PL/SQL

- It is a programming language which is used to define our own logic
- It is used to execute block of statements at a time, and increase the performance
- It supports variables, conditional statements, and loops
- It supports object oriented programming and supports composite data type
- It supports error handling mechanism

1. BLOCK

- It is one of the areas which is used to write a programmic logic.
- The block has 3 sections
  - Declaration section
  - Executable section
  - Exception section

a. Declaration section

- It is one of the section which is used to declare variables, cursors and lexceptions and so on.
- It is optional section

a. Executable section

- It is one of the section used to write coding
- It is mandatory section

b. Exception section

- It is one of the section which is used to handle errors at run time
- It is optional section

➡ There are two types of blocks are supported by pl/sql

- Anonymous block
- Named block

a. Anonymous block

- These block doesn't have name and also not stored in database
- SYNT: declare

```
----------
Begin
----------
----------
End;
```

EX:

```
BEGIN
Dbms_output.put_line ('welcome to oracle');
End;
```

b. Named blocks

- These blocks are having a name and also stored in database
- EX: procedures, functions, packages, and triggers

➡ **Variables:**

- It is one of the memory location which is used to store data
- Generally we declare the variable in declaration section
- These are supported default and not null
- Synt: variable_name datatype(size);
- EX: declare
  A number (5);
  B number (5) not null:=10;
  C number (5) default 10;

EX1: declare
A varchar (5);
Begin
A: = 'hello e bs';
Dbms_output.put_line (a);
End;

➡ **Storing a value into variable**

Using assigning operator (: =) we storing a value into variable

Synt: a: =50;

➡ **Display message or variable value**

We have pre-defined package which is used to display the message or value in the program

EX: Dbms_output.put_line ('message');
Dbms_output.put_line (a);

➡ **Select --------- into --------- clause**

This clause is used to retrieve the data from table and storing into pl/sql variables

EX: select col1,col2 into var1, var2;

1. DATA TYPES

- % TYPE
- % ROW TYPE
- PL/SQL RECORD

1. %TYPE

- It is one of the data type, which is used to assign the column datatype to variable
- It is used to store one value at time
- It is not possible to hold more than one column values or row values

Synt: variable_name table_name.colum_name%type

EX-1: declare
Vno emp.empno%type: =:no;
Vname emp.ename%type;
Begin
Select ename into vname from emp where empno=vno
Dbms_output.put_line(vname);
End;

2. % ROW TYPE

- It is one of the datatype which is used to assign all the column datatypes of a table to a variable
- It holds entire record of the same table
- Each of the time it override one record only
- It is not possible to capture the more than one table data

Synt: v_name tab_nam%rowtype

EX-1: declare
Vrow emp%rowtype;
Vno emp.empno%type: =:no;
Begin
Select * into vrow from emp where empno=vno;
Dbms_output.put_line ('details:'vrow.ENAME);
End;

3. RECORD TYPE OR PL/SQL RECORD

- It is one of the user defined temporary datatype which is used to store more than one table data (or) to assign more than one column datatypes
- They must contain atleast one element
- Pin point of data is not possible

Synt: Type typename is record (val-1 datatype, val2 datatype, ……)
Var typename

Ex: declare
Type rec is record (vname emp.ename%type
Vsal emp.sal%type
Vloc dept.loc%type) ;
Vrec rec;
Vno emp.empno%type := :no;
Begin

```
Select ename, sal, loc into vrec from emp,dept where
emp.deptno = dept.deptno and emp.empno=vno
Dbms_output.put_line(vrec.vname||','||vrec.vsal||','||vrec.
emp||','||vrec.vloc);
End;
```

2. CONTROL STATEMENTS

- IF
- IF ELSE
- ELSE IF
- CASE

1. IF condition

Synt:
```
if condition then
Statements;
End if;
```

Ex-1:
```
declare
    a number (5):=&age;
    b char (1);
begin
    if a<20 then
    b:= 'yes';
    end if;
    Dbms_output.put_line (b);
End;
```

2. IF ELSE CONDITON

Synt:
```
if condition then
Statements;
Else
Statements;
End if;
```

Ex-1:
```
declare
    A number (5);
    B char (5);
```

```
Begin
        If a< 20 then
        B:= 'yes';
        Else
        B:= 'no';
        End if;
        Dbms_output.put_line (b);
End;
```

3. ELSE IF

```
Synt:  if cond-1 then
        Statements;
        Elsif cond-2 then
        Statements;
        Elsif cond-3 then
        Statements;
        Else
        Statements;
        End if;

Ex-1: declare
        A number (4);
        B char (15);
        Begin
        If a<20 then
        B:= 'low value';
        Elsif a>=20 and a<40 then
        B:='avg value';
        Elsif a>=40 and a<=100 then
        B:='high value';
        Else
        B:='enter correct value';
        End if;
        Dbms_output.put_line (b);
End;
```

4. CASE CONDITION

```
Synt:  case (column)
       When condition then
       Statements;
       When condition then
       Statements;
       Else
       Statements;
       End case;
```

```
Ex-1   declare
           Vsal number(10):=:no;
       Begin
           Case
               When vsal<=2000 then
               Dbms_output.put_line ('low sal');
               When vsal>2000 and vsal<=7500 then
               Dbms_output.put_line ('avg sal');
               When vsal>7500 and vsal<=10,000 then
               Dbms_output.put_line ('high sal');
               Else
               Dbms_output.put_line ('enter correct sal');
           End case;
       End;
```

3. LOOPS

- SIMPLE LOOP
- WHILE LOOP
- FOR LOOP

1. SIMPLE LOOP

```
Synt:      loop
           Statements;
           End loop;
```

```
Ex-1:      begin
               Loop
```

```
        Dbms_output.put_line ('welcome to oracle');
        End loop;
End;
```

Ex-2:

```
declare
        N  number (5): =1;
Begin
        Loop
        Dbms_output.put_line (n);
        Exit when n>=10;
        N:=n+1;
        End loop;
End;
```

Ex-3

```
declare
        N number (5):=1;
Begin
        Loop
        Dbms_output.put_line (n);
        If n>=10 then
        Exit ;
        End if;
        N:=n+1;
        End loop;
End;
```

Note that n:=n+1 condition is used just before end loop

2. WHILE LOOP

```
Synt:  while (condtion)
       Loop
       Statements
       End loop;
```

```
Ex-1   declare
           N number (2):= 1;
```

```
Begin
        While n<=10
        Loop
        Dbms_output.put_line (n);
        N:=n+1;
        End loop;
End;
```

NOTE:

IN SIMPLE LOOP FIRST WE PRINT THE DATA AND THEN APPLY THE CONDITION
IN WHILE LOOP FIRST WE APPLY CONDITION AND THEN WE PRINT THE DATA

3. FOR LOOP

Synt: for variable_name in lowerbound ..outerbound

```
        Loop
        Statements;
        End loop;
```

Ex-1

```
declare
        N number (2):=1;
Begin
        For n in 1..10
        Loop
        Dbms_output.put_line (n);
        End loop;
End;
```

Ex-2

```
declare
        N number (2):=1;
Begin
        For n in reverse 1..10
        Loop
        Dbms_output.put_line (n);
        End loop;
End;
```

BIND VARIABLES / HOST VARIABLES:

These are session variables

Ex-1 variable a number

```
Declare
        B number(5):=500;
Begin
        A:=B/2;
End;

Print A;
```

4. CURSORS

- It is a buffer area which is used to process multiple records & also record by record process
- Types of cursors

➢ IMPLICIT CURSOR
➢ EXPLICIT CURSOR

➢ IMPLICIT CURSOR

- SQL statements returns a single record is called implicit cursor
- Implicit cursors operations are done by system

1. Open by the system
2. Fetch the records by the system
3. Close by the system

Ex-1
```
declare
        X emp%rowtype;
Begin
        Select * into x from emp where empno=7369;
        Dbms_output.put_line (x.empno||','||x.ename);
End;
```

