EXERCISE – 9

PL/SQL BASIC PROGRAMS

In addition to SQL commands, PL/SQL can also process data using flow of statements. The flow of control statements are classified into the following categories.

- Conditional control -Branching
- Iterative control - looping

- Sequential control

BRANCHING in PL/SQL:

Sequence of statements can be executed on satisfying certain condition . If statements are being used and differe forms of if are:

1) Simple IF
2) If-Else
3) Nested IF

SIMPLE IF:

Syntax

IF condition THEN statement1; statement2;

END IF;

IF-THEN-ELSE STATEMENT:

Syntax:

IF condition THEN statement1;

ELSE statement2;

END IF;

ELSIF STATEMENTS:

Syntax:

IF condition1 THEN statement1;

ELSIF condition2 THEN statement2;

ELSIF condition3 THEN statement3;

ELSE statementn;

END IF;

NESTED IF :

Syntax:

IF condition THEN statement1;

ELSE

  IF condition THEN statement2;

  ELSE statement3;

  END IF;

END IF;

ELSE statement3;

END IF;

SELECTION IN PL/SQL(Sequential Controls)

SIMPLE CASE

Syntax:

CASE SELECTOR

WHEN Expr1 THEN statement1;

WHEN Expr2 THEN statement2;

:

ELSE Statement n;
END CASE;

SEARCHED CASE:

CASE
WHEN searchcondition1 THEN statement1;
WHEN searchcondition2 THEN statement2;
:
:
ELSE statement n;
END CASE;

ITERATIONS IN PL/SQL

Sequence of statements can be executed any number of times using loop construct.
It is broadly classified into:

- Simple Loop
- For Loop

- While Loop

SIMPLE LOOP

Syntax:

LOOP statement1;

EXIT [WHEN Condition];

END LOOP;

Example:

Declare

A number:=10;

Begin

Loop

```
a := a+25;
exit when a=250; end loop;
dbms_output.put_line(to_char(a));
 end;
/
```

WHILE LOOP

Syntax

WHILE condition LOOP statement1; statement2;

END LOOP;

Example:

```
Declare
number:=0; j number:=0; begin

While i<=100 Loop j := j+i;
:= i+2;
end loop;
dbms_output.put_line('the value of j is' ||j); end;
/
```

FOR LOOP

Syntax:

FOR counter IN [REVERSE] LowerBound..UpperBound

LOOP
statement1;
statement2;
END LOOP;

Example:

Begin
For I in 1..2 Loop
Update emp set field = value where condition End loop;
End;
/

LAB-9 :

SIMPLE PL/SQL PROGRAMS

AIM : To execute simple PL/SQL programs.

1. Write a PL/SQL code to set the sales commission to 10%, if the sales revenue is greater than 200,000. Else, the sales commission is set to 5%.

CODE :

```
SQL> DECLARE
  2      sales_revenue NUMBER(8,2) := 100000;
  3      sales_commission NUMBER(8,2) := 0;
  4  BEGIN
  5      IF sales_revenue > 200000 THEN
  6          sales_commission := sales_revenue * 0.1;
  7      ELSE
  8          sales_commission := sales_revenue * 0.05;
  9      dbms_output.put_line('Sales Revenue    = ' || sales_revenue);
 10      dbms_output.put_line('Sales Commission = ' || sales_commission);
 11      END IF;
 12  END;
 13  /
```

OUTPUT :

```
Sales Revenue    = 100000
Sales Commission = 5000

PL/SQL procedure successfully completed.
```

2. Use PL/SQL CASE statement where if monthly_value is equal to or less than 4000, then income_level will be set to 'Low Income'. If monthly_value is equal to or less than 5000, then income_level will be set to 'Avg Income'. Otherwise, income_levelwill be set to 'High Income'.

