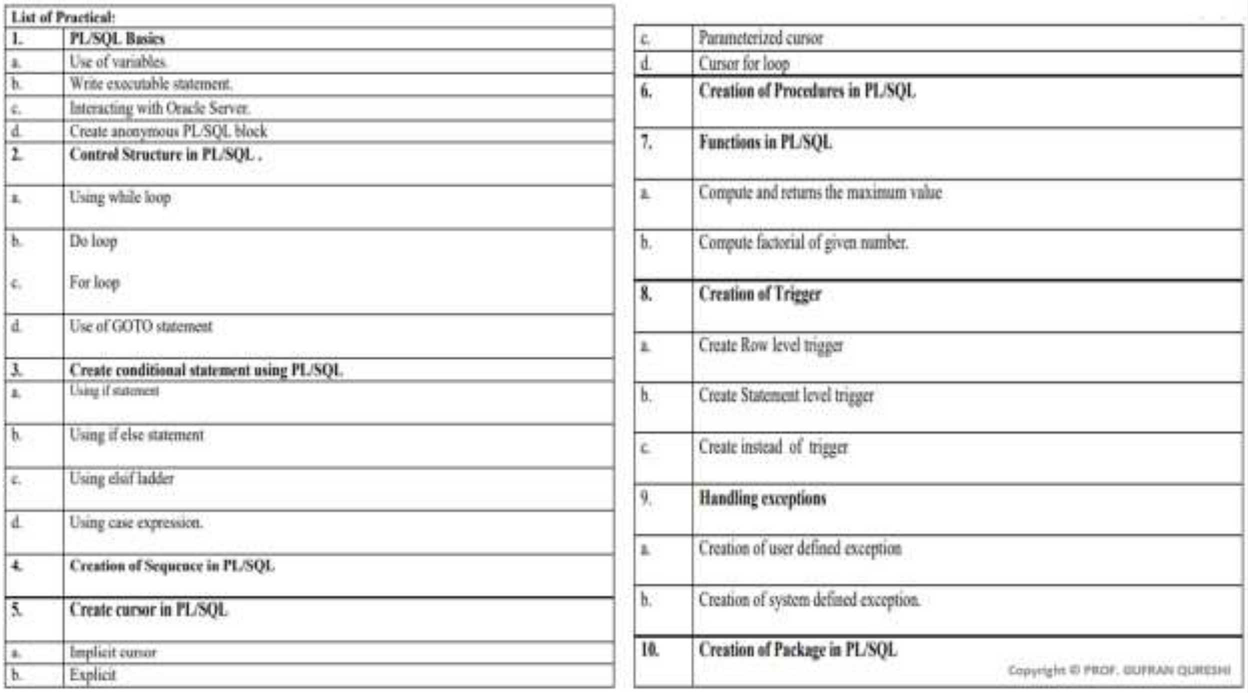
L.S. Raheja College Of Arts & Commerce

PRESENTS,

PL/SQL Practical

F.Y.B.Sc.I.T – SEM II

Mumbai University Course Code: USIT2P5

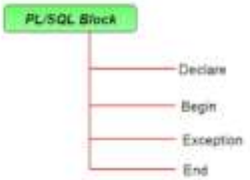


List of Practical:	
1.	**PL/SQL Basics**
a.	Use of variables.
b.	Write executable statement.
c.	Interacting with Oracle Server.
d.	Create anonymous PL/SQL block
2.	**Control Structure in PL/SQL .**
a.	Using while loop
b.	Do loop
c.	For loop
d.	Use of GOTO statement
3.	**Create conditional statement using PL/SQL**
a.	Using if statement
b.	Using if else statement
c.	Using elsif ladder
d.	Using case expression.
4.	**Creation of Sequence in PL/SQL**
5.	**Create cursor in PL/SQL**
a.	Implicit cursor
b.	Explicit
c.	Parameterized cursor
d.	Cursor for loop
6.	**Creation of Procedures in PL/SQL**
7.	**Functions in PL/SQL**
a.	Compute and returns the maximum value
b.	Compute factorial of given number.
8.	**Creation of Trigger**
a.	Create Row level trigger
b.	Create Statement level trigger
c.	Create instead of trigger
9.	**Handling exceptions**
a.	Creation of user defined exception
b.	Creation of system defined exception.
10.	**Creation of Package in PL/SQL**

Introduction to PL/SQL

PL/SQL Inherits Database Robustness, Security, and Portability

PL/SQL is a procedural language designed specifically to embrace SQL statements within its syntax. PL/SQL program units are compiled by the Oracle Database server and stored inside the database. And at run-time, both PL/SQL and SQL run within the same server process, bringing optimal efficiency. PL/SQL automatically inherits the robustness, security, and portability of the Oracle Database.



PL/SQL is **a block of codes that used to write the entire 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 things needs to be done. Execute as a single statement.

SQL is data oriented language. PL/SQL is application oriented language. SQL is used to write queries, create and execute DDL and DML statments. PL/SQL is used to write program blocks, functions, procedures, triggers and packages.

SQL: declarative -> "What to do"

PL/SQL: "What to do" + "How to do"

PL/SQL block : 1. Declarative -> start 'declare'
2. Executive -> start with 'begin'
3. Exception -> end with 'end'

Software Required

1. **Oracle: https://www.oracle.com/database/technologies/xe-prior-release-downloads.html**
Steps: https://www.geeksforgeeks.org/how-to-install-oracle-database-11g-on-windows/

Oracle Database: Oracle Database (known as Oracle *RDBMS*) is a Database Management System produced and marketed by Oracle Corporation.
The Most Fundamental and common usage of Oracle Database is to store a Pre-Defined type of Data. It supports the *Structured Query language* (SQL) to Manage and Manipulate the Data that it has. It is one of the most Reliable and highly used Relational Database Engines.

There are many versions of Oracle Database like Oracle Database 10g, Oracle Database 11g, Oracle Database 12c, Oracle Database 19c, etc. from which *Oracle 19c* is the Latest Version. In this article, we will learn how to Install version 11g on Windows. Oracle Database offers market-leading performance, scalability, reliability, and security, both on-premises and in the cloud. Oracle Database 19c is the current long term release, and it provides the highest level of release stability and longest time-frame for support and bug fixes.

Practical 1: PL/SQL Basics

a. Use of variables.
b. Write executable statement.
c. Interacting with Oracle Server.
d. Create anonymous PL/SQL block.

PL/SQL variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in PL/SQL has a specific data type, which determines the size and the layout of the variable's memory; the range of values that can be stored within that memory and the set of operations that can be applied to the variable.
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. You cannot use a reserved PL/SQL keyword as a variable name.
PL/SQL programming language allows to define various types of variables, such as date time data types, records, collections, etc. which we will cover in subsequent chapters. For this chapter, let us study only basic variable types.

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.
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

Practical 1: PL/SQL Basics

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';

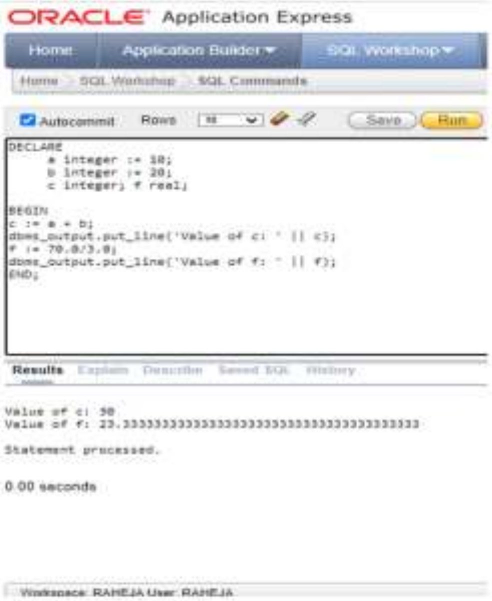
Practical 1: PL/SQL Basics

1a. Use 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;
/
```

Output:

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



```
DECLARE
      a integer := 10;
      b integer := 20;
      c integer; f real;

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

```
Value of c: 30
Value of f: 23.33333333333333333333333333333333333333

Statement processed.

0.00 seconds
```

Practical 1: PL/SQL Basics

Variable Scope in PL/SQL

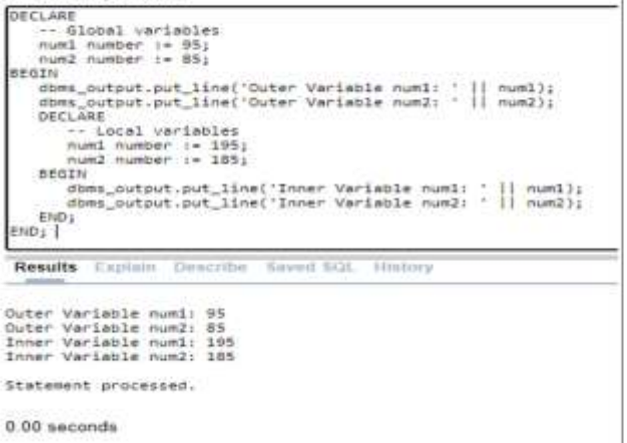
PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope −

Local variables − Variables declared in an inner block and not accessible to outer blocks.

Global variables − Variables declared in the 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;
/
```



```
DECLARE
   -- Global variables
   num1 number := 95;
   num2 number := 85;
BEGIN
   dbms_output.put_line('Outer Variable num1: ' || num1);
   dbms_output.put_line('Outer Variable num2: ' || num2);
   DECLARE
      -- Local variables
      num1 number := 195;
      num2 number := 185;
   BEGIN
      dbms_output.put_line('Inner Variable num1: ' || num1);
      dbms_output.put_line('Inner Variable num2: ' || num2);
   END;
END;
```

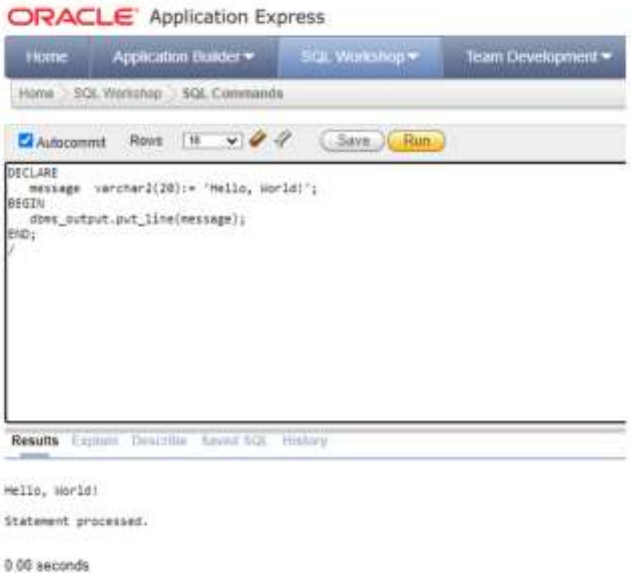
```
Outer Variable num1: 95
Outer Variable num2: 85
Inner Variable num1: 195
Inner Variable num2: 185

Statement processed.

0.00 seconds
```

Practical 1: PL/SQL Basics

1b. Write executable statement.

The 'Hello World' Example

```
DECLARE
  message  varchar2(20):= 'Hello, World!';
BEGIN
  dbms_output.put_line(message);
END;
/
```



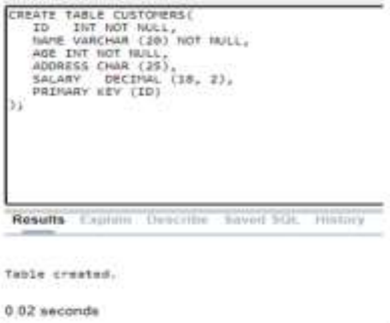
```
DECLARE
   message  varchar2(20):= 'Hello, World!';
BEGIN
   dbms_output.put_line(message);
END;
/
```

```
Hello, World!