➢ EXPLICIT CURSOR

- Sql statements returns a multiple records is called explicit cursor
- Operations are done by user

1. Declare by user
2. Open by user
3. Fetch records by user
4. Close by the user

Ex-1:
```
declare
        Cursor c1 is select empno, ename, sal from emp;
        Vno emp.empno%type;
```

```
        Vname emp.ename%type;
        Vsal emp.sal%type;
    Begin
        Open c1
        Fetch c1 into vno, vname, vsal;
        Dbms_output.put_line (vno||','||vname);
        Close c1;
    End;
```

o/p: it will show only one record, we should use loop to get all records

➡ **Using loop:**

```
begin
    open c1
    loop
    fetch c1 into vno, vname, vsal
    Dbms_output.put_line (vno);
    End loop
    Close c1;
End;
```

o/p: here we get all records , but with some extra records so we use cursor attributes

➡ **Multiple fetch:**

```
begin
    Open c1
    Fetch c1 into vno, vname, vsal;
    Dbms_output.put_line (vno);
    Fetch c1 into vno, vname, vsal;
    Dbms_output.put_line (vno);
    Fetch c1 into vno, vname, vsal;
    Dbms_output.put_line (vno);
    Close c1;
End;
```

o/p: we get 3 records b/c of 3 fetches.

Cursor attributes: every explicit cursor having fallowing four attributes

- %notfound
- %found
- %isopen
- %rowcount

- All cursors attributes should be used along with cursor name
  Synt: % cur_nam attributename

Note: Except % rowcount all other cursors attributes returns Boolean Value (either true or false) whereas %rowcount return number type

1. %NOT FOUND

- Returns 'invalid cursor' if cursor is declared, but not open or if cursor has been closed
- Returns 'null' if cursor is open, but fetch has not been executed
- Return 'false' if a successful fetch has been executed
- Returns 'true' if no row was returned

```
Ex-1:  declare
            Cursor c1 is select empno, ename, job from emp;
            Vno         emp.empno%type
            Vname       emp.ename%type
            Vjob        emp.job%type
       Begin
            Open c1
            Loop
            Fetch c1 into vno, vname, vjob;
            Exit when c1%notfound;
            Dbms_output.put_line (vno);
            End loop;
            Close c1;
       End;
```

2. %FOUND

- Returns 'invalid cursor ' if cursor is declared, but not open or if cursor is closed
- Returns 'null' if cursor is open, but fetch has not been executed
- Returns 'true' if a successfull fetch has been done
- Returns 'false' if no rows are returned

```
Ex-1:  declare
            Cursor c1 is select * emp;
            X emp%rowtype;
       Begin
            Open c1
            Loop
            Fetch c1 into x;
```

```
        If c% found then
        Dbms_output.put_line (x.empno);
        Else
        Exit;
        End if;
        End loop;
        Close c1;
    End;
```

3. %IS OPEN

- Is to determine whether the cursor is open or not

Ex-1:
```
declare
        Cursor c1 is select * emp;
        Vrow emp%rowtype;
Begin
        Open c1
        If c1%isopen then
        Dbms_output.put_line ('cursor is open')
        Loop
        Fetch c1 into vrow
        Dbms_output.put_line (vrow.empno);
        Exit when c1%notfound
        End loop;
        Close c1;
        Else
        Dbms_output.put_line ('cursor not open');
        End if;
End;
```

4. % ROWCOUNT

- Returns 'invalid cursor' if cursor is declared, but not open
- Returns the number of rows fetched by the cursor

Ex-1:
```
declare
        Cursor c1 is select * from emp
        Vrow emp%rowtype;
Begin
```

```
Open c1
Loop
Fetch c1 into vrow;
Dbms_output.put_line (vrow.ename);
Exit when c1%notfound;
End loop;
Dbms_output.put_line ( 'total no.of emp='||c1%rowcount);
Close c1;
End;
```

SUB TOPIC IN CURSORS

- PARAMETER CURSOR
- CURSOR WITH FOR LOOP
- NESTED CURSOR WITH FOR LOOP
- CURSOR WITH DML OPERATIONS

a. PARAMETER CURSOR

i. We pass the parameter in the cursor

Ex-1:

```
declare
        Cursor c1 (p_dno number) is select * from emp where
        deptno=:p_dno
        Vrow emp%rowtype;
Begin
        Open c1 (10); / open c1 (:p_dno);
        Loop
        Fetch c1 into vrow
        Dbms_output.put_line (vrow.empno);
        Exit when c1%notfound;
        End loop;
        Dbms_output.put_line ('total no.of emp='c1%rowcount);
        Close c1;
End;
```

b. CURSOR WITH FOR LOOP

- In cursor for 'for loop' no need to open, fetch, close the cursor.
- For loop itself automatically will perform these functions

Ex-1:

```
declare
        Cursor c1 is select * from emp;
```

```
        Vrow emp%rowtype;
    Begin
        For vrow in c1
        Loop
        Dbms_output.put_line (vrow.ename);
        End loop;
    End;
```

Here in cursor for loop no need to declare a variable

c. NESTED CURSOR WITH FOR LOOP

```
Ex-1:  declare
        Cursor c1 is select *  from dept
        Cursor c2(d_no number) is select * from emp where
deptno=d_no;
    Begin
        For i in c1
        Loop
        Dbms_output.put_line (i.deptno);
        For j in c2 (i.deptno)
        Loop
        Dbms_output.put_line (j.empno||','||j.ename);
        End loop;
        End loop;
    End ;
```

d. CURSOR WITH DML OPERATIONS

i. Insert

1. Create an empty table

```
Ex-1:  create emp_c as select * from emp where 1=2;

        Declare
            Cursor c1 is select * from emp;
        Begin
            For i in c1
            Loop
            Inset into emp_c values (i.empno,i.ename…);
```

```
        End loop;
End;
```

ii. Update

1. Update salary for manager in 10 department

```
Ex-1  declare
        Cursor c1 is select * from emp_c;
      Begin
        For i in c1
        loop
        If i.job='clerk' then
        Update emp_c set sal=sal+100 where
        Empno=i.empno;
        Elsif i.job='salesman' then
        Update emp_c set sal=sal+1000 where
        Empno=i.empno;
        End if;
        End loop;
      End;
```

IF THIS CONDITION IS NOT USED, THEN LAST RECORD IS ONLY UPDATED

iii. Delete

```
Ex-1  declare
        Cursor c1 is select * from emp_c;
      Begin
        For i in c
        Loop
        Delete from emp_c where empno=i.empno;
        End loop;
      End;
```

REF CURSOR/ DYNAMIC CUROSR

- Ref cursors are user define types, which is used to process multiple records and also record by record process
- Generally through the static cursors we are using one select statement at a time for single active set area, where as in ref cursor we are executing no.of select statements dynamically for single active set area
- By using ref cursor we can return large amount of data from oracle database into client application

- **There are two type of ref cursors**
  - **Strong ref cursor**
  - **Weak ref cursor**

Strong ref cursor

- **It is one of the ref cursor which is having return type**

Weak ref cursor

- **It is one of the ref cursor which does not have a return type**

Note: in ref cursor we are executing select statements using open…. For statement

Ex-1

```
Declare
        Type t1 is ref cursor;
        X           t1;
        Vrow        emp%rowtype;

Begin
        Open x for select * from emp where sal>2000;
        Loop
        Fetch x into vrow;
        Exit when x%notfound;
        Dbms_output.put_line (vrow.ename);
        End loop;
End;
```