CODE :

```
SQL> DECLARE
  2      monthly_value NUMBER := 6000;
  3      income_level VARCHAR(20);
  4  BEGIN
  5      CASE
  6          WHEN monthly_value <= 4000 THEN
  7              income_level := 'low income';
  8          WHEN monthly_value <= 5000 THEN
  9              income_level := 'average income';
 10          ELSE
 11              income_level := 'high income';
 12      END CASE;
 13      dbms_output.put_line('Monthly salary = ' || monthly_value);
 14      dbms_output.put_line('Income level   = ' || income_level);
 15  END;
 16  /
```

OUTPUT :

```
Monthly salary = 6000
Income level   = high income

PL/SQL procedure successfully completed.
```

3. Write a PL/SQL program to add numbers in a given range.

CODE :

```
SQL> DECLARE
  2      n INT := &n;
  3      sum1 INT := 0;
  4      i INT;
  5  BEGIN
  6      FOR i in 1..n
  7          LOOP
  8              sum1 := sum1 + i;
  9          END LOOP;
 10      dbms_output.put_line('Sum = ' || sum1);
 11  END;
 12  /
```

OUTPUT :

```
Enter value for n: 10
old   2:      n INT := &n;
new   2:      n INT := 10;
Sum = 55

PL/SQL procedure successfully completed.
```

4. Write a PL/SQL program by using SELECT INTO statement to get the name of a customer based on the customer id, which is the primary key of the customers table.

CREATE TABLE :

```
SQL> create table CUSTOMER
  2  (
  3       customer_id number PRIMARY KEY,
  4       customer_name varchar(20),
  5       address varchar(30),
  6       mobile INTEGER
  7  );
```

```
SQL> INSERT INTO CUSTOMER(customer_id, customer_name, address, mobile)
  2  WITH input AS
  3      ( SELECT 1, 'Ramcharan', 'Telangana', 9090897867  FROM DUAL
  4        UNION ALL
  5        SELECT 2, 'Prabhas', 'Andhra Pradesh', 8768757890 FROM DUAL
  6        UNION ALL
  7        SELECT 3, 'Allu Arjun', 'Tamil Nadu', 8659090897 FROM DUAL
  8        UNION ALL
  9        SELECT 4, 'Yash', 'Tamil Nadu', 8659090897 FROM DUAL
 10      )
 11  SELECT * FROM input;
```

```
SQL> SELECT * FROM CUSTOMER;
```

```
Table created.
```

```
4 rows created.
```

```
CUSTOMER_ID CUSTOMER_NAME        ADDRESS                        MOBILE
----------- -------------------- ------------------------------ ----------
          1 Ramcharan            Telangana                      9090897867
          2 Prabhas              Andhra Pradesh                 8768757890
          3 Allu Arjun           Tamil Nadu                     8659090897
          4 Yash                 Tamil Nadu                     8659090897
```

CODE :

```
SQL> DECLARE
  2      id CUSTOMER.customer_id%TYPE;
  3      name CUSTOMER.customer_name%TYPE;
  4  BEGIN
  5      id := &id;
  6      SELECT customer_name INTO name FROM CUSTOMER WHERE customer_id = id;
  7      dbms_output.put_line('Name of the customer with id = ' || id || ' is ' || name || '.');
  8  END;
  9  /
```

OUTPUT :

```
Enter value for id: 2
old   5:      id := &id;
new   5:      id := 2;
Name of the customer with id = 2 is Prabhas.

PL/SQL procedure successfully completed.
```

RESULT :

Hence successfully executed simple PL/SQL programs.

EXERCISE – 10

PROCEDURES IN PL/SQL

A **subprogram** is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design'. A subprogram can be invoked by another subprogram or program which is called the **calling program**.

A subprogram can be created −

- At the schema level
- Inside a package
- Inside a PL/SQL block
-

At the schema level, subprogram is a **standalone subprogram**. It is created with the CREATE PROCEDURE or the CREATE FUNCTION statement. It is stored in the database and can be deleted with the DROP PROCEDURE or DROP FUNCTION statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database and can be deleted only when the package is deleted with the DROP PACKAGE statement. We will discuss packages in the chapter **'PL/SQL - Packages'**.

PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL provides two kinds of subprograms –

- **Functions** − These subprograms return a single value; mainly used to compute and return a value.

- **Procedures** − These subprograms do not return a value directly; mainly used to perform an action.

