Course Route Map

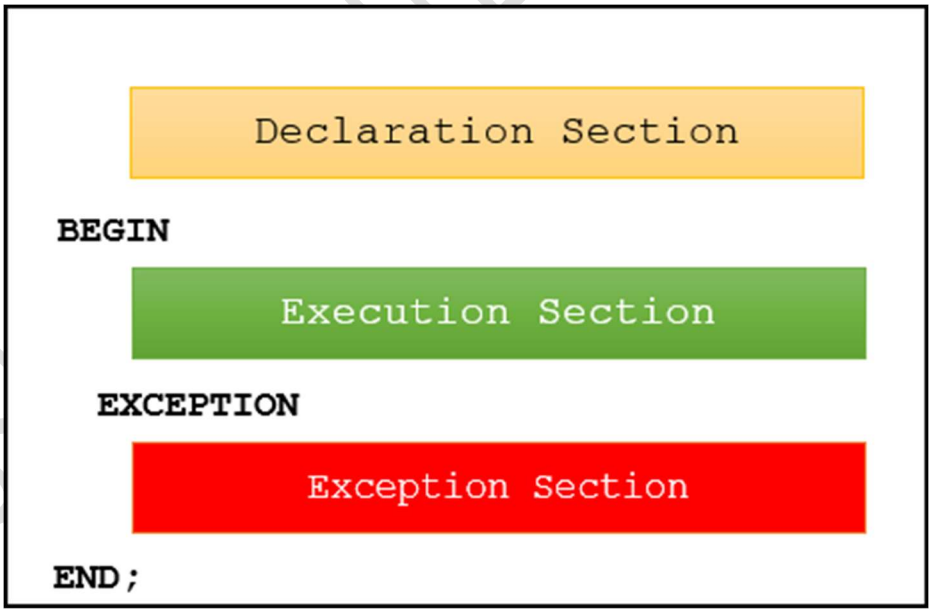
- PL-SQL (Procedure Language – SQL)
- Procedures in PL/SQL
- Functions in PL/SQL
- Packages in PL/SQL
- EXCEPTIONS in PL/SQL
- Database Triggers in PL/SQL

- PL-SQL (Procedure Language – SQL)
  - Introduction to PL/SQL
  - PL/SQL Architecture
  - PL/SQL Data types
  - Variable and Constants
  - Using Built in Functions
  - Conditional and Unconditional Statements
  - Simple if, if… else, nested if..else, if..else Ladder
  - Selection Case, Simple Case, GOTO Label and EXIT
  - Iterations in PL/SQL
  - Simple LOOP,WHILE LOOP,FOR LOOP and NESTED LOOPS
  - SQL within PL/SQL
  - Composite Data types (Complete)
  - Cursor Management in PL/SQL
    - Implicit Cursors
    - Explicit Cursors
    - Cursor Attributes
    - Cursor with Parameters
    - Cursors with LOOPs Nested Cursors
    - Cursors with Sub Queries
    - Ref. Cursors
  - Record and PL/SQL Table Types

Introduction to PL/SQL:

- It stands for procedural language/structure query language in SQL we can't execute the same statement repeatedly to a particular no of times but where as in PL/SQL we can execute the same statement repeatedly because it supports looping mechanism.
- In SQL we can't execute more than one statement at a time. But where as in PL/SQL we can execute more than one statement concurrently.
- PL/SQL program is a combination of procedural language statements and structure query language statements.
- PL/SQL supports all the principles of procedure language such as procedural functions, control statements, conditional statements e.t.c., and also it supports some principles of OOPS.
- PL/SQL is not a consecutive language and every statement of PL/SQL program should ends with ;(semicolon).
- PL/SQL is the high performance transaction processing language.

Structure of PL/SQL



Declaration Section:

- The Declaration section of a PL/SQL Block starts with the reserved keyword DECLARE.

- This section is optional and is used to declare any placeholders like variables, constants, records and cursors, which are used to manipulate data in the execution section.

Execution Section:

- The Execution section of a PL/SQL Block starts with the reserved keyword BEGIN and ends with END.
- This is a mandatory section and is the section where the program logic is written to perform any task.
- The programmatic constructs like loops, conditional statement and SQL statements form the part of execution section.

Exception Section:

- The Exception section of a PL/SQL Block starts with the reserved keyword EXCEPTION.
- This section is optional. Any errors in the program can be handled in this section, so that the PL/SQL Blocks terminates gracefully.
- If the PL/SQL Block contains exceptions that cannot be handled, the Block terminates abruptly with errors.

Program 1:

Write a PL/SQL program to display a welcome message

```
begin
   dbms_output.put_line('Welcome');
end;
/
```

Note:

To display the output we have to set the serveroutput on.

```
SQL> SET SERVEROUTPUT ON;
```

Data types in PL/SQL:

The data types which are using in SQL same data types are supported in PL/SQL.

