PL/SQL

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

⇨ **Block**

➔ **It is one of the area which is used to write a programming logic.**
➔ **This block is have 3 sections.**

- **Declaration Section**
- **Executable Section**
- **Exception Section**

1) **Declaration Section**
   ➔ **It is one of the section which is used declare variables, cursors and exceptions and so on.**
   ➔ **It is optional section.**
2) **Executable Section**
   ➔ **It is one of the section which is used to write a program coding.**
   ➔ **It is mandatory section.**
3) **Exception Section**
   ➔ **It is one of the section which is used to handle the errors at runtime.**
   ➔ **It is optional section.**

⇨ **There are two types of blocks are supported by pl/sql.**

- **Anonymous Block**
- **Named Block**

1) **Anonymous Block**
   ➔ **These blocks does not have a name and also not stored in database.**

```
Ex : Declare
        -----------------
        Begin
        -----------------
        -----------------
        End;
```

```
Ex-1 :  Begin
        Dbms_Output.Put_Line( 'welcome to E Business Solutions' );
        End;
```

2) **Named Block**
   ➔ **These blocks are having a name and also stored in database.**
   Examples : Procedures , Functions, Packages and Triggers etc..

⇨ **Variable**

➔ **It is one of the memory location which is used to store the data.**

- ➔ **Generally we are declare the variables in declaration section.**
- ➔ **These are supported default and not null.**
  Syntax : Variable_Name Datatype (Size);

Ex :
```
Declare
 A      Number ( 5 );
 B      Number ( 5 ) not null :=10;
 C      Number ( 5 )  default      10;
```

Ex-1 :
```
Declare
A  Varchar2(20);
Begin
A := 'Hello EBS';
Dbms_Output.Put_Line( A );
End;
```

⇨ Storing a value into variable

- ➔ **Using assignment operator (:=) we storing a value into variable.**
  Syntax : Variable_Name := value;

  Ex : a :=50;

⇨ Display Message (or) Varaible Value

- ➔ **We have one pre defined package which is used display the message or value in a program.**
  Syntax : dbms_output.put_line ('message');
  dbms_output.put_line (variable_name);

⇨ Select ------ Into ------ Clause

- ➔ **This clause is used to retrieve the data from table & storing into pl/sql variables.**
  Syntax : select col1, col2 into var1, var2;

⇨ Data Types

- ➢ **% Type**
- ➢ **% RowType**
- ➢ **Record Type (or) Pl/sql Record**
- ➢ **Index By Table (or) Pl/sql Table**

1) % Type:

- ➔ **It is one of the datatype which is used to assign the column datatype to a variable.**
- ➔ **It is used to store one value at a time.**
- ➔ **It is not possible to hold more than one column values or row values.**
  Syntax : variable_name table_name.column_name%type;

Ex-1 :
```
Declare
Vno          emp.empno%type:=&n;
Vname        emp.ename%type;
```

```
Begin
Select ename into vname from emp where empno=vno;
Dbms_output.put_line ( ‘ employee name is : ‘ || ‘ ‘ || vname );
End;
```

2) **% RowType**

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

  Syntax : variable_name table_name%rowtype;

Ex-1 :

```
Declare
Vrow  emp%rowtype;
Vno     emp.empno%type:=&n;
Begin
Select * into vrow from emp where empno=vno;
Dbms_output.put_line ( vrow.ename || ‘ ‘ || vrow.sal );
End;
```

3) **Record Type (or) Pl/Sql Record**

- ➔ Is is one of the user defined temporary data type which is used to store more than one table data (or) to assign more than one column datatypes.
- ➔ They must at least contain one element.
- ➔ Pinpoint of data is not possible.

  Syntax : Type Typename is Record (Val-1 Datatype, Val-2 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:=&n;
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.vloc);
End;
```

⇨ **Conditional Statements**

1) If Condition
2) If Else Condition
3) Elsif Condition
4) Case Condition

1) If Condition

```
Syntax :        If condition then
          Statements;
          End if;

Ex-1 :  Declare
        A       Number ( 4 ) :=&n;
        B       Char ( 1 );
        Begin
        If a<20 then
        B:='Yes';
        End if;
        Dbms_output.put_line ( B );
        End;
```

2) If Else Condition

```
Syntax :        If condition then
                Statements ;
                Else
                Statements ;
                End if;

Ex-1 :  Declare
        A       Number ( 4 ) :=&n;
        B       Char ( 10 );
        Begin
        If a<20 then
        B:='TRUE';
        Else
        B:='FALSE';
        End if;
        Dbms_output.put_line ( B );
        End;
```

3) Elsif Condition

```
Syntax :        If condition-1 then
          Statements;
          Elsif condition-2 then
          Statements;
          Elsif condition-3 then
          Statements;
          Else
          Statements;
          End if;
```

```
Ex-1 :  Declare
        A         Number ( 4 ) :=&n;
        B         Char ( 15 );
        Begin
        If a<20 then
        B:='Low Value';
        Elsif a>20 and a<100 then
        B:='High Value';
        Else
        B:='Invalid Value';
        End if;
        Dbms_output.put_line ( B );
        End;
```

4) Case Condition

```
Syntax :        Case ( column name )
                When condition then
                Statements;
                When condition then
                Statements;
                Else
                Statements;
                End Case;
```

```
Ex-1 :  DECLARE
        VSAL NUMBER(10):=&N;
        BEGIN
        CASE
        WHEN VSAL<2000 THEN
        DBMS_OUTPUT.PUT_LINE('VSAL IS'||' '||'LOW');
        WHEN VSAL>2000 THEN
        DBMS_OUTPUT.PUT_LINE('VSAL IS'||' '||'HIGH');
        ELSE
        DBMS_OUTPUT.PUT_LINE('VSAL IS'||' '||'INVALID');
        END CASE;
        END;
```

⇨ Loops

1) Simple Loop
2) While Loop
3) For Loop

1) Simple Loop

```
Syntax :        Loop
                Statements;
                End loop;
```

```
 Syntax :       Loop
                Code;
                Exit when condition;
                End loop;

Ex-1 :  Begin
        Loop
        Dbms_output.put_line ( ‘Welcome to k-onlines.com' );
        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;
```

2) **While Loop**

```
Syntax :        While ( Condition )
                Loop
                Statements;
                End loop;

Ex-1:  Declare
        N       Number(4):=1;
        Begin
        While n>=10
        Loop
        Dbms_output.put_line ( N );
        N:=N+1;
```

```
End loop;
End;
```

3) For Loop

```
Syntax :    For   variable_name     in    lowerbound..outerbound
            Loop
            Statements;
            End loop;
```

```
Ex-1:  Declare
       N       number(5);
       Begin
       For     n    in 1..10
       Loop
       Dbms_output.put_line ( N );
       End loop;
       End;
```