Parts of a PL/SQL Subprogram

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

S.No	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

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

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

Where,

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

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

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

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

```
Procedure created.
```

Executing a Standalone Procedure

A standalone procedure can be called in two ways −

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

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

```
EXECUTE greetings;
```

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

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

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

The above call will display −

```
Hello World

PL/SQL procedure successfully completed.
```

Deleting a Standalone Procedure

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

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

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

```
DROP PROCEDURE greetings;
```

Parameter Modes in PL/SQL Subprograms

The following table lists out the parameter modes in PL/SQL subprograms −

S.No	Parameter Mode & Description
1	**IN** An IN parameter lets you pass a value to the subprogram. **It is a read-only parameter**. Inside the subprogram, an IN parameter acts like a constant. It cannot be assigned a value. You can pass a constant, literal, initialized variable, or expression as an IN parameter. You can also initialize it to a default value;

	however, in that case, it is omitted from the subprogram call. **It is the default mode of parameter passing. Parameters are passed by reference**.
2	**OUT** An OUT parameter returns a value to the calling program. Inside the subprogram, an OUT parameter acts like a variable. You can change its value and reference the value after assigning it. **The actual parameter must be variable and it is passed by value**.
3	**IN OUT** An **IN OUT** parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and the value can be read. The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression. Formal parameter must be assigned a value. **Actual parameter is passed by value.**

IN & OUT Mode Example 1

This program finds the minimum of two values. Here, the procedure takes two numbers using the IN mode and returns their minimum using the OUT parameters.

```
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;
/
```

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

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

PL/SQL procedure successfully completed.
```

IN & OUT Mode Example 2

This procedure computes the square of value of a passed value. This example shows how we can use the same parameter to accept a value and then return another result.

```
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;
/
```

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

```
Square of (23): 529

PL/SQL procedure successfully completed.
```

Methods for Passing Parameters

Actual parameters can be passed in three ways −

- Positional notation
- Named notation
- Mixed notation

Positional Notation

In positional notation, you can call the procedure as −

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

In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

Named Notation

In named notation, the actual parameter is associated with the formal parameter using the **arrow symbol (=>)**. The procedure call will be like the following −

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

Mixed Notation

In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

The following call is legal −

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

However, this is not legal:

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

LAB-10 :

PROCEDURE

AIM : To implement procedure in PL/SQL code.

1. Write a PL/SQL block to get the salary of the employee who has empno=7369 and update his salary as specified below

- if his/her salary < 2500, then increase salary by 25%
- otherwise if salary lies between 2500 and 5000, then increase salary by 20%
- otherwise increase salary by adding commission amount to the salary.

```
Declare
        Salary number(5);
Begin
        Select sal into salary from emp where empno=7369;
        -- complete remaining statements


End;
/
```

CREATE TABLE :

```
CREATE TABLE EMP
(
    "emp_id" INT,
    "first_name" VARCHAR2(20),
    "last_name" VARCHAR2(20),
    "job_id" VARCHAR2(20),
    "mobile" NUMBER(10),
    "salary" NUMBER(10,2),
    "dept_id" INT,
    PRIMARY KEY ("emp_id")
);

INSERT INTO EMP
WITH INPUT AS
(   SELECT 12, 'Anil', 'Ravipudi', 'SALES_MAN', 9678909686, 10000, 10 FROM DUAL
    UNION ALL
    SELECT 2556, 'Siva', 'Koratala', 'HR', 9567890897, 30000, 80 FROM DUAL
    UNION ALL
    SELECT 2365, 'Prasanth', 'Neel', 'IT', 9878907834, 40000, 24 FROM DUAL
    UNION ALL
    SELECT 7678, 'Surender', 'Reddy', 'MANAGER', 8679856787, 50000, 3 FROM DUAL
    UNION ALL
    SELECT 7369, 'Sreenu', 'Boyapati', 'MANAGER', 8675848483, 56000, 80 FROM DUAL
) SELECT * FROM input;
```

```
select * from emp;
```

```
Table created.


