**Experiment No.-5-6**

**Aim** :The Run-time errors arise from design faults, coding mistakes, hardware failures, and many other sources. Although you cannot anticipate all possible errors, you can plan to handle certain kinds of errors meaningful to your PL/SQL program. With PL/SQL mechanism called exception handling design "bulletproof” program so that it can continue operating in the presence of errors.

**What is Exception Handling in PL/SQL?**

An exception occurs when the PL/SQL engine encounters an instruction which it cannot execute due to an error that occurs at run-time. These errors will not be captured at the time of compilation and hence these needed to handle only at the run-time.

For example, if PL/SQL engine receives an instruction to divide any number by '0', then the PL/SQL engine will throw it as an exception. The exception is only raised at the run-time by the PL/SQL engine.

Exceptions will stop the program from executing further, so to avoid such condition, they need to be captured and handled separately. This process is called as Exception-Handling, in which the programmer handles the exception that can occur at the run time.

**Exception-Handling Syntax:**

Exceptions are handled at the block, level, i.e., once if any exception occurs in any block then the control will come out of execution part of that block. The exception will then be handled at the exception handling part of that block. After handling the exception, it is not possible to resend control back to the execution section of that block

BEGIN

<execution block>

.

.

EXCEPTION

WHEN <exceptionl\_name>

THEN

<Exception handling code for the “exception 1 \_name’' >

WHEN OTHERS

THEN

<Default exception handling code for all exceptions > END;

**Syntax Explanation:**

* In the above syntax, the exception-handling block contains series of WHEN condition to handle the exception.
* Each WHEN condition is followed by the exception name which is expected to be raised at the run time.
* When any exception is raised at runtime, then the PL/SQL engine will look in the exception handling part for that particular exception. It will start from the first 'WHEN' clause and, sequentially it will search.
* If it found the exception handling for the exception which has been raised, then it will execute that particular handling code part.
* After executing the exception, part control will go out of the current block.
* If none of the 'WHEN' clause is present for the exception which has been raised, then PL/SQL engine will execute the 'WHEN OTHERS' part (if present). This is common for all the exception.
* Only one exception part can be executed for a block at run-time. After executing it, the controller will skip the remaining exception handling part and will go out of the current block.

**Types of Exception:**

There are two types of Exceptions in Pl/SQL.

1. Predefined Exceptions
2. User-defined Exception

**Predefined Exceptions:**

Oracle has predefined some common exception. These exceptions have a unique exception name and error number. These exceptions are already defined in the 'STANDARD' package in Oracle. In code, we can directly use these predefined exception name to handle them.

There are pre-defined exceptions:

|  |  |  |  |
| --- | --- | --- | --- |
| EXCEPTIONS | | DESCRITION | |
|  | |  | |
| CURSOR\_ALREADY\_OPEN | | Trying to open a cursor which is | |
|  | | already opened | |
| INVALID\_CURSOR | | Illegal cursor operations like closing | |
|  | | an unopened cursor | |
| NO\_DATA\_FOUND | | When 'SELECT' statement that | |
|  | | contains INTO clause fetches no | |
|  | | rows. | |
| TOO\_MANY\_ROWS | | When a 'SELECT' statement with | |
|  | | INTO clause returns more than one | |
|  | | row | |
| ZERO\_DIVIDE | | Dividing a number by '0' | |
|  | |  | |
| VALUE\_ERROR | | Arithmetic or size constraint error | |
|  | | (eg: assigning a value to a variable | |
|  | | that is larger than the variable size) | |
| DUP\_VAL\_ON\_INDEX | | Storing a duplicate value in a | |
|  | | database column that is a | |
|  | | constrained by unique index | |
|  |  | |  | |  |
|  |  | |  | |  |
|  |  | |  | |  |
|  | LOGIN\_DENIED | | It is raised when a program attempts | |  |
|  |  | | to log on to the database with an | |  |
|  |  | | invalid username or password. | |  |
|  | INVALID\_NUMBER | | Conversion of character to a number | |  |
|  |  | | failed due to invalid number | |  |
|  |  | | character | |  |
|  | NOT\_LOGGED\_ON | | It is raised when a database call is | |  |
|  |  | | issued without being connected to | |  |
|  |  | | the database. | |  |

**Query 1:**

Enter any number raise exception when divided by zero.

