**PL/SQL**

* Introduction to Programming Language
* Procedures Vs Non-Procedures Language
* Limitation of ANSI SQL and Oracle SQL
* Introduction to Oracle PL/SQL
* PL/SQL Usage in Production Database
* Key benefits of PL/SQL over SQL
* Anchor datatypes or Attributes
* Composite datatypes
* Collections
* PL/SQL block structure & Designing
* Scope and Visibility
* Constructs of PL/SQL
* Assignment operations
* Debugging statement
* Flow Control Statement
* IF / NESTED IF / EXIT / GOTO /
* Iterative statements
* Simple Loop / While Loop / For Loop
* Continue statement(11g)
* Embedded SQL
* Introduction to Embedded SQL
* Role of Embedded SQL in PL/SQL
* Constructs of Embedded SQL
* Transaction Mngmt Using Embedded SQL
* Dynamic SQL
* Introduction to Dynamic SQL
* Usage of Dynamic SQL in PL/SQL
* Introduction to Exceptions
* Importance of Exceptions in PL/SQL
* Type of Exceptions
* Exception handling
* Save exceptions
* Introduction and STANDARD Package
* Introduction to User Defined Exceptions
* Non predefined exceptions
* Exception cases
* Usage of PRAGMA EXCEPTION\_INIT()
* Cursor Management in PL/SQL
* Introduction to cursor management
* Pictorial presentation of cursor mechanism
* Introduction and usage of implicit cursor
* Introduction and usage of Explicit cursors
* Cursor attributes
* Cursor using simple loop
* Cursor using while loop
* Cursor using for loop
* Cursor exceptions
* Cursor expression
* Data Locking
* Data Manipulation through Cursor
* REF cursor(strong and weak)
* Using ref cursor variable as parameter
* Bulk Fetch and Bulk Data Retrieval in PL/SQL
* Bulk Collection
* Bulk Binding mechanism of Cursor (for all statement)
* Dynamic behavior of cursor mnt.
* SUB PROGRAMS
* Types of PL/SQL blocks
* Labelled blocks
* Anonymous PL/SQL blocks
* Named PL/SQL blocks
* Forward Declaration of Local Block
* Introduction to storage PL/SQL block
* Stored Procedures
* Stored Functions
* Nocopy(9i)
* Autonomous Transaction Management of PL/SQL
* PACKAGES
* Introduction to Package
* Stand-alone schema Vs Packaged Object
* Encapsulation mechanism of Package
* Data security
* Function overloading mechanism of Package
* Introduction and Usage of Package
* Oracle supplied packages
* Package data
* Restrict Reference and Complex Hints
* Usage of Pragma Serially\_ reusable
* DB TRIGGERS
* Introduction to Database Trigger
* Types of Triggers
* Triggering events
* Usage of Old & New Reference
* Instead of  Trigger
* Enforcing the referential integrity constraint
* Compound Trigger(11g);
* Follows key word(11g);
* Defining a disable trigger(11g)
* Trigger Cascading
* Enabling/Disabling Trigger
* Schema Trigger
* Table Mutation Error
* Transaction Audit Trigger
* Advanced Pl/sql Topics
* User Defined Types (RECORDS)
* Subtypes of Pl/sql
* Automation Transaction
* Advantages of Autonomous Transaction
* Usage of Autonomous Transaction
* Scope of autonomous Transaction
* Usage of Autonomous Transaction in Trigger
* Suing FORALL Statement
* About % BULK\_ROWCOUNT
* FGA and FGAC(VPD)
* Table functions
* Managing database dependencies
* Designing pl/sql code
* Using collections
* Working with lobs
* Using secure file lobs
* C ompiling pl/sql code
* Tuning pl/sql code
* Pragma inline(11g)
* Caching to improve performance
* Analysing pl/sql code
* Profiling pl/sql code
* Tracing pl/sql code
* Safeguarding pl/sql code against sql injection
* Pl/sql Architecture.

**PL/SQL**

**TOPICS**

1. introduction
2. composite data types
   1. %type attribute or anchor data types
      1. %type
      2. %row type

User defined or temporary data types

* + 1. Record type or pl/sql record
    2. Index by table or pl/sql table or associate array
  1. User defined or permanent data types
     1. Objects
     2. Varrays
     3. Nested tables/ pl/sql table

1. Blocks
   1. Unnamed or anonymous block
   2. Named or stored procedures
   3. Labeled blocks
2. Control statements or control structures
   1. Sequential (default)
   2. Conditional (if, case)
3. Cursors
   1. Implicit
   2. Explicit
   3. Ref cursor
4. Exceptions (To handle run time errors)
   1. predefine
   2. user define
   3. non predefine
5. stored procedure
   1. procedures
   2. functions
   3. packages
   4. triggers
6. pragmas
7. no copy
8. forward declaration
9. bulk collect & bulk bind
10. mutating triggers
11. dynamic sql (execute immediate)
12. table function
13. advanced topics

* This is a procedural language or programming language
* It consist of unit blocks
* Makes the application design easy
* Provides security and is portable
* It is used to handle multiple statements
* To define our own logics we use Pl/sql
* It supports control statements like IF conditions, Loops, go to.. etc.
* By using pl/sql we can handle runtime errors
* Explicitly we can define our own cursors in pl/sql. It allows Boolean data types
* PL/SQL has tight integration with oracle data types.

**Blocks:**

**Unnamed Blocks:-**

**Syntax:**

Declare

begin

code;

Exception

End;

Optional (declarative section)

Mandatory (Executable Section)

Optional (Exception Section)

Declare

begin

code;

Exception

End;

Optional (declarative section)

Mandatory (Executable Section)

Optional (Exception Section)

**Declarative Section:-**

* We use declarative section to define variables, constants, cursors, exceptions, etc.,
* We have to define the things in declarative section which are not understand by PL/SQL engine which is optional?

**Excitable Section:-**

* Here we define or we provide coding path
* Execution takes place in executable section which is mandatory
* It starts from begin to end

**Exception section:-**We use exception section to handle runtime errors and which is optional

**Nested Blocks:** Block within another block is called nested block

Declare

begin

Exception

End;

Inner block

Declare

begin

code;

Exception

End;

Outer block

or

Enclosing Block

Declare

begin

Exception

End;

Declare

begin

code;

Exception

End;

or



**Example:-**

**1. Write a program to display the message on to the screen?**

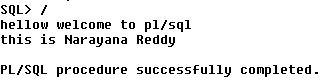
begin

dbms\_output.put\_line('hello welcome to pl/sql');

dbms\_output.put\_line('this is Narayana Reddy');

end;

**Output:-**



**Using ‘NULL’ as a statement:-**

Begin

NULL;

End;

pl/

sql

procedure successfully completed

Begin

End;

o/p

not coming w/o using stmt.

Begin

NULL;

End;

pl/

sql

procedure successfully completed

Begin

End;

o/p

not coming w/o using stmt.

2 Using Constant:**-** We don’t change modifications in entire block

Declare

A constant number (5,3) := 3.142;

B number (5) := 100;

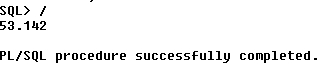
Begin

B:=50;

Dbms\_output.put\_line(a+b);

End;

**Output:**



3. Change output automatically by using the program

Declare

A number (5): = &n;

B number (5):= &m;

Begin

Dbms\_output.put\_line (a+b);

End;

4. Write a program to calculate the area of the circle by knowing the radius

declare

a number (5,3);

r number (5) := &n;

p constant number (5,3) := 3.142;

begin

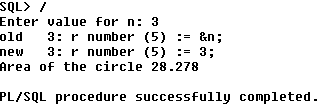
a:=p\*power(r,2); (or you can use p\*r\*r)

dbms\_output.put\_line('Area of the circle '||a);

end;

/

**Out Put:**



Nesting of comments is not possible

**Ex:-**

/\*

Stmts

/\*

Stmts

\*/

stmts \*/

**2) Composite datatypes**

**a) %type**

**Ex:-**Write a program to retrieve and display the employee name by providing employee number?

declare

vno number(5):=&n;

vname varchar2(10);

begin

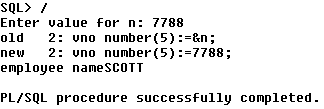
select ename into vname from emp where empno = vno;

dbms\_output.put\_line('employee name'||vname);

end;

/

**Output:-**



* It is used to provide column data type to a variable.
* No need to remember the column data type
* It is dynamic in nature

**Syn:**Variablename tablename.columnname%type;

**Ex:-**variable emp.ename%type;

declare

vno emp.empno%type:=&n;

vname emp.ename%type;

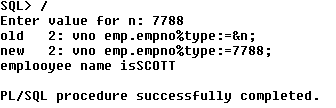
begin

select ename into vname from emp where empno = vno;

dbms\_output.put\_line('emplooyee name is'||vname);

end;/

**Output:**



**Disadvantage:-** Not possible to store entire record

**b) %row type:** It assigns the entire column data types of a table. Useful to store entire record.

Syn:- Variable tablename%rowtype;

Ex:

declare

vno emp.empno%type:=&n;

vrow emp%rowtype;

begin

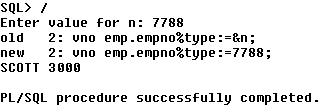
select \* into vrow from emp where empno = vno;

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

end;

/

**Output:**



**Disadvantage:-** by using %rowtype it is not possible to store table record along with user data or user information

**4. CONTROL STATEMENTS**

Control flow

1. if statement
2. case statement

1. If condition:

**Syntax:-**

if condition then

Do1; (stmts)

Else if condition then

Do2; (stmts)

Else do3; (stmts)

End if;

2. Nested if:

**Syntax:-**

If condition then

Do1;

If condition then

Do3;

Else

Do4;

End if;

Else

Do2;

End if;

**Ex:-**

declare

v number(5) :=&n;

begin

if v>1000 then

dbms\_output.put\_line('hellow given number is > 1000');

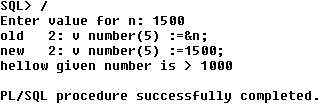
else

dbms\_output.put\_line('number is < 1000');

end if;

end;/

**Output:-**



**Case:**

1. Simple case
2. Search case

**Syntax:**

Case [columns/expressions]

When condition then

Do1; (stmts)

When condition then

Do2; (stmts)

When condition then

Do3; (stmts)

Else

Do4; (stmts)

End case;

**2. Search case:**

**Ex:-**

SELECT sal,CASE WHEN sal =5000

THEN'A'

WHEN sal >=3000

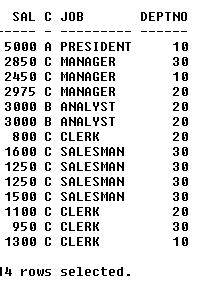
THEN'B'

ELSE'C'