5 row(s) inserted.
```

emp_id	first_name	last_name	job_id	mobile	salary	dept_id
12	Anil	Ravipudi	SALES_MAN	9678909686	10000	10
2556	Siva	Koratala	HR	9567890897	30000	80
2365	Prasanth	Neel	IT	9878907834	40000	24
7678	Surender	Reddy	MANAGER	8679856787	50000	3
7369	Sreenu	Boyapati	MANAGER	8675848483	56000	80

Download CSV

5 rows selected.

CODE :

```
CREATE PROCEDURE update_salary("empid" IN emp."emp_id"%TYPE)
AS
    "sal" emp."salary"%TYPE;
    "incr_per" NUMBER(3,2);
    "commission" NUMBER(3,2) := 0.10;
BEGIN
    select "salary" into "sal" from emp where "emp_id"="empid";
    IF "sal"<25000 THEN
        "incr_per" := 0.25;
    ELSIF "sal">=25000 AND "sal"<=50000 THEN
        "incr_per" := 0.20;
    ELSE
        "incr_per" := "commission";
    END IF;
    UPDATE emp
        SET "salary" = "sal" + "sal"*"incr_per" WHERE "emp_id" = "empid";
    dbms_output.put_line('Salary updated');
END;
```

```
DECLARE
    "empid" emp."emp_id"%type := 7369;
BEGIN
    UPDATE_SALARY("empid");
END;
```

```
select "salary" from emp where "emp_id"=7369;
```

OUTPUT :

```
Procedure created.
```

```
Procedure created.
Salary updated
```

salary
61600

Download CSV

2. Write a PL/SQL Block to modify the department name of the department 71 if it is not 'HRD'.

```
Declare
        deptname dept.dname%type;
Begin       -- complete the block


End;
/
```

CREATE TABLE :

```
CREATE TABLE DEPT
(
    "dept_id" INT,
    "dept_name" VARCHAR2(20),
    "loc_id" INT,
    PRIMARY KEY ("dept_id")
);
```

```
INSERT INTO DEPT
WITH input AS
(   SELECT 3, 'Marketing', 1450 FROM DUAL
    UNION ALL
    SELECT 10, 'Software', 1700 FROM DUAL
    UNION ALL
    SELECT 24, 'Management', 1500 FROM DUAL
    UNION ALL
    SELECT 71, 'Sales', 452 FROM DUAL
    UNION ALL
    SELECT 48, 'Administration', 1700 FROM DUAL
) SELECT * FROM input;

SELECT * FROM DEPT;
```

```
Table created.


5 row(s) inserted.
```

DEPT_ID	DEPT_NAME	LOC_ID
3	Marketing	1450
10	Software	1700
24	Management	1500
71	Sales	452
48	Administration	1700

Download CSV

5 rows selected.

CODE :

```
CREATE PROCEDURE modify_dept_name("deptid" IN dept."dept_id"%type)
AS
    "deptname" dept."dept_name"%type;
BEGIN
    select "dept_name" into "deptname" from dept where "dept_id"="deptid";
    IF "deptname" != 'HRD' THEN
        UPDATE DEPT
            SET "dept_name"='HRD' WHERE "dept_id" = "deptid";
        dbms_output.put_line('Modified department name to HRD');
    ELSE
        dbms_output.put_line('Not Modified department name ( already HRD )');
    END IF;
END;
```

```
DECLARE
    "deptid" dept."dept_id"%type := 71;
BEGIN
    MODIFY_DEPT_NAME("deptid");
END;

```

```
SELECT "dept_name" FROM DEPT WHERE "dept_id"=71;
```

OUTPUT :

```
Procedure created.
```

```
Procedure created.
Modified department name to HRD
```

dept_name
HRD

Download CSV

RESULT :

Hence successfully implemented procedures in PL/SQL codes.

EXERCISE – 11

FUNCTIONS IN PL/SQL

A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

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, which we had created in the PL/SQL Variables chapter −

```
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 −

```
Function created.
```

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block −

```
DECLARE
   c number(2);
BEGIN
```

```
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces 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.
```

LAB-11 :

FUNCTIONS

AIM : To implement functions in PL/SQL code.