```
Ex-1:  Declare
       N       number(5);
       Begin
       For     n    in reverse 1..10
       Loop
       Dbms_output.put_line ( N );
       End loop;
       End;
```

⇨ Bind Variable

- ➔ These variables are session variable.
  Syntax : variable a number;

```
Ex-1:  sql>   Variable V Number;
       Sql>   Declare
              A       number(5):=500;
              Begin
              :v:=a/2;
              End;
       Sql>   Print V;
```

⇨ CURSORS

- ➔ Cursor is a buffer area which is used to process multiple records and also record by record by process.
- ➔ There are two types

a) Implicit Cursors
b) Explicit Cursors

a) Implicit Cursors

- **Sql statements returns a single record is called implicit cursors**
- **Implicit cursor operations done by the system.**
- **Open by the system.**
- **Fetch the records by the system**
- **Close by the system.**

Ex :
```
Declare
      X  emp%rowtype;
      Begin
      Select * into X from emp where empno=7369;
      Dbms_output.put_line(x.empno||’,’||x.ename);
      End;
```

b) Explicit Cursors

- **Sql statements return a multiple records is called explicit cursors**
- **Explicit cursor operations done by the user.**
- **Declare by the user**
- **Open by the user**
- **Fetch the records by the user**
- **Close by the user**

Ex -1:
```
Declare
      Cursor c1 is select ename,sal from emp;
      V_Name        varchar2(10);
      V_Sal         number(10);
      Begin
      Open C1;
      Fetch c1 into v_name,v_sal;
      Dbms_output.put_line(v_name||’,’||v_sal);
      Close C1;
      End;
```

Ex-2:
```
Declare
      Cursor c1 is select ename,job from emp;
      Vvname varchar2(10);
      Job  varchar2(10);
      Begin
      Open c1;
      Fetch c1 into vname,job;
      Dbms_output.put_line(vname||','||job);
      Fetch c1 into vname,job;
      Dbms_output.put_line(vname||','||job);
      Close c1;
      End;
```

Ex-3

```
Declare
 Ccursor c1 is select ename,job from emp;
Vname varchar2(10);
Vjob     varchar2(10);
 Begin
 Open c1;
 Loop
Fetch c1 into vname,vjob;
Dbms_output.put_line(vname||','||vjob);
End loop;
Close c1;
End;
```

⇨ **CURSOR Attributes**

- Every explicit cursor having following four attributes

1) %NotFound
2) %Found
3) %Isopen
4) %Rowcount

- All these cursor attributes using along with cursor name only
  Syntax : cursorname % attributename

Note : Except %rowcount all other cursor attribute records Boolean value return either true or false where as %rowcount return number datatupe.

1) **%NotFound**

- 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.
- Returns FALSE if a successful fetch has been executed.
- Returns TRUE if no row was returned.

Ex-1

```
Declare
 Ccursor c1 is select ename,job from emp;
Vname varchar2(10);
Vjob     varchar2(10);
 Begin
 Open c1;
 Loop
Fetch c1 into vname,vjob;
Exit when c1%notfound;
Dbms_output.put_line(vname||','||vjob);
 End loop;
Close c1;
End;
```

2) %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.
- Returns TRUE if a successful fetch has been executed.
- Returns FALSE if no row was returned.

Ex-1:
```
Declare
Cursor c1 is select * from emp;
I emp%rowtype;
Begin
Open c1;
Loop
Fetch c1 into i;
If c1%found then
Dbms_output.put_line(i.empno||','||i.ename);
Else
Exit;
End if;
End loop;
Close c1;
End;
```

3) %IsOpen

- Returns TRUE if the cursor is open,
- Retuens FALSE if the cursor is closed.

Ex-1
```
Declare
Cursor c1 is select * from emp;
I emp%rowtype;
Begin
Open c1;
If c1%isopen then
Dbms_output.put_line('cursor is open');
Loop
Fetch c1 into i;
If c1%found then
Dbms_output.put_line(i.ename);
Else
Exit;
End if;
End loop;
Close c1;
If not c1%isopen then
Dbms_output.put_line('cursor is closed');
End if;
End if;
End;
```

4) %Rowcount

- Returns INVALID_CURSOR if cursor is declared, but not open or if cursor has been closed.
- Returns the number of rows fetched by the cursor.

Ex-1
```
Declare
Cursor c1 is select * from emp;
I emp%rowtype;
Begin
Open c1;
Loop
Fetch c1 into i;
Exit when c1%notfound;
Dbms_output.put_line(i.empno||','||i.ename);
End loop;
Dbms_output.put_line('Total no of employee: '|| c1%rowcount);
Close c1;
End;
```

⇨ PARAMETER CURSOR

- Passing a parameter in cursor is call it as a parameter cursor.

Syntax : Cursor cursor_name (parameter_name) is select * from table_name where column_name=parameter_name

Ex-1
```
Declare
Cursor c1 (p_deptno number) is select * from emp where deptno=p_deptno;
I emp%rowtype;
Begin
Open c1(10);
Loop
Fetch c1 into i;
Exit when c1%notfound;
Dbms_output.put_line(i.ename);
End loop;
Close c1;
End;
```

Ex-2
```
Declare
Cursor c1 ( p_job varchar2) is select * from emp where job=p_job;
I emp%rowtype;
Begin
Open c1('MANAGER');
Loop
Fetch c1 into i;
Exit when c1%notfound;
Dbms_output.put_line(i.empno||','||i.ename||','||i.job);
End loop;
```

```
Close c1;
Open c1('CLERK');
Loop
Fetch c1 into i;
Exit when c1%notfound;
Dbms_output.put_line(i.empno||','||i.ename||','||i.job);
End loop;
Close c1;
End;
```

⇨ **CURSOR WITH FOR Loop**

- **In cursor for loop no need to open, fetch, close the cursor. For loop it self automatically will perform these functionalities**

Ex-1
```
Declare
Cursor c1 is select * from emp;
I emp%rowtype;
Begin
For i in c1 loop
Dbms_output.put_line(i.empno||','||i.ename);
End loop;
End;
```

⇨ **NESTED CURSOR WITH FOR Loop**

Ex-2
```
Declare
Cursor c1 is select * from dept;
Cursor c2(p_deptno number) is select * from emp where deptno=p_deptno;
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||','||j.sal);
End loop;
End loop;
End;
```

⇨ **CURSOR WITH DML Operations**

Ex-1
```
Declare
Cursor c1 is select * from emp;
Begin
For i in c1
Loop
Insert into t1 values (i.ename,i.sal);
End loop;
End;
```

Ex-2

```
Declare
Cursor c1 is select * from t1;
Begin
For i in c1
Loop
Delete from t1 where sal=3000;
End loop;
End;
```

Ex-3