ENDCASE,job, deptno

FROMemp

**Output:**

****

**Ex:-**

SELECTSUM(CASE

WHEN sal =5000

THEN sal

WHEN sal >=3000

THEN sal

ELSE sal

END

) sum\_sal

FROMemp

**Output:**

****

**Ex:-**

Declare

vsal emp.sal%type;

vno emp.empno%type:=&n;

begin

select sal into vsal from emp where empno = vno;

case when vsal = 5000 then

dbms\_output.put\_line('sal = 5000');

when vsal >= 3000 then

dbms\_output.put\_line('sal = 3000');

else

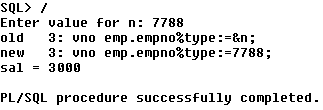
dbms\_output.put\_line('sal < 3000');

end case;

end;

/

**Output:**

****

**Note:-** In the absence of else part if all of the conditions are false then it throws on an error i.e. case not found. (it throws err ORA-06592, pls check below o/p)

**Ex:-**

declare

vsal emp.sal%type;

vno emp.empno%type:=&n;

begin

select sal into vsal from emp where empno = vno;

case

when vsal > 5000 then

dbms\_output.put\_line('sal = 5000');

when vsal > 6000 then

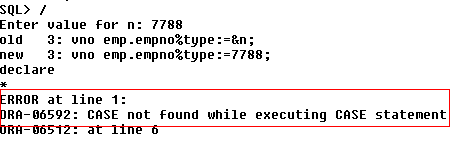
dbms\_output.put\_line('sal = 6000');

end case;

end;

/

**Output:**



**Simple case:-** It is not allow special operators

Ex:-

declare

vsal emp.sal%type;

vno emp.empno%type:=&n;

begin

select sal into vsal from emp where empno = vno;

case vsal

when 5000 then

dbms\_output.put\_line('sal = 5000');

when 3000 then

dbms\_output.put\_line('sal = 3000');

else

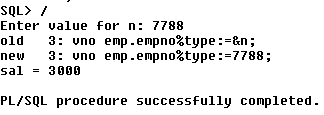
dbms\_output.put\_line('sal < 3000');

end case;

end;

/

**Output:**

****

**c) Iterations:**

**Loops:-** to execute the same statement or coding for repeated times we use loops

1. Simple loop
2. While loop
3. for loop
   1. Numeric for loop
   2. Cursor for loop
4. **Simple loop:-** it is an infinite loop explicitly we have to stop the loop. It is uses in blocks.

**Syntax:-**

Loop

Code;

End loop;

Ex:-

Begin

Loop

Exit [when condition];

Code;

End loop;

End;

**Ex:-** Write a program to display 1 to 10 numbers

declare

x number(5) :=1;

begin

loop

exit when x > 10;

dbms\_output.put\_line(x);

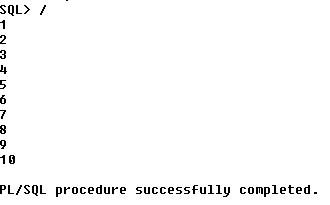
x:=x+1;

end loop;

end;

/

**Output:**

****

**Ex:-** To display 1 to 5 numbers

declare

x number(5) :=1;

begin

loop

if x > 5 then

exit;

end if;

dbms\_output.put\_line(x);

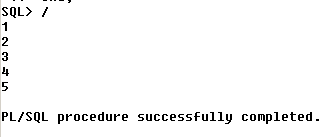
x:=x+1;

end loop;

end;

/

**Output:**

****

**Ex:-**

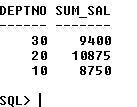
SELECT deptno,SUM(NVL(sal,100)) sum\_sal

FROMemp

GROUPBY deptno

HAVINGSUM(sal)>200

**Output:**

****

**While loop:-** It is a pre conditional loop

**Syntax:-** While condition loop

Code;

End loop;

**Ex:-** to display 1 to 10 numbers

declare

a number(5) := 1;

begin

while a <=10 loop

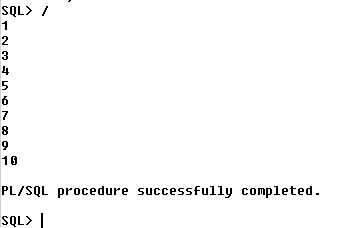
dbms\_output.put\_line(a);

a:=a+1;

end loop;

end;

**Output:**

****

* Write a program to display tables from 1 to 10 by using simple loop

**Ex:-**

declare

x number(5) :=1;

y number(5) := 1;

begin

loop

y:=1;

loop

dbms\_output.put\_line(x||'\*'||y||'='||x\*y);

y:=y+1;

exit when y>10;

end loop;

x:=x+1;

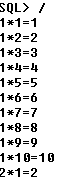
exit when x>10;

end loop;

end;

/

**Output:**

****

It will print 1 to 10 tables;

* Write a program to reverse the give string by using simple loop (without using reverse function)

**For loop:**

**Syntax:**

For var in [reverse] val1..val2 loop

Code;

End;

**Ex:-**Begin

For I in 1..10 loop

Dbms\_output.put\_line(i);

End loop;

End;

**Ex:-**

Begin

For I in reverse 1..10 loop

Dbms\_output.put\_line(i);

End loop;

End;

**Continue:** (11g introduced)

**Syntax:** Continue [When condition];

* It is a 11g feature. Which is used in loop (in any loop)
* Continue statement skips the current iteration

**Ex:-**

Begin

For I in 1..10 loop

Continue when i>5;

Dbms\_output.put\_line(i);

End loop;

End;

**Ex:-**

Begin

For I in 1..10 loop

If I <5 then

Continue;

End if;

Dbms\_output.put\_line(i);

End loop;

End;

**CURSORES**

* Oracle will make use of internal memory areas (implicit cursors) for sql statements to process the records
* This memory areas will be defined in a area called SGA (system global area)
* Oracle allows to provide our own memory areas (explicit cursors) to get control over each of the record

**Definition:-**It is a pointer pointing towards the arranged data in a context area

In explicit cursors user has to declare, open, fetch and close the cursors. Whereas in the case of impolicit cursors system has to look after all this functionalities.

**Cursor functionalities:**

**Step 1:** Declaring the cursor

**Syntax:-**cursor cursor\_name is select …

**Step 2:**opening a cursor

**Syntax**: open cursor\_name

**Step 3:** Fetching records into variable from cursor

**Syntax**: fetch cursor\_name into variable;

**Step 4:** closing cursor

**Syntax:** close cursor\_name

**Cursor Attributes**

1. Cursor\_name%isopen;
2. Cursor\_name%found;
3. Cursor\_name%notfound;
4. Cursor\_name%rowcount;
5. Cursor\_name%bulk\_rowcount;
6. Cursor\_name%bulk\_exception; (save exception 11g)

Cursor\_name%isopen: returns true if cursor is opened else false

Cursor\_name%found: returns true if records are found else false

Cursor\_name%notfound: reverse to the found

Cursor\_name%rowcount: returns number of records fetched to that state

**Example for explicit cursor:**

Declare

Cursor c is select ename for emp;

Vname c%rowtype;

Begin

Open c;

Loop

Fetch c into vname;

Exit when c%not found;

Dbms\_output.put\_line(vname.ename);

End loop;

End;

**Ex:** Declare

Cursor c is select ename for emp;

Vname emp.ename%type;

Begin

Open c;

Loop

Fetch c into vname;

Exit when c%notfound;

Dbms\_output.put\_line(vname);

End loop;

Close c;

End;

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cursor event |  | %isopen | %found | %notfound | %rowcount |
| Open cursor | Before | F | E | E | E |
|  | AFTER | T | N | N | O |
| 1ST FETCH | BEFORE | T | N | N | O |
|  | AFTER | T | T | F | I |
| 2ND FETCH (FETCHES) | BEFORE | T | T | F | I |
|  | AFTER | T | T | F | DD |
| LAST FETCH | BEFORE | T | T | F | DD |
|  | AFTER | T | F | T | DD |
| CLOSE CURSOR | BEFORE | T | F | T | DD |
|  | AFTER | F | E | E | E |

E -> Exceptions DD -> data dependence

In for loops no need to open fetch and close the cursors. It happens through for loop

**For loop cursors:**

Declare

cursor c is select ename for emp;

begin

for I in c loop

dbms\_output.put\_line(i.ename);

end loop;

end;

**Parameterised Cursors:**

In cursors we use “In” parameters

**Ex:**

Declare

Cursor c(x emp.deptno%type) is select ename from emp where deptno = x;

Vname emp.ename%type;

Vno number(5):=&n;

Begin

Open c(vno);

Loop

Fetch c into vname;

Exit when c%notfound;

Dbms\_output.put\_line(vname);

End loop;

Close c;

End;

**Using for loop:**

Ex:

Declare

Cursor c(x emp.deptno%type) is select ename from emp where deptno = x;

Begin

For I in c(vno) loop

Dbms\_output.put\_line(i.ename);

End loop;

End;

**Static cursors:**

we can open the cursor n no.of times

Ex:-

Declare cursor c is select ename from emp;

Vname emp.ename%type;

Begin

Open c;

Loop

Fetch c into vname;

Exit when c%notfound;

Dbms\_output.put\_line(vname);

End loop;

Close c;

Open c;

End;

**a) Implicit Cursors**

Declare

Vno emp1.deptno%type:=&n;

Begin

Delete from emp1 where deptno = vno;

If sql%rowcount >3 then

Dbms\_output.put\_line(sql%rowcount||’employees not possible to delete’);

Rollback;

Else

Dbms\_output.put\_line(sql%rowcount||’employees possible to delete’);

Commit;

End if;

End;

We use implicit cursors to find out the status of DML operations.