1. Write a PL/SQL Function to find factorial of a given number.

CODE :

```
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:= 8;
   factorial := fact(num);
   dbms_output.put_line('Factorial of '|| num || ' is ' || factorial);
END;
```

OUTPUT :

```
Statement processed.
Factorial of 8 is 40320
```

2. Write a PL/SQL Function that computes and returns the maximum of two values.

CODE :

```
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:= 10;
   b:= 40;
   c := findMax(a, b);
   dbms_output.put_line('Maximum value of 10 and 40 : ' || c);
END;

```

OUTPUT :

```
Statement processed.
Maximum value of 10 and 40 : 40
```

RESULT :

Hence successfully implemented functions in PL/SQL code.

EXERCISE - 12

CURSORS

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. There are two types of cursors −

- Implicit cursors
- Explicit cursors

Implicit Cursors

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, 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 −

S.No	Attribute & Description
1	**%FOUND** Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
2	**%NOTFOUND**

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

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 and used in the previous chapters.

```
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 customerS.No.ame%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.
```

LAB-12 :

CURSOR

AIM : To implement cursors in PL/SQL code.

CREATE TABLE :

```
CREATE TABLE EMP
(
    emp_id INT,
    first_name VARCHAR2(20),
    last_name VARCHAR2(20),
    job_id VARCHAR2(20),
    mobile NUMBER(10),
    salary NUMBER(10,2),
    dept_id INT,
    PRIMARY KEY (emp_id)
);

INSERT INTO EMP
WITH INPUT AS
(   SELECT 12, 'Anil', 'Ravipudi', 'SALES_MAN', 9678909686, 10000, 10 FROM DUAL
    UNION ALL
    SELECT 2556, 'Siva', 'Koratala', 'HR', 9567890897, 30000, 20 FROM DUAL
    UNION ALL
    SELECT 2365, 'Prasanth', 'Neel', 'IT', 9878907834, 40000, 24 FROM DUAL
    UNION ALL
    SELECT 7678, 'Surender', 'Reddy', 'MANAGER', 8679856787, 50000, 3 FROM DUAL
    UNION ALL
    SELECT 7369, 'Sreenu', 'Boyapati', 'MANAGER', 8675848483, 56000, 20 FROM DUAL
) SELECT * FROM input;

SELECT * FROM EMP;
```

```
Table created.


5 row(s) inserted.
```

EMP_ID	FIRST_NAME	LAST_NAME	JOB_ID	MOBILE	SALARY	DEPT_ID
12	Anil	Ravipudi	SALES_MAN	9678909686	10000	10
2556	Siva	Koratala	HR	9567890897	30000	20
2365	Prasanth	Neel	IT	9878907834	40000	24
7678	Surender	Reddy	MANAGER	8679856787	50000	3
7369	Sreenu	Boyapati	MANAGER	8675848483	56000	20

Download CSV

5 rows selected.

1. Write a PL/SQL Block, to update salaries of all the employees who work in deptno 20 by 15%. If none of the employee's salary are updated display a message 'None of the salaries were updated'. Otherwise display the total number of employee who got salary updated.

CODE :

```
DECLARE
    num number(5);
BEGIN
    UPDATE EMP
        SET salary = salary + salary*0.15 where dept_id=20;
    IF SQL%NOTFOUND THEN
        dbms_output.put_line('None of the salaries were updated');
    ELSIF SQL%FOUND THEN
        num := SQL%ROWCOUNT;
        dbms_output.put_line('Salaries for ' || num || ' employees are updated');
    END IF;
END;
```

```
SELECT * FROM EMP;
```

OUTPUT :

```
Table created.
Salaries for 2 employees are updated
```

EMP_ID	FIRST_NAME	LAST_NAME	JOB_ID	MOBILE	SALARY	DEPT_ID
12	Anil	Ravipudi	SALES_MAN	9678909686	10000	10
2556	Siva	Koratala	HR	9567890897	34500	20
2365	Prasanth	Neel	IT	9878907834	40000	24
7678	Surender	Reddy	MANAGER	8679856787	50000	3
7369	Sreenu	Boyapati	MANAGER	8675848483	64400	20

```
Download CSV
5 rows selected.
```

2. Create a table emp_grade with columns empno & grade. Write PL/SQL block to insert values into the table emp_grade by processing the emp table with the following constraints.