We can't use **for loop** for ref cursor

Ex-2:

```
Declare
        Type t1 is ref curosor;
        V_rc        t1;
        V_x         emp%rowtype;
        V_y         dept%rowtype;
        P_no        number:= &pno
Begin
        If p_no=1 then
        Open v_rc for select * from emp;
        Loop
        Fetch v_rc into x;
        Exit when v_rc%notfound;
        Dbms_output.put_line (x.ename);
```

```
End loop;
Elsif p_no=2 then
Open v_rc for select * from dept;
Loop
Fetch v_rc into y;
Exit when v_rc%notfound;
Dbms_output.put_line (y.dname);
End loop;
End if;
End;
```

5. EXCEPTIONS

- Exceptions are the activities which are used to handle errors at run time
- There are 3 types of exceptions
  - i. Pre-defined exceptions
  - ii. User defined exceptions
  - iii. Non predefined exceptions

1. PRE DEFINED EXCEPTIONS

i. It is one of the exception which are defined by oracle

ii. These are 20 exceptions available

Synt:
```
when exceptin1 then
    Statements;
    When exception2 then
    Statements;
    When others then
    Statements;
```

iii. Predefined exceptions are

1. No_data_found
2. Too_many_rows
3. Invalid_cursor
4. Cursor_already_open
5. Invalid_number
6. Value_error
7. Zero_divide
8. Others-++

1) No data found:

a. When a pl/sql block contains select-----into clause and also if requested data not available in table oracle server returns an error.
b. Error is ora.01403:no data found
c. To handle this error we are using no_data_found exception

```
Ex-1:   declare
        V_ename   varchar (20);
        V_sal     number (10);
    Begin
        Select ename, sal into v_ename, v_sal from emp where
        Empno = :no;
        Dbms_output.put_line (v_ename||','||v_sal);
        End;

Ex-2:   declare
        V_ename   varchar (20);
        V_sal     number (10);
    Begin
        Select ename, sal into v_ename, v_sal from emp where
        Empno = :no;
        Dbms_output.put_line (v_ename||','||v_sal);
    Exceptions
        When no_data_found then
        Dbms_output.put_line ('employees doesn't exist');
    End;
```

- **Too_many_rows**

1. When a select ----into clause try to return more than one record or more than one values then oracle server will return an error
2. Error is ora.01422: exact fetch return more than requested number of rows
3. To handle these errors we use too_many_rows exception

```
Ex-1:   declare
        V_ename   varchar (20);
        V_sal     number (10);
    Begin
        Select ename, sal into v_ename, v_sal from emp where
```

```
                Empno = :no;
                Dbms_output.put_line (v_ename||','||v_sal);
                End;

Ex-2:           declare
                V_ename     varchar (20);
                V_sal       number (10);
          Begin
                Select ename, sal into v_ename, v_sal from emp where
                Empno = :no;
                Dbms_output.put_line (v_ename||','||v_sal);
          Exceptions
                When too_many_rows then
                Dbms_output.put_line ('program returns more than
     one row');
          End;

Ex-3  declare
          Vrow emp%rowtype
      Begin
          Select * into vrow from emp where deptno=:dno
          Dbms_output.put_line (vrow.ename);
      Exception
          When no_data_found
          Dbms_output.put_line ('emps doesn't exist in this dept')
          When too_many_rows then
          Dbms_output.put_line ('program returns more than one
      row');
      End;
i/p: 40 then o/p: emps doesn't exist in this dept
i/p: 10 then o/p: program returns more than one row
```

- **Invalid cursor:**
  1. Whenever we are performing invalid operations on the cursor, server will return an error i.e., if you are try to close the cursor with opening it.
  2. Error is ora. 01001 : invalid cursor
  3. To handle this error we use invalid_cursor exception

- **Cursor already open:**

1. When we try to reopen the cursor without closing the cursor oracle server returns an error
2. Error is ora.06511: cursor is already open
3. To handle this error we use cursor_already_open.

- **Invalid number:**
  1. Whenever we try to convert string type to number type server return error.
  2. Error is ora.01722 invalid number.
  3. To handle this error we are using invalid_number exception

```
Ex-1:  begin
            Insert into emp (empno,ename) values ('abcd','abcd');
       Exception
            When invalid_number then
            Dbms_output.put_line ('plz check the datatype')
       End;
```

- **Value_error:**
  1. Whenever we are try to convert string type to number type based on the condition then oracle server returns an error.
  2. Whenever we are try to store large amount of data, than the specified datatype size then oracle server returns same error.
  3. Error is ora.06502: numeric or value error :character to number conversion error
  4. To handle this error we use value_error

```
Ex-1: (pt-1)

       Declare
            A      number(5);
       Begin
            A:=':x'+':y';
            Dbms_output.put_line (a);
       Exception
            When value_error then
            Dbms_output.put_line ('plz enter values only');
       End;
```

If u give any char to either x or y we get this error

```
Ex-2: (pt-2)
```

```
Declare
        A       number(2);
Begin
        A:=':x'+':y';
        Dbms_output.put_line (a);
Exception
        When value_error then
        Dbms_output.put_line ('give no.of addition < 1');
End;
```

- **Zero_error**
  1. Whenever we try to divide by zero then oracle server returns an error
  2. Error is ora-01476: divisor is equal to zero
  3. To handle this error we are using zero_divide exception

```
Ex-1:  declare
        A       number (5):=:a;
        B       number (5):=:b;
        C       number (5):=:c;
Begin
        A:=b/c;
        Dbms_output.put_line (a);
Exception
        When Zero_divide then
        Dbms_output.put_line ('enter c value >1');
End;
```

EXCEPTION PROPOGATION

- Exceptions are also raised in
  - ✓ Declaration section
  - ✓ Executable section
  - ✓ Exception section

- If the exceptions are raised in executable section then those exceptions are handled using either an outer block
- Whereas if exception are raised in declaration section or in exception section those exceptions are handled using outer blocks only

```
                Ex-1
Ex-1:  begin
       Declare
              A      number (3): =’abcde’;
              Begin
                     A:= ‘abcd’;
                     Dbms_output.put_line (a);
              Exception
                     When value_error then
                     Dbms_output.put_line (‘a    value is more than 3 char’);
              End;
       exception
              When value_error then
              Dbms_output.put_line (‘plz check the size in variable’);
       End;
```

2. USER DEFINED EXCEPTIONS

- We can also create our exception and also raise them whenever it is necessary, these are called userdefined exceptions.
- These exceptions are divided into 3 steps
  - Declare exception
  - Raise exception
  - Handle exception

1. In declare section of pl/sql program we are defining our own exception name using exception type

   Ex: declare
   A exception;

2. Whenever it is required, raise user defined exception either in executable section or in exception section ,in this case we are using raise keyword

   Ex: declare
   A exception;
   Begin
   Raise a;

```
End;
```

3. **We can handle user defined exceptions as same as predefined exception using predefined handler**

Ex-1:
```
declare
        A       exception;
Begin
        If to_char (sysdate,'dy')='sun' then
        Raise a;
        End if;
Exception
        When a then
        Dbms_output.put_line ('exception raised`today');
End;
```

Ex-2:
```
declare
        V_sal number (10);
        A exception;
Begin
        Select sal into v_sal from k where empno=7902;
        If v_sal>2000 then
        Raise a;
        Else
        Update k set sal=sal+100 where empno=7902;
        End if;
Exception
        When a then
        Dbms_output.put_line ('salary already high');
End;
```

RAISING PREDEFINED EXCEPTIONS

We can also raising predefined exception by using raise statement

Ex-1:
```
declare
        Cursor c1 is select * from emp where job=&job;
        Vrow emp%rowtype;
Begin
        Open c1;
```

```
loop
Fetch c1 into vrow;
if c1%found then
Dbms_output.put_line (vrow.eno);
Else
Raise no_data_found;
End if;
End loop;
Close c1;
Exception
When no_data_found then
Dbms_output.put_line ('your job not available');
End;
```