**REF CURSOR**

* Ref cursor is a datatype
* We use ref cursors to handle multiple select statements
* We can pass ref cursor variable to a parameter value

Syntax: type typename is ref cursor;

Var typename;

Ex:-

Declare

Type rec is refcursor;

Vrec rec;

Vemp emp.ename%type;

Vdept dept.loc%type;

Begin

Open vrec for select ename from emp;

Loop

Fetch vrec into vemp;

Exit when vrec%notfound;

Dbms\_output.put\_line(vemp);

End loop;

Close vrec;

Dbms\_output.put\_line(‘++++++++’);

Open vrec for select dname from dept;

Loop

Fetch vrec into vdept;

Exit when vrec%notfound;

Dbms\_output.put\_line(vdept);

End loop;

Close vrec;

End;

**For update cursor:**

* We use for update clause to provide row level locking in a table
* Where current of cursorname
* We use current of cursorname to refer the records of a table which are processed by a cursor
* We mainly use this to simplify the coding path, without for update it is not possible to provide/refer where current of clause

**Ex:**

Declare

Cursor c is select \* from emp1 where deptno = &n for update no wait;

Vrow emp1%rowtype;

Begin

Open c;

Loop

Fetch c into vrow;

Exit when c%notfound;

Update emp1 set sal = 6000 where current of c;

End loop;

End;

**Ref cursors are two types**

1. Strong ref cursors
2. Weak ref cursors

* If you restrict the ref cursor datatype with return datatype such refcursors are called “strong ref cursors”
* In strong ref cursors we have to process the only the records which are compartable to the return datatype.
* In weak refcursors there is no restrictions which means we can process records of variable data types (different data types)

**Ex:-**

Declare

Type rec is ref cursor return emp%rowtype;

Vrec rec;

Vemp emp%rowtype;

Vdept dept%rowtype;

Begin

Open vrec for select \* from emp;

Loop

Fetch vrec into vemp;

Exit when vrec%notfound;

End loop;

Close vrec;

Dbms\_output.put\_line(‘++++++’)

Open vrec for select \* from dept;

End;

**Cursor expressions:**

**EXCEPTIONS**

Basically or generally errors are of two types

1. Syntactical
2. Runtime

* We will get syntactical errors ar compilation time they itself we can rectify the compilation errors
* To handle the runtime errors we use exceptions
* Types of exceptions

User defined

Pre-defined

Non predefined

* In user defined exceptions user has to declared the exceptions, raise and the handle the exceptions
* We handle the errors in exception section by using exception handlers
* Whereas in the case of predefined manufacturer as to define system has to raise and user as to handle

**Syntax**

Declare

Exceptioname exception;

Begin

Raise exceptionname;

Exception

When exceptionname then

When

Exception then

End;

**Predefined Exceptions**

**Ex:-**

Declare

Vno emp.empno%type :=&n;

Vname emp.ename%type;

Begin

Select ename into vname from emp where empno = vno;

Dbms\_output.put\_line(‘hello’);

Exception

When too\_many\_rows then

Dbms\_otuput.put\_line(‘more than one row’);

When no\_data\_found then

Dbms\_output.put\_line(‘no such employee’);

When others then

Dbms\_output.put\_line(‘other error’);

End;

|  |  |
| --- | --- |
| Oracle Err # | Exception name |
| NO\_data\_found | 100 |
| TOO\_MANY\_ROWS | -1422 |
| CASE\_NOT\_FOUND | -6592 |
| INVALID\_NUMBER | -1722 |
| VALUE\_ERROR | -6502 |
| CURSOR\_ALREADY\_OPEN | -6511 |
| INVALID\_CURSOR | -1001 |
| ZERO\_DIVIDED | -1476 |
| DUP\_VAL\_ON\_INDEX | -0001 |
| COLLECTION\_IS\_NULL | -6531 |
| SUBSCRIPT\_BEYOND\_COUNT | -6533 |
| SUBSCRIPT\_OUTSIDE\_LIMIT | -6532 |
| PROGRAM\_ERROR | -6501 |
| STORAGE\_ERROR | -6500 |
| LOGIN\_DENIED | -1017 |
| MEMORY\_ERROR |  |
| SELF\_IS\_NULL | -30625 |
| ROWTYPE\_MISMATCH | -6504 |
| NO\_DATA\_NEEDED | -6548 |
| OTHERS |  |

**User\_defined exceptions:**

Declare

Ex exception;

Vno emp.empno%type :=&n;

Vrow emp%rowtype;

Begin

Select \* into vrow from emp where empno = vno;

If vrow.comm is null then

**CASES IN EXCEPTIONS OR EXCEPTIONAL CASES**

**CASE: 1**

DECLARE

----------------;

BEGIN

----------------;

EXCEPTION

WHEN OTHERS THEN;

----------------------------------;

WHEN EX THEN;

---------------------------------;

END;

**NOTE:** It throws an error. Others should not be at first in the list of exceptions. It should be always at last among exceptions.

**CASE: 2**

DECLARE

----------------;

BEGIN

----------------;

EXCEPTION

WHEN EX1 OR EX2 THEN;

---------------------------------------;

END;

**NOTE:** It is possible to mention the exception in series by separating with ‘OR’ operator.

**CASE: 3**

DECLARE

-------------

BEGIN

-------------

EXCEPTION

WHEN EX1 OR OTHERS THEN;

-----------------------------------------------

END;

**NOTE:** It is not possible to mention the exception in series by separating with’OR’ operator.

**CASE: 4**

DECLARE

----------------;

BEGIN

----------------;

EXCEPTION

WHEN EX1 AND EX2 THEN;

------------------------------------------;

END;

**NOTE:** It is not possible to mention the exception in series by separating

with AND operator.

**CASE: 5**

DECLARE

------------------

BEGIN

BEGIN

RAISE EX;

EXCEPTION;

WHEN EX2 THEN;

END;

EXCEPTION

WHEN EX THEN;

END;

**NOTE**: Exceptions raised in inner block can be handled in the outer

block. This is called as EXCEPTION PROPAGATION.

**CASE:6**

DECLARE

---------------;

BEGIN

RAISE EX2;

BEGIN

RAISE EX;

EXCEPTION

WHEN EX2 THEN;

END;

EXCEPTION

WHEN EX THEN;

END;

**NOTE:** Exception raised in outer block cannot be handled in inner block.

**CASE: 7**

DECLARE

----------------

BEGIN

BEGIN

RAISE EX

EXCEPTION

WHEN EX THEN;

--------------------------

RAISE

WHEN EX THEN;

-----------------------;

END;

EXCEPTION

WHEN EX THEN;

-------------------------;

END;

**NOTE**: Exception raised in exception section cannot be handled in the same block exceptional section. But possible to handle outer bock exceptional sections which is also called as EXCEPTION PROPAGATION

**CASE: 8**

DECLARE

---------------

BEGIN

DECLARE

V NUMBER (2):=12345

BEGIN

----------------

EXCEPTION

WHEN VALUE\_ERROR THEN

----------------------------------------;

END;

EXCEPTION

WHEN VALUE\_ERROR THEN

------------------------------------------;

END;

**NOTE:** Exceptions raised in declarative section cannot be handled in same block exception sections.

**TRACING AN EXCEPTION**

DECLARE

BEGIN

SELECT...................................;

SELECT...................................;

SELECT...................................;

SELECT...................................;

SELECT...................................;

SELECT...................................;

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

-------------------------------------------------;

END;

**NOTE:** In above case we are not able to identify in which select statement the exception has occurred. so to overcome this situation we writ as follows

DECLARE

----------------- ;

BEGIN

DBMS\_OUTPUT.PUT\_LINE('H');

SELECT............................................;

DBMS\_OUTPUT.PUT\_LINE('A');

SELECT............................................;

DBMS\_OUTPUT.PUT\_LINE('R');

SELECT............................................;

DBMS\_OUTPUT.PUT\_LINE('I');

SELECT............................................;

DBMS\_OUTPUT.PUT\_LINE('S');

SELECT............................................

DBMS\_OUTPUT.PUT\_LINE ('H');

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

END;

**NOTE:**

* If we get only 'H' as output then in first select statement the exception has occurred.
* If we get 'H','A' as output then in second select statement the exception has occurred.
* If we get 'H' ,'A','R' as output then in third select statement the exception has occurred.
* If we get 'H' ,'A','R','I' as output then in fourth select statement the exception has occurred.
* If we get 'H' ,'A','R' ,I','S' as output then in third select statement the exception has occurred.
* If we get 'H' ,'A','R','S',H' as output then there are no error in select statements.

**STORED PROCEDURE/ (NAMED BLOCKS)**

* They are the stored programs which get stored in DB as a DB object.
* Re-usability is possible in terms reduces the redundancy.(avoids code repetetion)
* Primarily (initially) we have two types of stored procedures.

1. Application stored procedures: They are application specific.
2. Database stored procedure:
3. Procedures
4. Functions
5. Triggers
6. Packages

Mostly we use these and cursors,loops and also TCL commands.

**Usages of stored procedures:-**

1. Portability
2. Security/data hiding
3. Encapsulation
4. Modularity
5. Scalability
6. Easy maintenance
7. Enhancing performance
8. Makes application design easy
9. Monitoring application

**Tables regarding with stored procedure:-**

1. User-arguments
2. User-dependencies
3. User-object
4. User-source
5. User-trigger
6. User-procedures

**Procedure:-**

* It is program unit (or) PL/SQL block which takes the parameters.
* In the procedure we attach header to the PL/SQL block.
* Procedure is a program unit which is used to execute statements.
* It acts as procedural statements and we use it for statements executions (more often).
* It is a stored program unit.
* **A procedure has to be four sections:-**

They are:

1. Header section
2. Declarative section
3. Execution section
4. Exception section

**Syntax:-**create [or replace] procedure <procedure\_name> (Para)[authid\_user definer/current\_user] is

begin

-----------------------

Exception

-----------------------

End [procedure name];

**Note: -**In the above syntax “is” represents standard alone procedure.

**Calling procedure:-**

* SQL/Plus
* SQL developer(new tool)
* Blocks
* Unix
* Front end applications(apps)
* Web applications

**Parameters:-**

Basically parameters of 3 types ate:

1. IN (read)
2. OUT (write)
3. INOUT (read and write)

**Procedure without any parameters:-**

create or replace procedure p is

begin

DOPL(‘hello’);

end p;

**procedure created**

Desc p;

Procedure p

**Calling in SQL\*plus:-**

Exec p;

**Calling in blocks:-**

begin

P;

end;

**procedure with ‘IN’ parameter:-**

Create or replace procedure p(X in number) is V number (5);

begin

v:=X+5000;

DOCL (V);

End;

Note:-

Number (5) is invalid because if you define size it will throw error.

**SQl\*Plus:-**

**Calling environment:**

Var a number;

We will declare this variable in executable section.

Exec:a:=1000;

Here assigning value to variable(a), from this variable(a) to I am sending value to’X’.

Exec p(:a);

Here calling the procedure.

declare

a number:=&n;(calling in blocks)

begin

p(a); (this is calling environment)

end;

**Note:-**

x number;

x In number

above both two statements are same and ‘In’ is default parameter.

Declare

vname varchar2(10):=&n;

a number(5);

begin

select sal into a from emp where ename=vname;

p(a);

end;

**create or replace procdure p(X number, Y number) is vname varchar2(10);**

begin

select ename into vname from emp where empno=X and sal=Y;

DOPL(‘employee name’||vname);

end;/

**-procedure created.**

declare

Vno number (5):=&n;

Vsal number (5):=&m;

begin

p(vno,vsal);

end;

**SQL/Plus:-** Calling environment

var a number;

var b number;

exec :a:=7839;

exec :b:=5000;

exec p(:a,:b);

**procedure with ‘OUT’ parameters:-**

Create or replace procedure p(X in number,Y out varchar) is

begin

select ename into Y from emp where empno=x;

end;

**SQL/Plus:-**

var a number;

var b varchar2(10);

exec :a:=7839;

exec p(:a,:b);

print b;

exec DOCL(:b);

**blocks:-**

declare

a number(5):=7839;

b varchar2(10);

begin

p(a,b);

DOCL(b);

end;

**Spool concept:-**

Sql>spool c:/gv.sql;

>select \* from emp;

Sql>spool off;

Sql>spool c:/gv.sql append ; (it will append to previous one)

**Procedure with ‘IN-OUT’ parameters:-**

* create or replace procedure p(X in out number) is

begin

select sal into X from emp where empno=x

end p;

**SQL \* Plus:-**

var a number;

exec :a:=7788;

exec p(:a);

print a;

**Blocks:-**

declare

a number(5):=&n;

begin

p(a);

DOPL(a);

End;

* While creating time we will declare (arguments/parameters eg: X in number) we will call that as “formal parameters”.