Statement processed.

0.00 seconds
```

Practical 1: PL/SQL Basics

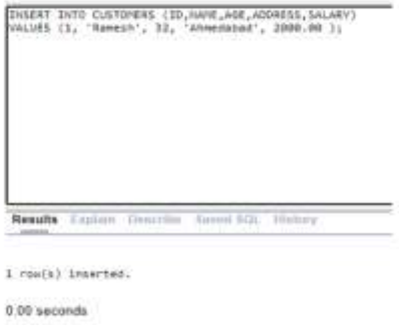
1c. Interacting with Oracle Server.

```
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) );

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 );
```



```
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.

0.02 seconds
```

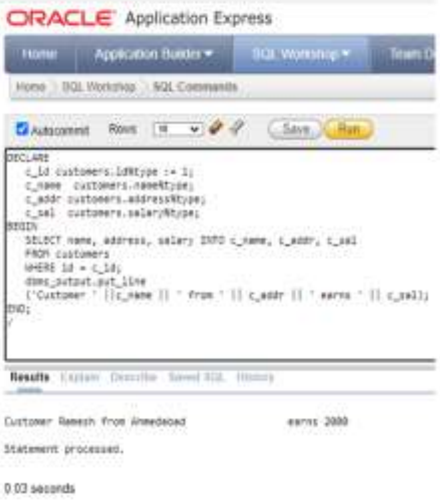


```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
```

```
1 row(s) inserted.

0.00 seconds
```

Practical 1: PL/SQL Basics

```
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;
/
```



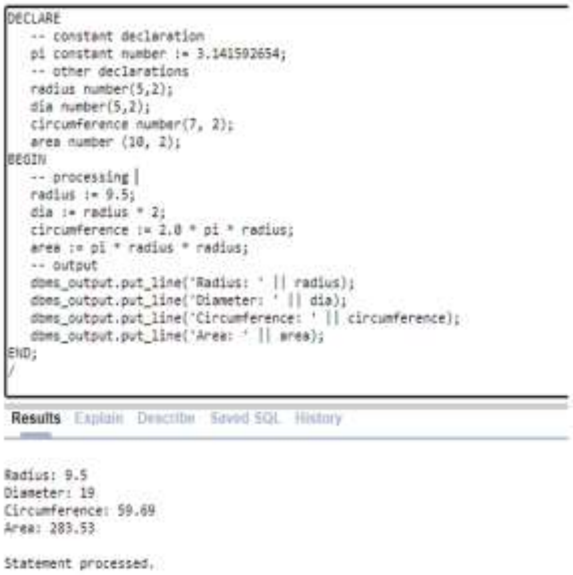
```
DECLARE
   c_id customers.id%type := 1;
   c_name  customers.name%type;
   c_addr customers.address%type;
   c_sal  customers.salary%type;
BEGIN
   SELECT name, address, salary INTO c_name, c_addr, c_sal
   FROM customers
   WHERE id = c_id;
   dbms_output.put_line
   ('Customer ' ||c_name || ' from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

```
Customer Ramesh from Ahmedabad                earns 2000

Statement processed.

0.03 seconds
```

Practical 1: PL/SQL Basics

1d. Create anonymous PL/SQL block

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



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

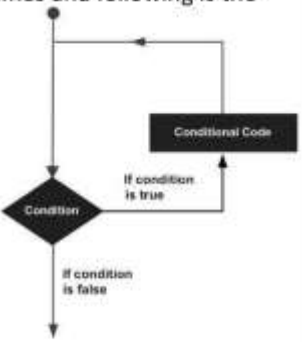
```
Radius: 9.5
Diameter: 19
Circumference: 59.69
Area: 283.53

Statement processed.
```

Practical 2: Control Structure in PL/SQL

a. Basic Loop
b. Using while loop
c. For Loop
d. Use of GOTO statement

There may be a situation when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages –



PL/SQL Basic Loop: Basic loop or simple loop is preferred in PL/SQL code when there is no surety about how many times the block of code is to be repeated. When we use the basic loop the code block will be executed at least once.

While using it, following two things must be considered:

•Simple loop always begins with the keyword LOOP and ends with a keyword END LOOP.

•A basic/simple loop can be terminated at any given point by using the exit statement or by specifying certain condition by using the statement exit when.

Practical 2: Control Structure in PL/SQL

1a. Basic loop

Syntax:

```
LOOP
        sequence of statements
END LOOP;
```

Example:

```
//set serveroutput on;

DECLARE
        i int;
BEGIN
        i := 1;
        LOOP
                if i>10 then
                        exit;
                end if;
                dbms_output.put_line(i);
                i := i+1;
        END LOOP;
END;
```

Practical 2: Control Structure in PL/SQL

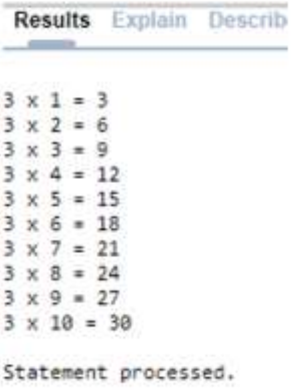
1a. Basic loop

Syntax:

```
LOOP
        sequence of statements
END LOOP;
```

Example:

```
//set serveroutput on;

DECLARE
        i int;
        res int;
BEGIN
        i := 1;
        LOOP
                if i>10 then
                        exit;
                end if;
                res:=3*i;
                dbms_output.put_line(‘3'||‘ x '||i||‘ = '||res);
                i := i+1;
        END LOOP;
END;
```



```
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30

Statement processed.
```

Practical 2: Control Structure in PL/SQL

1b. While loop

It is an entry controlled loop which means that before entering in a while loop first the condition is tested, if the condition is **TRUE** the statement or a group of statements get executed and if the condition is **FALSE** the control will move out of the while loop.

Syntax:

```
WHILE <test_condition> LOOP
        <action>
END LOOP;
```

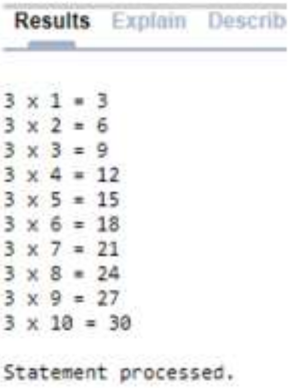
Example:

```
//set serveroutput on;

DECLARE
        num int:=1;
BEGIN
        while(num <= 10) LOOP
                dbms_output.put_line( ' '|| num);
                num := num+2;
        END LOOP;
END;
```

Practical 2: Control Structure in PL/SQL

1b. While loop

```
DECLARE
        i int;
        n int;
        res int;
BEGIN
        i:= 1;
        n:=:n;
        while(i<=10) LOOP
                res:=n*i;
                dbms_output.put_line(n||' x '||i||' = '||res);
                i := i+1;
        END LOOP;
END;
```



```
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30

Statement processed.
```

Practical 2: Control Structure in PL/SQL

1c. For loop

This loop is used when some statements in PL/SQL code block are to be repeated for a fixed number of times.
When we use the for loop we are supposed to define a counter variable which decides how many time the loop will be executed based on a starting and ending value provided at the beginning of the loop.
The for loop automatically increments the value of the counter variable by 1 at the end of each loop cycle.
The programmer need not have to write any instruction for incrementing or decrementing value.

Syntax:

```
FOR counter_variable IN start_value..end_value LOOP
        statement to be executed
END LOOP;
```

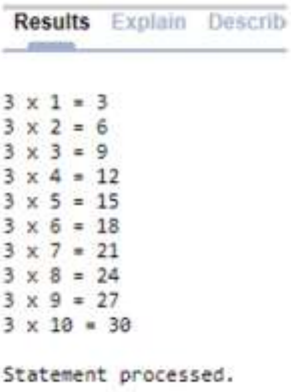
Example:

```
//set serveroutput on;
DECLARE
        i number(2);
BEGIN
        FOR i IN 1..10 LOOP
                dbms_output.put_line(i);
        END LOOP;
END;
```

Practical 2: Control Structure in PL/SQL

1c. For loop

```
DECLARE
        i int;
        n int;
        res int;
BEGIN
        i:= 1;
        n:=:n;
        FOR i IN 1..10  LOOP
                res:=n*i;
                dbms_output.put_line(n||' x '||i||' = '||res);
        END LOOP;
END;
```



```
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
3 x 10 = 30

Statement processed.
```

Practical 2: Control Structure in PL/SQL

1d. Goto Statement

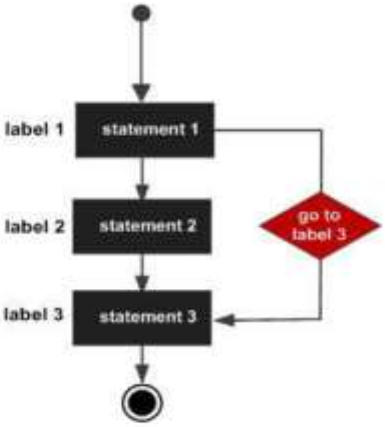
A **GOTO** statement in PL/SQL programming language provides an unconditional jump from the GOTO to a labeled statement in the same subprogram.
The use of GOTO statement is not recommended in any programming language because it makes it difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a GOTO can be rewritten so that it doesn't need the GOTO.

Syntax:

```
GOTO label;
..
..
<< label >>
statement;
```

Example:

```
DECLARE
 a number(2) := 10;
BEGIN
 <<loopstart>>
 -- while loop execution
 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;
/
```



Practical 3: Create conditional statement using PL/SQL

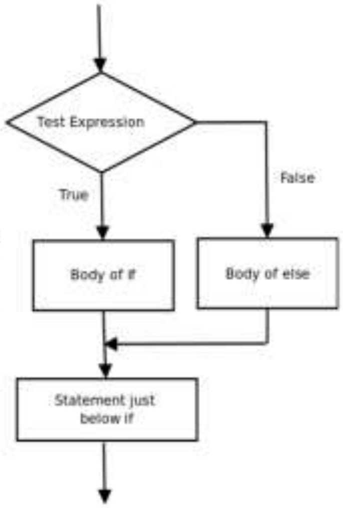
The conditional selection statements, IF and CASE, run different statements for different data values.
The IF statement either runs or skips a sequence of one or more statements, depending on a condition. The IF statement has these forms:

IF THEN

IF THEN ELSE

IF THEN ELSIF

The CASE statement chooses from a sequence of conditions, and runs the corresponding statement. The CASE statement has these forms:
Simple, which evaluates a single expression and compares it to several potential values.
Searched, which evaluates multiple conditions and chooses the first one that is true.
The CASE statement is appropriate when a different action is to be taken for each alternative.



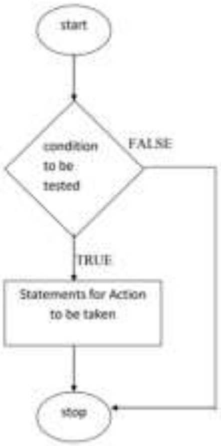
Practical 3: Create conditional statement using PL/SQL

1a: Using if statement

The if statement, or the if...then statement can be used when there is only a single condition to be tested. If the result of the condition is TRUE then certain specified action will be performed otherwise if it is FALSE then no action is taken and the control of program will just move out of the if code block.

Syntax:

```
if <test_condition> then
        body of action
end if;
```



Example:

```
DECLARE
        x int:=10;
        y int:=80;
BEGIN
        if(y>x) then
                dbms_output.put_line('Result: ' ||y|| ' is greater than ' ||x);
        end if;
END;
```

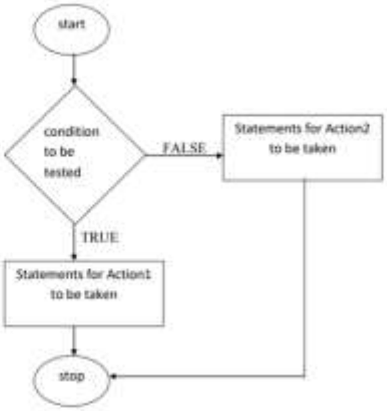
Practical 3: Create conditional statement using PL/SQL

1b: Using if else statement

Using this statement group we can specify two statements or two set of statements, dependent on a condition such that when the condition is true then one set of statements is executed and if the condition is false then the other set of statements is executed.