Variable declaration in PL/SQL:

Syntax:

variable data type(SIZE);

Exampe:

a number(10);

Input Statement in PL/SQL

- There is no input statements in PL/SQL to input the values at run time.
- If we want input values at run time then we use insertion operator (&).

Program 2:

Write a pl/sql program input any numbers and find out their sum.

```
declare
    a number:=&a;
    b number:=&b;
    c number;
begin
    c:=a+b;
    dbms_output.put_line('sum is'||c);
end;
```

Operators in PL/SQL:

The operators which are using in SQL same operators are by PL/SQL except assignment operator.

WRITE A PLSQL PROGRAM TO FIND THE BIGGEST OF TWO NUMBERS.

```
declare
 a number;
 b number;
```

```
begin
a:=&a;
b:=&b;
if(a>b) then
dbms_output.put_line('a is big');
elsif(a=b) then
dbms_output.put_line('both are same');
else
dbms_output.put_line('b is big');
end if;
end;
/
```

WRITE A PLSQL PROGRAM TO FIND THE BIGGEST OF THREE NUMBERS.

```
declare
a number;
b number;
c number;
begin
a:=&a;
b:=&b;
c:=&c;
if(a>b and a>c)
then
dbms_output.put_line('A is big');
else
if(b>c) then
```

```
dbms_output.put_line('B is big');
else
dbms_output.put_line('C is big');
end if;
end if;
end;
/
```

WRITE A PLSQL PROGRAM TO FIND THE BIGGEST OF THREE NUMBERS USING NESTED IF.

```
declare
a number;
b number;
c number;
begin
a:=&a;
b:=&b;
c:=&c;
if(a>b)
then
  if (b>c) then
    dbms_output.put_line('A is big');
  else
    dbms_output.put_line('C is big');
  end if;
else
if(b>c) then
```

```
dbms_output.put_line('B is big');
else
dbms_output.put_line('C is big');
end if;
end if;
end;
/
```

WRITE A PLSQL PROGRAM THAT SHOULD ACCEPT EMPLOYEE DETAILS AND DECIDE THE BASIC ACCORDING TO DESIGNATION.

```
declare
eid number(5) := &empid;
ename varchar2(20) := '&ename';
desig varchar2(10) := '&desig';
deptid number(3) := &deptid;
basic number(8);
basic1 number(8);
ename1 varchar2(20);
k number(5);
begin
if(deptid = 10) then
  if(desig = 'manager') then
   basic:=5000;
  else
   basic:=4000;
  end if;
else
  if(deptid = 20) then
```

```
   if(desig = 'manager') then
     basic:=10000;
    else
      basic:=9000;
   end if;
  end if;
end if;
dbms_output.put_line('Basic = ' || basic);
insert into employee values(eid,ename,desig,deptid,basic);
k:=eid;
select ename,basic into ename1,basic1 from employee where eid = k;
dbms_output.put_line('Ename is' || ename1);
dbms_output.put_line('Basic is' || basic1);
end;
/
```

LOOPS IN PLSQL

WRITE A PLSQL PROGRAM TO PRINT THE EVEN NUMBERS BETWEEN 1 TO 10.

```
declare
 i number;
 begin
 for i in  1..10 loop
  if( i mod 2 =0) then
      dbms_output.put_line(i);
  end if;
 end loop;
```

```
 end;
/

declare
a number(5) := &a;
b number(5) := &b;
c number(5) := &c;
begin
if (a>b) then
  if(a>c) then
     dbms_output.put_line('A is big');
  else
    dbms_output.put_line('C is big');
  end if;
elsif (b>c) then
      dbms_output.put_line('B is big');
  else
    dbms_output.put_line('C is big');
  end if;
end;
/
```

WRITE A PLSQL PROGRAM TO FIND THE SUM OF NATURAL NUMBERS BETWEEN 1 TO 10

```
declare
i number(2);
add number(3) := 0;
```

```
begin
for i in 1..10
loop
   add:=add+i;
end loop;
dbms_output.put_line('Result is:' || add);
end;
/
```

WRITE A PLSQL PROGRAM TO FIND REVERSE OF A GIVEN NUMBER.

```
declare
n number(10):=&n;
r number(10);
s number(10):=0;
begin
while(n>0)
loop
   r:=n mod 10;
   s:= s * 10 + r;
   n:=floor(n/10);
end loop;
dbms_output.put_line('Reverse is:' || s);
end;
/
```

WRITE A PLSQL PROGRAM TO PRINT THE NUMBERS BETWEEN 1 TO 10 IN REVERSE ORDER.

```
declare
 i number;
 begin
 for i in reverse 1..10 loop
 dbms_output.put_line(i);
 end loop;
 end;
/
```