**EX:** exec p(:a,:b) these are “actual parameters”.

* We will refer the values from actual to formal in three notations:

1. Positional notation; (based on position value will be referred).
2. Named notation; (by using names (formal parameters) we can implies the value)
3. Mixed notation.

create or replace procedure p(X number, Y number, Z number) is

begin

DOCL(X||’ ‘||Y||’ ‘||Z);

end;

/

-Procedure created.

**Positional notations:-**

Exec p(100,200,300);

**Named notation:-**

Exec p(Z=>300,Y=>200,X=>500);

**Mixed notation:-**

**Note:-**In mixed notation, named notation has to follow positional ` notation.P(100,200,Z=>300);

P(100,Y=>200,300);

P(100,Y=>300,X=>400,Z=>200);

In the above statements first will be valid remaining are in-valid because here we are assigning values in two times to ‘X’.

* values get referred from actual to formal and formal to actual in two methods they are:

1. Reference method; (enhances performance).
2. Copy method; (or) (pass by value method).

* By default ‘In’ parameter takes the values through reference method which is faster than the copy method and desirable, where as in the case of out and in out parameters values get passed through copy method. Which degrades
* the performance, so to avoid this thing we have a reserved word called ‘No Copy’ which is introduced from 9i onwards.

**No copy:-**

* we mention No Copy for Out and InOut parameters so, to take the values through reference method (which in turn) accelerates/alters the performance.

**Note:-**Mentioning NOCOPY to the ‘IN’ parameters throws an error.

**Eg:-** Create or replace procedure p(X number, Y out nocopy number) is

Begin

Y:=X+1000;

End p;

**Note:-** Here ‘nocopy’ is reserved word or key word.

**SQL\*Plus:-**

Var a number;

Var b number;

Exec :a:=100;

Exec p(:a, :b);

Print b;

* Once we pass the value to ‘In’ parameter throughout the program we can’t vary the value.

**Note:-** Always ‘In’ parameters should not be at left side of assignment operator which means ‘In’ parameters acts as a ‘constant’ in the scope of program.

Create or replace procedure p(X in number, Y out number) is

Begin

X:=200;

Y:=X+100;

End;

* Exec p(100,:b)

**Local subprograms:-**

* They are procedures/functions which are defined in the declarative section named and unnamed blocks.
* We have to call these local programs within the scope of that block.
* It won’t get stored anywhere else on by themselves.

Declare

Procedure p is

Begin local sub programs

DOCL(‘hello’);

End p;

Begin p;

End; Here we won’t mention create/replace.

**Declarative section we will handle these 8:-**

* Variables, exceptions, cursors, constants, data types, programs, defining local programs.

**Forward declaration:-**

* Basically if you want to call a program it has to be get defined very previously.
* Whenever you call a program before defining throws an error.
* For mutual exchange purpose sometimes we have a need to call a procedure before defining.
* To fulfill this requirement we will declare the procedure very previously which is calling before defining.
* If you want to call a procedure before defining you have to declare those procedures vary previously.

Declare

V number (5):=5;

Procedure p2(Y inout number); //forward declaration

Procedure p1(X inout number) is

Begin

If X>0 then

Docl(X);

X:=X-1;

P2(X);//calling

End if;

End p1;

Procedure p2(Y inout number) is //defining

Begin

P1(Y);

End p2;

Begin

P2(v); // calling and executable section first compiler comes here.

First define then call any procedure without defining if

you need to make a call at least we need to declare.

End;

* Create or replace procedure p3 is

(1)

End p3;

Exec p3;

**Note:-**

* A procedure can optionally contain ‘return’ statement but it won’t through any value as in the given eg.

Create or replace procedure p3 is

V number(5):=100;

Begin

DOCL(v);

Return v;

DOPL(v);

End p3;

**Functions**

* They are the PL/SQL program units which allow parameters similar to that of procedures.
* But unlike procedures they return value by default.
* Functions are useful for calculation purpose and for data manipulation purpose (DML).
* Functions makes the queries simple, readable and also enhance the performance.

**Note:-** Providing out and inout parameters’ in functions is not preferable.

**Syntax:-** create or replace function <function\_name> (Para) return datatype [pipelined|aggregate|parallelenabled|deterministic|authiduser] is/as

Begin

Code;

Return statement;

Exception

-----------

End[procedure name];

**Calling functions:-**

1. SQL/Plus
2. Blocks
3. SQl Developer
4. Apps
5. Select statement
6. Objects

**Functions without any parameter:-**

create or replace function fun return number is

v number(5):=&n;

begin

return v; // we can give assign value/exp/literals/collections.

end;

**block:-**

declare

a number(5);

begin

a:= fun;

DOPL(a);

End;

**SQL/Plus:-**

Var a number;

Exec :a:=fun; //where ‘a’ holds the value

Print a;

Create or replace procedure p is

B number(5);

Begin

B:=fun;

DOPL(b);

End p;

Create or replace function f1 return number is

Begin

Return fun;

End f1;

Select fun from dual;

**Note:-** In the return statement we can mention value directly, expression, another function, cursor variables, index by table (collection), Boolean value, and so on…..

**Function with IN parameter:-**

Create or replace function fin(X number) return variable is Vname emp.ename%type;

Begin

Select ename into vname from emp where empno:=X;

End fin;

**SQL/Plus:-**

Var a carchar2(10);

Var b number;

Exec :b:=7788;

Exec :a:=fin(:b);

Print a;

**Blocks:-**

Declare

A varchar2(10);

B number(5):=&n;

Begin

A:=fin(b);

DOPL(a);

End;

Select fin(empno) from emp;

**Create or replace function fn(X varchar2) return number is Vno number(5);**

Begin

Select empno into vno from emp where ename=X;

Return vno;

End fn;

SQL>select fn(fin(empno)) from emp;

Function can contain ‘n’ no of ‘return’ statements but alwaya executes only one return statements.

**Using multiple return statements in functions:-**

Create or replace function f(X number) return number is v number(5):=1000;

Begin

If x>100 then

Return(x);

Else

Return v;

End if;

End;

Create or replace function f return….

Begin

Return X;

Return Y;

End;

* + If control looks first return statement the control automatically comes out of function them it won’t goes to second return statement.
  + DOPL won’t supportsT(or) F

**SQL\*Plus:-**

Var a number;

Exec :a:=F(500);

Print a;

**Function without parameters:-**

**Returning Boolean value from the function:-**

Create or replace function f return Boolean is v number(5):=100;

Begin

Return null;

End;

**Blocks:-**

Declare

A Boolean;

Begin

A:=f;

DOCL(‘value ‘||a);

End;

SQL won’t supports Boolean datatype.

**EX for out:-**

Create or replace function f(X out number) return number is v number(5):=100;

Begin

X:=v+500;

Return v;

End;

**SQL\*Plus:-**

var a number;

var b number;

exec :a:=f(:b);

print a;

100

Print b;

600

**Blocks:-**

Declare

A nmber(5);

B number(5);

Begin

A:=f(b);

DOPL(a||’ ‘||b);

End;

**SQL> select f(:b) from dual;**

**Note:** It is not possible to call the function which are having out and in out parameters in ‘select statement’.

**Function with inout parameter:-**

Create or replace function f(X inout number) return number is v umber(5):=100;

Begin

X: =v+500+X;

Return v;

End;

**SQL\*Plus:-**

Var a number;

Var b number;

Exec: b: =500;

Exec: a: =f (: b);

Print a;

Print b;

**Blocks:-**

Declare

A number (5);

B number (5);

Begin

A: =f (b);

DOPL (a||’ ‘||b);

End;

**Using functions:-**

We can use the functions in the following areas

1. Where clause
2. Start with
3. Connect by
4. Having
5. Group by
6. Order by
7. Select
8. Update set clause
9. Insert value clause
10. From clause

We cannot use the functions in the following areas

1. Default(alter and create)
2. Check constraint

**Restrictions of function:-**

**Functions**

**Select statement** **DML statements**

* In the above diagram select statements DML is not possibls and select is possible.
* In the DML statements DML will not possible on same function table. But it will possible on different function tables.
* In DML statements select possible on different (or) same function table.

**NOTE:-**

* We can use select statement in DML operations.
* Using DML operations in select statements is not passible either directly (or) indirectly.
* Functions/procedures i.e DML
* Select f (DML) from dual;
* Select f (select) //invalid
* Update t1 set val=f(DML(T1)) //invalid
* Update t2 set val =f(DML(T2)) //valid
* Generally SQL statements won’t allow us to call the functions which are having DML operations. If you do so, resulting into error. If you won’t to restrict the user not to provide DML operations in a function which are frequently used in select statements, we need to provide “program restrict reference” while creating a function, in this manner we will eliminate the impurities.

Create or replace function f return number is

Begin

Null;

End f;

**Note:-**

We can define a function without return statement but at the time of calling it throws error as such in above eg.

Var a number;

Exec :a:=f; //throws an error.

**Diff between procedure and function:-**

**Procedure:-**

* Procedure may (or) may not return a value.
* Not passable to call in SQL statements.
* It acts as a PL/SQL statement execution (for statements execution purpose).

**Functions:-**

* By default function returns single value.
* It is possible to call in SQL statements.
* For calculations (or) computing purpose.
* Using functions in select statement will enhance the performance and simplifies the coding.

**Forward declaration:-**

* Basically if you want to call a program it has to be get defined very previously.
* Whenever you call a program before defining throws an error.
* For mutual exchange purpose sometimes we have a need to call a procedure before defining.
* To fulfill this requirement we will declare the procedure very previously which is calling before defining.
* If you want to call a procedure before defining you have to declare those procedures vary previously.

Declare

V number (5):=5;

Procedure p2(Y input number); //forward declaration

Procedure p1(X input number) is

Begin

If X>0 then

Docl(X)

X: =X-1;

P2(X); //calling

End if;

End p1;

Procedure p2(Y input number) is //defining

Begin

P1(Y);

End p2;

Begin

P2 (v); // calling and executable section first compiler comes here.

First define then call any procedure without defining if You need to make a call at least we need to declare.

End;

Create or replace procedure p3 is

(1)

End p3;

Exec p3;

**Note:-**

A procedure can optionally contain ‘return’ statement but it won’t through any value as in the given eg.

Create or replace procedure p3 is V number(5):=100;

Begin

DOCL(v);

Return v;

DOPL(v);

End p3;

**Diff between procedure and function:-**

**Procedure:-**

* Procedure may (or) may not returns a value.
* Not passable to call in SQL statements.
* It acts as a PL/SQL statement execution(for statements execution purpose).