Syntax:

```
if <test_condition> then
        statement 1/set of statements 1
else
        statement 2/set of statements 2
end if;
```


start
condition to be tested
FALSE
Statements for Action2 to be taken
TRUE
Statements for Action1 to be taken
stop

Example:

```
DECLARE
        x int;
BEGIN
        x :=: x;
        if mod(x,2) = 0 then
                dbms_output.put_line('Even Number');
        else
                dbms_output.put_line('Odd Number');
        end if;
END;
```

Output:

```
Enter value for x:6
Even Number
```

Practical 3: Create conditional statement using PL/SQL

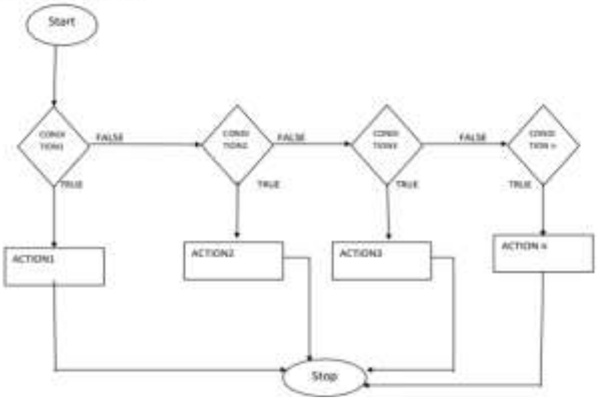
1c: Using elsif ladder

It is used to check multiple conditions. Sometimes it is required to test more than one condition in that case if...then...else statement cannot be used. For this purpose, if...then...elsif...else statement is suitable in which all the conditions are tested one by one and whichever condition is found to be TRUE, that block of code is executed. And if all the conditions result in FALSE then the else part is executed.

In the following syntax, it can be seen firstly condition1 is checked, if it is true, the statements following it are executed and then control moves out of the complete if block but if the condition is false then the control checks condition2 and repeats the same process. If all the conditions fail then the else part is executed.

Syntax:

```
if <test_condition1> then
        body of action
elsif <test_condition2>then
        body of action
elsif<test_condition3>then
        body of action
...
...
...
else
        body of action
end if;
```



Practical 3: Create conditional statement using PL/SQL

1c: Using elsif ladder

```
DECLARE
        a int;
        b int;
BEGIN
        a :=:a;
        b :=:b;
        if(a>b) then
                dbms_output.put_line('a is greater than b');
        elsif(b>a) then
                dbms_output.put_line('b is greater than a');
        else
                dbms_output.put_line('Both a and b are equal');
        end if;
END;
```

Output:

```
Enter value for a: 8
Enter value for b: 5
a is greater than b
```

Practical 3: Create conditional statement using PL/SQL

1d: Using case expression

If we try to describe the case statement in one line then, then we can say means "one out of many". It is a decision making statement that selects only one option out of the multiple available options.
It uses a selector for this purpose. This selector can be a variable, function or procedure that returns some value and on the basis of the result one of the case statements is executed. If all the cases fail then the else case is executed.

Syntax:

```
CASE selector
        when value1 then Statement1;
        when value2 then Statement2;
        ...
        ...
        else statement;
end CASE;
```

Example:

```
DECLARE
        a int;
        b int;
BEGIN
        a :=:a;
        b := mod(a,2);
        CASE b
        when 0 then dbms_output.put_line('Even Number');
        when 1 then dbms_output.put_line('Odd Number');
        else dbms_output.put_line('User has not given any input value to check');
        END CASE;
END;
```

Output:

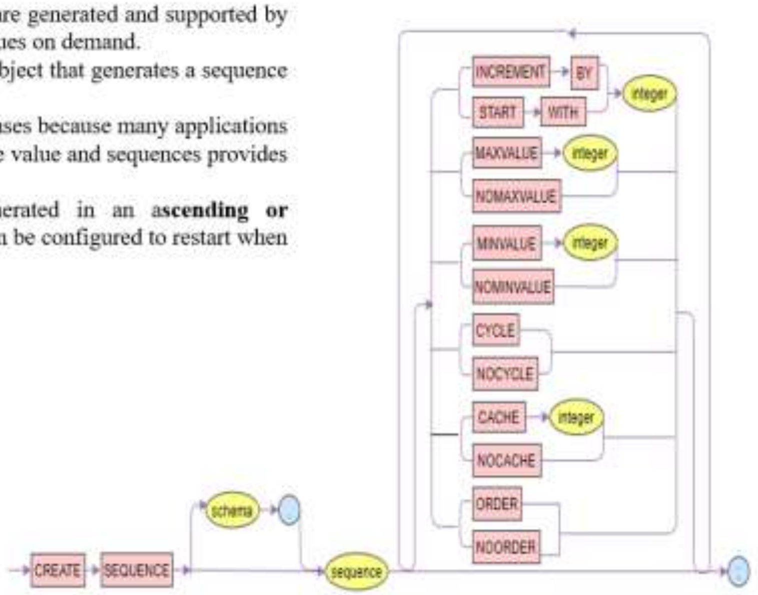
```
Enter the value for a:7
Odd number
```

Practical 4: Creation of Sequence in PL/SQL

Sequence is a set of integers 1, 2, 3, … that are generated and supported by some database systems to produce unique values on demand.
A sequence is a user defined schema bound object that generates a sequence of numeric values.
Sequences are frequently used in many databases because many applications require each row in a table to contain a unique value and sequences provides an easy way to generate them.
The sequence of numeric values is generated in an a**scending or descending order** at defined intervals and can be configured to restart when exceeds max_value.

Syntax:

```
CREATE SEQUENCE sequence_name
START WITH initial_value
INCREMENT BY increment_value
MINVALUE minimum value
MAXVALUE maximum value
CYCLE|NOCYCLE
CACHE cache_size | NOCACHE
ORDER | NOORDER ;
```



Practical 4: Creation of Sequence in PL/SQL

sequence_name: Name of the sequence.

initial_value: starting value from where the sequence starts.
Initial_value should be greater than or equal to minimum value and less than equal to maximum value.

increment_value: Value by which sequence will increment itself.
Increment_value can be positive or negative.

minimum_value: Minimum value of the sequence.
maximum_value: Maximum value of the sequence.

cycle: When sequence reaches its set_limit it starts from beginning.

nocycle: An exception will be thrown if sequence exceeds its max_value.

cache: Specify the number of sequence values that Oracle will preallocate and keep in the memory for faster access. The minimum of the cache size is 2. The maximum value of the cache size is based on this formula:
(CEIL (MAXVALUE - MINVALUE)) / ABS (INCREMENT)

order: Use ORDER to ensure that Oracle will generate the sequence numbers in order of request. This option is useful if you are using Oracle Real Application Clusters. When you are using exclusive mode, then Oracle will always generate sequence numbers in order.

noorder: Use NOORDER if you do not want to ensure Oracle to generate sequence numbers in order of request. This option is the default.

Practical 4: Creation of Sequence in PL/SQL

1) Basic Oracle Sequence example

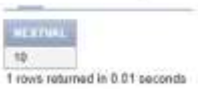
The following statement creates an ascending sequence called id_seq, starting from 10, incrementing by 10, minimum value 10, maximum value 100. The sequence returns 10 once it reaches 100 because of the CYCLE option.

```
CREATE SEQUENCE id_seq
  INCREMENT BY 10
  START WITH 10
  MINVALUE 10
  MAXVALUE 100
  CYCLE
  CACHE 2;
```

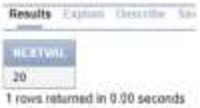
To get the next value of the sequence, you use the NEXTVAL pseudo-column:

```
SELECT
  id_seq.NEXTVAL
FROM
  dual;
```

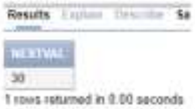
Here is the output:



```
NEXTVAL
10
1 rows returned in 0.01 seconds
```



```
NEXTVAL
20
1 rows returned in 0.00 seconds
```



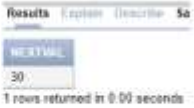
```
NEXTVAL
30
1 rows returned in 0.00 seconds
```

Practical 4: Creation of Sequence in PL/SQL

To get the current value of the sequence, you use the CURRVAL pseudo-column:

```
SELECT
  id_seq.CURRVAL
FROM
  dual;
```

The current value is 30:

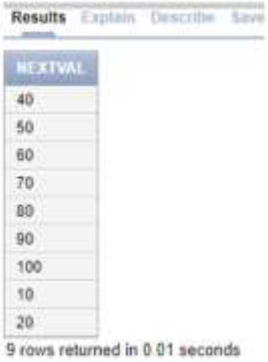


```
NEXTVAL
30
1 rows returned in 0.00 seconds
```

This SELECT statement uses the id_seq.NEXTVAL value repeatedly:

```
SELECT
  id_seq.NEXTVAL
FROM
  dual
CONNECT BY level <= 9;
```

Here is the output:



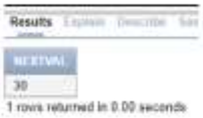
```
NEXTVAL
40
50
60
70
80
90
100
10
20
9 rows returned in 0.01 seconds
```

Practical 4: Creation of Sequence in PL/SQL

Because we set the CYCLE option for the id_seq sequence, the next value of the id_seq will be 10:

```
SELECT id_seq.NEXTVAL FROM dual;
```

And here is the output:



```
NEXTVAL
30
1 rows returned in 0.00 seconds
```

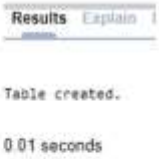
2) Using a sequence in a table column example

Prior Oracle 12c, you can associate a sequence indirectly with a table column only at the insert time.

See the following example.

First, create a new table called tasks:

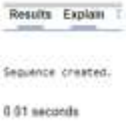
```
CREATE TABLE tasks(
  id NUMBER PRIMARY KEY,
  title VARCHAR2(255) NOT NULL
);
```



```
Table created.

0.01 seconds
```

Second, create a sequence for the id column of the tasks table:

```
CREATE SEQUENCE task_id_seq;
```



```
Sequence created.

0.01 seconds
```

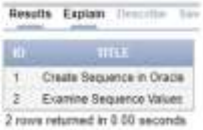
Practical 4: Creation of Sequence in PL/SQL

Third, insert data into the tasks table:
INSERT INTO tasks(id, title)
VALUES(task_id_seq.NEXTVAL, 'Create Sequence in Oracle');

INSERT INTO tasks(id, title)
VALUES(task_id_seq.NEXTVAL, 'Examine Sequence Values');

Finally, query data from the tasks table:

```
SELECT
   id, title
FROM
   tasks;
```



ID	TITLE
1	Create Sequence in Oracle
2	Examine Sequence Values

2 rows returned in 0.00 seconds

In this example, the tasks table has no direct association with the task_id_seq sequence.

3) Using the sequence via the identity column example

From Oracle 12c, you can associate a sequence with a table column via the identity column.

First, drop the tasks table:
DROP TABLE tasks;

Table dropped.

[illegible] seconds

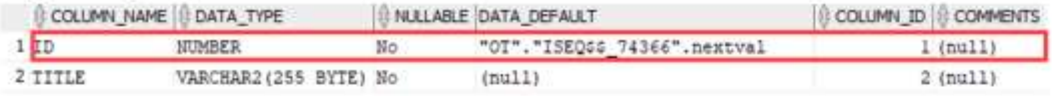
Practical 4: Creation of Sequence in PL/SQL

Second, recreate the tasks table using the identity column for the id column:

```
CREATE TABLE tasks(
  id NUMBER GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  title VARCHAR2(255) NOT NULL
);
```

Behind the scenes, Oracle creates a sequence that associates with the id column of the tasks table.