ERROR TRAPPING FUNCTION

There are two error trapping functions supported by oracle

1. Sql code: it returns number
2. Sql errm: it return number with message

Ex:

```
Declare
V_sal number (10);
Begin
Select sal into V_sal from emp where empno=7368;
Dbms_output.put_line (sqlcode);
Dbms_output.put_line (sqlerrm);
End;
```

RAISE APPLICAION ERROR

- If you want to display your own user defined exception number and exception message then we use this raise application error
- Synt: Raise_application_error (error_number, error_message)

Error_number: it is used to give error number between -20000 to -20999
Error_message: it is used to give message up to 512 characters

Ex:

```
Declare
V_sal number (10);
A exception;
```

```
Begin
Select sal into V_sal from emp where empno=7369
If v_sal>2000 then
Raise a;
Else
Update emp set sal=v_sal+100 where empno=7369;
End if;
Exception
When a then
Raise_application_error (-20999, 'sal already high');
End;
```

3. NON PREDEFINED/ UN NAMED EXCEPTIONS

- If you want to handle other than oracle 20 predefined errors we are using unnamed method
- Because oracle defined exception names for regular accuring errors other than 20 they are not defined exception names
- In this case we are providing exception names and also associate this exception name with appropriate error no using exception_init function.
- Synt: pragma exception_init (userdefined_exception_name, error_number);
- Here pragma is compiler directive i.e., at the time of compilation only pl/sql runtime engine associate error number with exception name.
- This function is used to declare section of pl/sql block.

Ex:

```
declare
V_no number (10);
E exception;
Pragma exception_init(e,-2291);
Begin
Select empno into v_no from emp where empno=&no;
Dbms_output.put_line(v_no);
Exception
When e then
Dbms_output.put_line('pragma error');
End;
```

5. SUBPROGRAMS

- Sub programs are named pl/sql blocks which are used to solve particular tasks
- There are two types of sub programs supported by oracle
  i. Procedures

ii. Functions

1. PROCEDURES

- Procedures may or may not return a value
- Procedures return more than one value while using parameter
- Procedure can only execute in 3 ways
    1. Anonymous block
    2. Exec
    3. Call
- Procedure cannot execute in select statement
- Procedure internally having one time compilation process
- Procedures are used to improve the performance of business application
- Every procedure having two parts

  a. procedure specification

  in procedure specification we are specifying name of the procedure and types of the parameter

  b. procedure body

  in procedure body we are solving the actual task

Ex-1:
```
create or replace procedure p1 (p_empno number) is
V_ename     number (5);
V_sal       number (10);
Begin
Select ename,sal into v_ename,v_sal from emp where empno=p_empno;
Dbms_output.put_line (v_ename);
End;
```

Execute the procedure the 3 ways

Method-1:
```
begin
P1 (7902);
End;
```

Method-2:
```
exec p1(7902);
```

Method-3:
```
call p1 (7902);
```

Ex-2:
```
create or replace procedure p111 (p_empno number) is
Cursor c1 is select * from emp where empno=p_empno;
```

```
V       emp%rowtype;
Begin
        Open c1;
        loop
        Fetch c1 into v;
        Exit when c1%notfound;
        Dbms_output.put_line(v.empno);
        End loop;
        Close c1;
End p111;
```

PROCEDURE WITH PARAMETERS:

Parameters are used to pass the value into procedures and also return values from the procedure

In this case we must use two types of parameters

a. Formal parameters
b. Actual parameter

a. FORMAL PARAMETERS

- formal parameters are defined in procedure specification
  in formal parameter we are defining parameter name, mode of the parameter
- there are three types of modes supported by oracle
  - IN MODE
  - OUT MODE
  - INOUT MODE

- **IN MODE**
  - By default procedure parameters having 'in parameter'
  - IN mode is used to pass the parameters into body
  - This mode behaves like a constant in procedure body, through this IN Mode we can also pass default values using := operater

```
EX: create or replace procedure p1 (dno in number
                                    Dname   in varchar
                                    Loc in varchar)
        Is
        Begin
                Insert into dept values (dno, dname, loc);
```

```
        Dbms_output.put_line ('records are inserted');
    End;
```

- There are three type of execution with in parameter
  - Position notations: exec p1 (1,'a','b')
  - Named notation: exec p1 (dname=>'x',loc=>'y',dno=>2);
  - Mixed notations: exec p1 (1,dname=>'m', loc=>'n')

- **OUT MODE**
  - This mode is used to return a value from the procedure body
  - Outmode internally behave like a uninitialized variable in procedure body

```
Ex-1:  create or replace p1 (a in number,
                             B out number);
       Is
       Begin
            B:=a+a;
            Dbms_output.put_line (b);
       End;
```

Note:

In oracle if a subprogram contain out, inout parameter these subprograms are executed using fallowing two methods

Method-1 using bind variables ------this method is better

Method-2 using anonymous block

```
Method-1
    Variable b number;

    Exec p1 (10, :b);


    Method-2
            Declare
                B number;
            Begin
                P1 (5, b);
                Dbms_output.put_line (b);
            End;
```

Q: DEVELOP A PROGRAM FOR PASSNG AN EMPLOYEE NAME AS IN PARAMETER RETURN SALRY OF THAT EMPLOYEE USITNG OUT PARAMETER FORM EMP TABLE

Prgm:

```
Create or replace procedure p1 (p_ename varchar,
                                P_sal out number)
Is
Begin
    Select sal into p_sal from emp where ename=p_ename;
    Dbms_output.put_line (p_sal);
End
```

Exec:

```
Variable a number;

Exec p1 ('king', :a);
```

Q: develop a program for passing deptno as a parameter return how many employees are working in a deptno from emp table

Progm

```
Create or replace procedure p2 (pdno number, pcount out number) IS
BEGIN
    Select count (*) into pcount from emp where deptno=pdno
    Dbms_output.put_line (pcount);
End;
```

Exec:

```
Exec p2 (10,:a);
```

- **INOUT MODE**
  - **This mode is used to pass the values into subprograms and return the values from subprogram**

Ex-1:

```
Create or replace procedure p3 (a in out number) is
Begin
    A:=a+a;
    Dbms_output.put_line(a);
```

```
End;

Variable a number;
Exec :a:=10;
Exec p1 (:a);
```

2. FUNCTIONS

- Function is a named pl/sql block which is used to solve particular task and by default functions returns a single value
- Function is allow to write multiple return statements but it execute only first return statement
- Function can execute in 4 ways
  - Anonymous block
  - Select statement
  - Bind variable
  - Exec
- Function also have two parts
  - Function specification
  - Function body
- In fun specification we are specifying name of the function and type of the parameter where as in fun body we are solving the task

Ex-1

Prgm:
```
Create or replace function f1 (a varchar)
    Return varchar2
    Is
    Begin
    Return a;
    End;
```

Executions:

1. anonymous block

```
Declare
        A varchar(10);
Begin
        A:=f1('welcome');
        Dbms_output.put_line(a);
End;
```

2. bind variable

```
Variable a number
```

```
Begin
:a=:f1('welcome');
End;
```

3. exec

```
Exec dbms_output.put_line(f1('welcome'));
```

4. select statemet

```
Select f1('welcome') from dual
```

Q: EVEN OR ODD NUMBER

PRGM:

```
Create or replace function f1 (a number) return varchar is
Begin
    If mod(a,2)=0 then
    Return 'even number';
    Else
    Return 'odd  number';
    End if;
End;

Exec:  select f2(4) from dual;
```

Q: ADDITION OF NUMBERS

PRGM

```
Create or replace function f3 return number is
A       number(10):=10;
B       number (10):=20;
C       number (10);
Begin
    C:=a+b;
    Return (c);
End;

Exec:  select f3 from dual;
```

Q: ADDITION OF NUMBERS WITH PARAMETERS

PRGM

```
Create or replace function f3 (a number, b number) return number is
C       number (10);
Begin
        C:=a+b;
        Return(c);
End;
```

Note: without "return c;" function is create, but execution is not possible
With just "return "function created with compilation errors

Ex:

Prgm Create or replace function f4 (a number, b number) return number is

```
C       number (10);
D       number (10);
Begin
        C:=a+b;
        D:=a-b;
        Return(c);
        Return(d);
End;
```

Exec: select f4(20,10) from dual;

o/p: func(20,10)
30 → first return
Suppose if we want to return multiple values we use out parameters

Ex:

Prgm create or replace function f4 (p_deptno in number,p_dname out varchar2, p_loc out varchar)

```
Return varchar is
Begin
        Select dname,loc into p_dname, p_loc from dept
        Where deptno=p_deptno;
        Return p_dname;
End;
```

Exec: bind variables

```
Variable    A    varchar;
Variable     b   varchar;
Variable    c    number;

Begin
:c:=f4(10,:a,:b);
End;

Print b,      print a;
```

Q: WRITE A PL/SQL STORED FUNCTION FOR PASSING EMPNO AS PARAMETER, RETURN GROSS SALARY FROM EMP TABLE BASED ON FALLOWING CONDITION?