PROGRAMS ON SELECT...INTO.... CLAUSE

WRITE A PL/SQL PROGRAM THAT ACCEPTS THE EMPID AND PRINT THE NAME AND SALARY OF AN EMPLOYEE.

```
declare
empid number(5);
empname varchar2(20);
empsalary number(5);
begin
dbms_output.put_line('Enter employee ID');
empid:=&empid;
select ename,sal into empname,empsalary from emp  where empno = empid;
dbms_output.put_line('Employee Id = ' || empid);
dbms_output.put_line('Employee Name = ' || empname);
dbms_output.put_line('Employee Salary = ' || empsalary);
end;
/
```

WRITE A PL/SQL PROGRAM THAT ACCEPTS THE DEPTNUMBER AND FIND THE HIGHEST SALARY IN THAT DEPARTMENT.

```
declare
dno number(5);
highestsalary number(5);
begin
dbms_output.put_line('Enter department Number');
dno:=&dno;
select max(sal) into highestsalary from emp  group by deptno having deptno = dno;
dbms_output.put_line('Employee Salary = ' || Highestsalary);
end;
/
```

WRITE A PL/SQL PROGRAM THAT ACCEPTS THE EMPID AND PRINT THE NAME AND SALARY OF AN EMPLOYEE USING COLUMN TYPE.

```
declare
empid employee.empno%type;
empname employee.ename%type;
empsalary employee.sal%type;
begin
dbms_output.put_line('Enter employee ID');
empid:=&empid;
select ename,sal into empname,empsalary from emp1 where empno = empid;
dbms_output.put_line('Employee Id = ' || empid);
dbms_output.put_line('Employee Name = ' || empname);
dbms_output.put_line('Employee Salary = ' || empsalary);
```

```
end;
/
```

WRITE A PL/SQL PROGRAM THAT ACCEPTS THE EMPID AND PRINT THE NAME AND SALARY OF AN EMPLOYEE USING ROW TYPE.

```
declare
v_emp emp1%rowtype;
empid emp1.empno%type;
begin
dbms_output.put_line('Enter employee ID');
empid:=&empid;
select * into v_emp from emp1 where empno = empid;
dbms_output.put_line('Employee Id = ' || v_emp.empno);
dbms_output.put_line('Employee Name = ' || v_emp.ename);
dbms_output.put_line('Employee Salary = ' || v_emp.sal);
end;
/
```

Important Commands:

- SAVE THE PROGRAM INTO A FILE
  - SQL> SAVE <<PATH>><<FILENAME>>.SQL
- RUN THE PROGRAM IN THE BUFFER
  - SQL> RUN <<PATH>><<FILENAME>>.SQL
- LOAD THE FILE FROM THE DIRECTORY TO BUFFER
  - SQL> GET <<PATH>><<FILENAME>>.SQL

Procedures in PL/SQL

- STORED PROCEDURES
- PROCEDURE with Parameters (IN,OUT and IN OUT)
- POSITIONAL Notation and NAMED Notation
- Procedure with Cursors
- Dropping a Procedure

PROGRAM: 1

```
create or replace procedure sample10(p_name varchar2)
is
begin
dbms_output.put_line('Welcome'|| p_name);
end;
/
```

```
SQL> exec sample10('SSSIT');
Welcome SSSIT
```

```
SQL> ed
Wrote file afiedt.buf
```

PROGRAM:2

```
1  create or replace procedure addition(p_val1 in number,p_val2 number)
 2  is
 3  begin
 4  dbms_output.put_line('sum is'|| (p_val1 + p_val2));
 5* end;
SQL> /
```

Procedure created.

SQL> call addition(10,15);

sum is25

Call completed.

SQL> ed

Wrote file afiedt.buf

PROGRAM: 3

```
1  create or replace procedure addition(p_val1 in number,p_val2 number)
2  is
3  begin
4  dbms_output.put_line('sum is'|| (p_val1 + p_val2));
5* end;
SQL> /
```

Procedure created.

SQL> call addition(10,15);

sum is25

Call completed.

PROGRAM: 4

```
SQL> ed
Wrote file afiedt.buf

  1  create or replace procedure multiply(a in number,b out number)
  2  is
  3  begin
  4   b := a*a;
  5* end multiply;
  6  /

Procedure created.

SQL> ed
Wrote file afiedt.buf

  1  declare
  2  val number;
  3  begin
  4  multiply(2,val);
  5  dbms_output.put_line('Result is:' || val);
  6* end;
  7  /
Result is:4

PL/SQL procedure successfully completed.
```