Because Oracle generated the sequence automatically for the id column, in your Oracle instance, the name of the sequence may be different.



	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	ID	NUMBER	No	"OT"."ISEQ$$_74366".nextval	1	(null)
2	TITLE	VARCHAR2(255 BYTE)	No	(null)	2	(null)

Oracle uses the sys.idnseq$ to store the link between the table and the sequence.

Practical 4: Creation of Sequence in PL/SQL

This query returns the association of the tasks table and ISEQ$$_74366 sequence:

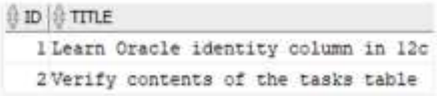
```
SELECT
   a.name AS table_name,
   b.name AS sequence_name
FROM
   sys.idnseq$ c
   JOIN obj$ a ON c.obj# = a.obj#
   JOIN obj$ b ON c.seqobj# = b.obj#
WHERE
   a.name = 'TASKS';
```

Third, insert some rows into the tasks table:

```
INSERT INTO tasks(title)
VALUES('Learn Oracle identity column in 12c');

INSERT INTO tasks(title)
VALUES('Verify contents of the tasks table');
```

Finally, query data from the tasks table:

```
SELECT
   id, title
FROM
   tasks;
```



ID	TITLE
1	Learn Oracle identity column in 12c
2	Verify contents of the tasks table

Practical 5: Create cursor in PL/SQL

1a: Implicit cursor

When an SQL statement is processed, Oracle creates a memory area known as context area. A cursor is a pointer to this context area. It contains all information needed for processing the statement. In PL/SQL, the context area is controlled by Cursor. A cursor contains information on a select statement and the rows of data accessed by it.

A cursor is used to referred to 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

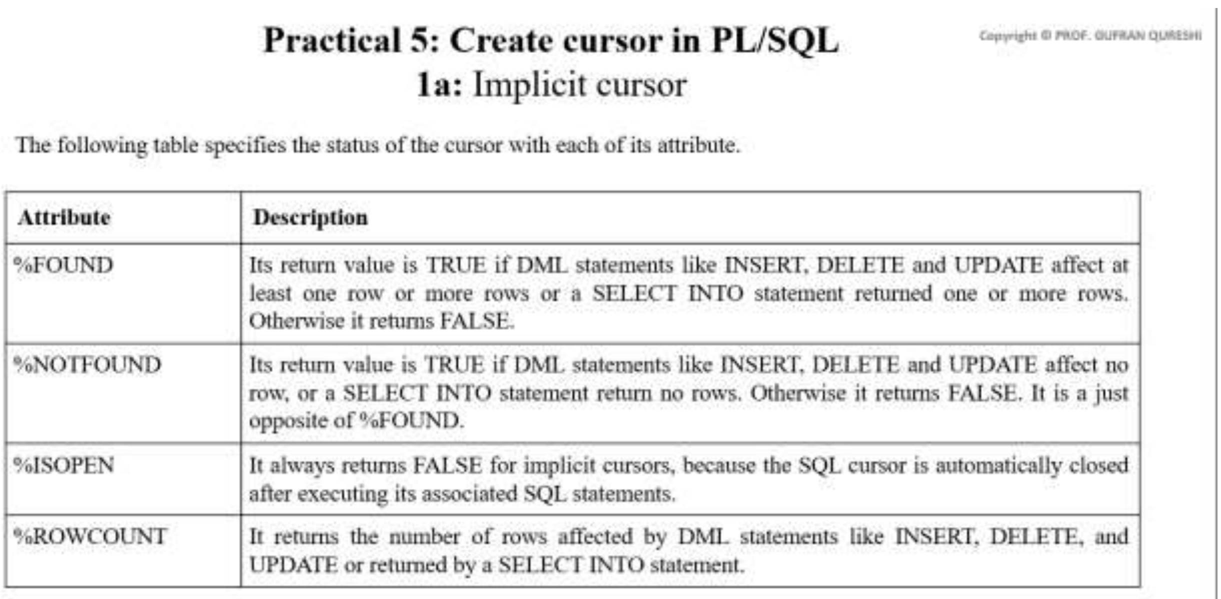
1) PL/SQL Implicit Cursors

The implicit cursors are automatically generated by Oracle while an SQL statement is executed, if you don't use an explicit cursor for the statement.

These are created by default to process the statements when DML statements like INSERT, UPDATE, DELETE etc. are executed.

Orcale provides some attributes known as Implicit cursor's attributes to check the status of DML operations. Some of them are: %FOUND, %NOTFOUND, %ROWCOUNT and %ISOPEN.

For example: When you execute the SQL statements like INSERT, UPDATE, DELETE then the cursor attributes tell whether any rows are affected and how many have been affected. If you run a SELECT INTO statement in PL/SQL block, the implicit cursor attribute can be used to find out whether any row has been returned by the SELECT statement. It will return an error if there no data is selected.

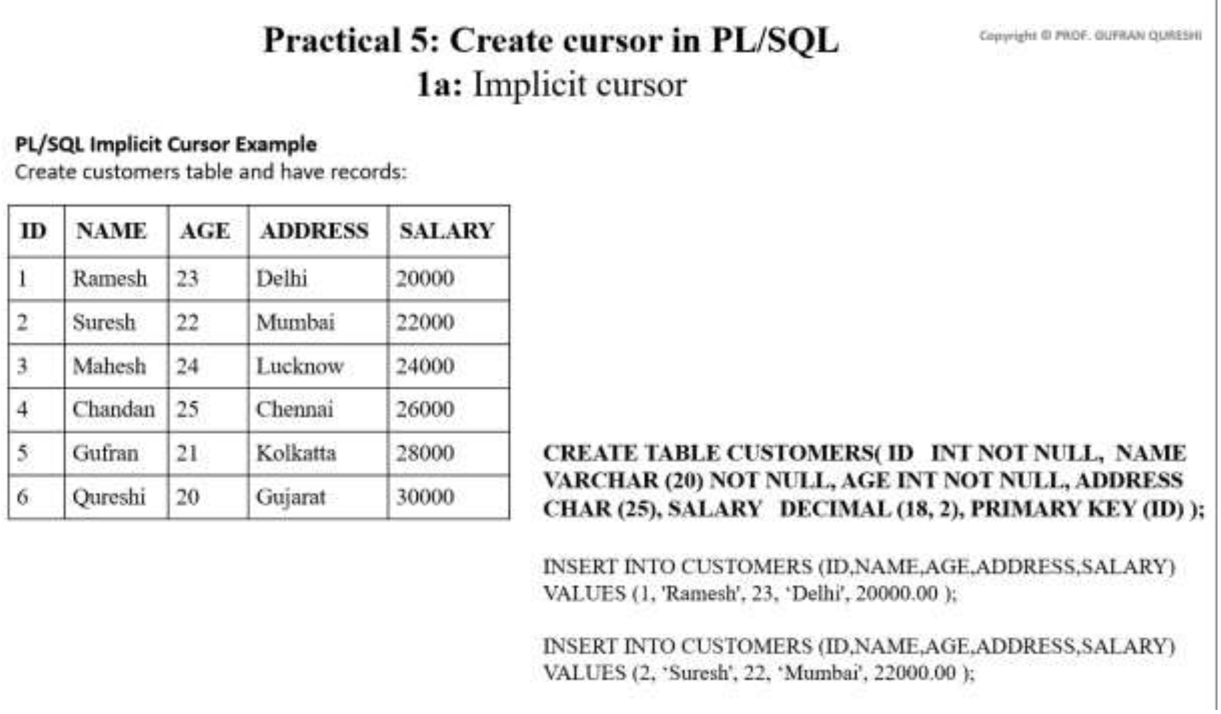


Practical 5: Create cursor in PL/SQL

1a: Implicit cursor

The following table specifies the status of the cursor with each of its attribute.

Attribute	Description
%FOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect at least one row or more rows or a SELECT INTO statement returned one or more rows. Otherwise it returns FALSE.
%NOTFOUND	Its return value is TRUE if DML statements like INSERT, DELETE and UPDATE affect no row, or a SELECT INTO statement return no rows. Otherwise it returns FALSE. It is a just opposite of %FOUND.
%ISOPEN	It always returns FALSE for implicit cursors, because the SQL cursor is automatically closed after executing its associated SQL statements.
%ROWCOUNT	It returns the number of rows affected by DML statements like INSERT, DELETE, and UPDATE or returned by a SELECT INTO statement.



Practical 5: Create cursor in PL/SQL

1a: Implicit cursor

PL/SQL Implicit Cursor Example

Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Delhi	20000
2	Suresh	22	Mumbai	22000
3	Mahesh	24	Lucknow	24000
4	Chandan	25	Chennai	26000
5	Gufran	21	Kolkatta	28000
6	Qureshi	20	Gujarat	30000

```
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) );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 23, ‘Delhi', 20000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, ‘Suresh', 22, ‘Mumbai', 22000.00 );
```

Practical 5: Create cursor in PL/SQL

1a: Implicit cursor

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

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

Output:

```
6 customers updated
PL/SQL procedure successfully completed.
```

Now, if you check the records in customer table, you will find that the rows are updated.

```
select * from customers;
```

Practical 5: Create cursor in PL/SQL

1b: Explicit cursor

The Explicit cursors are defined by the programmers to gain more control over the context area. These cursors 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.

Following is the syntax to create an explicit cursor:
CURSOR cursor_name **IS** select_statement;;

Steps:

1. You must follow these steps while working with an explicit cursor.
2. Declare the cursor to initialize in the memory.
3. Open the cursor to allocate memory.
4. Fetch the cursor to retrieve data.
5. Close the cursor to release allocated memory.

1) Declare the cursor:
It defines the cursor with a name and the associated SELECT statement.
Syntax for explicit cursor decleration

```
CURSOR name IS
SELECT statement;
```

Practical 5: Create cursor in PL/SQL

1b: Explicit cursor

2) Open the cursor:

It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.

```
OPEN cursor_name;
```

3) Fetch the cursor:

It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:

```
FETCH cursor_name INTO variable_list;
```

4) Close the cursor:

It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.

```
Close cursor_name;
```

Practical 5: Create cursor in PL/SQL

1b: Explicit cursor

PL/SQL Explicit Cursor Example

Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Delhi	20000
2	Suresh	22	Mumbai	22000
3	Mahesh	24	Lucknow	24000
4	Chandan	25	Chennai	26000
5	Gufran	21	Kolkatta	28000
6	Qureshi	20	Gujarat	30000

```
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) );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 23, ‘Delhi', 20000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, ‘Suresh', 22, ‘Mumbai', 22000.00 );
```

Practical 5: Create cursor in PL/SQL

1b: Explicit cursor

Let's execute the following program to update the table and increase salary of each customer by 5000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

```
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;
/
```

Output:

```
1 Ramesh  Delhi
2 Suresh  Mumbai
3 Mahesh  Lucknow
4 Chandan  Chennai
5 Gufran  Kolkatta
6 Qureshi  Gujarat
PL/SQL procedure successfully completed.
```