**Sol:**

declare

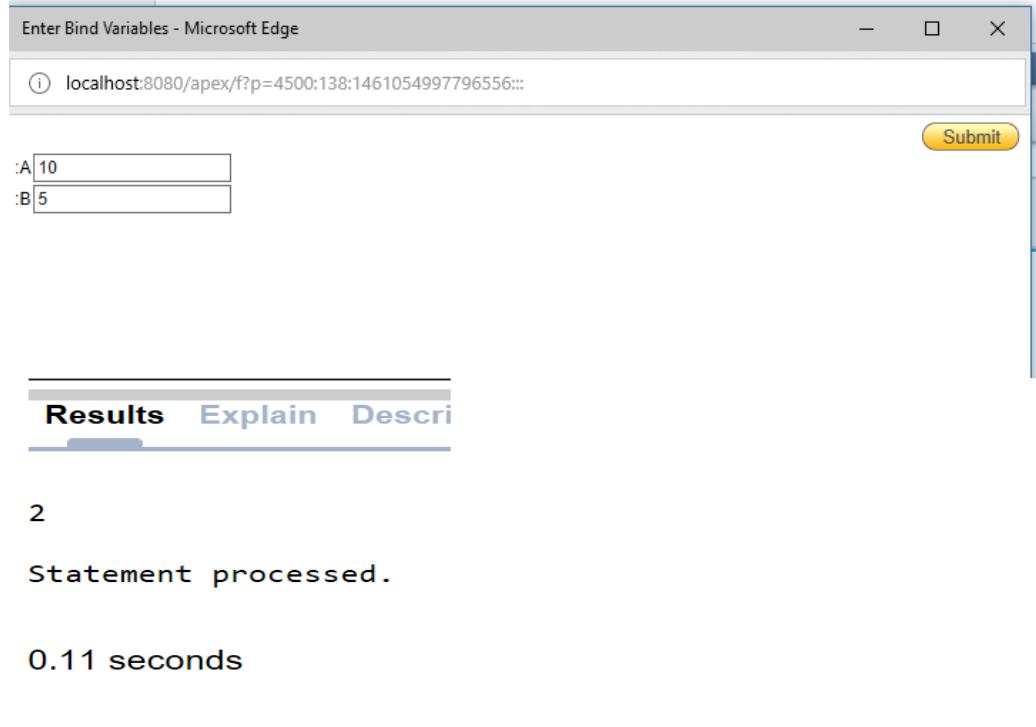
1. number;
2. number;
3. number; begin a:=:a; b:=:b; c:=a/b;

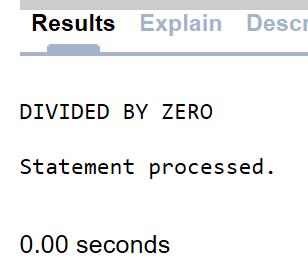
dbms\_output.put\_line(c); exception

when zero\_divide then

dbms\_output.put\_line('DIVIDED BY ZERO'); end;

**Output: -**

****



**Query 2:**

Display name of employee from employee table raise excption if deptment not found or data not found.

**Sol:**

declare

name emp.ename%type;

begin

select ename into name from emp where deptno=:deptno; dbms\_output.put\_line('ename is'||name);

exception

when zero\_divide then

dbms\_output.put\_line('dividing by 0');