PROGRAM: 5

```
SQL> ed
Wrote file afiedt.buf

  1  create or replace procedure incr(p_val in out number) is
  2  begin
  3  p_val := p_val + 1;
  4 end;
SQL> /
Procedure created.
SQL> ed
Wrote file afiedt.buf
  1  declare
  2  val number := 3;
  3  begin
  4  incr(val);
  5  dbms_output.put_line('Result is:' || val);
  6 end;
SQL> /
Result is:4

PL/SQL procedure successfully completed.
```

- Functions in PL/SQL
  - Difference between Procedures and Functions
  - User Defined Functions
  - Nested Functions
  - Using stored function in SQL statements

SQL> ed

Wrote file afiedt.buf

Create or replace function <fun_name(par datatype,par2 datatype,…) return value datatype

Is

Begin

Return value;

end

```
 1  create or replace function incrby1(p_val number)
 2  return number
 3  is
 4  p_val1 number;
 5  begin
 6  p_val1 := p_val + 1;
 7  return p_val1;
 8* end;
SQL> /

Function created.

SQL> ed
Wrote file afiedt.buf
```

```
1  declare
2  res number:=10;
3  res1 number;
4  begin
5  res1:=incrby1(res);
6  dbms_output.put_line(res1);
7 end;
8  /
```

PL/SQL procedure successfully completed.

Drop a function:

SQL>drop function <<function-name>>;

Function dropped.

Cursors in PL-SQL

- It is an Temporary buffer used to hold the transactional data for manipulation purpose.
- It is not stored in database.
- It is not Re-usable.
- It is valid in Pl/sql Block only.
- It is created in the Logical memory only.

2 Types :

1. Implicit cursors: Automatically created by oracle whenever " DML" operations are performed by user.

2. Explicit Cursors: Created by user in Pl/sql Block.

* Used to retrieve multiple rows from multiple tables into pl/sql block for manipulation purpose. It is based on SELECT stmt.

Explicit Cursors : 3 Steps

I. Declaring Cursor

II. Cursor Operations

III. Cursor Attributes

I . Declaring Cursor

Syntax: cursor <cursor name> is <select stmt>;

ex: cursor c1 is select * from emp;

II. Cursor Operations:

i> open <cursor name>;

used to open the cursor.

memory will be allocated to cursor after opening it.

ii>Fetch <cursor name> into < Variables >;

Used to retrieve data from cursor to pl/sql variables.

At a time it can retrieve only one row into variables. Generally Fetch will be placed in loop .

iii> close <cursor name>;

used to close the cursor.

memory alloted will be Deallocated .

III. Cursor Attributes

Gives the status of cursor

<cursor name>%<attribute>

a> %isopen --- Returns True / False

Returns True if cursor is opened successfully

b> %found --- Returns True / False

Returns True if Fetch statement sucessfully retrieves the data into pl/sql variables.

c> %notfound --- Returns True / False

Returns True if Fetch statement fails to retrieve the data into Pl/sql variables.

d> %rowcount --- Returns Number

Returns the No.of rows sucessfully retrieved from cursor so far.

Initially it holds 0.

After every sucessful "Fetch" it is incremented by 1.

Write a PL/SQL Query to display first five highest salary from employee table.

```
Declare
Cursor e1 is select * from employee order by sal desc;
V_sal number(10);
```

```
Begin
Open c1;
Loop
Fetch c1 into v_sal;
Dbms_output.put_line(v_sal);
Exit when c1%rowcount = 5;
End loop
Close c1;
End;
/
```

Write a PL/SQL Query to display employee names whose salaries are > 2000 from employee table.

```
Declare
Cursor e1 is select * from employee;
I employee%rowtype;
Begin
Open c1;
Loop
Fetch e1 into i;
Exit when c1%notfound;
If i.sal > 2000 then
Dbms_output.put_line(i.empid || i.empname||i.sal);
End loop
Close c1;
End;
/
```

Write a PL/SQL Query to display even number of rows from emp table

```
Declare
```

```
Cursor e1 is select * from employee;
I emp%rowtype;
Begin
Open c1;
Loop
Fetch c1 into i;
Exit when c1%notfound;
If mod(c1%rowcount,2) = 0 then
Dbms_output.put_line(i.ename||i.esal);
End if
End loop
Close c1;
End;
/
```

Note:

Using Cursor for loop we can eliminate the Cursor life cycle i.e. we need not to use open, fetch, close

Internally PL/SQL runtime engine uses these statements.

Write a PL/SQL Query to display employee names whose salaries are > 2000 from employee table.

```
Declare
Cursor e1 is select * from employee;
Begin
For I in e1
loop
Dbms_output.put_line(i.empid || i.empname||i.sal);
End loop
End;
```

/

Note:

We can eliminate declare section of the cursor using for loop.

In this we are using select statement in place of cursor name in cursor for loop.

```
Begin
For I in (select * from emp)
Loop
Dbms_output.put_line(i.empid || i.empname||i.sal);
End loop
End;
/
```

Parameterized cursors

We can pass parameters to the cursors same like a sub program in parameters.