```
Declare
Cursor c1 is select * from kuncham;
Begin
For i in c1
Loop
If i.job='CLERK' then
Update kuncham set sal=i.sal+1111 where empno=i.empno;
Elsif i.job='MANAGER' then
Update kuncham set sal=i.sal+2222 where empno=i.empno;
End if;
End loop;
End;
```

⇨ **Ref Cursor**

- Ref Cursors are user define types which is used to process multiple records and also this is record by record process
- Generally through the static cursors we are using only one select statement at a time for single active set area where as in ref cursors we are executing no of select statements dynamically for single active set area.
- Thats why these type of cursors are also called as dynamic cursors.
- By using ref cursors we return large amount of data from oracle database into client applications.
- There are 2 Types
  - ✓ 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;
v_t t1;
i emp%rowtype;
begin
open v_t for select * from emp where sal>2000;
loop
fetch v_t into i;
```

```
exit when v_t%notfound;
dbms_output.put_line(i.ename||' '||i.sal);
end loop;
close v_t;
end;
```

Ex - 2

```
declare
type t1 is ref cursor;
v_t t1;
i emp%rowtype;
j dept%rowtype;
v_no number(5):=&no;
begin
if v_no=1 then
open v_t for select * from emp;
loop
fetch v_t into i;
exit when v_t%notfound;
dbms_output.put_line(i.ename||' '||i.deptno);
end loop;
close v_t;
elsif v_no=2 then
open v_t for select * from dept;
loop
fetch v_t into j;
exit when v_t%notfound;
dbms_output.put_line(j.deptno||' '||j.dname);
end loop;
close v_t;
end if;
end;
```

Ex - 3

```
create or replace package pg1
is
type t1 is ref cursor return emp%rowtype;
type t2 is ref cursor return dept%rowtype;
procedure p1 (p_t1 out t1);
procedure p2 (p_t2 out t2);
end;

create or replace package body pg1 is
procedure p1 (p_t1 out t1)
is
begin
open p_t1 for select * from emp;
end p1;
```

```
procedure p2 (p_t2 out t2)
is
begin
open p_t2 for select * from dept;
end p2;
end;
```

Execution

```
variable a refcursor
variable b refcursor
exec pg1.p1(:a);
exec pg1.p2(:b);
print a b;
```

⇨ Where Current of and For Update Clause

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

  Syntax : 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 of clause uniquely identifying a record in each process because where current of clause internally uses 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 k set sal=i.sal+1000 where current of c1;
end if;
end loop;
commit;
close c1;
end;
```

⇨ **EXCEPTIONS**

- **Exception is one of the activity which is used to handle the errors at runtime.**
- **There are 3 types of exceptions**

1) **Predefined Exception**
2) **Userdefined Exception**
3) **Unnamed Exception**

1) **Predefined Exception**

- **It is one of the exception which are defined by oracle.**
- **There are 20 exceptions available.**

```
Syntax : when exception1 then
           statements;
           when exception2 then
           statements;
           when others then
           statements;
```

⇨ **Predefined Exceptions are**

- **no_data_found**
- **too_many_rows**
- **invalid_cursor**
- **cursor_already_open**
- **invalid_number**
- **value_error**
- **zero_devide**
- **others**

etc.....

1) **No_Data_Found**

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

```
Ex : declare
       v_ename varchar2(20);
       v_sal number(10);
       begin
       select ename,sal into v_ename,v_sal from k where empno=&no;
       dbms_output.put_line(v_ename||' '||v_sal);
       end;
```

```
Ex : declare
       v_ename varchar2(20);
       v_sal number(10);
       begin
       select ename,sal into v_ename,v_sal from k where empno=&no;
       dbms_output.put_line(v_ename||' '||v_sal);
```

```
exception
when no_data_found then
dbms_output.put_line('employee does not exit');
end;
```

2) Too_Many_Rows

- **When a select ----- into clause try to return more than one record or more than one value then oracle server return an error.**
- **Error is ora-01422 : exact fetch returns more than requested number of rows.**
- **To handle this error we are using too_many_rows exception**

```
Ex : declare
        v_ename varchar2(20);
        v_sal number(10);
        begin
        select ename,sal into v_ename,v_sal from k;
        dbms_output.put_line(v_ename||' '||v_sal);
        end;
```

```
Ex : declare
        v_ename varchar2(20);
        v_sal number(10);
        begin
        select ename,sal into v_ename,v_sal from k;
        dbms_output.put_line(v_ename||' '||v_sal);
        exception
        when too_many_rows then
        dbms_output.put_line('program return more than one row');
        end;
```

3) Invalid_Cursor

- **Whenever we are performing invalid operations on the cursor server returns an error i.e if you are try to close the cursor with out opening cursor then oracle server returns an error.**
- **Error is ora-01001 : invalid cursor**
- **To handle this error we are using invalid_cursor exception.**

```
Ex : declare
        cursor c1 is select * from emp;
        i emp%rowtype;
        begin
        loop
        fetch c1 into i;
        exit when c1%notfound;
        dbms_output.put_line(i.ename||i.sal);
        end loop;
        close c1;
        end;
```

```
Ex : declare
        cursor c1 is select * from emp;
        i emp%rowtype;
        begin
        loop
        fetch c1 into i;
        exit when c1%notfound;
        dbms_output.put_line(i.ename||i.sal);
        end loop;
        close c1;
        exception
        when invalid_cursor then
        dbms_output.put_line('first you open the cursor');
        end;
```

4) Cursor_Already_Open

- When we are try to reopen the cursor without closing the cursor oracle server returns an error.
- Error is ora-06511 : cursor already open
- To handle this error we are using cursor_already_open exception

```
Ex : cursor c1 is select * from emp;
        i emp%rowtype;
        begin
        open c1;
        loop
        open c1;
        fetch c1 into i;
        exit when c1%notfound;
        dbms_output.put_line(i.ename||i.sal);
        end loop;
        close c1;
        end;
```

```
Ex : declare
        cursor c1 is select * from emp;
        i emp%rowtype;
        begin
        open c1;
        loop
        open c1;
        fetch c1 into i;
        exit when c1%notfound;
        dbms_output.put_line(i.ename||i.sal);
        end loop;
        close c1;
        exception
        when cursor_already_open then
```

```
dbms_output.put_line('cursor already open');
end;
```

5) Invalid_Number

- Whenever we are try to convert string type to number type oracle server return error.
- Error is ora-01722 : invalid number
- To handle this error we are using invalid_error exception

```
Ex : begin
        insert into emp(empno,sal) values (111,'abcd');
        end;