- If sal <= 1400 then grade is 'C'
- Else if sal between 1401 and 2000 then the grade is 'B'.
- Else the grade is 'A'.

CODE :

```
CREATE TABLE EMP_GRADE(emp_id INT, grade CHAR(1));

DECLARE
    CURSOR cur IS SELECT emp_id, salary FROM EMP;
    empid EMP.emp_id%TYPE;
    sal EMP.salary%TYPE;
BEGIN
    OPEN cur;
    IF cur%ISOPEN THEN
        LOOP
            FETCH cur INTO empid, sal;
            IF cur%NOTFOUND THEN
                EXIT;
            END IF;
            IF sal <= 20000 THEN
                INSERT INTO EMP_GRADE VALUES(empid,'C');
            ELSIF sal BETWEEN 20001 AND 40000 THEN
                INSERT INTO EMP_GRADE VALUES(empid, 'B');
            ELSE
                INSERT INTO EMP_GRADE VALUES(empid, 'A');
            END IF;
        END LOOP;
    ELSE
        OPEN cur;
    END IF;
END;
```

```
SELECT * FROM EMP_GRADE;
```

OUTPUT :

```
Table created.


Table created.
```

EMP_ID	GRADE
12	C
2556	B
2365	B
7678	A
7369	A

Download CSV

5 rows selected.

RESULT :

Hence successfully implemented cursors in PL/SQL code.

EXERCISE - 13

TRIGGERS IN PL/SQL

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, 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).

Triggers can be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

Triggers can be written for the following purposes −

- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

The syntax for creating a trigger is −

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

Where,

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

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters −

```
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 creates a **row-level** trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values −

```
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;
/
```

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

```
Trigger created.
```

The following points need to be considered here −

- OLD and NEW references are not available for table-level triggers, rather you can use them for record-level triggers.
- If you want to query the table in the same trigger, then you should use the AFTER keyword, because triggers can query the table or change it again only after the initial changes are applied and the table is back in a consistent state.
- The above trigger has been written in such a way that it will fire before any DELETE or INSERT or UPDATE operation on the table, but you can write your trigger on a single or multiple operations, for example BEFORE DELETE, which will fire whenever a record will be deleted using the DELETE operation on the table.

Triggering a Trigger

Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table −

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

When a record is created in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

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

Because this is a new record, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table −

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

When a record is updated in the CUSTOMERS table, the above create trigger, **display_salary_changes** will be fired and it will display the following result −

```
Old salary: 1500
New salary: 2000
```

LAB-13 :

TRIGGERS

AIM : To implement triggers in PL/SQL code.

```
SQL> CREATE TABLE EMP1
  2  (
  3       "emp_id" INT,
  4       "emp_name" VARCHAR2(20),
  5       "job_id" VARCHAR2(20),
  6       "mobile" NUMBER(10),
  7       "salary" NUMBER(10,2),
  8       "dob" DATE,
  9       "dept_id" INT,
 10       PRIMARY KEY ("emp_id")
 11  );

Table created.
```

1. Create a Trigger to check if the entered age is valid or not.

CODE :

```
SQL> CREATE OR REPLACE TRIGGER age_validation
  2  BEFORE INSERT on EMP1
  3  FOR EACH ROW
  4
  5  DECLARE
  6  emp_age number;
  7
  8  BEGIN
  9      -- Finding employee age by date of birth
 10      SELECT MONTHS_BETWEEN(TO_DATE(sysdate,'DD-MON-YYYY'), TO_DATE(:new."dob",'DD-MON-YYYY'))/12
 11     INTO EMP_AGE FROM DUAL;
 12
 13     -- Check whether employee age is greater than 18 or not
 14      IF (EMP_AGE < 18) THEN
 15        RAISE_APPLICATION_ERROR(-20000,'Employee age must be greater than or equal to 18.');
 16      END IF;
 17
 18      -- Allow only past date of death
 19      IF(:new."dob" > sysdate) THEN
 20        RAISE_APPLICATION_ERROR(-20000,'Date of birth can not be Future date.');
 21      END IF;
 22  END;
 23  /
```

OUTPUT :

```
Trigger created.
```

```
SQL> INSERT INTO EMP1
  2      VALUES(12, 'Anil', 'SALES_MAN', 9678909686, 10000, DATE '2015-09-11', 10);
INSERT INTO EMP1
            *
ERROR at line 1:
ORA-20000: Employee age must be greater than or equal to 18.
ORA-06512: at "RA1911026010115.AGE_VALIDATION", line 11
ORA-04088: error during execution of trigger 'RA1911026010115.AGE_VALIDATION'
```

2. Create a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on that table.