**Functions:-**

* By default function returns single value.
* It is possible to call in SQL statements.
* For calculations (or) computing purpose.
* Using functions in select statement will enhance the performance and simplifies the coding.

**Packages:-**

* It is a container/program unit area which is useful to store related things at one place.
* It provides modularity, scalability, encapsulation, data security, portability, code analysation, debugging the code, tracing the code, profiling the code, location monitoring and so on…

Alters/decreases the redundancy.

**Note:-**

Packages won’t allow parameters, nesting, and calling.

**Packages consist of two parts:**

1. Package specification(PS)
2. Package body

**Package specification:-**

* It is a prototype for package body program.
* In specification we declare variables, cursors, exceptions, procedures, functions and so on…
* This is for information purpose.
* It can exist without body.
* Declaring cursor variables is not possible; defining ref cursor data type is possible.
* A package body can’t exist without package specification but reverse is not so…
* Package specification and body stores in different memory areas.
* PL /SQL objects defined in package specifications are considered as global objects, won’t allow coding part.

**Package body:-**

* It consists of program coding.
* A package body can optionally has executable section.
* Variables and programs which are defined in package body without specifying I package specification are called as local variables, local programs.
* We can drop a package body without dropping package specification

Package specification drops.

**Syntax for package specification:-**

Create or replace package <package\_name> is variables, cursor,exceptions,datatypes……

Declaring procedure,functions…….

End packagename;

**Pragmaautonomytransaction (Tx):-**

It is an independent Tx happens individually irrespective of parent Tx.

**Limitations:-**

* Package specification won’t allow pragma; we can apply for packeged procedures and packages.

Create or replace procedure p is pragma autonomius\_transaction;

Begin

Insert into nestab values(10);

End p;

Create table nestab(sno number(5));

Select \* form nestab;

**Calling environment:**

Begin

Insert into nestab values(11);

P;

Insert into nestab values(12);

Rollback;

End;

**Output:** 10

* While using pragma autonomus Tx we need to mention commit, rollback as mandatory.
* DDl- autocommit commands
* DML- non-autocommit.

Create or replace procedure p is pragma autonomius\_transaction;

Begin

Insert into nestab values(10);

Rollback;

End p;

Begin

Insert into nestab values(11);

P;

Insert into nestab values(12);

Commit;

End;

Select \* from nestab;

**Out put:-**

11

10

12

Create or replace procedure p is

Begin insert into nestab values(10);

Rollback;

End p;

Begin insert into nestab values(11);

P;

Insert into nestab values(12);

Commit;’

End p;

**Out put:**

12

**Note:-**We avoiding mutating error using pragma.atonomous\_transaction.

**Pragma inline:- (11g)**

* Which is used to including the programs(11g).
* Pragma inlining will enhance the performance.

**Deff:**

In lining a program means replacing the procedure call with actual executable code copy of a program.

**Note:-**

Programs which are having static code and frequently used programs (or) subject to the pragma inline(preferrable).

**Syntax:-**

Pragma inline(‘procedure name’,{‘yes/no’});

Declare

Stime integer;

Etime integer;

V number;

Function f(x number) return number is

Begin

Return x;

End f;

Begin

Pragma inline(‘f’,’yes’);

Stime :=dbms\_utility.get\_time;

For I in 1..10000 loop

V:=f(i);

End loop;

Etime:=dbms\_utility.get\_time;

DOPL(etime-stime);

Pragma inline(‘f’,’no’);

Stime:=dbms\_utility.get\_time;

For I in 1..10000 loop

V:=f(i);

End loop;

Etime:=dbms\_utility.get\_time;

DOPL(etime-stime);

End;

**Packages:-**

* It is a container/program unit area which is useful to store related things at one place.
* It provides modularity, scalability, encapsulation, data security, portability, code analysation, debugging the code, tracing the code, profiling the code, location monitoring and so on…
* Alters/decreases the redundancy.

**Note:-**

Packages won’t allow parameters, nesting, and calling.

**Some built-in packages:-**

Dbms-lob-handles lob related values // cannot call package in package

Dbms-fga(11g)-handles fine grain editing. //no parameters

Dbms-lock-to provide locks and latches

Dbms-describe-display the information regarding overload, level and arguments

Dbms-profile-profiling the code

Dbms-monitor-for application monitoring

Dbms-sql-to handle DDl commands in PL/SQL

Dbms-job-for defining and scheduling the jobs

Dbms-DDL

Dbms-transaction-to handle the transactions

Dbms-trace(11g)-tracing the code

Dbms-hprof(11g)-for hierarchical profiling

Dbmssession-session related data

Dbms-metadata-handles the metadata(data to the data)

Utl-file-for file handling

Dbms-utility-for miscellaneous

Dbms-rls-for row level security

Dbms-dependency-handles dependency objects

Dbms-revalidation-

Utl-tcp-

Dbms-output-

Dbms-result-cache (11g)-for data caching

Dbms-pipe-handles inter related session

dbms-sechdule-

dbms-warning-

dbms-debug-

**Packages consist of two parts:**

* Package specification(PS)
* Package body

**Package specification:-**

* It is a prototype for package body program.
* In specification we declare variables, cursors, exceptions, procedures, functions and so on…
* This is for information purpose.
* It can exist without body.
* Declaring cursor variables is not possible; defining ref cursor data type is possible.
* A package body can’t exist without package specification but reverse is not so…
* Package specification and body stores in different memory areas.
* PL /SQL objects defined in package specifications are considered as global objects, won’t allow coding part.

**Package body:-**

* It consists of program coding.
* A package body can optionally has executable section.
* Variables and programs which are defined in package body without specifying I package specification are called as local variables, local programs.
* We can drop a package body without dropping package specification
* Package specification drops.

**Syntax for package specification:-**

Create or replace package <package\_name> is variables, cursor,exceptions,datatypes……

Declaring procedure, functions…….

End packagename;

**Sysntax for package body:-**

Create or replace package body <package\_name> is variables, cursor….

Defining subprograms

Optional executable section;

End package\_name;

**Eg:**

Crate or replace package pack is v number(5):=400;

Ex exception ;

Cursor c is select \* from emp;

Procedure p(X inout number);

Function f(y in out number);

Return number;

End pack;

* Create or replace package body pack is L number(5):=500;

Function lf(Z in number )return number is lv number(5);

Begin

Lv:=Z+L;

Return lv;

End lf;

Procedure p (X inout number)is pv number(5);

Begin

Pv:=2000+lf(X);

X:=pv+5000;

End p;

Function f(Y inout number)return number is

Begin

Y:=Y+lf(v);

V:=Y+L;

Return v;

End f;

If you want to define package body first we need to define package specification.

**SQL\*PLUS:**

**SQL>**var a number;

Exec :a:=1000;

Exec pack.p(:a);

Print a;

**BLOCKS:**

Declare

a number(5):=2000;

begin

pack.p(a);

DOPL(a);

End;

**FUNCTIONS:**

**SQL\*PLUS**

Var a number;

Var b number;

Exec :b:=4000;

Exec :a:=pack.f(:b);

**BLOCKS:**

Declare

A number(5);

B number(5):=1000;

Begin

A:=pack.f(:b);

DOPL(a||’ ‘||b);

End;

**Example for Packaged Crusors:**

declare

vrow emp%rowtype;

begin

open pack.c;

loop

fetch pack.c into vrow;

DOPL(vrow.ename);

If vrow.ename=’scott’ then

Raise pack.ex;

End if;

End loop;

Exception

When pack.ex then

DOPL(‘packex’)

End;

Declare

Vrow emp%rowtype;

Begin

Fetch pack.c into vrow;

DOPL(vrow.ename);

End;

OutPut: Ford

**NOTE:-**

* If you won’t close the packaged (pack’s) cursor in any program then that is last for entire session.
* We can call the packaged procedures and functions outside of package which are specified in package specification(global accessing).
* Attempting to call packaged local sub programs to call outside of package throws an error.

**Using cursor variable as a parameter value:**

Create or replace package pack is type rec is ref cursor;

Procedure p(X rec);

End pack;

Create or replace package body pack is Procedure p(X rec)is vrow emp%rowtype;

Begin

Loop

Fetch X into vrow ;

DOPL(vrow.ename);

Exit when X%not found;

End loop;

End p;

End pack;

**BLOCK:**

Declare

Type recl is ref cursor;

Vrec recl;

Begin

Open vrec for select \* from emp;

Pack.p(vrec); //calling

End;

**Note:** Defining cursor variable in package specification will not be allowed but defining ref cursor is possible.

**We can use cursor variable as an ‘out’ parameter:**

Create or replace package p(X out sys\_refcursor) is vrec sys\_refcursor;

Begin

Open vrec for select \* from emp;

X:=vrec;

End p;

**SQL\*PLUS:**

Var v refcursor;

Exec p(:v);

**Using cursor variable in return statement of a function:**

Create or replace function f return sys\_refcursor is vrec sys\_refcursor;

Begin

Open vrec for select \* from emp,dept;

Return vrec;

End f;

Exec :v:=f;

Print v;

**Note:**If procedure and functions is present in specification then we go for package body otherwise no need.

**Packaged Cursors:**

Create or replace package pack is cursor c return emp%rowtype is select \* from emp;

End pack;

Here we are defining ‘cursor’ in package specification we can use this anywhere.

Begin

For i in pack.c

Loop

DOPL(i.ename);

End loop ;

End;

**We can hide the cursor select statement through package concept as shown in the snippent:**

Create or replace package pack is cursor c return emp%rowtype ;

End pack;

Create or replace package body pack is cursor c return emp%rowtype is select \* from emp;

End pack;

Begin

For i in pack.c

Loop

DOPL(i.ename);

End loop ;

End;

**Out Put:**

Emp 14 records

* A package body can optionally contain executable section. Which is one time initialization but it should be at last in the package body if we have any sub programs.

Create or replace package pack is v number (5);

End pack;

Create or replace package body pack is

Begin

V: =5000;

End pack;

**Note:-**

We can also call the packaged function in select statement but that should not contain out and inout parameter.

Drop package <package\_name>

* Here along with package specification, body alo will drop.
* Dropping packaged body without dropping package specification

Drop package body pack;

**Polymorphism(overloading):-**

* Defining multiple local subprograms with the very same name but by differing number, order and data types of parameters.

Create or replace package pack is Procedure p(x number);

Procedure p (x number, y number);

Procedure P(x number, y varchar2);

Procedure p(y varchar2, x number);

End pack;

**Note:-**The data types should not be same family while comparing to procedures.

Creating or replace package body pack is procedure p(x number) is

Begin

DOPL(x);

End p;

Procedure p(x nmber, y number) is

Begin

DOPL(x||y);

End p;

Procedure p(x number, y varchar2) is

Begin DOPL(x||’ ‘||y);

End p;

Procedure p(y varchar2, x number)is

Begin

DOPL(y||’ ‘||x);

End p;

End;

**Calling environment:-**

SQL>exec p(100);

SQL>exec p(‘a’,200);

SQL>exec p(200,’a’);

**Note:-**Overloading is not possible for standard alone program(schema level programs).

* V$parameter=>all DB related parameters will store
* PL/SQL-optimize-level-initialization parameter.