Ex : begin
        insert into emp(empno,sal) values (111,'abcd');
        exception
        when invalid_number then
        dbms_output.put_line('insert proper data only');
        end;
```

6) Value_Error

- Whenever we are try to convert string type to number type based on the condition then oracle server returns an error
- Whenever we are try to store large amount of data than the specified data type size in varaible declaration then oracle server return same error
- Error is ora-06502 : numeric or value error: character to number conversion error
- To handle this error we are using value_error exception

```
Ex : declare
        z number(10);
        begin
        z:='&x'+'&y';
        dbms_output.put_line(z);
        end;

Ex : declare
        z number(10);
        begin
        z:='&x'+'&y';
        dbms_output.put_line(z);
        exception
        when value_error  then
        dbms_output.put_line('Enter the proper data only');
        end;

Ex : declare
        z number(3);
        begin
        z:='abcd';
        dbms_output.put_line(z);
        end;
```

7) Zero_Devide

- Whenever we are try to divide by zero then oracle server return a error
- Error is ora-01476 : divisor is equal to zero
- To handle this error we are using zero_divide exception

```
Ex : declare
        a number(10);
        b number(10):=&b;
        c  number(10):=&c;
        begin
        a:=b/c;
        dbms_output.put_line(a);
        end;

Ex : declare
        a number(10);
        b number(10):=&b;
        c  number(10):=&c;
        begin
        a:=b/c;
        dbms_output.put_line(a);
        exception
        when zero_divide then
        dbms_output.put_line('c does not contain zero');
        end;
```

⇨ EXCEPTION PROPAGATION

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

- If the exceptions are raised in executable section those exceptions are handled using either inner block or an outer block.
- Where as if exception are raised in declaration section or in exception section those exceptions are handled using outer blocks only.

```
Ex : begin
        declare
        z varchar2(3);--:='abcd';
        begin
        z:='abcd';
        dbms_output.put_line(z);
        exception
        when value_error then
```

```
dbms_output.put_line('invalid string lenght');
end;
exception
when value_error then
dbms_output.put_line('the lenght is more');
end;
```

2) Userdefined Exception

- We can also create our own exception names and also raise whenever it is necessary. these types of exceptions are called user defined exceptions.
- These exceptions are divided into 3 steps
  1) Declare Exception
  2) Raise Exception
  3) Handle Exception

1) Declare Exception

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

  Syntax : userdefinedexception_name exception

  Ex :
  ```
  declare
      a  exception;
  ```

2) Raise Exception

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

  Syntax : raise userdefinedexception_name

  Ex :
  ```
  declare
      a    exception;
      begin
      raise  a;
      end;
  ```

3) Handle Exception

- We can also handle user defined exceptions as same as predefined exception using predefined handler.

  Syntax :
  ```
  when userdefinedexception_name1 then
  statements;
  when userdefinedexception_name2 then
  statements;
  ----------
  ----------
  when others then
  statements;
  ```

```
Ex : declare
     a  exception;
     begin
     if to_char(sysdate,'dy')='sun' then
     raise a;
     end if;
     exception
     when z then
     dbms_output.put_line('my exception raised today');
     end;

Ex : 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 alredy high');
     end;
```

RIASING Predefined Exception

- **We can also raising predefined exception by using raise statement.**

Syntax : raise predefinedexceptionname;

```
Ex : declare
     cursor c1 is select * from emp where job='CLRK';
     i emp%rowtype;
     begin
     open c1;
     fetch c1 into i;
     if c1%notfound then
     raise no_data_Found;
     end if;
     close c1;
     exception
     when no_data_found then
     dbms_output.put_line('your job not available');
     end;
```

EXCEPTION Raised in Exception Section

- We can also raise the exception in exception section

```
Ex : declare
        a1 exception;
        a2 exception;
        begin
        begin
        raise a1;
        exception
        when a1 then
       dbms_output.put_line('a1 handled');
       --raise a2;
       end;
       exception
       when a2 then
       dbms_output.put_line('a2 handled');
       end;
```

ERROR Trapping Functions

- There are two error trapping functions supported by oracle.
  1) SQL Code
  2) SQL Errm

1) SQL Code : It returns numbers
2) SQL Errm : It returns error number with error message.

```
Ex : declare
        v_sal number(10);
        begin
        select sal into v_sal from emp where empno=7369;
        dbms_output.put_line(sqlcode);
        dbms_output.put_line(sqlerrm);
        end;
```

RAISE APPLICATION ERROR

- If you want to display your own user defined exception number and exception message then we can use this raise application error

  Syntax : raise_application_error (error_number,error_message);

  Error_Number : It is used to give the error numbers between -20000 to -20999
  Error_Message : It is used to give the message upto 512 characters.

```
Ex : declare
        v_sal number(10);
        a exception;
        begin
```

```
select sal into v_sal from k where empno=7369;
if v_sal < 2000 then
raise a;
else
update k set sal=sal+100 where empno=7369;
end if;
exception
when a then
raise_application_error ( -20999,'salary alreday high');
end;
```

3) Un Named Exception

- If you want to handle other than oracle 20 predefined errors we are using unnamed method.
- Because oracle define exception names for regularly accured errors other than 20 they are not defining exception names.
- In this case we are providing exception names and also associate this exception name with appropriate error no using exception_init function.

  Syntax : pragma exception_init (userdefined_exception_name, error_number);

- Here pragma is a compiler directive i.e at the time of compilation only pl/sql runtime engine associate error number with exception name.
- This function is used in declare section of the 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;
```

⇨ SUB PROGRAMS

- Sub programs are named pl/sql blocks which is used to solve particular task.
- There are two types of sub programs supported by oracle.
  1) Procedures
  2) Functions

1) Procedures

- Procedures may or may not return a value.
- Procedures return more than one value while using the out parameter.
- Procedure can execute only 3 ways
  a) Anonymous Block

b) Exec
c) Call

- Procedure can not execute in select statement.
- Procedure internally having one time compilation process.
- Procedure are used to improve the performance of business applications
- Every procedure is having two parts
  a) Procedure Specification
    - In procedure specification we are specifying name of the procedure and types of the parameters.

  b) Procedure Body
    - In procedure body we are solving the actual task.

```
Ex : create or replace procedure p11(p_empno number) is
     v_ename varchar2(10);
     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||','||v_sal);
     end;
```

⇨ **Execute The Procedure in 3 ways**

```
Method : 1 - Exec P11 ( 7902 )

Method : 2 -  Begin
              P11 ( 7902 );
              end;

Method : 3 -  Call P11 ( 7902 )
```

```
Ex : create or replace procedure p111(p_deptno number) is
     cursor c1 is select * from emp where deptno=p_deptno;
     i emp%rowtype;
     begin
     open c1;
     loop
     fetch c1 into i;
     exit when c1%notfound;
     dbms_output.put_line(i.ename||','||i.sal||','||i.deptno);
     end loop;
     close c1;
     end;
```

⇨ **Parameters in Procedures**

- 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 Parameters

a) Formal Parameters

- Formal Parameters are defined in procedure specification
- In Formal Parameters we are defining parameter name & mode of the parameter
- There are three types of modes supported by oracle.