Practical 5: Create cursor in PL/SQL

1c: Parameterized cursor

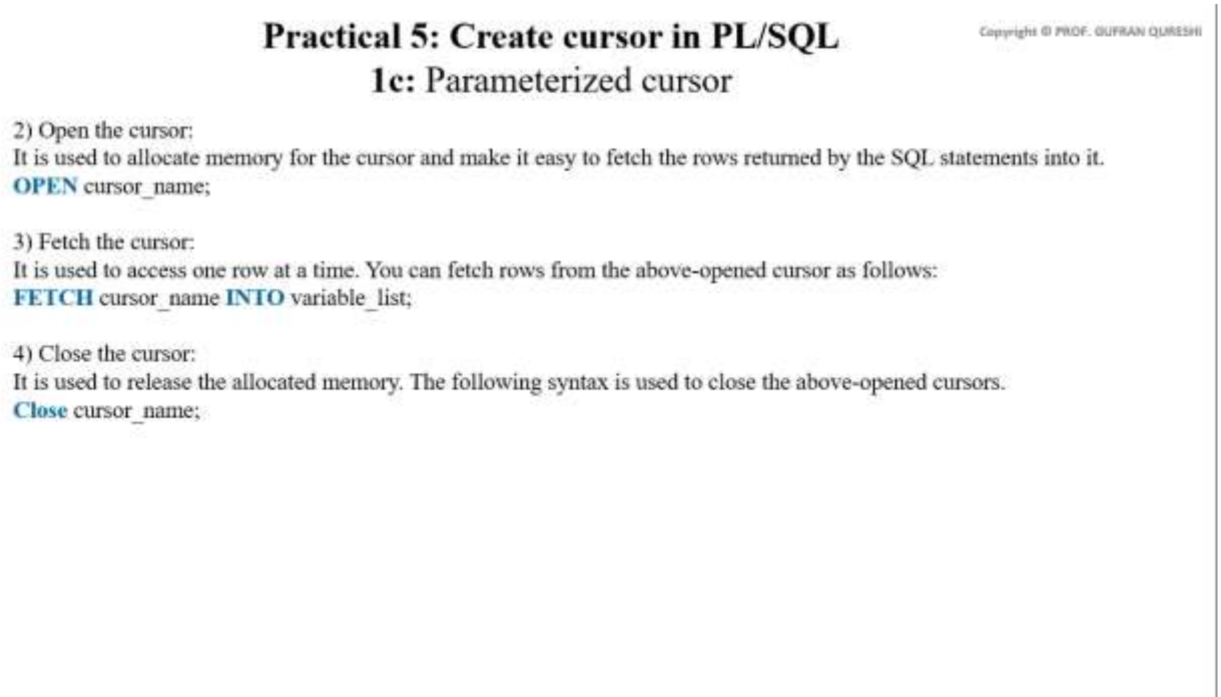
Parameterized cursors are static cursors that can accept passed-in parameter values when they are opened.

Following is the syntax to create an explicit cursor:
CURSOR cursor_name (parameter) **IS** select_statement;;

Steps:
You must follow these steps while working with a parameterized cursor.

1. Declare the parameterized cursor to initialize in the memory.
2. Open the cursor to allocate memory.
3. Fetch the cursor to retrieve data.
4. Close the cursor to release allocated memory.

1) Declare the cursor:
It defines the cursor with a name and the associated SELECT statement.
Syntax for explicit cursor decleration
CURSOR name (parameter) IS
SELECT statement;



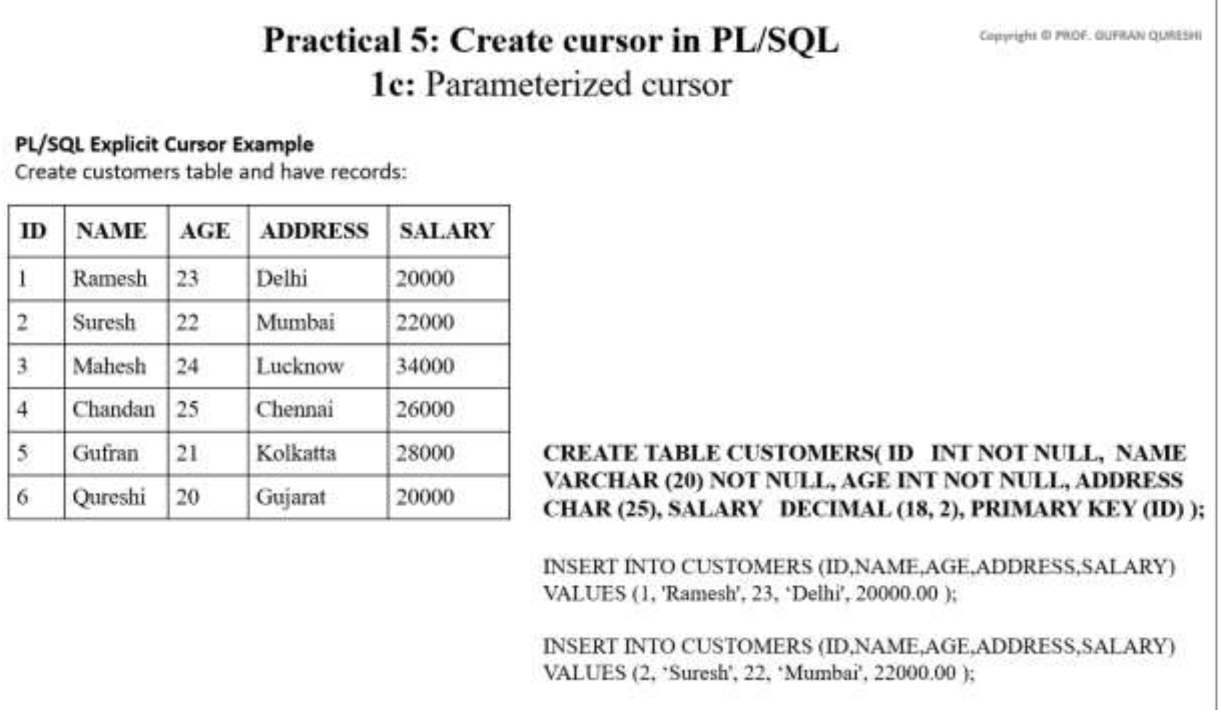
Practical 5: Create cursor in PL/SQL

1c: Parameterized cursor

2) Open the cursor:
It is used to allocate memory for the cursor and make it easy to fetch the rows returned by the SQL statements into it.
OPEN cursor_name;

3) Fetch the cursor:
It is used to access one row at a time. You can fetch rows from the above-opened cursor as follows:
FETCH cursor_name **INTO** variable_list;

4) Close the cursor:
It is used to release the allocated memory. The following syntax is used to close the above-opened cursors.
Close cursor_name;



Practical 5: Create cursor in PL/SQL

1c: Parameterized cursor

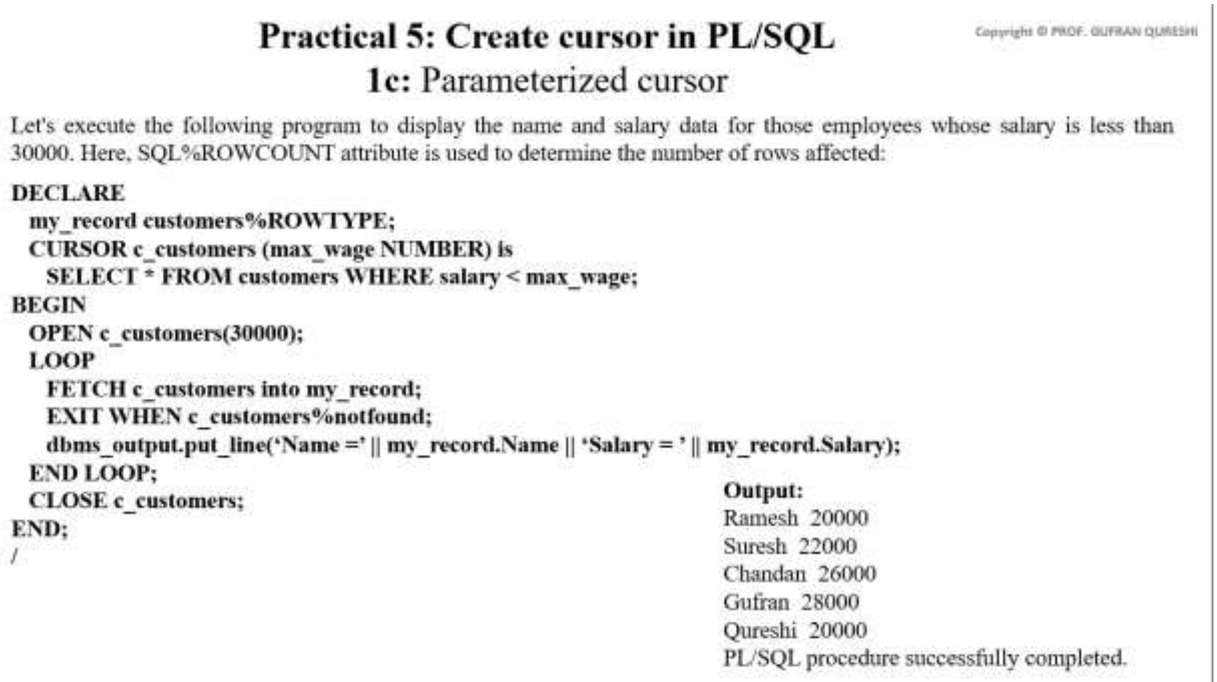
PL/SQL Explicit Cursor Example
Create customers table and have records:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Delhi	20000
2	Suresh	22	Mumbai	22000
3	Mahesh	24	Lucknow	34000
4	Chandan	25	Chennai	26000
5	Gufran	21	Kolkatta	28000
6	Qureshi	20	Gujarat	20000

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));

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (1, 'Ramesh', 23, ‘Delhi', 20000.00);

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY) VALUES (2, ‘Suresh', 22, ‘Mumbai', 22000.00);



Practical 5: Create cursor in PL/SQL

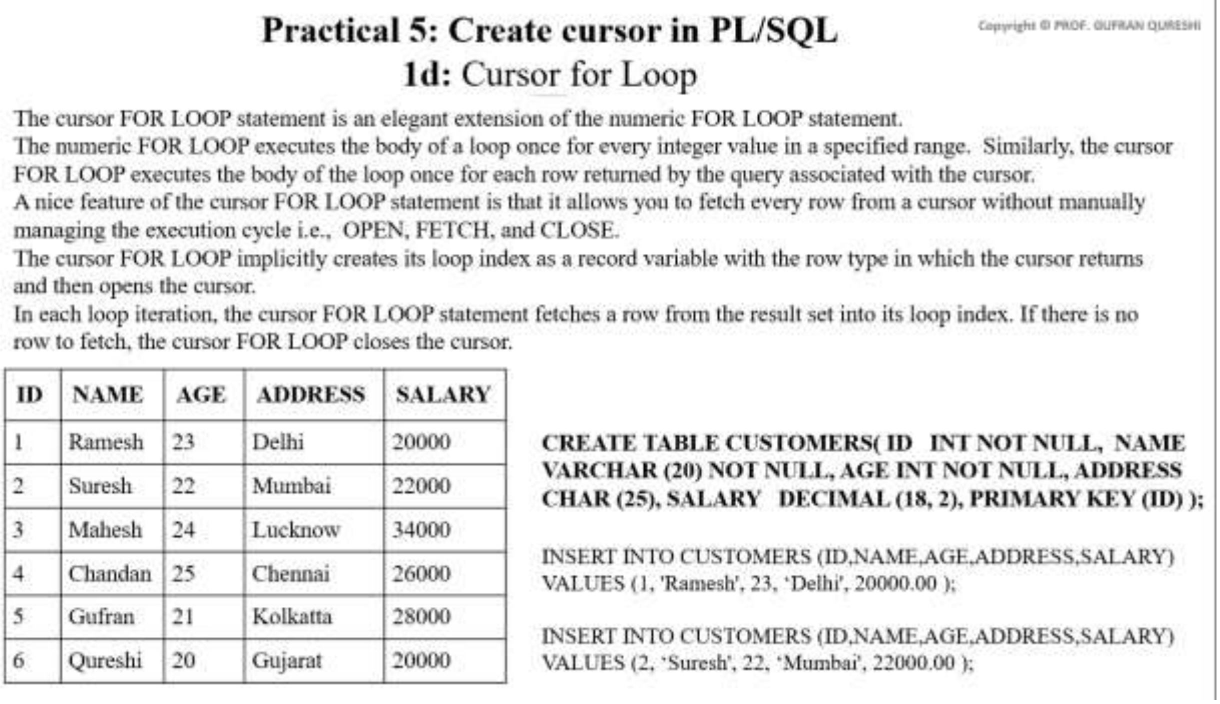
1c: Parameterized cursor

Let's execute the following program to display the name and salary data for those employees whose salary is less than 30000. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

```
DECLARE
  my_record customers%ROWTYPE;
  CURSOR c_customers (max_wage NUMBER) is
    SELECT * FROM customers WHERE salary < max_wage;
BEGIN
  OPEN c_customers(30000);
  LOOP
    FETCH c_customers into my_record;
    EXIT WHEN c_customers%notfound;
    dbms_output.put_line(‘Name =’ || my_record.Name || ‘Salary = ’ || my_record.Salary);
  END LOOP;
  CLOSE c_customers;
END;
/
```