```
SQL> create table CUSTOMER
  2  (
  3      "customer_id" number PRIMARY KEY,
  4      "customer_name" varchar(20),
  5      "address" varchar(30),
  6      "mobile" INTEGER
  7  );

Table created.
```

CODE :

```
SQL> CREATE OR REPLACE TRIGGER customer_update
  2  BEFORE DELETE OR INSERT OR UPDATE ON customer
  3  FOR EACH ROW
  4  WHEN (NEW."customer_id" > 0)
  5
  6  BEGIN
  7      dbms_output.put_line('Changes to CUSTOMER table triggered');
  8  END;
  9  /
```

```
SQL> INSERT INTO CUSTOMER
  2  WITH input AS
  3      ( SELECT 1, 'Ramcharan', 'Telangana', 9090897867  FROM DUAL
  4        UNION ALL
  5        SELECT 2, 'Prabhas', 'Andhra Pradesh', 8768757890 FROM DUAL
  6        UNION ALL
  7        SELECT 3, 'Allu Arjun', 'Tamil Nadu', 8659090897 FROM DUAL
  8        UNION ALL
  9        SELECT 4, 'Yash', 'Tamil Nadu', 8659090897 FROM DUAL
 10      )
 11  SELECT * FROM input;
```

OUTPUT :

```
Trigger created.
```

```
Changes to CUSTOMER table triggered
Changes to CUSTOMER table triggered
Changes to CUSTOMER table triggered
Changes to CUSTOMER table triggered

4 rows created.
```

RESULT :

Hence successfully implemented triggers in PL/SQL code.

EXERCISE - 14

EXCEPTION HANDLING IN PL/SQL

In PL/SQL a warning or error condition is called an exception. Exceptions can be internally defined (by the runtime system) or user-defined. Examples of internally defined exceptions include *division by zero* and *out of memory*.

Predefined Exceptions

CURSOR_ALREADY_OPEN is raised if you try to OPEN an already open cursor.

DUP_VAL_ON_INDEX is raised if you try to store duplicate values in a database column that is constrained by a unique index.

INVALID_CURSOR is raised if you try an illegal cursor operation. For example, if you try to CLOSE an unopened cursor.

INVALID_NUMBER is raised in a SQL statement if the conversion of a character string to a number fails.

LOGIN_DENIED is raised if you try logging on to ORACLE with an invalid username/password.

NO_DATA_FOUND is raised if a SELECT INTO statement returns no rows or if you reference an uninitialized row in a PL/SQL table.

NOT_LOGGED_ON is raised if your PL/SQL program issues a database call without being logged on to ORACLE. **PROGRAM_ERROR** is raised if PL/SQL has an internal problem.

STORAGE_ERROR is raised if PL/SQL runs out of memory or if memory is corrupted.

TIMEOUT_ON_RESOURCE is raised if a timeout occurs while ORACLE is waiting for a resource.

TOO_MANY_ROWS is raised if a SELECT INTO statement returns more than one row.

VALUE_ERROR is raised if an arithmetic, conversion, truncation, or constraint error occurs.

ZERO_DIVIDE is raised if you try to divide a number by zero.

Handling Raised Exception

Syntax :

...

EXCEPTION

WHEN ... THEN

- handle the error differently

WHEN ... OR ... THEN

- handle the error differently

...

WHEN OTHERS THEN

- handle the error differently

END;

1) QB 1 Handling **NO_DATA_FOUND** and **ZERO_DIVIDE** Exceptions