Syntax:

- Cursor cursorname (parameter datatype) is select * from table where column name = parameter name.
- Open cursor(actual parameters)

Example:

```
Declare
Cursor c1(p_deptno number) is select * from employee where deptno=p_deptno;
I employee%rowtype;
Begin
Open c1(10);
Fetch e1 into i;
Exit when c1%notfound;
Dbms_output.put_line(i.empid || i.empname||i.sa||i.deptnol);
End loop
```

```
Close c1;

End;

/
```

Note: when ever we are passing formal parameter to the cursor, then we can't declare the size of the data type in formal parameter declaration.

Write a PL/SQL Parameterized query to display the list of employees who are working as a managers and analyst from employee table.

```
Declare

Cursor c1(p_job number) is select * from employee where job=p_job;

I employee%rowtype;

Begin

Open c1('manager');

Dbms_output.put_line('Employees who are working as a managers');

Fetch e1 into i;

Exit when c1%notfound;

Dbms_output.put_line(i.empid || i.empname);

End loop

Close c1;

Open c1('analyst');

Dbms_output.put_line('Employees who are working as a Analyst');

Fetch e1 into i;

Exit when c1%notfound;

Dbms_output.put_line(i.empid || i.empname);

End loop

Close c1;

End;

/
```

Note:

Before reopening the cursor, we must close the cursor properly. Otherwise it raises an error "CURSOR ALREADY OPEN"

If we are trying to close the cursor without opening the cursor, then it raises "INVALID CURSOR".

Example using For Loop

```
Declare
Cursor c1(p_deptno number) is select * from employee where deptno=p_deptno;
I employee%rowtype;
Begin
For I in C1(10)
Loop
Dbms_output.put_line(i.empid || i.empname||i.sa||i.deptnol);
End loop
Close c1;
End;
/
```

Write a pl/sql statement to retrieve the employee details department wise.

```
Declare
Cursor c1 is select deptid from dept;
Cursor c2(p_deptid number) is select * from employee where deptid=p_deptid;
Begin
For I in c1
Loop
Dbms_output.put_line('Dept Number is:'|| i.deptid);
For j in c2(i.deptid)
Loop
Dbms_output.put_line('Employee ID is:'|| j.empid);
End loop
```

End loop

End;

- EXCEPTIONS in PL/SQL
  - Types of exceptions
  - User Defined Exceptions
  - Pre Defined Exceptions
  - RAISE_APPLICATION_ERROR
  - SQL Error Code Values

Exception Handling: Errors are classified into two types those are

i. Syntactical errors
ii. Logical errors.

Syntactical errors: These errors will be raise when the user violate the language rules, these errors will be raise at compile time.

Logical Errors: These errors will be raise if the program contains any logical mistakes and those will be raised at runtime. We can handle these errors by using the exception handling mechanism.

HANDLING EXCEPTIONS

When exception is raised, control passes to the exception section of the block. The exception section consists of handlers for some or all of the exceptions. An exception handler contains the code that is executed when the error associated with the exception occurs, and the exception is raised.

Syntax:

```
EXCEPTION
        When exception_name then
                Sequence_of_statements;
        When exception_name then
                Sequence_of_statements;
        When others then
                Sequence_of_statements;
END;
```

Exceptions are classified into two types those are

i. System defined ex caption
ii. User defined exceptions.

i. **System defined exceptions:** An exception which is defined by the system internally those exceptions can be called as System defined exceptions.

ii. **User defined Exceptions:** An exception which is defined by the user manually and programmatically those exceptions are called User defined exceptions.

1. NO_DATA_FOUND EXCEPTION: This exception will be raised when the user's data not available in the table.

P) Write a PL/SQL block to raise NO_DATA_FOUND exception.

```
declare
a emp%rowtype;
begin
select * into a from emp where empno=&a;
dbms_output.put_line('ename is'||a.ename);
dbms_output.put_line('salary is'||a.sal);
dbms_output.put_line('department no is'||a.deptno);
exception
when no_data_found then
dbms_output.put_line('Data is not available for given number plese enter another number');
end;
```

P) Write a PL/SQL block to raise zero divide exception.

```
declare
a number:=&a;
b number:=&b;
c number;
```

```
begin
c:=a/b;
dbms_output.put_line('division is'||c);
exception
when zero_divide then
dbms_output.put_line('infinity');
end;
```

1. **Too many rows:** This exception will be raised when the user is trying to fetch more than one record at a time.