  1) IN Mode
  2) OUT Mode
  3) INOUT Mode

1) IN Mode :

- By default procedure parameters having IN mode.
- IN Mode is used to pass the values into procedure body.
- This mode behaves like a constant in procedure body, through this IN Mode we can also pass default values using default or ":=" operator

Ex :

```
Create or replace procedure P1 ( p_deptno in number,
                                 p_dname in varchar2,
                                 p_loc in varchar2)
is
begin
insert into dept values (p_deptno,p_dname,p_loc);
dbms_output.put_line('record is inserted through procedure');
end;
```

- There are three types of execution methods supported by in parameter.
  1) Positional Notations
  2) Named Notations
  3) Mixed Notations

1) Positional Notations

Ex : exec p1(1, 'a','b');

2) Named Notations

Ex : exec p1 (p_dname=>'x', p_loc=>'y', p_deptno=>2)

3) Mixed Notations

Ex : exec p1 (1, p_dname=>'m', p_loc=>'n');

2) OUT Mode :

- This mode is used to return values from procedure body.
- OUT Mode internally behaves like a uninitialized variable in procedure body

Ex :

```
Create or replace procedure p1 (a in number, b out number) is
begin
```

```
b:=a*a;
dbms_output.put_line(b);
end;
```

Note : In oracle if a subprogram contains OUT, INOUT Parameters those subprograms are executed using following two methods.

Method - 1 : Using Bind Variable
Method - 2 : Using Annonymous Block

Bind Variable:

- **These variables are session variables.**
- **These variables are created at host environment that's why these variables are also called as host variables.**
- **These variables are not a pl/sql variables, but we can also use these variables in PL/SQL to execute subprograms having OUR Parameters.**

Method - 1 : Bind Variable

Ex :
```
Variable b number;
exec p1 ( 10, :b);
```

Method - 2 : Annonymous Block

Ex :
```
Declare
b  number(10);
begin
p1( 5, b )
dbms_output.put_line( b );
end;
```

Ex : Develop a program for passing employee name as in parameter return salary of that employee using out parameter from emp table?

Prog :
```
Create or replace procedure p1 ( p_ename in varchar2, p_sal out number ) is
begin
select sal  into p_sal from emp where empno=p_ename;
end;
```

Method - 1 : Bind variable

```
variable a number;
exec p1 ( 'KING', :a);
```

Method - 2 : Annonymous Block

```
Declare
a  number(10);
```

```
begin
exec p1( ' ALLEN ', a );
dbms_output.put_line( a );
end;
```

Ex : Develop a program for passing deptno as a parameter return how many employees are working in a dept from emp table?

```
Prog : Create or replace procedure pe2 ( p_deptno in number, p_t out number) is
       begin
       select count(*) into p_t from emp where deptno=p_deptno;
       dbms_output.put_line(p_t);
       end;
```

3) IN OUT Mode

- **This mode is used to pass the values into sub program & return the values from sub programs.**

```
Ex :   Create or replace procedure p1 ( a in out number ) is
       begin
       a := a*a;
       dbms_output.put_line ( a );
       end;
```

Method - 1 : Bind Variable

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

Method - 2 : Annonymous Block

```
Declare
a  number(10) := &n;
begin
p1( a );
dbms_output.put_line ( a );
end;
```

```
Ex :   Create or replace procedure pe4 ( a in out number) is
       begin
       select sal into a from emp where empno=a;
       dbms_output.put_line( a );
       end;
```

⇨ PRAGMA AUTONOMOUS TRANSACTION

- Autonomous transactions are independent transactions used in either procedures or in triggers.
- Generally autonomous transactions are used in child procedures, These procedures are not effected from the main transactions when we are using commit or rollback.

Ex : Create table test (name varchar2(10));

Program :
```
Create or replace procedure P1 is
pragma autonomous_transaction;
begin
insert into ptest values ('india');
commit;
end;
```

Execute The Program:
```
Begin
insert into ptest values ('usa');
insert into ptest values ('uk');
P1;
rollback;
end;
```

With out Autonomous Transaction

Program :
```
Create or replace procedure P1 is
begin
insert into ptest values ('india');
commit;
end;
```

Execute The Program:
```
Begin
insert into ptest values ('usa');
insert into ptest values ('uk');
P1;
rollback;
end;
```

2) Functions

- Function is a named pl/sql block which is used to solve particular task and by default functions return a single value.
- Function is allow to write multiple return statements but it execute only first return statement.
- Function can execute in 4 ways
  1) Annonymous Block
  2) Select Statement
  3) Bind Variable
  4) Exec
- Function also having two parts
  1) Function Specification

2) Function Body

- **In Function Specification we are specifying name of the function and type of the parameters where as in function body we are solving the actual task.**

Ex :
```
Create or replace function fun1( a varchar2)
return varchar2
is
begin
return a;
end;
```

Method - 1 : Select Clause
```
Select fun1('hi') from dual
```

Method - 2 : Annonymous Block
```
Declare
a  varchar2(10);
begin
a :=fun1('hi');
dbms_output.put_line(a);
end;
```

Method - 3 : Bind Variable
```
Variable V Varchar2(20);
Begin
:a:=fun1('hi');
end;
```

Method - 4 : Exec
```
Exec    Dbms_output.put_line(fun1('hi'));
```

Ex :
```
Create or replace function fun2 (a number)
return varchar2
is
begin
if mod(a,2)=0 then
return 'even number';
else
return 'odd number';
end if;
end;
```

Note : We can also use user defined function in insert statement.

Ex :
```
Create table t1(sno number(10), msg varchar2(10));
Insert into t1 values ( 1, fun2(5));
Select * from t1;
```

Ex : Write a pl/sql stored function for passing empno as parameter return gross salary from emp table based on following condition?

Condition => gross:=basic+hra+da+pf;

hra	=>	10% of Sal
da	=>	20% of Sal
pf	=>	10% of Sal

Prog :
```
Create or replace function fun3 (p_empno number)
return number
is
vsal number(10);
gross number(10);
hra number(10);
da number(10);
pf number(10);
begin
select sal into vsal from emp where empno=p_empno;
hra:=vsal*0.1;
da:=vsal*0.2;
pf:=vsal*0.1;
gross:=vsal+hra+da+pf;
return gross;
end;
```

Note : We can also use predefined functions in user defined functions and also this user defined functions in same table or different table.

Ex :
```
Create or replace function fm
return number
is
vsal number(10);
begin
select max(sal) into vsal from emp;
return vsal;
end;
```

Note : If we want to return more number of values from function we are using OUT Parameter.