Where level-0-ideal

Level-1

Level-2-preferrable

Level-3(11g)-aggressive much more faster.

**Tuning the PL/SQL code:- optimizing the PL/SQL code:-**

* We will tune the PL/SQL code based on initialisation parameter i.e PL/SQL-optimize-level in this we will set the levels from 0 to 3. Level 2 is default which allows us

1. In lining (11g) a program.
2. Avoiding the in lining program.

* Level 0 and 1 won’t allow us to in lining program.
* Level 3 won’t allow us to inline a program but allow us to avoid in lining program.

Declare

Sdate number:=0;

Edate number:=0;

D\_code number:=0; //dead code no use in program

L\_r number:=0;

Function f(x number) return number is

Begin

Return x;

End f;

Begin

Sdate:=dbms\_utility.get\_time; //here we are assigning to one variable

so call it as function.

For I in 1..1000000

Loop

D\_code:=0;

L\_r:=l\_r+f(i); //calling

End loop;

Edate:=dbms\_utility.get\_time;

DOPL(edate\_sdate);

End;

**SQL>**alter session set PLSQL\_optimize\_level=0;

Time:23session set PLSQL\_optimize\_level=1;

Time:22session set PLSQL\_optimize\_level=2;

Time:21session set PLSQL\_optimize\_level=3;

Time:9session set PLSQL\_optimize\_level=4;

* Dbms\_utility.put\_line;
* Get\_time;

SQL>show parameter PLSQL\_optimize\_level;

**Understanding the NOTNULL constraint for a PL/SQL variable:-**

Providing NOT NULL constraint for a PL/SQL variable is not preferable doing so, will degrades the performance because nullity will be checked through one virtual variable created implicitly by the engine.

Declare

Sdate number:=0;

Edate number:=0;

V number not null:=0

Begin

Sdate:=dbms\_utility.get\_time;

For i in 1..100000000 loop

V:=I;

End loop;

Edate:=dbms\_utility.get\_time;

DOPL(edate\_sdate);

End;

Output:

Time :- 72 minutes.

**Note:** Internally it takes one virtual variable and it checks v number NOT NULL(it is not preferable more time it will take if you use coading in program it is better like if v is not null).

Same as above program but variable using without NOTNULL.

**Output:**

Time: 44

Using pls integer is preferiable than the number since it is faster:-

V pls\_integer:=0; //we use these only in PL/SQL and remaining is same as above.

Time:-12

**Using simple integer is more refereable than pls\_integer:-**

Simple integer is a derivate of pls\_integer , but avoids overflow error and allow null values.

**Note:** Up to 10g every datatype allows null values but n 11g simple integer won’t allow null value. Pls\_integer size -2147483648 to 2147483647

**Note:** Pls\_integer can store max of 2147483647, if you assign above of this value throws an error i.e overflow error, to overcome this error we have a datatype called “simple-integer”(11g) and even simple integer won’t allow “null” values (NOTNULL datatype) .

Declare

V pls\_integer:=214783646;

Begin

V:=v+1;

DOPL(v);

V:=v+1;

DOPL(v);

End;

**Out put:** Numeric overflow error.

To overcome this

Declare

V simple\_integer:=2147483646;

Begin

V:=v+1;

DOPL(v);

V:=v+1;

DOPL(v);

End;

**Out put:**

2147483647

-2147483648

**Pragmaautonomoustransaction (Tx):-**

It is an independent Tx happens individually irrespective of parent Tx.

**Limitations:-**

Package specification won’t allow pragma; we can apply for packeged procedures and packages.

Create or replace procedure p is pragma autonomous\_transaction;

Begin

Insert into nestab values(10);

End p;

Create table nestab(sno number(5));

Select \* form nestab;

**Calling environment:**

Begin

Insert into nstab values(11);

P;

Insert into nestab values(12);

Rollback;

End;

**Output:** 10

* While using pragma autonomus Tx we need to mention commit, rollback as mandatory.
* DDl- autocommit commands
* DML- non-autocommit.

Create or replace procedure p is pragma autonomius\_transaction;

Begin

Insert into nestab values(10);

Rollback;

End p;

Begin

Insert into nestab values(11);

P;

Insert into nestab values(12);

Commit;

End;

Select \* from nestab;

**Out put:-**

11

10

12

Create or replace procedure p is Begin insert into nestab values(10);

Rollback;

End p;

Begin insert into nestab values(11);

P;

Insert into nestab values(12);

Commit;

End p;

**Out put:**

12

**Note:** We avoiding mutating error using pragma.atonomous\_transaction.

**Pragma inline:- (11g)**

* Which is used to including the programs(11g).
* Pragma inlining will enhance the performance.

**Deff:** Inlining a program means replacing the procedure call with actual executable code copy of a program.

**Note:** Programs which are having static code and frequently used programs (or) subject to the pragma inline(preferrable).

**Syntax:**

Pragma inline(‘procedure name’,{‘yes/no’});

Declare

Stime integer;

Etime integer;

V number;

Function f(x number) return number is

Begin

Return x;

End f;

Begin

Pragma inline(‘f’,’yes’);

Stime :=dbms\_utility.get\_time;

For I in 1..10000 loop

V:=f(i);

End loop;

Etime:=dbms\_utility.get\_time;

DOPL(etime-stime);

Pragma inline(‘f’,’no’);

Stime:=dbms\_utility.get\_time;

For I in 1..10000 loop

V:=f(i);

End loop;

Etime:=dbms\_utility.get\_time;

DOPL(etime-stime);

End;

**LOB datatype:**

Before of 8i if you want to store the images or huge information we opt(choose) for long datatypes but , we have so many restrictions on long datatypes to overcome this disadvantage we have LOB datatype.

**Cliassifications of LOB datatype:-**

LOB datatypes can be devided as below:

* Internal
  + Persistance
  + Non-persistence(temporary)
    - BLOB
    - CLOB
    - NCLOB
* External
  + Bfiles

**CLOB:** allows huge information upto 4GB from 11g onwords even it supports 8 to 128TB.

**BLOB:** stores binary data, allows images, audio,video files and so on…

**NCLOB:** it stores national database chracterset data. Supports multi languages.

**Bfiles:** useful to store files in the form of binary data.

Commit and rollback is not passible to provide on this data.

**1\*2**

Db

File not store in db

**C1CLOB**   **c2BLOB** **c3Bfile**

CLOB locater BLOB locator Bfile locator

abc

01010

* We’e to make use of transactions by using locators.
* We handle LOB datatype through a package is’DBMS\_LOB’

Locator starts-null,0,empty.

**Handling lob data by using DBMS\_lob package:\_**

If we are using Lob datatype’s we’ve to make use of packages compulsoraly.

Declare

VCLOB clob;

Max\_size integer;

Offset integer:=1;

Wdata varchar(100):=’welcomw to LOB world’;

Rdata varchar2(100);

Begin

Delete from clobdata;

Insert into clobtab values(10,empts\_clob);

Select ccol into vclob from clobtab;

Max\_size:=length(edata);

Dbms\_lob.open(vclob,dbms\_lob.lob\_readwrite);

Dbms\_lob.write(vclob,max\_size,offset,wdata);

Dbms\_lob.read(vclob,max\_size,offset,rdata);

Dbms\_lob.close(vclob);

DOPL(rdata);

End;

Create table clobtab(sno number(5),ccol clob);

Insert into clobtab values(10,’abc’);

Create table imagetab(sno number(5), image bfile);

Declare

Bimage bfile:=bfilename(‘DY’,’gimage.bmp’);

Image\_loc bfile;

Max\_data integer:=60;

Offset\_integer:=1;

Raw\_data raw(100);

Begin

Delete from imagetab;

Insert into imagetab values(10,bimage);

Select image into image\_loc from imagetab;

Dbms\_lob.open(image\_loc);

Dbms\_lob.read(image\_loc, max\_data, offset,raw\_data);

Dbms\_lob.close(image\_loc);

DOPL(rawtohex(raw\_data));

DOPL(utl\_raw.cast\_to\_varchar2(raw\_data));

End;

* Create directory dy as ‘c:\’;
* Grant read on directory by to apps;
* Create a file with name ‘gv.txt’ and provide data and save in path mentioned.

As in directory ‘dy’;

Create table direction(sno number(10),dcol clob);

Declare

Vdir bfile:=bfilenamme(‘DY’,’gv.txt’);

Colfile clob;

Max\_data integer:=10;

Des\_offset integer:=1;

S\_ofset integer:=1;

Lang\_ctx integer:=dbms\_lob.default\_lang\_ctx;

War\_mes integer;

Begin

Delete from direction;

Insert into direction values(10, empty\_clob);

Selection dcol into colfile from direction;

Dbms\_lob.open(colfile,dbms\_lob,lob\_readwrite);

Dbms\_lob.open(vdir);Dbms\_lob.load clob from file(colfile, vdir, dbms\_lob.maxsize, des\_offset, s\_offset, nls\_charset\_id(‘USTASCII’), lang\_ctx, war\_mes);

Dbms\_lob.close(colfile);

Dbms\_lob.close(vdir);

End;

**Select \* from direction;**

**Triggers:-**

* They are the stored programs fires implicitly unlike subprograms.
* Triggers fires automatically when event occurs.
* We can’t call trigger and pass parameters and can’t nest (trigger with in trigger).

**Usages:-**

* Data auditing
* Enforcing the referential integrity
* Security
* Data replication
* Application monitoring(AM)
* Enforcing the data
* Triggers have their own name space.
* Triggers won’t allow commit and rollback, but by using “pragma atonomous transaction we can mention commit and rollback in triggers.
* Basically triggers are of two types

1.Application trigger

2.Database triggers

**Database triggers:-**

1. DML triggers (table)
2. Instead of trigger(views)
3. DDL trigger(schema) // 3 and 4 never write these in DBA level.
4. DB trigger(database)
5. Compound trigger(11g)(table)

We will write triggers on tables, views, schema and database.

**DML triggers:-**

* We provide DML triggere on table
* Based on DML events execution of trigger takes place implicitly.
* This triggers are again of two types:

1. statement level trigger (or) table level trigger
2. row level trigger

**Statement level trigger:-**

This trigger executes only once for an entire table/for bunch of records execution of this trigger takes place atleast once. Even if table data won’t get effected.

**Rowlevel trigger:-**

* this trigger executes for each of row.
* This trigger won’t get executes atleast once if table data won’t get effected.

**Syntax of DML trigger:-**

Create or replace trigger <trigger\_name> (1)before/after DML operations on <table\_name>[disable/enable](2)[foreach row](3)[when condition][declare]

Begin

Code;

Exception

End;

**(1)-trigger timing:-**

It specifies when trigger has to fire.