EX:

```
declare
 a number;
 b varchar2(10);
 c number;
 begin
 select ename,sal into b,c from emp where deptno=&a;
 dbms_output.put_line(b||' ' ||c);
end;
```

```
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
```

EX:2

```
 declare
  a number;
  b varchar2(10);
  c number;
```

```
begin
select ename,sal into b,c from emp where deptno=&a;
dbms_output.put_line(b||' ' ||c);
exception
when too_many_rows then
dbms_output.put_line('It is not posible to fetch more than one row');
end;
```

OUTPUT:

```
Enter value for a: 30
old   6:   select ename,sal into b,c from emp where deptno=&a;
new   6:   select ename,sal into b,c from emp where deptno=30;
It is not posible to fetch more than one row
PL/SQL procedure successfully completed.
```

Using more than two exceptions.

```
declare
a emp%rowtype;
begin
select * into a from emp where deptno=&a;
dbms_output.put_line('ename is'||a.ename);
dbms_output.put_line('salary is'||a.sal);
dbms_output.put_line('department no is'||a.deptno);
exception
when no_data_found then
dbms_output.put_line('Data is not available for given number plese enter another number');
When too_many_rows then
dbms_output.put_line('It is not posible to fetch more than one row');
end;
```

2. **Dup_val_on_index:** This exception will be raised if the user trying to enter duplicate values under primary key constraint column or unique key constraint column.

EX:

```
declare
 a emp.empno%type;
 b emp.ename%type;
 c emp.sal%type;
 d emp.deptno%type;
 begin
 insert into emp(empno,ename,sal,deptno) values(&a,&b,&c,&d);
 dbms_output.put_line('Values inserted');
exception
when dup_val_on_index then
dbms_output.put_line('the entered number already in the table');
 end;
```

3. **Value error:** This exception will be raised if the data types are not matching.

```
declare
 a number;
 b number;
 c number;
 d number;
 begin
 select ename,sal,deptno into b,c,d from emp where empno=&a;
 dbms_output.put_line(b||' ' ||c||' '||d);
Exception
when value_error then
dbms_output.put_line('type not mathced here');
 end;
```

OUTPUT:

Enter value for a: 7788

old 7: select ename,sal,deptno into b,c,d from emp where empno=&a;

new 7: select ename,sal,deptno into b,c,d from emp where empno=7788;

type not mathced here

PL/SQL procedure successfully completed.

```
-- CURSOR_ALREADY_OPEN
            BEGIN
             open c;
             open c;
            EXCEPTION
             when cursor_already_open then
                    dbms_output.put_line('Cursor is already opened');
            END;

            -- INVALID_CURSOR
            BEGIN
             close c;
             open c;
             close c;
             close c;
             EXCEPTION
             when invalid_cursor then
                    dbms_output.put_line('Cursor is already closed');
             END;
```

User Defined Exceptions:

declare

```
a number:=&a;
bhargav exception;
begin
if(a=10) then
dbms_output.put_line('it is positive');
end if;
if (a=0) then
raise bhargav;
end if;
exception
when bhargav then
 dbms_output.put_line('a is nuetral');
end;
```

- Database Triggers in PL/SQL
  - Types of Triggers
  - Row Level Triggers
  - Statement Level Triggers
  - DDL Triggers

What is a Trigger?

A trigger is a pl/sql block structure which is fired when a DML statements like Insert, Delete, Update is executed on a database table. A trigger is triggered automatically when an associated DML statement is executed.

Syntax of 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)
BEGIN
  --- sql statements
END;
```

- CREATE [OR REPLACE] TRIGGER trigger_name - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.
- {BEFORE | AFTER | INSTEAD OF } - This clause indicates at what time should the trigger get fired. i.e for example: before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- {INSERT [OR] | UPDATE [OR] | DELETE} - This clause determines the triggering event. More than one triggering events can be used together

separated by OR keyword. The trigger gets fired at all the specified triggering event.

- [OF col_name] - This clause is used with update triggers. This clause is used when you want to trigger an event only when a specific column is updated.

- CREATE [OR REPLACE] TRIGGER trigger_name - This clause creates a trigger with the given name or overwrites an existing trigger with the same name.

- [ON table_name] - This clause identifies the name of the table or view to which the trigger is associated.

- [REFERENCING OLD AS o NEW AS n] - This clause is used to reference the old and new values of the data being changed. By default, you reference the values as :old.column_name or :new.column_name. The reference names can also be changed from old (or new) to any other user-defined name. You cannot reference old values when inserting a record, or new values when deleting a record, because they do not exist.

- [FOR EACH ROW] - This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed(i.e.statement level Trigger).

- WHEN (condition) - This clause is valid only for row level triggers. The trigger is fired only for rows that satisfy the condition specified.

For Example: The price of a product changes constantly. It is important to maintain the history of the prices of the products.

We can create a trigger to update the 'product_price_history' table when the price of the product is updated in the 'product' table.

1) Create the 'product' table and 'product_price_history' table

```
CREATE TABLE product_price_history
(product_id number(5),
product_name varchar2(32),
supplier_name varchar2(32),
unit_price number(7,2) );


CREATE TABLE product
```

(product_id number(5),

product_name varchar2(32),

supplier_name varchar2(32),

unit_price number(7,2));

2) Create the price_history_trigger and execute it.