Ex :
```
Create or replace function fun4
(p_deptno in number
,p_dname  out varchar2
,p_loc    out varchar2)
return varchar2
is
begin
select dname,loc into p_dname,p_loc from dept where deptno=p_deptno;
return p_dname;
end;
```

```
Variable  a  varchar2(10);
Variable  b  varchar2(10);
Variable  c  varchar2(10);

Begin
:a:=fun4 ( 10, :b, :c);
end;

Print b c;
```

Ex : Write a pl/sql stored function for passing empno,date as parameter return number of years that employee is working based on date from emp table?

Prog :

```
Create or replace function fun5(p_empno number,p_date date)
return number
is
a number(10);
begin
select months_between(p_date,hiredate)/12 into a from emp where empno=p_empno;
return (round(a));
end;
```

Execution :

```
Select empno,ename,hiredate,
fun5(empno,sysdate)||'Years' Exp
from emp where empno=7902
```

Ex : Write a pl/sql stored function for passing empno as parameter,calculate tax based on following conditions by using emp table.

Conditions:
1) if annual salary >10000 then tax=10%
2) if annual salary >20000 then tax=20%
3) if annual salary >50000 then tax=30%

Prog :

```
Create or replace function fun7 (p_empno number)
return number
is
vsal number(10);
asal number(10);
itax number(10);
begin
select sal into vsal from emp where empno=p_empno;
asal:=vsal*12;
if asal>10000 and asal<=15000 then
itax:=asal*0.1;
elsif asal>15000 and asal<=2000 then
itax:=asal*0.2;
elsif asal>20000 then
itax:=asal*0.3;
```

```
else
itax:=0;
end if;
return itax;
end;
```

⇨ **Packages**

- Package is a database object which is used encapsulate variables, constants, procedures, cursors, functions, types in to single unit.
- Packages does not accepts parameters, can not be nested, can not be invoked.
- Generally packages are used to improve performance of the application because when we are calling packaged sub program first time total package automatically loaded into memory area.
- Whenever we are calling subsequent sub program calls pl/sql run time engine calling those sub program from memory area.
- This process automatically reduces disk I/O that's why packages improves performance of the application.
- Packages have two types.
  1) Package Specification
  2) Package Body
- In Package Specification we are defining global data and also declare objects, sub programs where as in Package Body we are implementing sub programs and also package body sub program internally behaves like a private sub program.

Package Specification Syntax :

```
Syntax :    Create or Replace Package Package_Name
            Is/As
            Global Variable Declaration;
            Constant        Declaration;
            Cursor          Declaration;
            Types           Declaration;
            Procedure       Declaration;
            Function        Declaration;
            End;
```

Package Body Syntax :

```
Syntax :    Create or Replace Package Body
            Package_Name
            Is/As
            Procedure    Implementations;
            Function     Implementations;
            End;
```

Invoking Packaged Subprograms

1) Exec Package_Name.Procedure_Name (Actual Parameters);
2) Select Package_Name.Function_Name (Actual Parameters) from dual;

Package Specification

Ex :
```
Create or replace package pack1 is
procedure pr1;
procedure pr2;
end;
```

Package Body

Ex :
```
Create or replace package body pack1 is
procedure pr1
is
begin
dbms_output.put_line('first procedure');
end pr1;
procedure pr2
is
begin
dbms_output.put_line('second procedure');
end pr2;
end;

Exec Pack1.pr1;
Exec Pack2.pr2;
```

⇨ **Global Variable**

- ➢ It is one of the variable which is used to define in package specification and implement in package body that variables are call it as a global variables.

⇨ **Local Variable**

- ➢ It is one of the variable which is used to define in programs (Procedure, Function) and implement with in the program only.

Package Specification

Ex :
```
Create or replace package pck2 is
g number(5):=500;
procedure p1;
function f1 ( a number ) return number;
end;
```

Package Body

Ex :
```
create or replace package body pck2 is
procedure p1
is
z number(5);
begin
z:=g/2;
dbms_output.put_line(z);
end p1;
function f1( a number ) return number
```

```
is
begin
return a*g;
end f1;
end;
```

⇨ Procedures Overloading

- Overloading refers to same name can be used for different purposes i.e we are implementing overloading procedures through packages only, those procedures having same name and also different types of arguments.

Package Specification

Ex :
```
Create or replace package pck3 is
procedure p1(a number, b number);
procedure p1(x number, y number);
end;
```

Package Body

Ex :
```
Create or replace package body pck3 is
procedure p1 (a number, b number)
is
c number(10);
begin
c:=a+b;
dbms_output.put_line(c);
end p1;
procedure p1 (x number, y number)
is
z number(10);
begin
z:=x+y;
dbms_output.put_line(z);
end p1;
end;/

Exec Pack.p1 ( a=>10, b=>20 );
Exec Pack.p1 ( x=>100, b=>200);
```

⇨ Forward Declaration

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

Package Specification

Ex :
```
Create or replace package pack14 is
procedure p1;
end;
```

Package Body

```
Ex :    Create or replace package body pack14 is
        procedure p2;
        procedure p1
        is
        begin
        p2;
        end;
        procedure p2
        is
        begin
        dbms_output.put_line('local procedure');
        end p2;
        end;
```

⇨ **Triggers**

- Trigger is also same as stored procedure & also it will automatically invoked whenever DML Operation performed against table or view.
- There are two types of triggers supported by PL/SQL.

  1) Statement Level Trigger
  2) Row Level Trigger

- In Statement Level Trigger, Trigger body is executed only once for DML Statements.
- In Row Level Trigger, Trigger body is executed for each and every DML Statements.

```
Syntax :  create { or replace }       trigger trigger_name
          before / after              trigger event
          insert / update / delete    on table_name
          { for each row }
          { where condition }
          { declare }
          variable declarations, cursors
          begin
          -----
          end;
```

Execution order in Triggers

1) Before Statement Level
2) Before Row Level
3) After Row Level
4) After Statement Level

1) Statement Level Trigger

- In Statement Level Trigger, Trigger body is executed only once for each DML Statement. Thats 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 Operations in saturday and sunday?

```
Program) Create or replace trigger tr1 before insert or update or delete on tt
          begin
          if to_char(sysdate,'DY') in ('SAT','SUN')
          then
         raise_application_error(-20123,'we can not perform DMLs on sat and sunday');
         end if;
         end;
```

Q) Write a pl/sql statement level trigger on emp table not to perform DML Operation on last day of the month?

```
Program ) create or replace trigger tt2 before insert or update or delete on tt
           begin
           if sysdate=last_day(sysdate) then
           raise_application_error (-20111,'we can not perform dml operations on lastday ');
           end if;
           end;
```