Output:

```
Ramesh  20000
Suresh  22000
Chandan  26000
Gufran  28000
Qureshi  20000
PL/SQL procedure successfully completed.
```



Practical 5: Create cursor in PL/SQL

1d: Cursor for Loop

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.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Delhi	20000
2	Suresh	22	Mumbai	22000
3	Mahesh	24	Lucknow	34000
4	Chandan	25	Chennai	26000
5	Gufran	21	Kolkatta	28000
6	Qureshi	20	Gujarat	20000

```
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) );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 23, ‘Delhi', 20000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, ‘Suresh', 22, ‘Mumbai', 22000.00 );
```

Practical 5: Create cursor in PL/SQL

1d: Cursor for Loop

Let's execute the following program to print id, name and address customers using for loop. Here, SQL%ROWCOUNT attribute is used to determine the number of rows affected:

```
DECLARE
 CURSOR c_customers
 IS
  SELECT
    Name, Salary
  FROM
    Customers
  ORDER BY
    Salary DESC;
BEGIN
 FOR r_customers IN c_customers
 LOOP
  dbms_output.put_line( r_customers.Name || ': $' ||  r_customers.Salary);
 END LOOP;
END;
```

Practical 6: Creation of Procedures in PL/SQL

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.

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 in 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

IN Parameter:

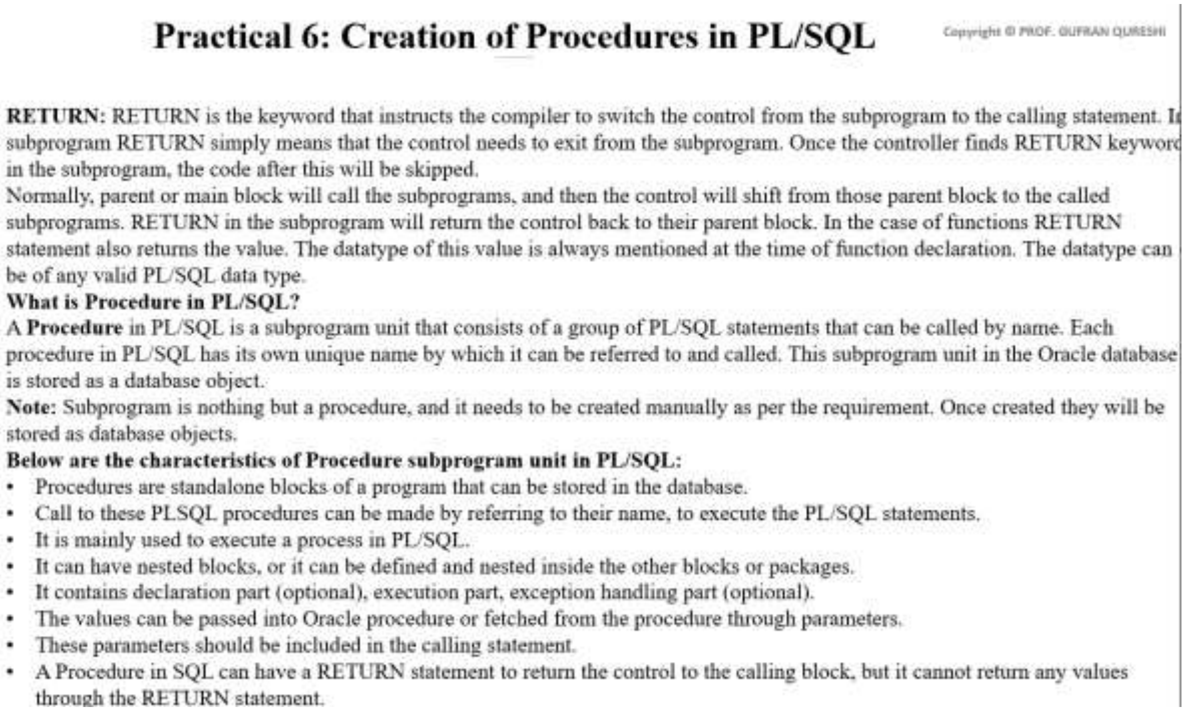
- This parameter is used for giving input to the subprograms.
- It is a read-only variable inside the subprograms. Their values cannot be changed inside the subprogram.
- In the calling statement, these parameters can be a variable or a literal value or an expression, for example, it could be the arithmetic 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.



Practical 6: Creation of Procedures in PL/SQL

RETURN: RETURN is the keyword that instructs the compiler to switch the control from the subprogram to the calling statement. I subprogram RETURN simply means that the control needs to exit from the subprogram. Once the controller finds RETURN keywor 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.

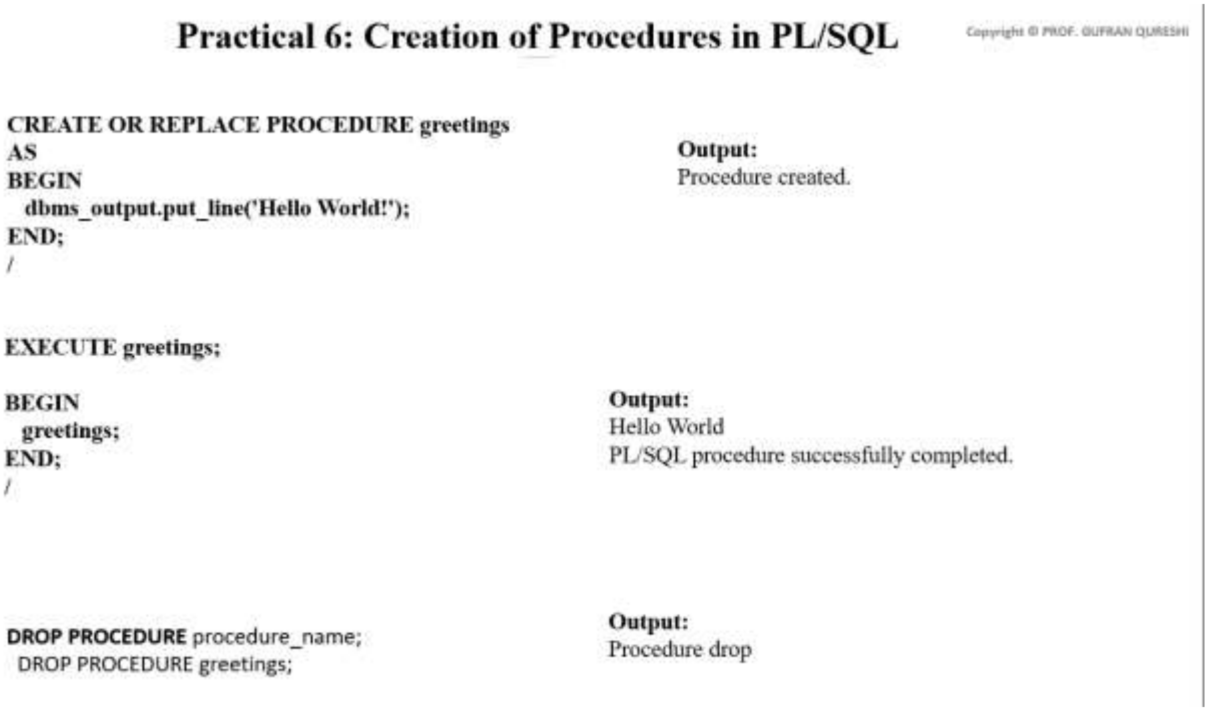
What is Procedure in PL/SQL?

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

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.

Below are the characteristics of Procedure subprogram unit in PL/SQL:

- Procedures are standalone blocks of a program that can be stored in the database.
- Call to these PLSQL 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 Oracle procedure or fetched from the procedure through parameters.
- These parameters should be included in the calling statement.
- A Procedure in SQL can have a RETURN statement to return the control to the calling block, but it cannot return any values through the RETURN statement.



Practical 6: Creation of Procedures in PL/SQL

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

Output:
Procedure created.

```
EXECUTE greetings;
```

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

Output:
Hello World
PL/SQL procedure successfully completed.

```
DROP PROCEDURE procedure_name;
 DROP PROCEDURE greetings;
```

Output:
Procedure drop

Practical 6: Creation of Procedures in PL/SQL

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

Output:

```
Minimum of (23, 45) : 23
PL/SQL procedure successfully completed.
```

Practical 6: Creation of Procedures in PL/SQL

```
DECLARE
  a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
 x := x * x;
END;
BEGIN
  a:= 23;
  squareNum(a);
  dbms_output.put_line(' Square of (23): ' || a);
END;
/
```

Output:

```
Square of (23): 529
PL/SQL procedure successfully completed.
```

Practical 6: Creation of Procedures in PL/SQL

```
create table user(id number(10) primary key,
name varchar2(100));
```

Output:
Table created.

```
create or replace procedure "INSERTUSER"
(id IN NUMBER, name IN VARCHAR2)
is
begin
insert into user values(id,name);
end;
/
```

Output:
Procedure created.

```
BEGIN
  insertuser(1,'Prof');
  insertuser(2,'Gufran');
  insertuser(1,'Qureshi');
  dbms_output.put_line('record inserted successfully');
END;
/
```

Output:
record inserted successfully.

```
DROP PROCEDURE procedure_name;
DROP PROCEDURE INSERTUSER;
```

Output:
Procedure drop

Practical 7: Functions in PL/SQL

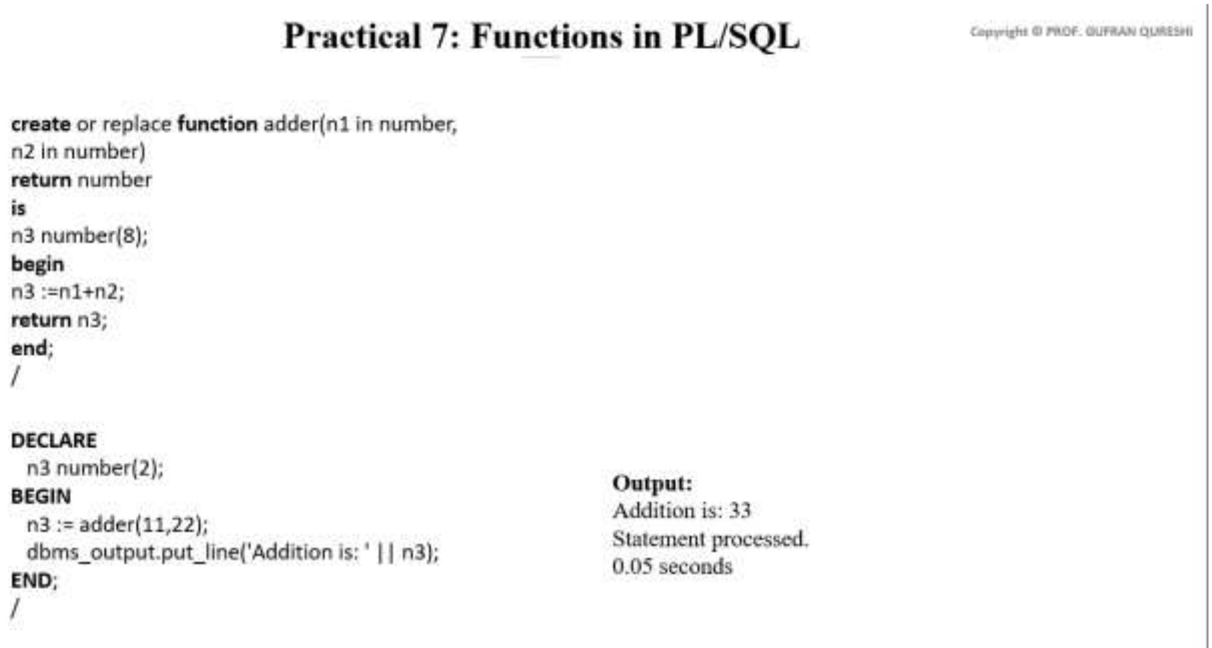
The PL/SQL Function is very similar to PL/SQL Procedure. The main difference between procedure and a function is, a function must always return a value, and on the other hand a procedure may or may not return a value. Except this, all the other things of PL/SQL procedure are true for PL/SQL function too.