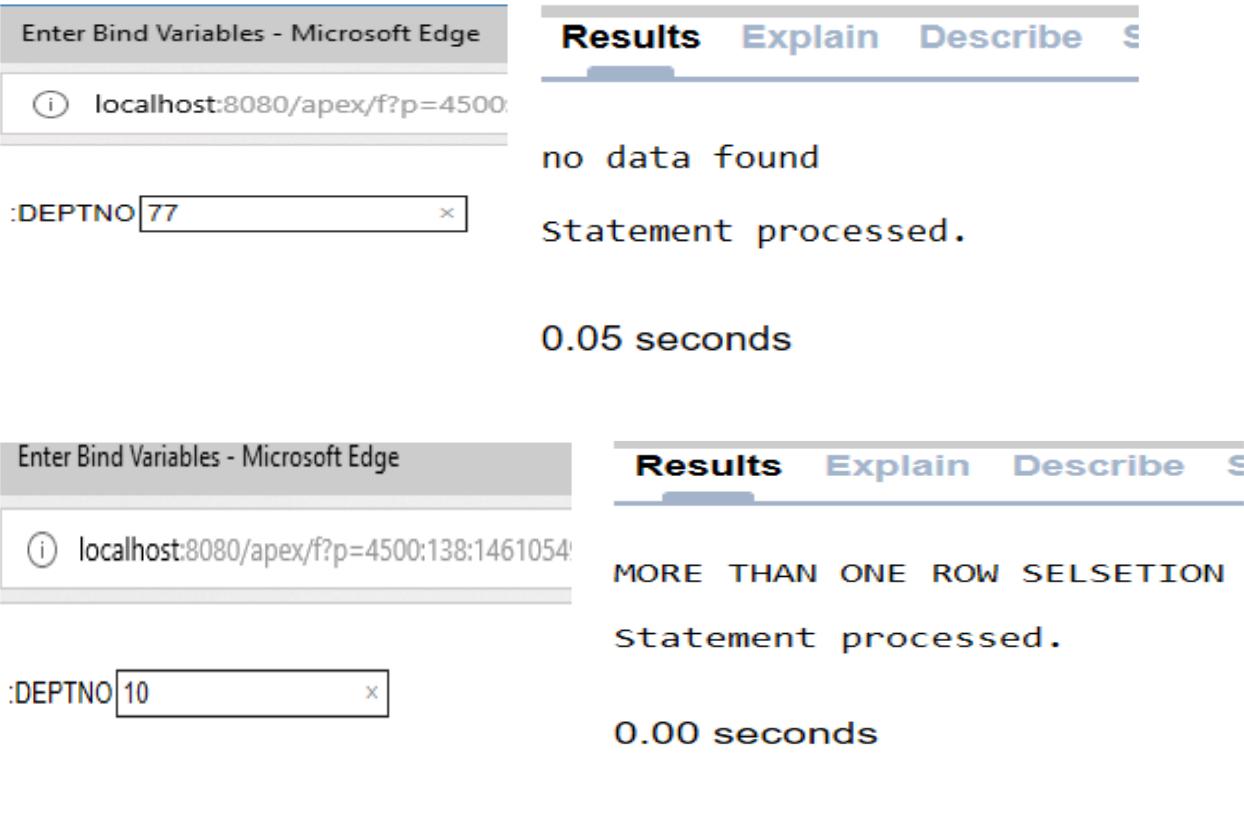
when too\_many\_rows then

dbms\_output.put\_line('MORE THAN ONE ROW SELSETION'); when others then

dbms\_output.put\_line('no data found');

end;

**Output: -**



**Query 3:**

To insert the value in table and raise the exception if duplicate value if inserted.

**Sol:**

declare

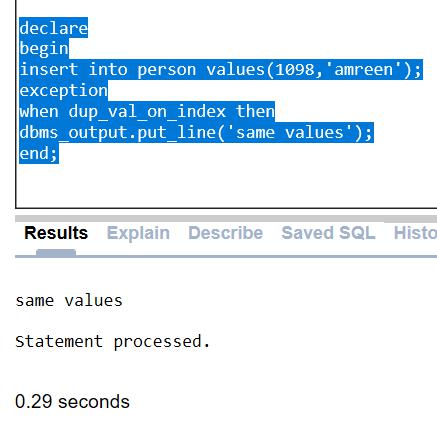
begin

insert into person values(1098,'amreen');

exception

when dup\_val\_on\_index then

dbms\_output.put\_line('same values');



**User-defined Exception:**

In Oracle, other than the above-predefined exceptions, the programmer can create their own exception and handle them. They can be created at a subprogram level in the declaration part. These exceptions are visible only in that subprogram. The exception that is defined in the package specification is public exception, and it is visible wherever the package is accessible.

**Syntax:**

DECLARE

<exception\_name> EXCEPTION;

BEGIN

<Execution block>

EXCEPTION

<Handler>

END;

**Query 1:**

Make an exception if name enter is null or no is null.

**Sol:**

declare

eno emp.empno%type;

name emp.ename%type;

exception\_name exception;

begin

eno:=:eno;

name:=:name;

if name is null then

raise exception\_name;

end if;

if eno is null then

raise exception\_no;

end if;

insert into emp(empno,ename) values (eno,name); dbms\_output.put\_line('data entered'); exception

when exception\_name then

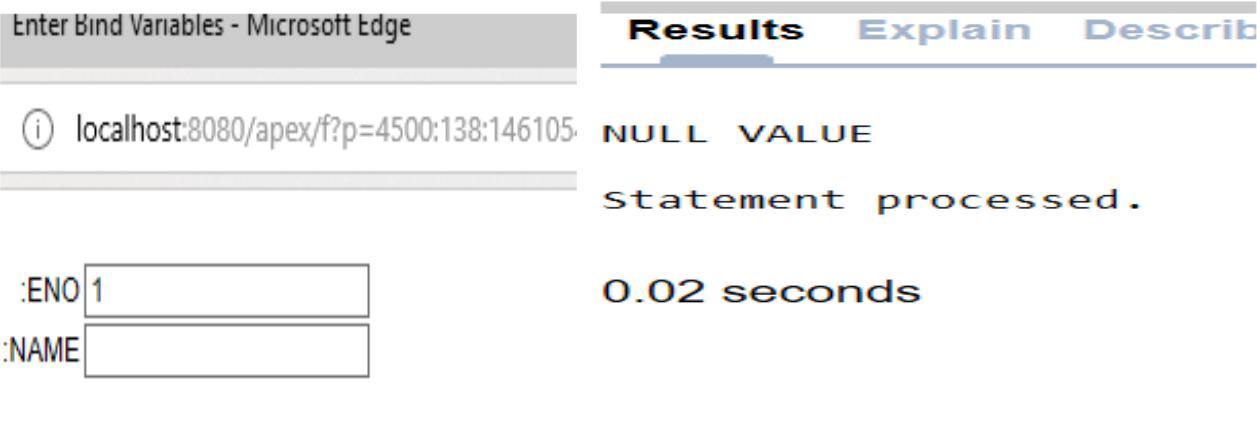
dbms\_output.put\_line('data is null');

when exception\_no then

dbms\_output.put\_line('data is null');

end;

**Output: -**



**Qurey 2:**

Raise exception if number enter is zero or name, id entered is duplicate.