CONDITION=> GROSS=BASIC+HRA+DA+PF

HRA =10% OF SAL
DA =20% OF SAL
PF =30% OF SAL

PRGM

```
Create or replace function f5 (p_empno) return number is
        V_sal        number (10);
        V_hrda       number (10);
        V_da         number (10);
        V_pf         number (10);
        V_gross      number (10);
Begin
        Select sal into v_sal from emp where empno=p_empno;
        V_hrda:=v_sal*0.1;
        V_da:=v_sal*0.2;
        V_pf:=v_sal*0.3;
        V_gross:=v_sal+ V_hrda+ v_da+ v_pf;
        Return (V_gross);
End;
```

Exec: select f45(7369) from dual;

Q: WRITE A PL/SQL STORED FUNCTION FOR PASSING EMPNO,DATE AS PARAMETER RETURN NUMBER OF YEARS THAT EMPLOYEE IS WORKING BASED ON DATE FROM TABLE

PRGM

```
Create or replace function f6 (p_empno number, p_date date) return number is
```

```
A number (10);
Begin
    Select month_between(p_date,hiredate)/12 into a from emp where
    empno=p_empno;
    Return(round(a));
End;
```

Exec: select f6(7369,10/3/12) from dual;

Q: WRITE A PL/SQL STORED FUNCION FOR PASSING EMPNOAS PARAMETER, CALCULATE TAX BASED ON FALLOWING CONDITIONS BY USING EMP TABLE

CONDITION:IF ANNUAL SAL>10,000 THEN TAX=10%
IF ANNUAL SAL>20,000 THEN TAX=20%
IF ANNUAL SAL>30,000 THEN TAX=30%

PRGM:

```
Create or replace function f7 (p_empno number) return number is
P_ann       NUMBER (10);
P_tax       number (10);
P_sal       number (10);
Begin
    Select sal into p_sal from emp where empno=p_empno
    P_ann:=sal*12;
    If p_ann>=10,000 and p_ann<20,000 then
    P_tax:=p_ann*0.1;
    elsIf p_ann>=20,000 and p_ann<30,000 then
    P_tax:=p_ann*0.2;
    elsIf p_ann>=30,000 then
    P_tax:=p_ann*0.3;
    Else
    p_tax:=0;
    end if;
    return (p_tax);
end;
```

exec: select f7(7369) from dual

6. PACKAGE

- **Packages are database objects which is used to encapsulate variables, constants, procedures, cursors, functions, types into single unit**

- Packages cannot accept parameters, cannot be nested, cannot be invoked
- Generally packages are used to improve performance of the application, because when we calling packaged subprogram first time total package automatically loaded into memory area
- Whenever we are calling subsequent subprogram, pl/sql engine calls those subprograms from memory area
- This process automatically reduces disk i/o that's why package improves performance of the application
- Package has two steps
    i. Package specification
    ii. Package body
- In package specification we are defining global data and also declare objects , subprograms where as in package body we are implementing subprograms and also package body subprogram internally behaves like a private subprogram

2. **Package specification synt:**

```
Create or replace package pac_name
Is/as
Global variable declaration;
Constant        declaration;
Cursor          declaration;
Types           declaration;
Procedure       declaration;
Function        declatation;
End;
```

3. **Package body synt:**

```
Create or replace package body pac_name
Is/as
Procedure   implementation;
Function    implementation;
End;
```

4. **Invoking packaged subprogram**
    i. Exec pak_name.proc_name(actual parameters);
    ii. Select pak_name.fun_name(actual parameters) from dual;

Note: there is no possibility to done all transactions, we can only execute one transaction at a time

Ex-1:

Spec:
```
Create or replace package pack1 is
Procedure p1;
```

```
        Procedure p2;
        End;

Body: create or replace package body pack1 is
        Procedure p1 is
        Begin
                Dbms_output.put_line('first procedure ');
        End p1;
        Procedure p2 is
        Begin
                Dbms_output.put_line('second procedure');
        End p2;
        End pack1;
```

Ex-2:

```
Spec: create or replace package pack2 is
        Procedure p1 (a number, b number);
        Procedure p2 (x number, y number);
        End;

Body: create or replace package body pack2 is
        Procedure p1 (a number, b number) is
                C number (10);
        Begin
                C:=a+b;
                Dbms_output.put_line (c);
        End p1;

        Procedure p2(x number, y number) is
                Z number (10);
        Begin
                Z:=x+y;
                Dbms_output.put_line (z);
        End p2;
        End pack2;
```

Ex-3: procedure and functions with packages

```
Spec: create or replace package pack3 is
        Procedure p1(a number, b number);
```

```
Function f1 (x number, y number) return number ;
End;
```

Body:
```
create or replace package body pack3 is
    Procedure p1 (a number, b number) is
    C number (10);
    Begin
        C:=a+b;
        Dbms_output.put_line (c);
    End p1;
    Function f1 (x number, y number) return number is
    Z number (10);
    Begin
        Z:=x*y;
        Dbms_output.put_line (z);
    End f1;
    End pack3;
```

Exec:
```
exec pack3.p1 (10, 20);
Select pack3.f1 (10, 20) from dual;
```

SUB TOPICS

- Global and local variables
- Procedure overloading
- Forward declaration

1. Global variable:

- It is one of the variable used to define in package specification and implement in package body are called global variables.

2. Local variable

- It is one of the variable which is used to define in programs (procedure, functions) and implement with in the program only

Ex:

Spec:
```
create or replace package pack4 is
    G_v    number (10):=500;
    Procedure p1;
    End;
```

Global variable

Body:
```
create or replace package body pack4 is
    Procedure p1 is
```

Local variable

```
        Z number (10);
    Begin
        z:=g_v/2;
    end p1;
    end pack4;
```

3. Procedure overloading

- Overloading refer to same name can be used for different purposed i.e., we are implementing overloading procedures through package only, those procedures having same name but different arguments

Ex:

Spec:

```
create or replace package pack5 is
    Procedure p1 (a number, b number);
    Procedure p1 (x number, b number);
    End;
```

Body:

```
create or replace package body pack5 is
    Procedure p1 (a number, b number) is
    Begin
        C:=a+b;
        Dbms_output.put_line(c);
    End p1;
    Procedure p1 (x number, y number) is
    Begin
        Z:=+x*y;
        Dbms_output.put_line (z);
    End p1;
    End pack5;
```

Exec:

```
exec pack4.p1 (10, 20);
Error: too many declarations of p1
```

Notations: exec pack4.p1 (a=>10, b=>20);

4. Forward declaration

- When we are calling procedure into another procedure then only we are using forward declaration i.e., whenever we are calling local procedure into global procedure first we must implement local procedure, before calling otherwise we use forward declaration in package body