Trigger Event (or) Trigger Predicate Clauses

- If you want to define multiple conditions on multiple tables then all database systems uses trigger events.
- These are inserting, updating, deleting clauses
- These clauses are used in either row level or statement level triggers.

```
Syntax : if inserting then
           statements;
           elsif updating then
           statements;
           elsif deleting then
           statements;
           end if;
```

Q) Write a pl/sql statement level trigger on emp table not to perform any dml operation in any days using triggering event?

```
Program ) create or replace trigger tr3 before insert or update or delete on tt
           begin
           if inserting then
           raise_application_error (-20121,'we can not perform inserting operation');
           elsif updating then
           raise_application_error (-20122,'we can not perfrom update operation');
           elsif deleting then
```

```
                raise_application_error (-20123,'we can not perform deleting operation');
                end if;
                end;
```

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

```
create or replace trigger tr4 after insert or update or delete on tt
declare
a varchar2(50);
begin
if inserting then
a := 'rows inserted';
elsif updating  then
a := 'rows updated';
elsif deleting then
a := 'rows deleted';
end if;
insert into testt values (a);
end;
```

2) Row Level Trigger

- In Row Level Trigger, Trigger body is executed for each row for DML Statement, Thats why we are using for each row clause in trigger specification and also data internally stored in 2 rollback segment qualifiers are OLD & NEW
- These qualifiers are used in either trigger specification or in trigger body. when we are using these modifiers in trigger body we must use colon prefix in the qualifiers.

  Syntax - :old.column_name (or) :new.column_name.

- When we are using these qualifiers in when clause we are not allow to use colon infront of the qualifiers.

Qualifier	Insert	Update	Delete
:new	YES	YES	NO
:old	NO	YES	YES

- In Before Triggers, Trigger body is executed before DML Statements are effected into database.
- In After Triggers, Trigger body is executed after DML Statements are effected into database.
- Generally if we want to restrict invalid data entry always we are using before triggers, where as if we are performing operation on the one table those operations are effected in 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 a emp table sal should be more than 5000?

Program) Create or replace trigger t90 before insert on tb
```
for each row
begin
if :new.sal<5000 then
raise_application_error (-20123,'salary should be more than 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?

Program) Create or replace trigger t1
```
 after delete on dept
 for each row
 begin
delete from emp where deptno=:old.deptno;
end;
```

Q) Write a PL/SQL Row Level Trigger on dept table whenever updating deptno's in dept table automatically those deptno's modified into emp table?

Program) Create or replace trigger t19
```
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 whenever user inserting data into ename column after inserting data must be converted into uppercase ?

Program) create or replace trigger t21
```
before insert on emp
for each row
begin
:new.ename:=upper(:new.ename);
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

Program) First we create 3 tables which are having the same structure of emp table.
```
Create or replace trigger te1
after insert or update or delete on t01
for each row
begin
```

```
if inserting then
insert into e1(empno,ename) values (:new.empno,:new.ename);
elsif updating then
insert into e2(empno,ename) values (:old.empno,:old.ename);
elsif deleting then
insert into e3(empno,ename) values (:old.empno,:old.ename);
end if;
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 the delete statment?

Program)
```
Create or replace trigger tp1 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

Ex :
```
Create or replace trigger tp1 after delete on emp
for each row
declare
a number(10);
begin
select count(*) into a from emp;
dbms_output.put_line('remaining records are: '||a);
end;
```

- **Into a Row Level Trigger based on a table trigger body can not read data from same table and also we can not perform DML Operations on same table.**
- **If we are trying to this oracle server returns an error is table is mutating.**
- **This Error is called Mutating Error**
- **This Trigger is called Mutating Trigger**
- **This Table is called Mutating Table**
- **Mutating Errors are not accured in Statement Level Trigger Because through these Statement Level Trigger when we are performing DML Operations automatically data Committed into database.**
- **Where as in Row Level Trigger 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 accured.**
- **To avoid this mutating error we are using autonomous transaction in triggers.**

Ex

```
Create or replace trigger tp1 after delete on t01
for each row
declare
pragma autonomous_transaction;
a number(10);
begin
select count(*) into a from t01;
dbms_output.put_line('remaining records are: '||a);
commit;
end;
```

DDL Triggers

- We can also create triggers on schema level, database level. These types of triggers are called DDL Triggers or System Triggers.
- These types of triggers are created by database administrator.

Syntax :

```
Create or replace trigger trigger_name
Before / After
Create / Alter / Drop / Truncate / Rename
On Username.Schema
```

Q) Write a PL/SQL Trigger on scott schema not to drop emp table?

Program)

```
Create or replace trigger td
before drop on apps.schema
begin
if ora_dict_obj_name = 'T100' and
ora_dict_obj_type = 'TABLE' then
raise_application_error(-20121,'we can not drop this table');
end if;
end;
```

Collections

- Oracle server supports following types
  1) PL/SQL Record (or) Record Type
  2) Index by table (or) PL/SQL table (or) Associative Arrays.
  3) Nested tables
  4) Varrays
  5) Ref Cursors

Index By Table

- This is an user defined type which is used to store multiple data items in to a single unit. Basically this is an unconstraint table
- Generally these tables are used to improve performance of applications because these tables are stored in memory area thats why these tables are also called as memory tables.
- Basically these table 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 character and also these values are either -ve or +ve.
- These indexes key behaves like a primary key i.e does not accept duplicate and null values. basically this key datatype is binary_integer.
- Index by table having following 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;
v_t t1;
begin
v_t(1):=10;
v_t(2):=20;
v_t(3):=30;
v_t(4):=40;
v_t(5):=50;
dbms_output.put_line(v_t(3));
dbms_output.put_line(v_t.first);
dbms_output.put_line(v_t.last);
dbms_output.put_line(v_t.prior(3));
dbms_output.put_line(v_t.next(4));
dbms_output.put_line(v_t.count);
dbms_output.put_line(v_t(5));
end;
```

Ex -2

```
declare
type t1 is table of number(10)
index by binary_integer;
v_t t1;
begin
v_t(1):=10;
v_t(2):=20;
v_t(3):=30;
v_t(4):=40;
v_t(5):=50;
dbms_output.put_line(v_t.count);
v_t.delete(2,3);
dbms_output.put_line(v_t.count);
v_t.delete;
dbms_output.put_line(v_t.count);
```

```
end;
```

Q) Write a PLSQL program to get all employee names from emp table and store it into index by table and display data from index by table?