```
Declare
n1 number;
n2 number;
Begin
n2 := &n2;
Select sal into n1 from emp where empno=7654;
n1 := n1/n2;
Exception
when zero_divide then
dbms_output.put_line('Zero Divide Error !');
when no_data_found then
dbms_output.put_line('No such Row in EMP table');
when others then
```

dbms_output.put_line(‘Unknown exception’);

end;

User Defined Exception

Unlike predefined exceptions, user-defined exceptions must be declared and must be raised explicitly by RAISE statements. Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the keyword EXCEPTION.

Exception Declaration

Ex.

DECLARE

past_due EXCEPTION;

acct_num NUMBER(5);

BEGIN

…

Exceptions and variable declarations are similar. But remember, an exception is an error condition, not an object. Unlike variables, exceptions cannot appear in assignment statements or SQL statements.

Syntax.

Exception-name **Exception;**

Using Raise statement

User-defined exceptions must be raised explicitly by RAISE statements.

Syntax

RAISE ***exception-name;***

Q2) Write PL/SQL block to raise ‘out-of-balance’ exception if balance fall below 100.

DECLARE

out_of_balance EXCEPTION;

```
bal NUMBER;
BEGIN
IF bal < 100 THEN
RAISE out_of_stock;
END IF;
.
EXCEPTION
WHEN out_of_balance THEN
dbms_output.put_line('Low balance. Unable to do Transactions');
END;
```

Raise_Application_Error

This is a procedure to issue user-defined error messages from a stored subprogram or database trigger.

Syntax : **raise_application_error(error_number, error_message);**

where error_number is a negative integer in the range -20000..-20999 and error_message is a character string up to 512 bytes in length.

Ex. ….

```
IF salary is NULL THEN
raise_application_error(-20101, 'Salary is missing');
….
```

LAB-14 :

EXCEPTION HANDLING

AIM : To implement exception handling in PL/SQL code.

CREATE TABLE :

```
SQL> create table CUSTOMER
  2  (
  3       customer_id number PRIMARY KEY,
  4       customer_name varchar(20),
  5       address varchar(30),
  6       mobile INTEGER
  7  );
```

```
SQL> INSERT INTO CUSTOMER(customer_id, customer_name, address, mobile)
  2  WITH input AS
  3      ( SELECT 1, 'Ramcharan', 'Telangana', 9090897867  FROM DUAL
  4        UNION ALL
  5        SELECT 2, 'Prabhas', 'Andhra Pradesh', 8768757890 FROM DUAL
  6        UNION ALL
  7        SELECT 3, 'Allu Arjun', 'Tamil Nadu', 8659090897 FROM DUAL
  8        UNION ALL
  9        SELECT 4, 'Yash', 'Tamil Nadu', 8659090897 FROM DUAL
 10      )
 11  SELECT * FROM input;
```

```
SQL> SELECT * FROM CUSTOMER;
```

```
Table created.
```

```
4 rows created.
```

```
CUSTOMER_ID CUSTOMER_NAME        ADDRESS                         MOBILE
----------- -------------------- ------------------------------ ----------
          1 Ramcharan            Telangana                      9090897867
          2 Prabhas              Andhra Pradesh                 8768757890
          3 Allu Arjun           Tamil Nadu                     8659090897
          4 Yash                 Tamil Nadu                     8659090897
```

1. Write a PL/SQL program that accepts a customer id as an input and returns the customer name using exception handling.

CODE :

```
SQL> DECLARE
  2     c_id CUSTOMER.customer_id%type := &c_id;
  3     c_name CUSTOMER.customer_name%type;
  4
  5  BEGIN
  6      SELECT customer_name INTO  c_name FROM CUSTOMER WHERE customer_id = c_id;
  7      dbms_output.put_line('Name: '||  c_name);
  8      EXCEPTION
  9          WHEN no_data_found THEN
 10              dbms_output.put_line('No such customer!');
 11          WHEN others THEN
 12              dbms_output.put_line('Error!');
 13  END;
 14  /
```

OUTPUT :

```
Enter value for c_id: 4
old   2:     c_id CUSTOMER.customer_id%type := &c_id;
new   2:     c_id CUSTOMER.customer_id%type := 4;
Name: Yash

PL/SQL procedure successfully completed.
```

RESULT :

Hence successfully implemented exception handling in PL/SQL code.