```
CREATE or REPLACE TRIGGER price_history_trigger
BEFORE UPDATE OF unit_price
ON product
FOR EACH ROW
BEGIN
INSERT INTO product_price_history
VALUES
(:old.product_id,
 :old.product_name,
 :old.supplier_name,
 :old.unit_price);
END;
/
```

3) Lets update the price of a product.

```
UPDATE PRODUCT SET unit_price = 800 WHERE product_id = 100
```

Once the above update query is executed, the trigger fires and updates the 'product_price_history' table.

4)If you ROLLBACK the transaction before committing to the database, the data inserted to the table is also rolled back.

Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.

1) Row level trigger - An event is triggered for each row upated, inserted or deleted.

2) Statement level trigger - An event is triggered for each sql statement executed.

PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

1) BEFORE statement trigger fires first.

2) Next BEFORE row level trigger fires, once for each row affected.

3) Then AFTER row level trigger fires once for each affected row. This events will alternates between BEFORE and AFTER row level triggers.

4) Finally the AFTER statement level trigger fires.

For Example: Let's create a table 'product_check' which we can use to store messages when triggers are fired.

```
CREATE TABLE product
(Message varchar2(50),
 Current_Date number(32)
);
```

Let's create a BEFORE and AFTER statement and row level triggers for the product table.

1) BEFORE UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' before a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER Before_Update_Stat_product
BEFORE
UPDATE ON product
Begin
INSERT INTO product_check
Values('Before update, statement level',sysdate);
END;
/
```

2) BEFORE UPDATE, Row Level: This trigger will insert a record into the table 'product_check' before each row is updated.

```
CREATE or REPLACE TRIGGER Before_Upddate_Row_product
BEFORE
UPDATE ON product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values('Before update row level',sysdate);
END;
/
```

3) AFTER UPDATE, Statement Level: This trigger will insert a record into the table 'product_check' after a sql update statement is executed, at the statement level.

```
CREATE or REPLACE TRIGGER After_Update_Stat_product
AFTER
UPDATE ON product
BEGIN
INSERT INTO product_check
Values('After update, statement level', sysdate);
End;
/
```

4) AFTER UPDATE, Row Level: This trigger will insert a record into the table 'product_check' after each row is updated.

```
CREATE or REPLACE TRIGGER After_Update_Row_product
AFTER
insert On product
FOR EACH ROW
BEGIN
INSERT INTO product_check
Values('After update, Row level',sysdate);
```

```
END;
/
```

Now lets execute a update statement on table product.

```
UPDATE PRODUCT SET unit_price = 800
WHERE product_id in (100,101);
```

Lets check the data in 'product_check' table to see the order in which the trigger is fired.

```
SELECT * FROM product_check;
```

Output:

Mesage	Current_Date
Before update, statement level	26-Nov-2008
Before update, row level	26-Nov-2008
After update, Row level	26-Nov-2008
Before update, row level	26-Nov-2008
After update, Row level	26-Nov-2008
After update, statement level	26-Nov-2008

The above result shows 'before update' and 'after update' row level events have occured twice, since two records were updated. But 'before update' and 'after update' statement level events are fired only once per sql statement.

The above rules apply similarly for INSERT and DELETE statements.

How To know Information about Triggers.

We can use the data dictionary view 'USER_TRIGGERS' to obtain information about any trigger.

The below statement shows the structure of the view 'USER_TRIGGERS'

```
DESC USER_TRIGGERS;
```

NAME	Type
TRIGGER_NAME	VARCHAR2(30)

TRIGGER_TYPE	VARCHAR2(16)
TRIGGER_EVENT	VARCHAR2(75)
TABLE_OWNER	VARCHAR2(30)
BASE_OBJECT_TYPE	VARCHAR2(16)
TABLE_NAME	VARCHAR2(30)
COLUMN_NAME	VARCHAR2(4000)
REFERENCING_NAMES	VARCHAR2(128)
WHEN_CLAUSE	VARCHAR2(4000)
STATUS	VARCHAR2(8)
DESCRIPTION	VARCHAR2(4000)
ACTION_TYPE	VARCHAR2(11)
TRIGGER_BODY	LONG

This view stores information about header and body of the trigger.

SELECT * FROM user_triggers WHERE trigger_name = 'Before_Update_Stat_product';

The above sql query provides the header and body of the trigger 'Before_Update_Stat_product'.

You can drop a trigger using the following command.

DROP TRIGGER trigger_name;

First create one student table

Suppose we have a follwing table.