**Sol:**

declare

exception1 exception;

exception2 exception;

1. number;
2. number;
3. number; begin a:=:a; b:=:b;

if b=0 then

raise exception1; end if;

insert into people1(id,name) values(1098,'Amreen'); raise exception2;

exception

when exception1 then

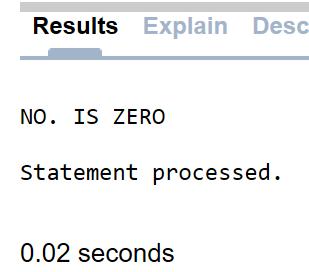
dbms\_output.put\_line('NO. IS ZERO ');

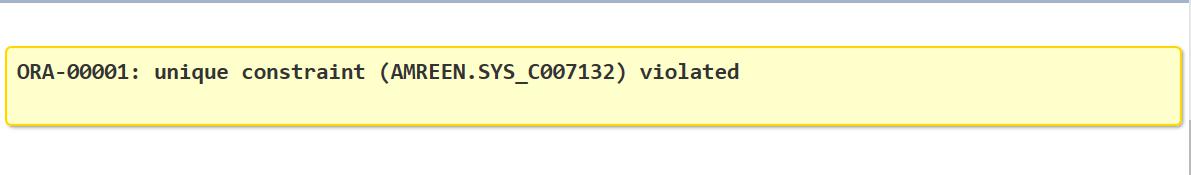
when exception2 then

dbms\_output.put\_line('DUPLICATE VALUES');

end;

**Output: -**



If duplicate value is entered:

**Experiment No.:-7**

**Aim: -** PL/SQL give you control to make your own exception base on oracle rules. User define exception must be declare yourself and RAISE statement to raise explicitly. Use PL/SQL user defined exception to make your own exception.

**PL/SQL Raise Exception**

All the predefined exceptions are raised implicitly whenever the error occurs. But the user-defined exceptions need to be raised explicitly. This can be achieved using the keyword 'RAISE'. This can be used in any of the ways mentioned below.

If 'RAISE' is used separately in the program, then it will propagate the already raised exception to the parent block.

**Syntax:**

CREATE [ PROCEDURE | FUNCTION ]

AS

BEGIN

<Execution block>

EXCEPTION

WHEN <exception\_name> THEN

<Handler>

RAISE;

END;

**Syntax Explanation:**

* In the above syntax, the keyword RAISE is used in the exception handling block.
* Whenever program encounters exception "exception\_name", the exception is handled and will be completed normally
* But the keyword 'RAISE' in the exception handling part will propagate this particular exception to the parent program.
* We can use keyword 'RAISE' followed by the exception name to raise that particular user-defined/predefined exception. This can be used in both execution part and in exception handling part to raise the exception.

**Query**:

Using re-raise concept change the salary of employee which is input wrong.

**Sol:**

declare

sal\_too\_high exception;

current\_sal number:=20000;

max\_sal number:=10000;

error\_sal number;

begin

begin

if current\_sal > max\_sal then

raise sal\_too\_high;

end if;

exception

when sal\_too\_high then

dbms\_output.put\_line('SALARY OUT OF RANGE'); dbms\_output.put\_line('MAX SALARY IS'||max\_sal); raise;

end;

exception

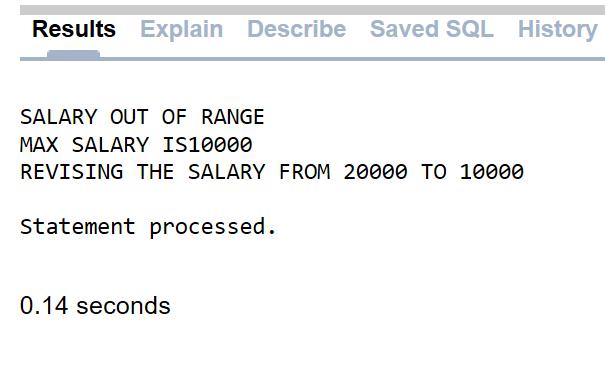
when sal\_too\_high then

error\_sal:=current\_sal;

current\_sal:=max\_sal;

dbms\_output.put\_line('REVISING THE SALARY FROM '||error\_sal||' TO '||current\_sal); end;

**Output: -**

****

**Pragma EXCEPTION\_INIT: -**

Pragma is a keyword directive to execute proceed at compile time. pragma EXCEPTION\_INIT function take this two argument,

1. exception\_name
2. error\_number

You can define pragrma EXCEPTION\_INIT in DECLARE BLOCK on your program.

**Syntax:**

DECLARE

user\_define\_exception\_name EXCEPTION;

PRAGMA EXCEPTION\_INIT(user\_define\_exception\_name,-error\_number); BEGIN

statement(s);

IF condition THEN

RAISE user\_define\_exception\_name;

END IF;

EXCEPTION

WHEN user\_define\_exception\_name THEN

User defined statement (action) will be taken;

END;

**Experiment No.: - 8-9**

**Aim :** Create a temporary work area in the system memory whenever a SQL statement is executed.This temporary work area will be used to store the data retrieved from the database, and manipulate this data. Create a cursor that can hold more than one row, but can process only one row at a time.

**Cursor**:

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

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

You can name a cursor so that it could be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. 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.

**%FOUND:**

Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE

**%NOTFOUND** **:**

The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

**%ISOPEN**

Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

**%ROWCOUNT**

Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

**Query 1:**

Count the number of rows and update the salary 100 times.