Ex:

Spec: create or replace package pack6 is
Procedure p1; → *global procedure*
End

Body: create or replace package body pack5 is
Procedure p2; → *we defined local procedure (forward declaration)*
Procedure p1 is
Begin
P2; → *exec local procedure*
End p1;
Procedure p2 is
Begin
Dbms_output.put_line ('local procedure');
End p2;
End pack6

7. TRIGGERS

- Trigger is also same as stored procedure and also it will automatically invoked whenever 'DML' operation performed against table or view
- There are two types of triggers
  - Statement level
  - Row level
- In statement level trigger body is executed only for DML statements
- In row level trigger , trigger body is executed for each and every DML operation

Synt:

```
Create or replace trigger        trigger_name
Before/after                     trigger event
Insert/ update/ delete     on    table_name
{for each row}
{where condition }
{ declare }
Varaibles, cursors declaration
Begin
----------
End;
```

Execution order in triggers:

- Before statement level trigger
- Before row level trigger
- After row level
- After statement level

1. Statement level trigger

- In this, trigger body is executed only once for each DML statement, that's why generally statement level triggers used to define type based condition and also used to implement auditing reports. These triggers does not contain new, old qualifiers.

Q: write a pl/sql statement level trigger on emp table not to perform DML operation in Saturday and Sunday?

Prgm:

```
Create or replace trigger tr1
Before
Insert or update or delete on tt/fnd_user;
Begin
        If to_char (sysdate, 'dy') in ('sat', 'sun');
        Then
        Raise_application_error (-20123,'we can't perform on sat and sun');
        End if;
End;
```

Q: write a pl/sql statement level trigger on emp table not to perform DML operations on last day of the month

Prgm;

```
Create or replace trigger tr2
Before
Insert or update or delete on tt;
begin
        If sysdate=last_date (sysdate) then
        Raise_application_error (-2011, 'we can't perform on last day of month');
        End if;
End;
```

TRIGGER EVENT:

- If you want to define multiple condition on multiple tables then all data base systems uses trigger event.
- These are insert, update, deleting
- These are either used in statement level or row level

Q: write a pl/sql statement level trigger on emp table not to perform any DML operations in any day using trigger event

Progm:

```
Create or replace trigger tr3
Before
Insert or update or delete on emp
Begin
        If inserting then
        Raise_application_error (-21012, 'we cant insert');
        elsIf updating then
        Raise_application_error (-21013, 'we cant update');
        elsIf delete then
        Raise_application_error (-21014, 'we cant delete');
        End if;
End;
```

Ex: create table test (msg varchar2(100));

```
Create or replace trigger tr5
After
Insert or update or delete on emp
Declare
        A       varchar(100);
Begin
        If inserting then
        A:='rows inserted';
        elsIf updating then
        A:='rows updated';
        elsIf deleting then
        A:='rows deleted';
        End if;
        Insert into test values(a);
```

End;

2. Row level trigger

- In row level triggers, trigger body is executed for each row for DML operation, that's the reason we are using for each row clause in trigger specification and also data internally stored in 2 roll back segment qualifiers are OLD and NEW
- These qualifiers are used in either trigger specification or trigger body. When we are using these qualifiers in trigger body we must use colon (:) prefix in the qualifiers
- Synt: :old.column_name (or) :new.column_name
- When we are using these qualifiers in when clause we are not allowed to use colon infront of the qualifiers

Qualifiers	Insert	Update	Delete
:new	Yes	Yes	No
:old	No	Yes	Yes

- In before trigger, trigger body is executed before DML statements are effected into database
- In after trigger, trigger body is executed after DML statements are effected into database
- Generally we want to restrict the invalid data entry ,always we are using before triggers, where as if we are performing operations on the one table those operations are effecting on another table then we are using after trigger
- Whenever we are inserting values into new qualifiers we must use before trigger otherwise oracle server returns an error

Q: write a pl/sql row level trigger on emp table whenever user inserting data into emp table sal should be more than 5000?

Prgm:

```
Create or replace trigger tr10
Before
Insert on emp_t
For each row
Begin
If :new.sal<5000 then
Raise_application_error (-20123, 'sal should be >5000');
End if;
End;
```

Q: write a pl/sql row level trigger on emp, dept tables while implement on delete cascade concept without using on delete cascade clause

Prgm:

```
Create or replace trigger t11
After  delete on dept
For each row
Begin
Delete from emp where deptno=:old.deptno;
End
```

Parent table

Q: write a pl/sql row level trigger on dept table, whenever updating deptno's in dept table automatically those deptno's modified in emp table

Prgm:

```
Create or replace trigger t12
After
Update on dept
For each row
Begin
Update emp set deptno=:new.deptno where deptno=:old.deptno;
End;
```

Q: write a pl/sql row level trigger on emp table by using below conditions

1. **Whenever user inserting data those values stored in another table**
2. **Whenever user updating data those values stored in another table**
3. **Whenever user deleting data those values stored in another table**

Prgm:

First we create 3 tables which are having the same structure of emp table

```
Create or replace trigger t14
After
Insert or update or delete on emp
For each row
Begin
        If inserting then
        Insert into e1 values (:new.empno,:new.ename);
        If updating then
        Insert into e2 values (:old.empno,:new.ename);
```

```
        If deleting e3 then
        Insert into e1 values (:old.empno,:new.ename);
    End;
```

Q: write a pl/sql trigger on emp table whenever user deleting records from emp table automatically display remaining number of existing record number in bottom of delete statement?

Prgm:

```
    Create or replace trigger t15
    After
    Delete on emp
    Declare
        A       number (10);
    Begin
        Select count (*) into a from emp;
        Dbms_output.put_line ('remaining records are:'||a);
    End;
```

MUTATING TRIGGER:

```
    Create or replace trigger t15
    After
    Delete on emp
    For each row
    Declare
        A       numer (10);
    Begin
        Select count (*) into a from emp;
        Dbms_output.put_line ('rmaining record are:'||a);
    End;
```

- In a row level trigger based on a table trigger body cannot read data from same table and also we cannot perform DML operations on same table
- If we are trying to do this oracle will return an error table is mutating
- This error is called mutating error
- This table is called mutating table
- This trigger is called mutating trigger
- Mutating errors are not accured in statement level triggers because through these statement level triggers when we are performing DML operations automatically data committed to data base

- Where as in row level triggers when we are performing transaction data is not committed and also again we are reading this data from the same table then only mutating error is occurred
- To avoid this mutating error, we are using autonomous transaction in triggers.

```
Create or replace trigger t15
After
Delete on emp
For each row
Declare
        Pragma autonomous_transaction;
        A       numer (10);
Begin
        Select count (*) into a from emp;
        Dbms_output.put_line ('rmaining record are:'||a);
Commit;
End;
```

DDL TRIGGERS

- We can also create triggers on schema level, data base level. These types of triggers are called DDL triggers / system triggers.
- These are types of triggers are created by database administrator

```
Synt:  create or replace trigger trigger_name
       Before/ after
       Create/ alter/rename/ drop
       On username.schema
```

Q: write a pl/sql trigger on schott schema not to drop emp table

Prgm:

```
Create or replace trigger tr17
Before
Drop on apps.schema
Begin
        If ora_dict_obj_name='emp' and
        Ora_dict_obj_type='TABLE' then
        Raise_application_error (-20123, 'we cannot drop table');
        End if;
```

End;

8. COLLECTIONS

5. Oracle server supports fallowing types
   i. Pl/sql record
   ii. Index by table or pl/sql table or associate arrays
   iii. Nested tables
   iv. Varrays
   v. Ref cursors