SQL> select * from student;

NO	NAME	MARKS
-----	-------	----------

1	a	100
2	b	200
3	c	300
4	d	400

Also we have triggering_firing_order table with firing_order as the field.

```
CREATE OR REPLACE TRIGGER TRIGGER1
    before insert on student
BEGIN
    insert into trigger_firing_order values('Before Statement Level');
END TRIGGER1;

CREATE OR REPLACE TRIGGER TRIGGER2
    before insert on student
    for each row
BEGIN
    insert into trigger_firing_order values('Before Row Level');
END TRIGGER2;

CREATE OR REPLACE TRIGGER TRIGGER3
   after insert on student
BEGIN
   insert into trigger_firing_order values('After Statement Level');
END TRIGGER3;

CREATE OR REPLACE TRIGGER TRIGGER4
   after insert on student
```

```
    for each row
BEGIN
    insert into trigger_firing_order values('After Row Level');
END TRIGGER4;
```

Output:

```
SQL> select * from trigger_firing_order;

no rows selected

SQL> insert into student values(5,'e',500);

1 row created.

SQL> select * from trigger_firing_order;

FIRING_ORDER
--------------------------------------------
Before Statement Level
Before Row Level
After Row Level
After Statement Level

SQL> select * from student;

        NO  NAME    MARKS
        ---- -------- ----------
         1       a         100
```

2	b	200
3	c	300
4	d	400
5	e	500

- Packages in PL/SQL
  - Creating PACKAGE Specification and PACKAGE Body
  - Private and Public Objects in PACKAGE

A package will have two mandatory parts –

1. Package specification
2. Package body or definition

Package Specification

- The specification is the interface to the package.
- It just DECLARES the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.
- In other words, it contains all information about the content of the package, but excludes the code for the subprograms.
- All objects placed in the specification are called public objects.
- Any subprogram not in the package specification but coded in the package body is called a private object.

Example:

```
CREATE PACKAGE cust_sal AS
   PROCEDURE find_sal(c_id customers.id%type);
END cust_sal;
/
```

Package Body

The package body has the codes for various methods declared in the package specification and other private declarations, which are hidden from the code outside the package.

Example:

```
CREATE OR REPLACE PACKAGE BODY cust_sal AS

   PROCEDURE find_sal(c_id customers.id%TYPE) IS
   c_sal customers.salary%TYPE;
   BEGIN
```

```
    SELECT salary INTO c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line('Salary: '|| c_sal);
  END find_sal;
END cust_sal;
/
```

The package elements (variables, procedures or functions) are accessed with the following syntax −

```
package_name.element_name;
```

Example:

```
DECLARE
  code customers.id%type := &cc_id;
BEGIN
  cust_sal.find_sal(code);
END;
/
```

Another Example for packages:

```
CREATE OR REPLACE PACKAGE c_package AS
   -- Adds a customer
   PROCEDURE addCustomer(c_id   customers.id%type,
   c_name customers.Name%type,
   c_age  customers.age%type,
   c_addr customers.address%type,
   c_sal  customers.salary%type);

   -- Removes a customer
   PROCEDURE delCustomer(c_id  customers.id%TYPE);
   --Lists all customers
   PROCEDURE listCustomer;

END c_package;
/

CREATE OR REPLACE PACKAGE BODY c_package AS
   PROCEDURE addCustomer(c_id  customers.id%type,
      c_name customers.Name%type,
      c_age  customers.age%type,
      c_addr  customers.address%type,
      c_sal   customers.salary%type)
   IS
   BEGIN
      INSERT INTO customers (id,name,age,address,salary)
         VALUES(c_id, c_name, c_age, c_addr, c_sal);
   END addCustomer;
```

```
  PROCEDURE delCustomer(c_id   customers.id%type) IS
  BEGIN
    DELETE FROM customers
    WHERE id = c_id;
  END delCustomer;

  PROCEDURE listCustomer IS
  CURSOR c_customers is
    SELECT  name FROM customers;
  TYPE c_list is TABLE OF customers.Name%type;
  name_list c_list := c_list();
  counter integer :=0;
  BEGIN
    FOR n IN c_customers LOOP
    counter := counter +1;
    name_list.extend;
    name_list(counter) := n.name;
    dbms_output.put_line('Customer(' ||counter|| ')'||name_list(counter));
    END LOOP;
  END listCustomer;

END c_package;
/

DECLARE
  code customers.id%type:= 8;
BEGIN
```

```
   c_package.addcustomer(7, 'Rajnish', 25, 'Chennai', 3500);
   c_package.addcustomer(8, 'Subham', 32, 'Delhi', 7500);
   c_package.listcustomer;
   c_package.delcustomer(code);
   c_package.listcustomer;
END;
/
```