**Sol:**

declare

row number(5);

begin

update emp set sal=sal\*100;

if SQL%notfound then

dbms\_output.put\_line('NO UPDATE');

elsif SQL%found then

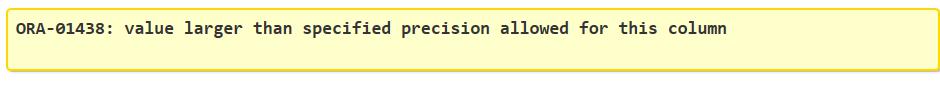
row:=SQL%rowcount;

dbms\_output.put\_line('COUNT OUTPUT '||row);

end if;

end;

**Output: -**

****

**Query 2:**

Delete the record if exist else not found

**Sol:**

declare

no number;

begin

delete from emp where empno=:no;

if sql%found then

dbms\_output.put\_line('RECORD FOUND AND DELETED'); insert into emp(empno,ename) values(101,'XYZ');

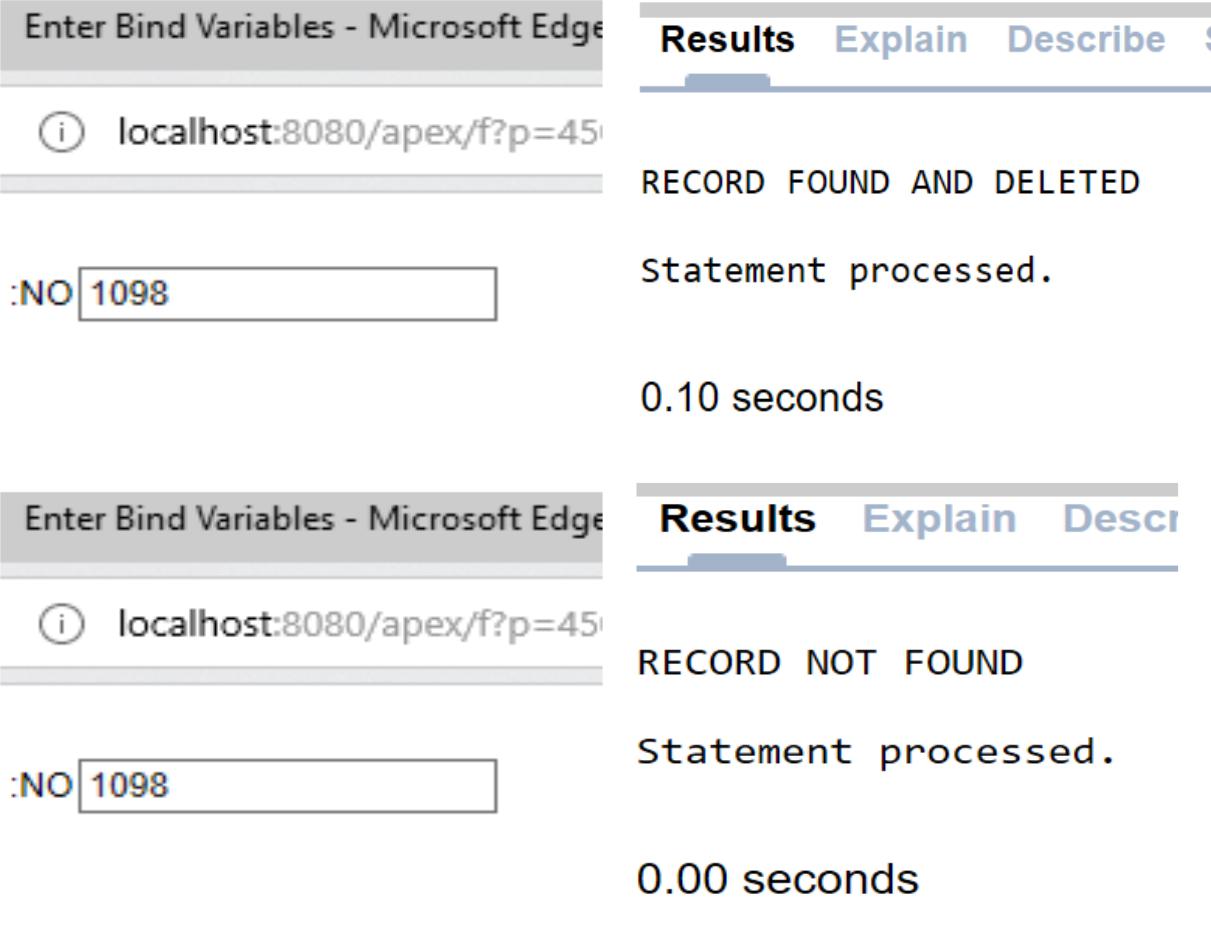
else

dbms\_output.put\_line('RECORD NOT FOUND');

end if;

end;

**Output**:



**Explicit Cursor: -**

Programmers are allowed to create named context area to execute their DML operations to get more control over it. The explicit cursor should be defined in the declaration section of the PL/SQL block, and it is created for the 'SELECT' statement that needs to be used in the code.

Below are steps that involved in working with explicit cursors.