Syntax:

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

Here:

Function_name: specifies the name of the function.
[OR REPLACE] option allows modifying an existing function.
The **optional parameter list** contains name, mode and types of the parameters.
IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
The function must contain a return statement.
RETURN clause specifies that 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.

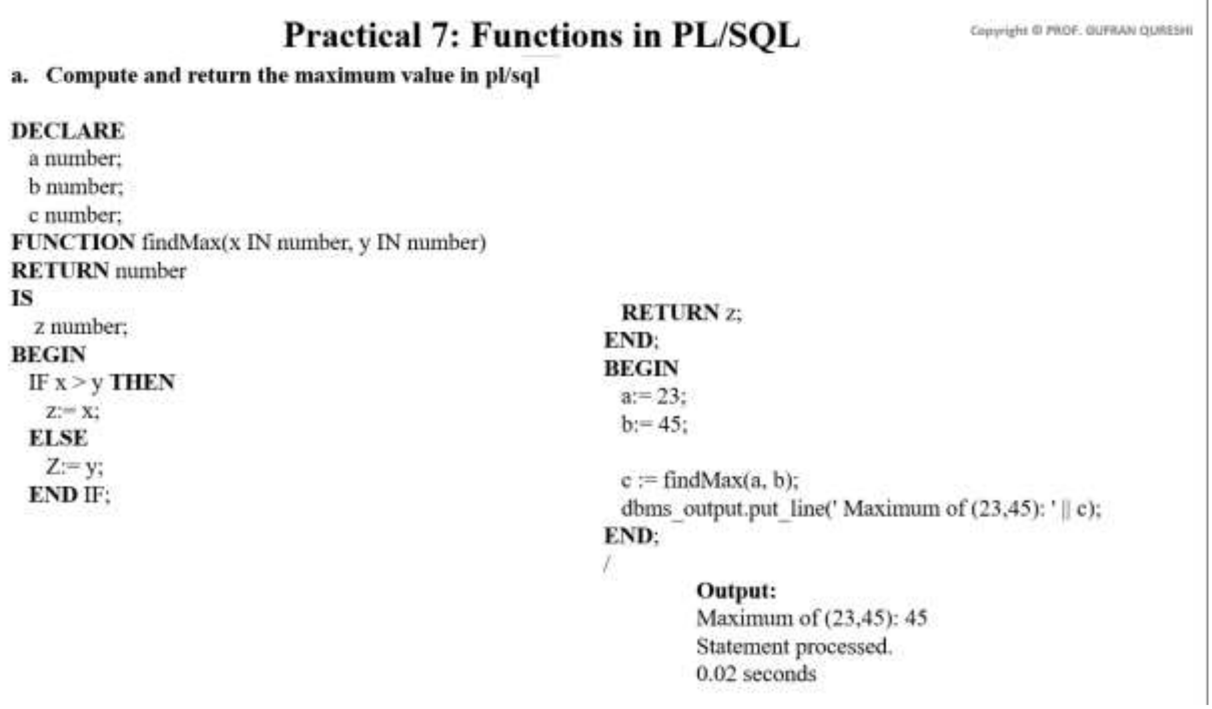


Practical 7: Functions in PL/SQL

```
create or replace function adder(n1 in number,
n2 in number)
return number
is
n3 number(8);
begin
n3 :=n1+n2;
return n3;
end;
/

DECLARE
  n3 number(2);
BEGIN
  n3 := adder(11,22);
  dbms_output.put_line('Addition is: ' || n3);
END;
/
```

Output:

```
Addition is: 33
Statement processed.
0.05 seconds
```



Practical 7: Functions in PL/SQL

a. Compute and return the maximum value in pl/sql

```
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;
/
```

Output:

```
Maximum of (23,45): 45
Statement processed.
0.02 seconds
```

Practical 7: Functions in PL/SQL

b. Compute factorial of given number

```
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 of '|| num || ' is ' || factorial);
END;
/
```

Output:

```
Factorial of 6 is 720
Statement processed.
0.02 seconds
```

Practical 8: Creation of Trigger

Trigger is invoked by Oracle engine automatically whenever a specified event occurs.Trigger is stored into database and invoked repeatedly, when specific condition match.

Triggers are stored programs, which are automatically executed or fired when some event occurs.

Triggers are written to be executed in response to any of the following events.

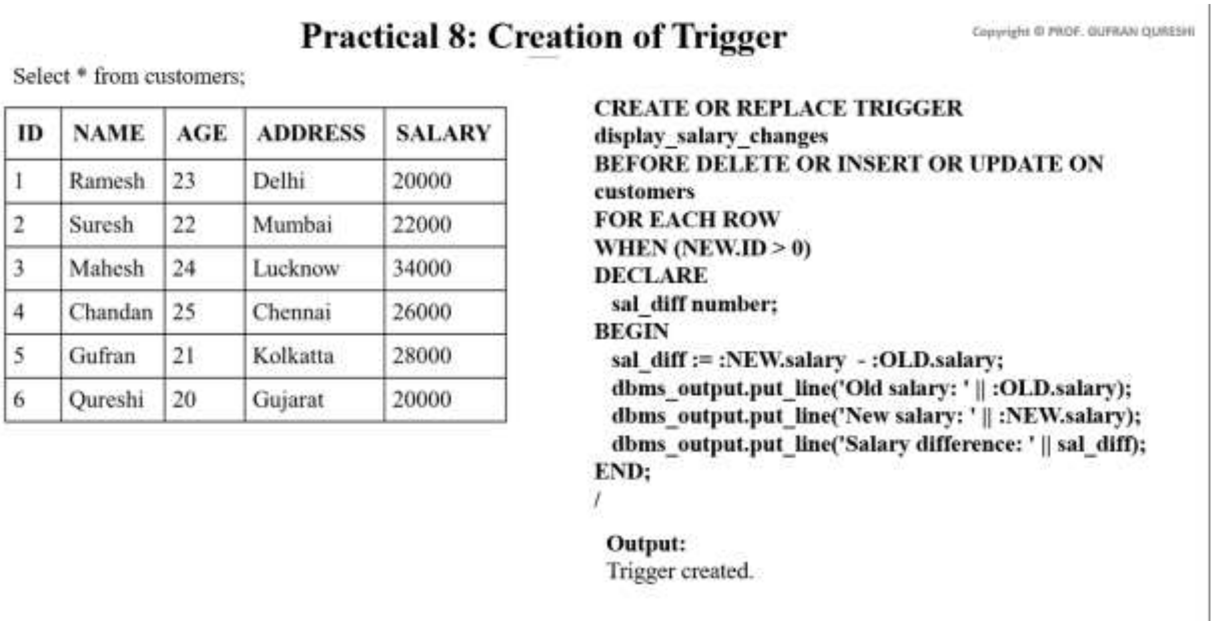
A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
A database definition (DDL) statement (CREATE, ALTER, or DROP).
A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).

Advantages of Triggers

These are the following advantages of Triggers:
Trigger generates some derived column values automatically
Enforces referential integrity
Event logging and storing information on table access
Auditing
Synchronous replication of tables
Imposing security authorizations
Preventing invalid transactions

Syntax:

```
TRIGGER trigger_name
  triggering_event
  [ trigger_restriction ]
BEGIN
  triggered_action;
END;
```



Practical 8: Creation of Trigger

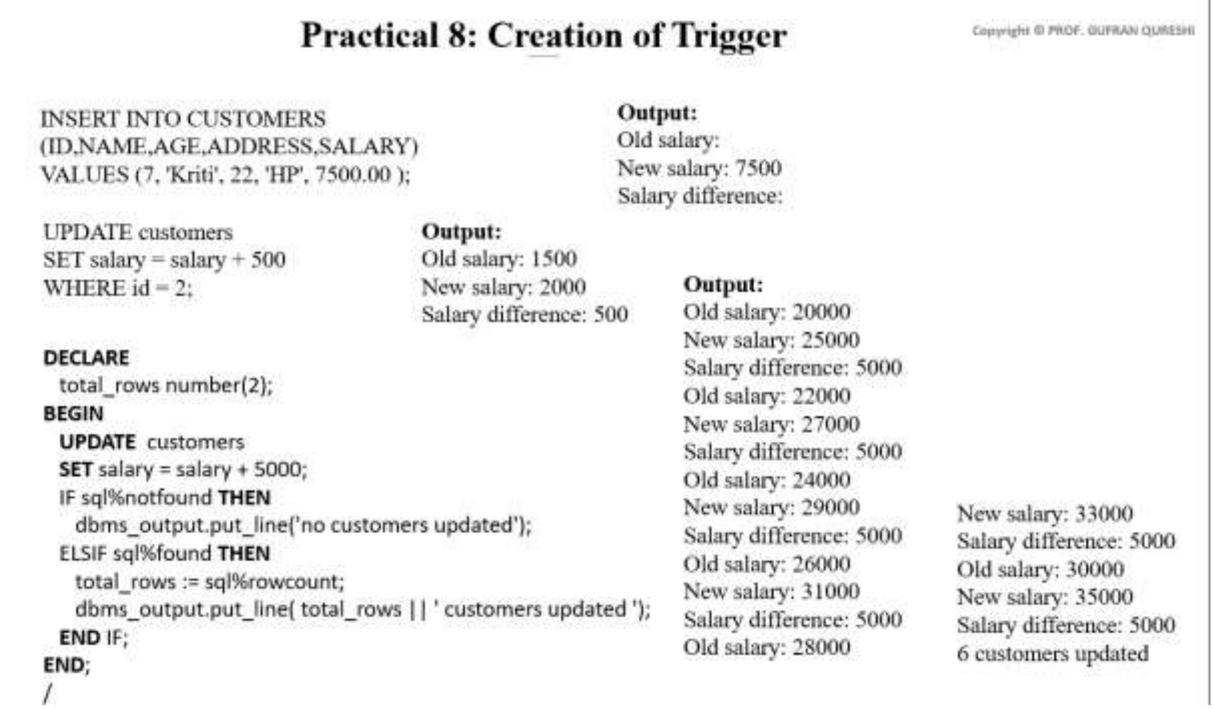
Select * from customers;

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Delhi	20000
2	Suresh	22	Mumbai	22000
3	Mahesh	24	Lucknow	34000
4	Chandan	25	Chennai	26000
5	Gufran	21	Kolkatta	28000
6	Qureshi	20	Gujarat	20000

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

Output:

```
Trigger created.
```



Practical 8: Creation of Trigger

```
INSERT INTO CUSTOMERS
(ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Kriti', 22, 'HP', 7500.00 );
```

Output:

```
Old salary:
New salary: 7500
Salary difference:
```

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

Output:

```
Old salary: 1500
New salary: 2000
Salary difference: 500
```

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

Output:

```
Old salary: 20000
New salary: 25000
Salary difference: 5000
Old salary: 22000
New salary: 27000
Salary difference: 5000
Old salary: 24000
New salary: 29000
Salary difference: 5000
Old salary: 26000
New salary: 31000
Salary difference: 5000
Old salary: 28000
New salary: 33000
Salary difference: 5000
Old salary: 30000
New salary: 35000
Salary difference: 5000
6 customers updated
```

Practical 9: Handling Exception

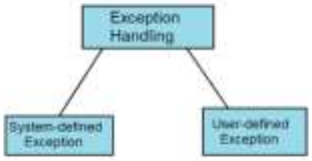
Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions. Exceptions can be caught and handled by the program.