1. Index by table

- This is a user defined data type, which is used to store multiple data items in to single unit, basically it is unconstrained table
- Generally these tables are used to improve performance of application because these tables are stored in memory area that's why these tables are also called as memory tables
- Basically these tables contains key value pairs i.e., value field is stored in actual data and key field stored in indexes
- Key field values are either integer or char and also these values or either +ve or –ve
- These indexes key behave like a primary key i.e., they don't allow duplicate and null values. Basically this key data type is binary_integer
- Index by table having fallowing collection methods
  1. Exists
  2. First
  3. Last
  4. Prior
  5. Next
  6. Count
  7. Delete (range of indexes)

Ex:1

```
Declare
Type t1 is table of number (10) index by binary_integer;
V1      t1;
Begin
V1(1):=10;
V1(2):=20;
V1(3):=30;
```

```
V1(4):=40;
V1(5):=50;
Dbms_output.put_line (v1(3));
Dbms_output.put_line (v1.first);
Dbms_output.put_line (v1.last);
Dbms_output.put_line (v1.prior (3));
Dbms_output.put_line (v1.next (2));
Dbms_output.put_line (v1.count);
End;
```

Ex-2:

```
Declare
Type t2 is table of number (10) index by binary_integer;
V1     t2;
Begin
V1(1):=10;
V1(2):=20;
V1(3):=30;
V1(4):=40;
V1(5):=50;
Dbms_output.put_line (v1.count);
V1.delete (1, 3);

Dbms_output.put_line (v1.count);
V1.delete (5);
Dbms_output.put_line (v1.count);
V1.delete;
Dbms_output.put_line (v1.count);
End;
```

(1,3) is range i.e., from 1,2,3

Single value with index 5 is deleted

All deleted

Q: write a pl/sql program to get all employee names from emp table and store it into index by table and display from index by table

Prgm:

```
Declare
Type t3 is table by varchar2 (10) index by binary_integer;
V1     t3;
Cursor c1 is select ename from emp;
N      number(5):=1;
Begin
       Open c1;
```

```
        Loop
        Fetch c1 into v1(n);
        Exit when c1%notfound;
        N:=n+1;
        End loop;
        Close c1;

        For I in v1.first..v1.last
        Loop
        Dbms_output.put_line (v1(i));
        End loop;
    End;
```

Prgm;

```
    Declare
    Type t4 is table of varchar2 (10) index by binary_integer;
    V1 t4;
    Begin
        Select ename bulk collect into v1 from emp;
        For I in v1.first..v1.last
        Loop
        Dbms_output.put_line (v1 (i));
        End loop;
    End;
```

Prgm:

```
    Declare
    Type t5 is  table of date index by binary_integer;
    V1     t5;
    Begin
    For I in 1..10
    Loop
    V1.(i):=sysdate+1;
    End loop;
    For I in v1.first..v1.last
    Loop
    Dbms_output.put_line (v1 (i));
```

```
End loop;
End;
```

Q: write a pl/sql program to retrieve all joining dates from emp table and store it into index by table and display content from index by table?

Prm:

```
Declare
Type t6 table of date index by binary integer
V1 t6;
Cursor c1 select hire date from emp;
N       number(2):=1;
begin
open c1;
loop
Fetch c1 into v1(n);
Exit when c1%notfound;
N:=n+1;
End loop;
Close c1;
For I in v1.first..v1.last
Loop
Dbms_output.put_line (v1(i));
End loop;
End;
```

Prgm:

```
Declare
        Type t7 is table of date index by binary_integer;
        V1 t5;
Begin
        Select hiredate bulk collect into v1 from emp;
        For I in v1.first..v1.last
        Loop
        Dbms_output.put_line (v1(i));
        End loop;
End;
```

Ex-1:

```
Declare
        Type t8 is table of varchar (10) index by varchar2 (10);
        V1      t8;
        X       varchar2 (10);
Begin
        V1 ('a'):='ARUN';
        V1 ('b'):='AJAY';
        V1 ('c'):='CHANU';
        X:=a;
        Loop
        Dbms_output.put_line (v1(x));
        X:=v1.next(x);
        Exit when x is null;
        End loop;
End;
```

Ex-2:

```
Declare
        Type t9 is table of emp%rowtype index by binary_integer;
        V1      t9;
        X       number (5);
Begin
        Select * bulk collect into v1 from emp;
        X:=1;
        Loop
        Dbms_output.put_line (v1(x).empno||','||v1(x).ename…);
        X:=v1.next (x);
        Exit when x is null;
        End loop;
End;
```

(OR)

```
Declare
        Type t10 is table of emp%rowtype index by binary_integer;
        V1      t10;
Begin
        Select * bulk collect into v1 from emp;
        For I in v1.first..v1.last
```

```
        Loop
        Dbms_output.put_line (v1(x).empno||','||v1(x).ename);
        End loop;
End;
```

2. NESTED TABLES

- This is also user defined datatype which is used to store multiple data items in a single unit but before are storing actual data we must initialize the data while using constructor.
- Here constructor name is same as type name. generally we are not allow to store index by tables permanently into database, to overcome this problem they are introduced nested tables, to extension of index by tables
- These user defined datatype store permanently into database using sql
- In index by tables we cannot add or remove indexes, where as in nested tables we can add or remove the indexes using extend, trim collection methods.
- In nested tables we can allocate the memory explicitly while using extend method

Synt:

```
Type type_name is table of datatype (size)
Variable_name type_name :=  type_name ();➔ constructor name
```

Ex-1:

```
        declare
Type    t1 is table of number (10);
        V1 t1:=t1();
Begin
        V1.extend(100);
        V1(100):=10;
        Dbms_output.put_line (v(100));
End
```

Ex-2

```
Declare
        Type t2 is table of number (10);
        V1 t2:=t2 (10, 20, 30, 40, 50);
Begin
        Dbms_output.put_line (v1.first);
        Dbms_output.put_line (v1.last);
        Dbms_output.put_line (v1.prior (3));
        Dbms_output.put_line (v1.next (3));
        Dbms_output.put_line (v1.count);
```

```
        Dbms_output.put_line (v1 (3));
        For I in v1.first ..v1.last
        Loop
        Dbms_output.put_line (v1 (i));
        End loop;
End;
```

Ex-3:

```
Declare
        Type t3 is table of number (10);
        V1 t3:=t3 (10, 20, 30, 40, 50);
Begin
        V1.trim (3);
        For I in v1.first..v1.last
        Loop
        Dbms_output.put_line (v1 (1));
        End loop;
End;
```

Last 3 records are deleted here,
Particular 1 record is not possible to delete

Ex-3:

```
Declare
        Type t4 is table of number (10);
        V1 t4
        V2 t4:=t4 ();
Begin
        If v1 is null then
        Dbms_output.put_line ('v1 is null)';
        Else
        Dbms_output.put_line ('v1 is not null)';
        End if;
        If v2 is null then
        Dbms_output.put_line ('v2 is null)';
        Else
        Dbms_output.put_line ('v2 is not null)';
        End if;
End;
```

Q: write a pl/sql program to get all employee names from emp table and store it to nested table and display data from nested tables

Prgm:

```
Declare
Type t5 is table of varchar2 (10);
V1 t5:=t5();
Cursor c1 is select ename from emp;
N number (5):=1;
Begin
For I in c1
Loop
V1. extend();
V(n):=i.ename;
N:=n+1;
End loop;
For I in v1.first..v1.last
Loop
Dbms_output.put_line(v(i));
End loop;
End;
```

(OR)

Prgm:

```
Declare
        Type t5 is table of varchar2 (10);
        V1 t5:=t5();
Begin
        Select ename bulk collect into v1 from emp
        For I in v1.first..v1.last
        Loop
        Dbms_output.put_line (v1(i));
        End loop;
End;
```

Prgm:

```
Declare
        Type t6 is table of emp%rowtype;
        V1 t6:=t6();
Begin
```

```
        Select * bulk collect into v1 from emp;
        For I in v1.first..v1.last;
        Loop
        Dbms_output.put_line ('v1 (i).ename||','||v1(i).empno)';
        End loop;
End;
```

3. VARRAYS

- This is also user defined data types which are used to store multiple data types in single unit, but before that we are storing actual data we must initialize the data while using constructor
- These user defined types stored permanently into database using sql
- Basically we are using Varrays for retrieving the huge data.