* **Declaring the cursor**

Declaring the cursor simply means to create one named context area for the 'SELECT' statement that is defined in the declaration part. The name of this context area is same as the cursor name.

**Opening Cursor** Opening the cursor will instruct the PL/SQL to allocate the memory for this cursor. It will make the cursor ready to fetch the records.

In this process, the 'SELECT' statement is executed and the rows fetched is stored in the allocated memory. These are now called as active sets. Fetching data from the cursor is a record-level activity that means we can access the data in a record-by-record way.

* **Fetching Data from the Cursor**

Each fetch statement will fetch one active set and holds the information of that particular record. This statement is same as 'SELECT' statement that fetches the record and assigns to the variable in the 'INTO' clause, but it will not throw any exceptions.

* **Closing the Cursor**

Once all the record is fetched now, we need to close the cursor so that the memory allocated to this context area will be released.

**Syntax: -**

DECLARE

CURSOR <cursor\_name> IS <SELECT statement^> <cursor\_variable declaration>

BEGIN

OPEN <cursor\_name>;

FETCH <cursor\_name> INTO <cursor\_variable>;

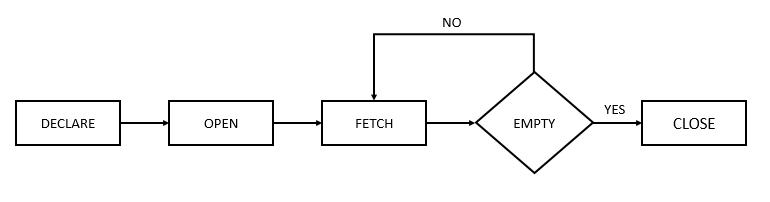
.

.

CLOSE <cursor\_name>;

END;

* In the above syntax, the declaration part contains the declaration of the cursor and the cursor variable in which the fetched data will be assigned.
* The cursor is created for the 'SELECT' statement that is given in the cursor declaration.
* In execution part, the declared cursor is opened, fetched and closed.

**Diagram: **

**Query :**

Display the employee id, name, salary using cursor.

**Sol:**

declare

eid emp.empno%type;

name emp.ename%type;

salary emp.sal%type;

cursor cursor1 is

select empno,ename,sal from emp;

begin

open cursor1;

loop

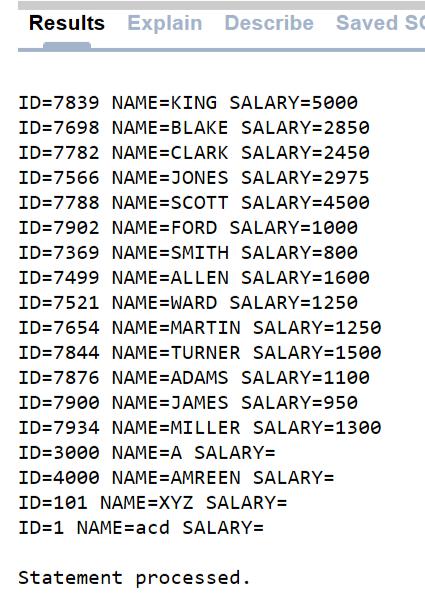
fetch cursor1 into eid,name,salary;

exit when cursor1 %NotFound;

dbms\_output.put\_line('ID='||eid||' NAME='||name||' SALARY='||salary); end loop;

close cursor1;

end;

**Output: -**

**FOR Loop Cursor statement: -**

"FOR LOOP" statement can be used for working with cursors. We can give the cursor name instead of range limit in the FOR loop statement so that the loop will work from the first record of the cursor to the last record of the cursor. The cursor variable, opening of cursor, fetching and closing of the cursor will be done implicitly by the FOR loop.

**Syntax:**

DECLARE

CURSOR <cursor\_name> IS <SELECT statement>;

BEGIN

FOR I IN <cursor\_name>

LOOP

.

.

END LOOP;

END;

* In the above syntax, the declaration part contains the declaration of the cursor.
* The cursor is created for the 'SELECT' statement that is given in the cursor declaration.
* In execution part, the declared cursor is setup in the FOR loop and the loop variable 'I' will behave as cursor variable in this case.

**Example: -**

declare

cursor c1 is select ename,sal from emp;

begin

for a in c1

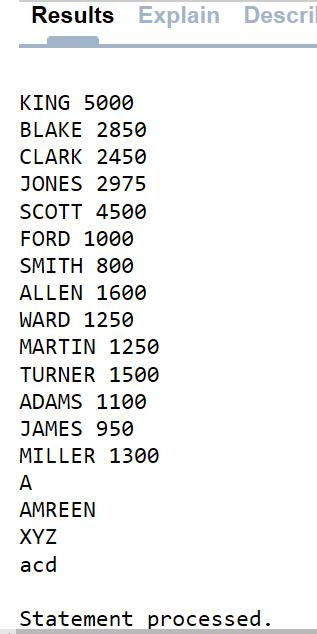
loop

dbms\_output.put\_line(a.ename||' '||a.sal);

end loop;

end;