PL/SQL provides us the exception block which raises the exception thus helping the programmer to find out the fault and resolve it.

There are two types of exceptions defined in PL/SQL

a. **System defined exceptions.**

b. **User defined exception.**



Syntax:

```
WHEN exception THEN
   statement;
DECLARE
declarations section;
BEGIN
executable command(s);
EXCEPTION
WHEN exception1 THEN
statement1;
WHEN exception2 THEN
statement2;
[WHEN others THEN]
/* default exception handling code */
END;
```

Practical 9: Handling Exception

9a: 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.

1. **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:

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	23	Delhi	20000
2	Suresh	22	Mumbai	22000
3	Mahesh	24	Lucknow	24000
4	Chandan	25	Chennai	26000
5	Gufran	21	Kolkatta	28000
6	Qureshi	20	Gujarat	30000

```
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) );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 23, ‘Delhi', 20000.00 );

INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (2, ‘Suresh', 22, ‘Mumbai', 22000.00 );
```

Practical 9: Handling Exception

9a: System defined exceptions

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

```
DECLARE
temp varchar(20);

BEGIN
SELECT id into temp from customers where name='Shahrukh';

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

Output:

```
ERROR
there is no name as Shahrukh in customers table
```

Practical 9: Handling Exception

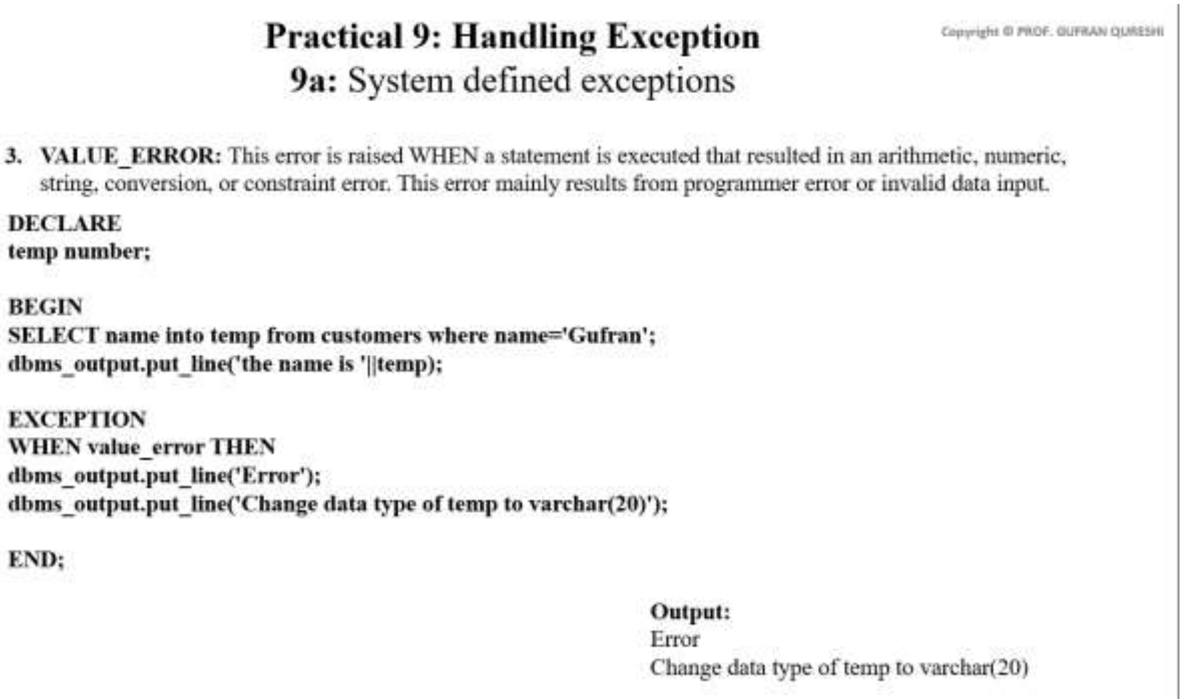
9a: System defined exceptions

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 name into temp from customers;
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
```



Practical 9: Handling Exception

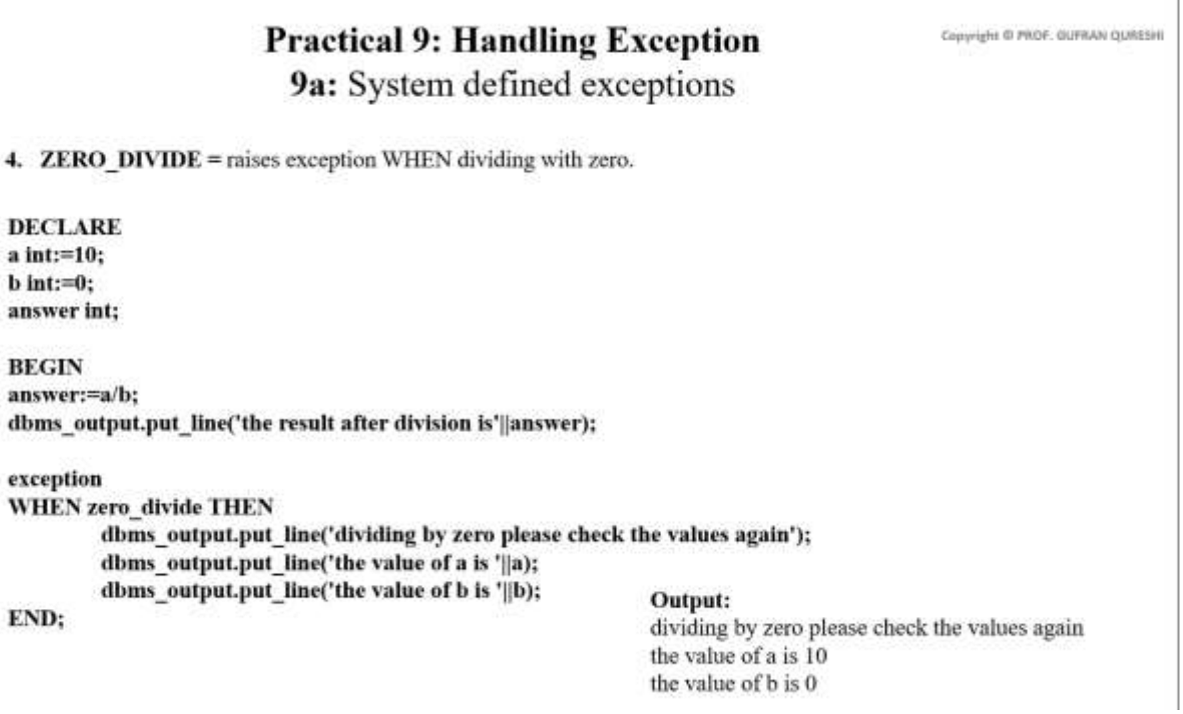
9a: System defined exceptions

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 name into temp from customers where name='Gufran';
dbms_output.put_line('the 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)
```



Practical 9: Handling Exception

9a: System defined exceptions

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
```

Practical 9: Handling Exception

9a: System defined exceptions

2. **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.

```
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 to Unnamed System Exceptions');
END;
```

Output:

```
1
4
9
16
25
36
Welcome to Unnamed System Exceptions
```

Practical 9: Handling Exception

9b: 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 by zero.
- If result is greater than or equal to 1 means y is less than or equal to x.

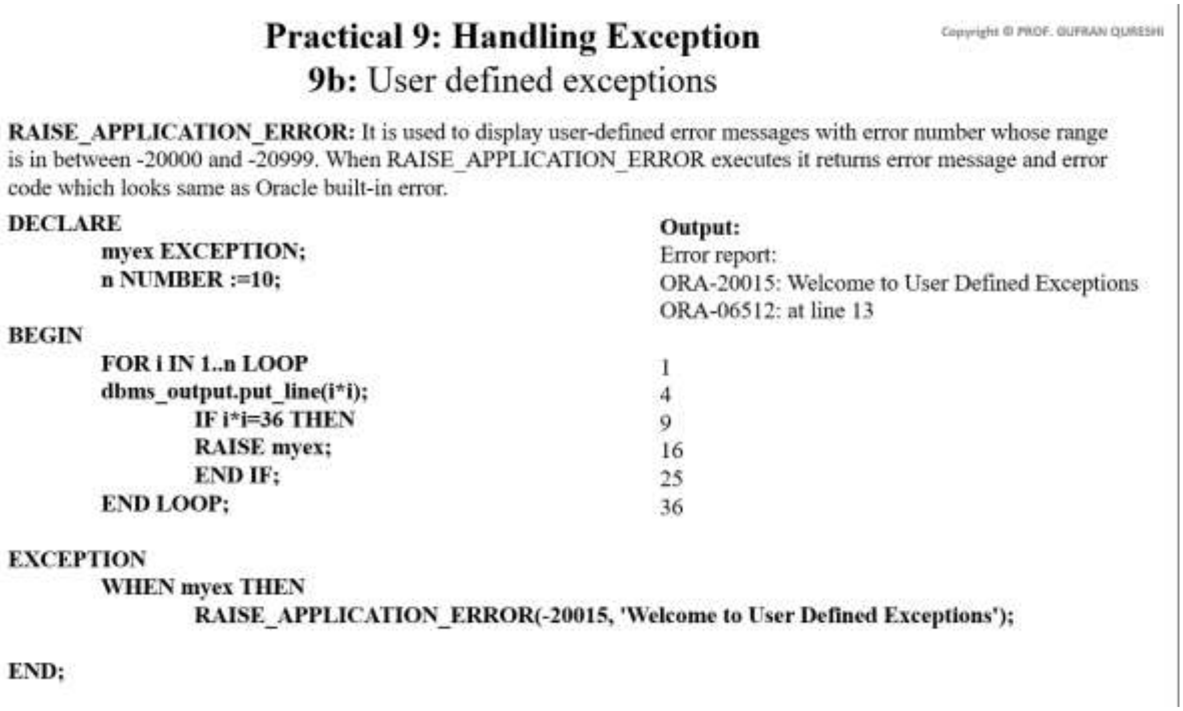
```
DECLARE
x int:=&x; /*taking value at run time*/
y int:=&y;
div_r float;
exp1 EXCEPTION;
exp2 EXCEPTION;
BEGIN
IF y=0 then
        raise exp1;
ELSEIF y > x then
        raise exp2;
ELSE
        div_r:= x / y;
        dbms_output.put_line('the result is '||div_r);
END IF;
```

```
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;
```

Output:

```
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:<.em>
Error
y is greater than x please check the input
```



Practical 9: Handling Exception

9b: User defined exceptions

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.

```
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 to User Defined Exceptions');

END;
```

Output:

```
Error report:
ORA-20015: Welcome to User Defined Exceptions
ORA-06512: at line 13

1
4
9
16
25
36
```