**(2)-trigger level:-**

Specifies what ever row level/statement level.if you mention for row i.e rowlevel else statement level but bydefault statement level.

**(3)-trigger condition:-(not preferrable)**

If further provides the additional restriction on trigger execution.in trigger conditon should always be a boolean expression statement level trigger won’t allow trigger condition.

**(4)-trigger body:-**

It specifies what action we have to perform through trigger.

**Ex for statement level trigger:-**

Write a statement level trigger so not tto allow any DML operation on a table on weekends.

Create table stab(sno number(5))

Create or replace trigger string before insert or update or delete on stab;

Begin

If to-char(sysdate,’dy’)in(‘thu’,’sun’)then raise\_application\_error

(-20003,’DML operations are not allowed on weekends’);

Else

DOPL(‘dml operations’);

End if;

End string;

* Trigger created.
* Insert into stab values(10); //invalid
* Select \* from stab;

No rows selected.

* Generally most of the trigges make use of ‘before’ instead of after. We can define a trigger name with same table name on which we are defining
* Create or replace trigger stab on stab.

**Note:** because trigger uses different namespaces, table uses different namespaces so possible.

**Rowlevel trigger:**

It will make use of pseudo records as shown in following table.

Virtual table:

|  |  |  |
| --- | --- | --- |
|  | old | new |
| insert  update  delete | null | null |

Here we wil represent new and old in this program i.e rowlevel trigger program.

Write a rowlevel trigger to update the salaries witch one specific value for an increment, if Any one of the existing salaries is greater than new salaries then I have to cancel all the transactions happened by trigger.

Create or replace trigger ustrig before update on emp1;

For each row

Begin

If :old.sal>:new.sal then

Raise\_application\_error(-20003,’sal cannot be decremented’);

Else

DOPL(‘icremented’);end if;

End;

* From the trigger if we get any error, all the previous transactions will get cancelled.

**Eg for trigger when condition:-**

* When condition skips the execution of trigger body when ‘when condition’ becomes ‘false’.(but insertion takes place).
* If when condition becomes true then allows trigger body to execute.
* Create or replace trigger logtrig before insert or update or delete on logtab;

For each row

when(to\_char(sysdate,’HH’)not between ‘05’ and ‘08’)

begin

DOPL(‘login time’);

End;

* Insert into logtab values(10);
* 1 row created.
* Create table logtab(sno number(5));

**Using :new and :old in when condition:-**

Create or replace trigger emp1trig before insert or update or delete on emp1;

For each row

When(new.sal<old.sal)

Begin

DOPL(‘oldsal<newsal’);

End;

* In when condition we won’t use :new and :old but in trigger body we can use, if you use also it throws an error.
* Update emp1 set sal=4000;

**Note:** Trigger when condition won’t allows colons(☺ to refer old and new values.

**Enforcing referential integrity:-**

* Triggers can’t chek the old/existing data, it checks only incoming data but where as in the case of constraints they check both incoming and existing data.
* Constraints can give guarantee for centralized data.
* Triggers are an advance of constraints.
* Create trigger contrig after insert on ltab;

For each row

Begin

Insert into ptab values(:new.sno);

End;

* Insert into ltab values(11);
* Before of constraint checking trigger execution takes place.

**Order of execution:**

* Before statement level trigger(BSLT).
* Before row level trigger(BRLT)>
* After row level trigger(ARLT).
* After statement level trigger(ASLT).

**DDL triggers:-**

**Syntax:**

Create or replace trigger <trigger\_name> before/after DDL on schema

Begin

……….

Exception

…………

End [trigger\_name]

* We write DDL on schema level(insert, update, delete).

**DDL attributes:\_**

In DDL we have more than 100 attributes but here we will go for

* Ora\_sysevent
* Ora\_DICT\_obj\_name
* Ora\_DICT\_obj\_type

We will provide DDL commands on schema, mainly for auditing purpose.

Create trigger audtrig before create or drop on schema

Begin

Insert into audtab values (user, ora\_sysevent, ora\_dict\_obj\_name, ora\_dict\_obj\_type, sysdate);

End audtrig;

Create table audtab(suser varchar2(10), sevent varchar2(10), sname varchar2(10), stype varchar2(10), sdate date);

Create table SAM(sno number(5));

Select \* from audtab;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Suser** | **Sevent** | **Sname** | **Stype** | **Sdate** |
| Apps | Create | Sam | Table | 11-mar-13 |

* Drop table sam;
* Select \* from audtab;

**We will write DB triggers on below mentioned:-**

Before

Befor(T)

Before(T)

before

After(T)

After

After

errornumber

After(T)

Log on

Log off

Shutdown

Servererror

Startup

* Attaching the procedure to the trigger in the place of trigger body with ‘call statement’, so to reduce the redundancy as in the following eg.

Create or replace procedure p is

Begin

DOPL(‘hello’);

End;

Create table RAM(sno number(5));

Create or replace trigger protrig before insert on RAM

Call p(don’t use semicolon here, if you use you will get error)

/

-trigger created.

* Insert into RAM values(10);

Hello

-1 row created.

**Advanced topics in triggers☹introduced from 11g)**

* It follows reserved word or keywords followed by 11g.
* Oracle won’t give gurantee for the order of trigger execution among the sametypes but, by using follows(11g) we can order the execution of triggers.

Create table RAM(sno number(5));

Create or replace trigger trig1 before insert on RAM

Begin

DOPL(‘trig1’);

End trig1;

Create trigger trig2 before insert on RAM follows trig1

Begin

DOPL(‘trig2’);

End trig2;

**Output:-**

Trig1

Trig2

**Note:-**

If we did not use follows we get out put like

Trig2

Trig1

Oracle won’t wives guarantee for order of execution, so from 11g onwords we use ‘follows’.

**Disable(11g):-**

* Prior to 11g we only have an option to disable an existing trigger fron 11g onwords we can disable a trigger while creating/defining.
* Create or replace trigger disable before insert on RAM disable

Begin

DOPL(‘this is disable trigger’);

End disable;

* Insert into RAM values(10);

-1 row created.

* To enable/disable the trigger we’ve to make use of ALTER.
* Alter trigger distrig enable;
* Insert into RAM values(10); **//this is disable trigger.**

-1 row created.

* We can define max of 12 triggers on a table(DML).

**We can write a trigger to fire when updating on specific column:-**

* Create table uptab(sno number(5), loc varchar2(10));
* Create trigger updating before updating of sno on uptab

Begin

DOPL(‘updating on column’);

End;

* Insert into uptabvalues(10, ‘x’);
* Updating utab set sno=20; **//updating on column**

-1 row updating.

**We can write trigger on multiple columns:-**

Create trigger uptrig1 before update of sno, loc on uptab

Begin

DOPL(‘updating on column’);

End;

**Analyzing PL/SQL code:-**

* Following things are useful to analyse the PL/SQL code so to increase the performance, easy identification, identifying the code flaws, so to protect from SQL injection.
* User\_object(to know the stored procedure valid/invalid)
* User\_source(to know the source code of stored procedures)

**Note:** Deterministic🡺 from the function, for eg if you call one function for every calling if it returns same value in that situation if you write deterministic, for next execution it won’t executs, it acts as buffer.

* **User\_objects:-**

Which is useful to verify the status of object i.e. whether valid or invalid.

* **User\_source:-**

Which is useful to extract the source code of a stored procedure.

* **User\_arguments:-**

Gives the information regarding arguments i.e name of argument, parameter\_mode, datatype, scale, precision, position of arguments.

* **User\_procedure:-**

Specifies whether the given program is deterministic, pipelined, parallel enabled, aggregate, result\_cache, authid user….

* **User\_identifires:-**Gives the information regarding activities of identifiers,

this table automatically get populated for a program (or) for session (or) for

a database when we enable the PL/SQL tool (by default disabled) with an

intialisation parameter called PL/scope-settings.

* **User\_trigger:-**

Gives information regarding triggers.

* **User\_dependences:-**

To find out reflection (or) information between referenced and dependent object we use this table.

**Note:**It won’t give information regarding remote dependenies for that we have to run a script file so to create one procedure and two tables. here deptree and ideptree are remote dependencies.

* **Dbms\_metadata:-**

To create DDL report for an object.(dbms\_utility.get\_ddl(emp))

* **PL/scope tool(11g):-**

It is an utility, to get the information of identifier activites.

* **Eanblig PL/SQL tool:-**

**Syntax:-**

Alter session set PLscpoe\_setting=’identifiers:all’(enable);

Alter session(or system) set PLscpoe\_setting=’identifiers:null’(disable);

**Virtual private database (or) fine grained access control(FGAC):-**

* Let us take eg:
* Select \* from emp where 1=2;

Note:-

In the above eg ‘where’ represents the condition should be added dynamically. And ‘1=2’ represents predicates.

* Vpd is an alias of fine grained access control(FGAC), the basic concept of FGAC is to provide the row level security(RLS) (some applications deliberately needs this process).
* Vpd provides security.
* Vpd dynamically modify the SQL statement at run time by appending (or) attaching predicate through policy function.
* Policy function is useful to return the predicate.
* We will provide a package called along with schema and object names.
* We can provide ‘N’ no of policy function on a single table and a single policy function can serve for multiple tables.

**Definging policy function:-**

Create or replace function pol\_fun(p\_schema varchar2,p\_object varchar2)

Return varchar2 is l\_pridicate varchar2(10);

Begin

L\_proidicate:=’1=2’;

Return(l\_pridicate);

End potfun;

**Note:**

Here while creating policy functions(p\_schema and p\_object) these 2 in parameters are mandatory. ‘1=2’ this condition will append to select statement dynamically.

**SQL/Plus:-**

Var a varchar2(10);

Exec :a:=pot\_fun(‘apps’,’emp’);

Print a;

Exec dbms\_rls.add\_policy(‘apps’,’emp1’,’res’,’appa’,’pol\_fun’,select,insert,update’);

Select \* from emp1;

Exec.dbms\_rls.drop\_policy(‘apps’,’emp1’,’res’);

**Note:-**

If you create next time with the very same name also same will be applicable.

**Collections:-**To store data in the form of arrays we use collections.

Eg: nested table.