Program)
```
declare
type t1 is table of varchar2(10)
index by binary_integer;
v_t t1;
cursor c1 is select ename from emp;
n number(5):=1;
begin
open c1;
loop
fetch c1 into v_t(n);
exit when c1%notfound;
n:=n+1;
end loop;
close c1;
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;
```

Program)
```
declare
type t1 is table of varchar2(10)
index by binary_integer;
v_t t1;
begin
select ename bulk collect into v_t from emp;
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;
```

Program)
```
declare
type t1 is table of date
index by binary_integer;
v_t t1;
begin
for i in 1..10
loop
v_t(i):=sysdate+i;
end loop;
for i in v_t.first..v_t.last
loop
```

```
dbms_output.put_line(v_t(i));
end loop;
end;
```

Q) Write a PLSQL Program to retrieve all joining dates from emp table and store it into index by table and display content from index by table?

Program)
```
declare
type t1 is table of date
index by binary_integer;
v_t t1;
begin
select hiredate bulk collect into v_t from emp;
for i in v_t.first..v_t.last
loop
dbms_output.put_line(v_t(i));
end loop;
end;
```

Ex :
```
declare
type t1 is table of varchar2(10)
index by varchar2(10);
v_t t1;
x varchar2(10);
begin
v_t('a'):= 'ARUN';
v_t('b'):= 'AJAY';
v_t('c'):= 'ABHI';
x :='a';
loop
dbms_output.put_line(v_t(x));
x := v_t.next(x);
exit when x is null;
end loop;
end;
```

Ex :
```
declare
type t1 is table of emp%rowtype
index by binary_integer;
v_t t1;
x number(5);
begin
select * bulk collect into v_t from emp;
x:=1;
loop
dbms_output.put_line(v_t(x).empno||','||v_t(x).ename);
x:=v_t.next(x);
exit when x is null;
```

```
end loop;
end;
            ( OR )

Ex :  declare
      type t1 is table of emp%rowtype
      index by binary_integer;
      v_t t1;
      begin
      select * bulk collect into v_t from emp;
      for i in v_t.first..v_t.last
      loop
      dbms_output.put_line(v_t(i).empno||','||v_t(i).ename);
      end loop;
      end;
```

Nested Tables

- This is also user defined type which is used to store multiple data items in a single unit but before we 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 introduce Nested Tables to extension of the index by tables.
- These user defined types stored permanently into database using sql.
- In Index by tables we can not add or remove the 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.

```
Syntax : Type type_name is Table of datatype( size );
         variable_name     type_name( ); =>  Constructor_Name
```

```
Ex :  Declare
      type t1 is table of number(10);
      v t1:=t1();
      begin
      v.extend(100);
      v(100):=10;
      dbms_output.put_line(v(100));
      end;

Ex :  Declare
      type t1 is table of number(10);
      v1 t1:=t1(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 :

```
Declare
type t1 is table of number(10);
v1 t1;
v2 t1:=t1();
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;
```

Ex :

```
declare
type t1 is table of number(10);
v t1:=t1();
begin
v.extend;
v(1):=5;
dbms_output.put_line(v(1));
end;
```

Q) Write a PLSQL program to get all employee names from emp table and store it into Nested Table and display data from Nested Table?

Program)

```
declare
type t1 is table of varchar2(10);
v t1:=t1();
cursor c1 is select ename from emp;
n number(10):=1;
begin
for i in c1
loop
v.extend();
v(n):=i.ename;
n:=n+1;
```

```
                end loop;
                for i in v.first..v.last
                loop
                dbms_output.put_line(v(i));
                end loop;
                end;
                        ( OR )
Program)  declare
                type t1 is table of varchar2(10);
                v t1:=t1();
                begin
                select ename bulk collect into v from emp;
                for i in v.first..v.last
                loop
                dbms_output.put_line(v(i));
                end loop;
                end;

Program)  declare
                type t1 is table of emp%rowtype;
                v t1:=t1();
                begin
                select * bulk collect into v from emp;
                for i in v.first..v.last
                loop
                dbms_output.put_line(v(i).empno||','||v(i).ename||','||v(i).job);
                end loop;
                end;
```

Varrays

- **This is also user defined type which is used to store multiple data items in a single unit but before 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 the Varrays for retrieving the huge data.**

Syntax : Type type_name is varray(maxsize) of datatype(size);
Variable_name Type_name := Type_name( );

```
Program ) Declare
                 type t1 is varray(50) of emp%rowtype;
                 v t1:=t1();
                 begin
                 select * bulk collect into v from emp;
                 for i in v.first..v.last
                 loop
                dbms_output.put_line(v(i).empno||','||v(i).ename||','||v(i).job);
                end loop;
                end;
```

➢ **Difference b/w Index by Table, Nested Table, Varrays**

Index by Table	Nested Table	Varrays
1) It is not stored permanently in database. 2) We can not add or remove indexes. 3) Indexes starting from negative to positive numbers and also having key value pairs.	1) It is stored permanently in database by using sql. 2) We can add or remove indexes using extend, trim method. 3) Indexes starting from 1.	1) It is stored permanently in database by using sql. 2) We can add or remove indexes using extend, trim method. 3) Indexes starting from 1.

Bulk Mechanism

- Bulk is one of the method which is used to improve the performance of the applications.
- Oracle introduce bulk bind process using collection 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
  1) Bulk Collect
  2) Bulk Bind

1) Bulk Collect

- In this clause we are used to fetch the data from resource into collection
- This clauses used in
  1) Select ....into........... clause
  2) Cursor...........Fetch.......... Statement
  3) Dml............Returning........ Clauses

1) Bulk Collect used in select .....into .....clause
Syntax : select * bulk collect into collection_name from table_name.

Program)
```
Declare
type t1 is table of emp%rowtype
index by binary_integer;
v t1;
begin
select * bulk collect into v from emp;
for i in v.first..v.last
loop
dbms_output.put_line(v(i).empno||','||v(i).ename||','||v(i).job);
end loop;
end;
```

2) Bulk Collect used in cursor......fetch...... statement
Syntax : fetch cursor_name bulk collect into collection_variable.

Program)
```
Declare
type t1 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 with out 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(z2-z1);
end;
```

Time Program with BULK

```
Declare
type t1 is table of varchar2(50) index by binary_integer;
v1 t1;
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 v1;
```

```
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(z2-z1);
end;
```

3) Bulk Collect used in DML .... Returning clauses.

Syntax : dml statement returning column_name into variable_name;

Ex :

```
Variable a varchar2(10);
Update emp set sal=sal+100 where ename ='KING' returning job into :a;
Print a;
```

Q) Write a PLSQL Stored Procedure modify salaries of the clerk from emp table and also these modified value immediately stored into index by table by using dml ... returning clause and also display content from index by table?

Program)

```
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;
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||','||v1(i).sal);
end loop;
end;
```

2) Bulk Bind

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

Syntax : forall indexvar in collectionvar.frist..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

```
Program )   Declare
            type t1 is table of number(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+111 where empno=v1(i);
            end;
```

Bulk Delete

```
Program )   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

```
Program )   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 (v1(i));
            end;
```