Synt:

```
Type type_name is varray(max size) of datatype(size);
variable_name:=type_name();
```

Prgm:

```
Declare type t1 is varrary (50) is emp%rowtype;
V1 t1:=t1();
Beign
Select * bulk collect into v1 from emp;
For I in v1.first..v1.last
Loop
Dbms_output.put_line (v1(i).empno||','||v1(i).ename);
End loop;
End;
```

Q: how to store these user defined types in database

A: create type t1 is table of number(100);

Difference b/w index by table, nested table, Varrays

Index by table	Nested by table	Varrays
1.it is not stored permanently in database	1.it is stored permanently in database using sql	1.it is stored permanently in database using sql
2.we can't add or remove indexes	2. we can add or remove indexes using extend, trim method	2. we can add or remove indexes using extend, trim method

3.indexes starting from –ve to +ve numbers and also having key value paris	3.indexes staring form 1	3. indexes starting from 1
4.For small and medium date	4.for small and medium data	4.for large date

9. BULK

- Bulk Is one of the method which is used to improve the performance of the applications
- Oracle introduce bulk bind process using collections i.e., in this process we are putting all sql statement related values into collection and in this collection we are performing insert, update, delete at a time using "for all" statement
- In this bulk we have two actions
    i. Bulk collect
    ii. Bulk bind

1. Bulk collect

- In this clause we are used to fetch the data from resource into collection
- This clauses used in

```
Select……..into……clause
Cursor……fetch……clause
Dml……returning……clause
```

1. Bulk collect used in select……into…….clause

Synt: select * bulk collect into collection_name from table_name

Ex:
```
declare
        Type t1 is table of emp%rowtype Index by binary_integer;
        V1     t1;
begin
        Select * bulk collect into v1 from emp;
        For I in v.first..v1.last
        Loop
        Dbms_output.put_line (v(i).empno||','||v(i).ename);
        End loop;
End;
```

2. **Bulk collect used in cursor……..fetch…….statement**

Synt: fetch cursor_name bulk collect into collection_variable;

Ex:

```
declare
        Type t2 is table of varchar2 (10) index by binary_integer;
        V1      t1;
        V2      t1;
        Cursor c1 is select ename, job from emp;
Begin
        Open c1;
        Fetch c1 bulk collect into v1,v2;
        Close c1;
        For I in v1.first..v1.last
        Loop
        Dbms_output.put_line (v1(i)||','||v2(i));
        End loop;
End;
```

Time program without bulk:

```
Declare
Vrow varchar2(50);
Cursor c1 is select object_name from all_objects;
Z1 number (10);
Z2 number (10);
Begin
Z1:=dbms_utility.get_time;
Open c1;
Loop
Fetch c1 into vrow;
Exit when c1%notfound;
End loop;
Close c1;
Z2:=dbms_utility.get_time;
Dbms_output.put_line (z1);
Dbms_output.put_line (z2);
Dbms_output.put_line (z1-z2);
End;
```

Time program without bulk:

```
Declare
Vrow varchar2(50);
Cursor c1 is select object_name from all_objects;
Z1 number (10);
Z2 number (10);
Begin
Z1:=dbms_utility.get_time;
Open c1;
Loop
Fetch c1 bulk collect into vrow ;
Exit when c1%notfound;
End loop;
Close c1;
Z2:=dbms_utility.get_time;
Dbms_output.put_line (z1);
Dbms_output.put_line (z2);
Dbms_output.put_line (z1-z2);
End;
```

3. **Bulk collect used in DML…..returning clauses**
   Synt: dml statement returning column_name into variable_name

   Ex: variable a varchar2 (10);
   Update emp set sal=sal+100 where ename='KING' returning job bulk collect into a;
   Print a;

Q: write a pl/sql stored procedure modify salaries of the clerk from emp table and also these modified values immediately stored in index table by using dml ……returning……clause And also display content form index by table?

Prgm:

```
Create or replace procedure p1 is
        Type t1 is table of emp%rowtype index by binary_integer;
        V1      t1;
Begin
Update emp set sal=sal+100 where job='clerk'
Returning empno, ename, job, mgr, hiredate, sal, comm, deptno
Bulk collect into v1;
```

Returning updated clerk information

```
Dbms_output.put_line ('updated no.of clerks are:'||sql%rowcount);
For I in v1.first..v1.last
Loop
Dbms_output.put_line (v1(i).ename||','||v1(i).job);
End loop;
End;
```

2. Bulk bind

- In bulk bind process we are performing bulk of operations using collections i.e., in this process we are using bulk update, bulk insert, bulk delete using "forall" statement
- Before we are using bulk bind process we are fetching data from database into collection using bulk collect clause

  Synt: forall indexvar in collectionvar.first..collectionvar.last

Ex:
```
declare
        Type t1 is varray (10) of number (10);
        V1 t1:=t1 (10, 20);
  Begin
        Forall I in v1.first..v1.last
        Update emp set sal=sal+100 where deptno=v1(i);
  End;
```

Bulk update:

Prgm:
```
declare
        Type t1 is table of numbe (5) index by binary_integer;
        V1     t1;
  Begin
        Select empno bulk collect into v1 from emp;
        Forall I in v1.first..v1.last
        Update emp set sal=sal+100 where empno=v1(i);
  End;
```

Bulk delete

Prgm:
```
declare
        Type t1 is varray(10) of number (10);
        V1 t1:=t1(20,30,40);
  Begin
        Forall I in v1.first..v1.last
```

```
            Delete from emp where empno=v1(i);
        End;
```

Bulk insert

```
Prgm: declare
        Type t1 is table of number (10) index by binary_integer;
        V1 t1;
        Begin
        For I in 1..100
        Loop
        V1(i):=I;
        End loop;
        Forall I in v1.first..v1.last
        Insert into bt values(v(i));
        End;
```

10. When current of and for update clause

- Generally when we are using update, delete statements automatically locks are generated in the database.
- If you want to generate locks before update, delete statements then we are using cursor locking mechanism in all database systems
- In this case we must specify for update clause in cursor definition

Synt: cursor cursor_name is select * from table_name where condition for update

- If you are specifying for update clause also oracle server does not generate the lock i.e., whenever we are opening the cursor then only oracle server internally uses exclusive locks
- After processing we must release the locks using commit statement
- Where current clause uniquely identifying a record in each process because where current of clause internally used ROWID
- Whenever we are using where current of clause we must use for update clause

Ex:

```
declare
Cursor c1 is select * from k for update;
I emp%rowtype;
```

```
Begin
        Open c1;
        Loop
        Fetch c1 into I ;
        Exit when c1% notfound;
        If i.job='CLERK' then
        Update set sal=i.sal+1000 where current of c1;
        End if
        End loop;
        Commit;                                  release lock
        Close c1;
End;
```

If don't release it, we cant do any operation on same table in other schema

11.PRAGMA AUTONOMUS TRANSACTION

- **When we use commit in child procedure then we use pragma autonomous_transaction**

Child procedure

```
        Create or replace procedure cp is
        Cursor c1 is select * from emp;
        Vrow emp%rowtype;
        Pragma autonomus_transaction;
        Begin
                Open c1;
                Loop
                Fetch c1 into vrow;
                Exit when c1%notfound;
                If vrow.deptno=10 then
                Update sal=vrow.sal+1000 where empno=vrow.empno;
                End if;
                End loop;
                Commit;
                Close c1;
        end;
```

o/p: procedure created

Main procedure

```
create or replace procedure mp is
begin
        delete from pt where deptno=20;
        cp;
        rollback;
end;
```