**Output:**

****

**Passing parameter to cursor: -**

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

**Example**: -

declare

cursor c1(p\_id number,p\_name varchar)is select empno,ename from emp where empno=p\_id and ename=p\_name;

begin

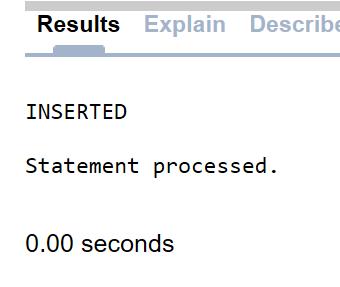
open c1(7896,'TOM');

dbms\_output.put\_line('INSERTED');

close c1;

end;

**Output:**



**Experiment No.: -10-11**

**Aim :** Creating and calling a standalone function. The function should return the total number ofCUSTOMERS in the customers table. Use the CUSTOMERS table, with different columns like name, salary, department, designation, DOJ etc.

**What is Function?**

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

* Functions are a standalone block that is mainly used for calculation purpose.
* Function use RETURN keyword to return the value, and the datatype of this is defined at the time of creation.
* A Function should either return a value or raise the exception, i.e. return is mandatory in functions.
* Function with no DML statements can be directly called in SELECT query whereas the function with DML operation can only be called from other PL/SQL blocks.
* It can have nested blocks, or it can be defined and nested inside the other blocks or packages.
* It contains declaration part (optional), execution part, exception handling part (optional).
* The values can be passed into the function or fetched from the procedure through the parameters.
* These parameters should be included in the calling statement.
* Function can also return the value through OUT parameters other than using RETURN.
* Since it will always return the value, in calling statement it always accompanies with assignment operator to populate the variables.

**Syntax: -**

CREATE OR REPLACE FUNCTION

<procedure\_name>

(

<parameterl IN/OUT <datatype>

)

RETURN <datatype>

[ IS | AS ]

<declaration\_part>

BEGIN

<execution part>

EXCEPTION

<exception handling part>

END;

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

**Advantages:**

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

**Functions are of two types:**

1. Local functions
2. Stored functions

**Local functions:**

The local function are those which are defined within the main code. These are not separately stored in oracle.

**Query 1:**

Make a function to check the greatest number between any 2 numbers.

**Sol:**

declare

1. number;
2. number;
3. number;

function great(a number,b number)

return number

is

begin

if a > b then

return a;

else

return b;

end if;

end;

begin

a:=:a;

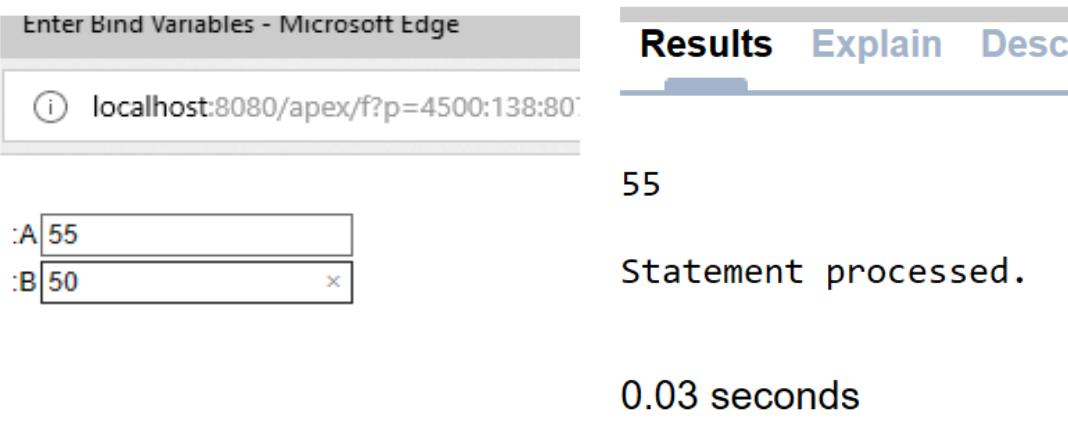
b:=:b;

c:=great(a,b);

dbms\_output.put\_line(c);

end;

**Output**: -



**Query 2:**

Get the salary of employee of which the employee id is entered .

**Sol:**

Declare

salary emp.sal%type;

eid emp.empno%type;

function fun1(eid number)

return number

is

begin

select sal into salary from emp where empno=:eid; return salary;

end;

begin

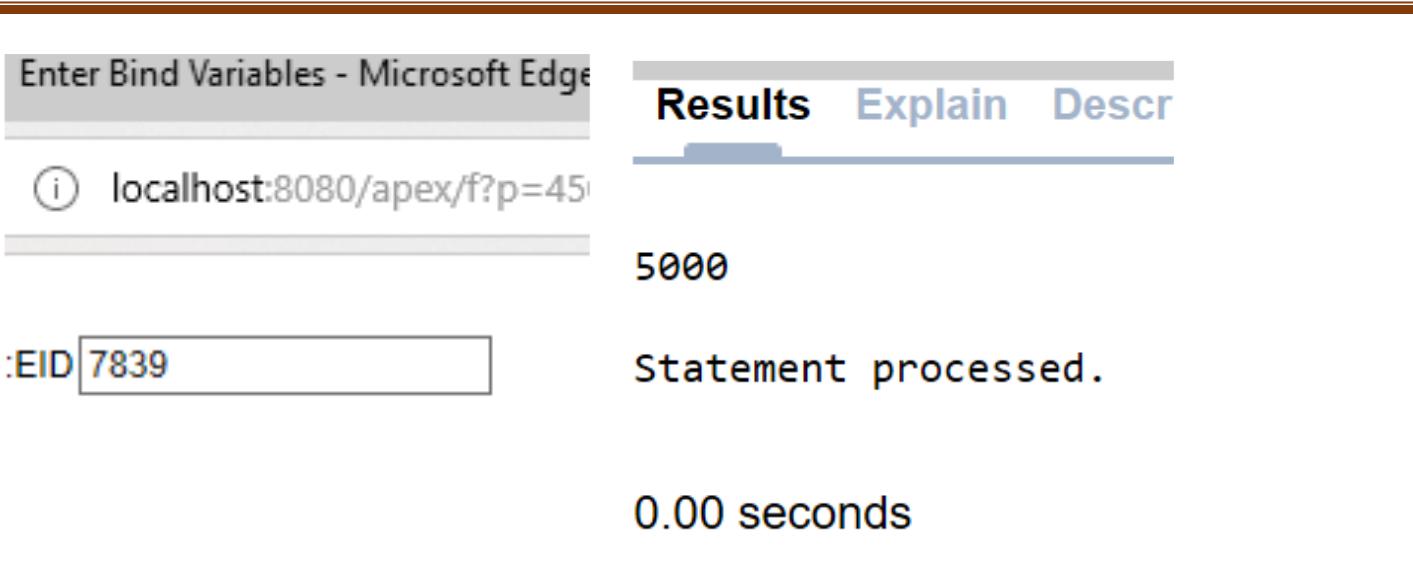
eid:=:eid;

salary:=fun1(eid);

dbms\_output.put\_line(salary);

end;

**Output: -**



**Stored functions:**

In the stored functions the functions are already stored in the oracle , these functions can be call later on in any code/or program in which required.

**Query 1:**

Write a code to find greater number in any 2 number.

**Sol:**

create or replace function great(a number,b number)

return number

is

1. number; begin

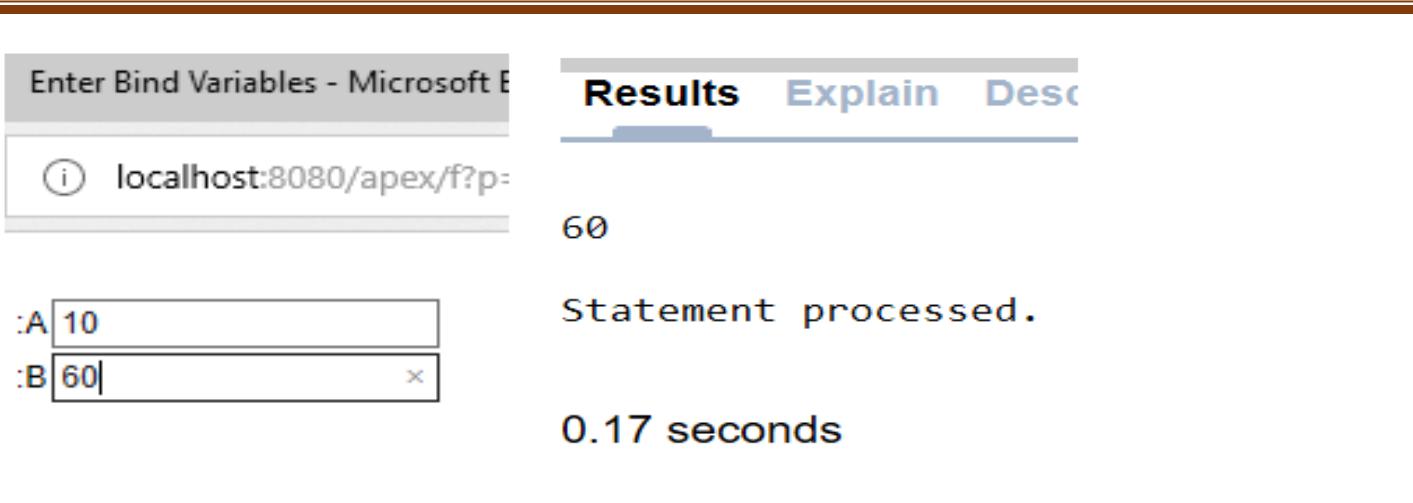
if a > b then return a; else return b; end if; end;

declare

1. number;
2. number;
3. number; begin a:=:a; b:=:b; c:=great(a,b);

dbms\_output.put\_line(c); end;

**Output**:



**Query 2:**

Write a code/program to multiply 2 numbers.

**Sol:**

create or replace function multi(a number,b number)

return number

is

1. number; begin c:=a\*b; return c; end;

declare

1. number;
2. number;
3. number; begin a:=:a; b:=:b; c:=multi(a,b);

dbms\_output.put\_line(c); end;

**Output:**