Declare

Type nes is table of varchar2(10);

Vnes nes:=nes(‘a’,’b’,’c’,’d’);

**// where ‘nes’ is datatype and ‘vnes’ is collection variable.**

If vnes.limit is null then

DOPL(‘vnes limit is limitness’);

Else

DOPL(‘vnes limit ‘||vnes.limit’);

End if;

DOPL(‘vnes count’||vnes.count);

DOPL(‘vnes first’||vnes.first);

DOPL(‘vnes last’||vnes.last);

DOPL(‘vnes prior’||vnes.prior(2));

DOPL(‘vnes next’||vnes.next(2));

DOPL(‘vnes values’);

For I in vnes.first..vnes.last loop

DOPL(‘vnes values(‘||I||’)’||vnes(I));

End loop;

if vnes.exists(2) then

dopl(‘vnes second value exists’||vnes(2));

dopl(‘vnes second value doesnot exists’);

end if;

vnes.extend;

vnes.exted(2);

DOPL(‘vnes value after extension’);

For I in vnes.first..vnes.last loop

DOPL(‘vnes values(‘||I||’)’||vnes(I));

End loop;

Vnes(5):=’E’;

Vnes(6):=’F’;

Vnes(7):=’G’;

Vnes.extend(2,3);

DOPL(‘nes value after providing element’);

For I in vnes.first..vnes.last loop

DOPL(‘vnes values(‘||I||’)|| vnes(I));

End loop;

vnes.trim;

vnes.trim(2);

DOPL(‘vnes values after trim’);

For I in vnes.first..vnes.last loop

DOPL(‘vnes values(‘||I||’)’||vnes(I));

End loop;

vnes.delete(1);

vnes.delete(2,5);

DOPL(‘vnes values after specific deletion’);

For I in vnes.first..vnes.last loop

DOPL(‘vnes values(‘||I||’)’||vnes(I));

End loop;

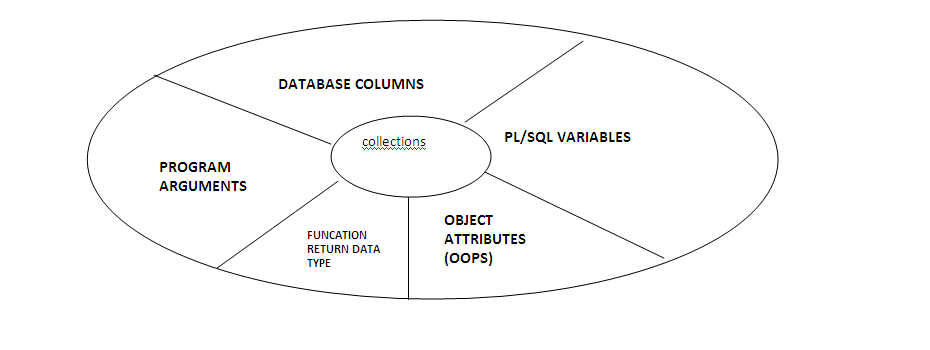
Vnes.delete;

DOPL(‘vnes values after complete deletion’);End;

collection variable should not be ‘NULL’, to avoid that in that case we will define elements.

The function should be empty nes()(or)nes(‘A’,’B’,’C”,’D’);

**Usage of collections:-**

****

**Deff:**To store the data in the form of arrays we use collections

Collections basically are of two types:

1.non-persistance(index by table)(temporary)

2.persistance(varrays, nested table)(permanent)

* Collections are useful to store complex/homogenous data.
* Collections defined in packages will persist for an entire session.
* Limit(is there any limit (or) not (upper boundary)) Extend(2,3);

Where 2 represents extensions will happen. And what ever the value is present in 3 element it will return that. After time only we will delete, reverse is not so we delete at that time empty cell will populate.

Densely packed(with out any cops)

sparse

|  |  |
| --- | --- |
|  |  |
| **1** | **A** |
|  |  |
| **3** | **C** |
|  |  |
| **5** | **C** |

**densely**

|  |  |
| --- | --- |
|  |  |
| **1** | **A** |
| **2** | **B** |
| **3** | **C** |
| **4** | **C** |
| **5** | **D** |

* Extend delete(n) (or) (m,n)

Here we can mention in three ways without paranthesis defining with number(n) .

* In index by table we can key values as var/char also.
* Key values should start with 1 and always positive values.
* Varray always densely.

**Eg:**

Declare

Type nes is table of varchar2(10) index by pls\_integer;

Vnes nes:=nes();

Begin

Null;

End;

**Error:**no function name ‘nes’ exists in this scope.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **NESTED TABLE** | **VARRAYS** | **INDEX BY TABLE** |
| First, Last, Next, prior, count  Limit,  Delete,  Extend,  Trim,  Persistence,  Subscript values,  Usage SQL/PL SQL,  Dense & paye,  Exception,  Storage | Limitless (non upper boundary)  Persistence  Always +ve values,start with 1  SQL & PL/SQL  Initially dense but may be sparse  Out of line storage. | Limit (exits upper bound)  Specific deletion is not possible  Possible within the max-limit  Persistence  Same as nested table..  SQL & PL/SQL  DENSE  INLINE | LIMIT LESS  Non-Persistance  Might be with –ve, allows caharacterstics  Both |

**Bulk bind:-**

* Concept of bulk bind is a mechanism which is used to enhance the performance drastically, by decreasing the context switches.
* Bulk bind reduces the interaction between SQLand PL/SQL engine.
* We will maintain the bulk bind data through collections.

**These are two types:**

**Bulk collect:- (clause)**

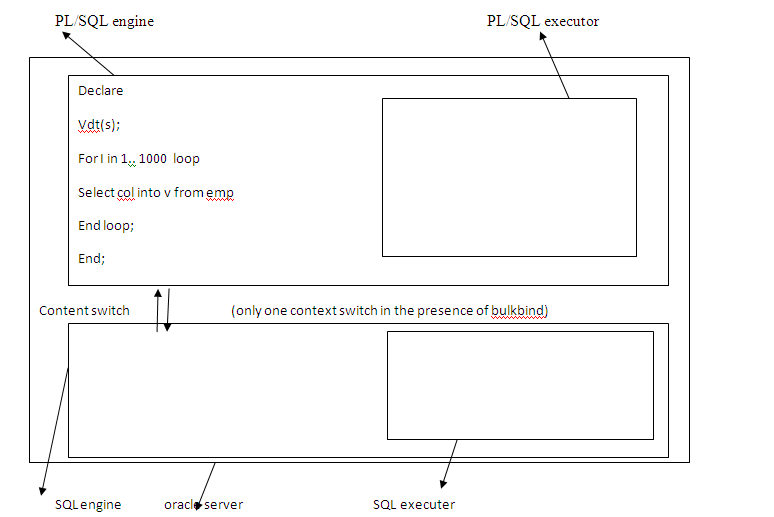
* select col bulkcollect into variable(this variable must be collection variable).
* Fetch cursor bulkcollect into variable.
* Return col bulk collect into variable(we can call out bind also).
* We can use bulk collect only above three other than this we can’t.

**For all statement(DML)**

**syntax:**

for all: in{ var1..var2|value of collection|indecies by collection}

[save exception](10g)DML operation;

* in for all statement we can perform single DML operation only.
* by using bind we can perform.
* In the presence of bulk bind ‘n’ no.of context switches will get compressed (or) bineoled into one single swich.

**Using bulk collect in select statement:-**

Declare

Sdate number:=0;

Edate number:=0;

Type nes is table of pls\_integer;

Vnes nes:=nes();

Begin

Sdate:=dbms\_utility.get\_times;

For I in (select \* from multab) loop

Vnes.extend; (if you don’t write extend it throws an error)

Vnes(vnes.last)=i.sno;

Enol loop;

Edate:=dbms\_utility.get\_time;

DOPL(edate-sdate);

Sdate:=dbms\_utility.get\_time;

Select sno bulk collect into vnes from multabs;

Edate:=dbms\_utility.get\_time;

DOPL(edate-sdate);

End;

* While using bulk collect the variable should be collection variable.

**Using bulk collect in fetch statement:-**

Declare

Sdate number:=0;

Edate number:=0;

Type nes is table of pls\_integer;

Vnes nes:=nes( );

Cursor c is select \* from multab;

Begin

Open c;

Sdate:=dbms\_utility.get\_time;

Loop

Vnes.extend;

Fetch c into vnes(vnes.last);

Exit when c%not found;

End loop;

Close c;

Edate:=dbms\_utility.get\_time;

DOPL(edate-sdate);

Open c;

Sdate:=dbms\_utility.get\_time;

Fetch c bulk collect into vnes;

Edate:=dbms\_utility.get\_time;

Close c;

DOPL(edate-sdate);

End;

For the same program

Fetch c bulk collect into vnes limit 500000;

We can also mention the limit so to specify the no.of records into a collection through fetch statement limit only possible in fetch statement.

**Collections:-**

**Note:-**We won’t initialize the ‘index by table’ datatype variable with a constructor (or) function trying to do so through an error.

Eg:-

Declare

Type nes is table of varchar2(10);

Vnes nes:=nes(‘a’,’b’,’c’);

Begin

Vnes.delete(2)

DOPL(‘vnes first’||vnes.first);

DOPL(‘vnes last’||vnes.last);

DOPL(‘vnes prior’||vnes.prior(3));

DOPL(‘vnes next’||vnes.next(1));

DOPL(‘vnes count’||vnes.count);

If vnes.exists(2) then

DOPL(‘vnes second value exists’||vnes(2)); **// not existed**

End if;

Vnes.extend;

If vnes.existts(4) then

DOPL(‘vnes fourth value exists’||vnes(4));

Else

DOPL(‘vnes fourth does not exists’);

End if;

End;

**Using collections as program parameters:-**

Declare

Type nes is table of varchar2(10);

Vnes nes:=nes(‘a’,’b’,’c’,’d’);

Procedure p(x nes) is

Begin

For I in x.first..x.last loop

DOPL(x(i));

End loop;

End p;

Begin

p(vnes);

end;

**using collections for return datatype:-**

declare

type nes is table of varchar2(10);

vnes:=nes(‘a’,’b’,’c’,’d’);

Y nes:=nes();

Function fun(x nes) return nes is

Begin

Return x; **//here we are returning also**

End fun;

Begin

Y:=fun(vnes);

For I in Y.first..Y.last loop

DOPL(Y(i));

End loop;

End;

**Note:-**

From 11g onwards for all statement allows merge command

Declare

Type nes is table of number;

Vnes nes:=nes( );

Sdate number:=0;

Edate number:=0;

Rdate number:=0;

Begin

Select sno bulk collect into vnes from multab;

Execute immediate ‘truncate table multab’;

Sdate:=dbms\_utility.get.\_time;

For iin vnes.first..vnes.last loop

Insert into multab values(vnes(i));

End loop;

Edate:=dbms\_utility.get\_time;

Rdate:=edate\_sdate;

DOPL(‘elapsed time ‘||rdate);

Execute immediate ‘truncate table multab;

Sdate:=dbms\_utility.get\_times;

For all i in vnes.first..vnes.last

Insert into multab values(vnes(i));

Edate:=dbms\_utility.get\_time;

Rdate:=edate-sdate;

DOPL(‘elapsed time ‘||rdate);

End;

Output:-

Elapsed time 198

5

Upto here we never used DDL commands in executable section.

**Another eg for all statement:-**

Declare

Type nes is table of number;

Vnes nes:=nes( );

Vnes1 nes:=nes( );

Begin

Select sno bulk collect into vnes from multab;

For all I in vnes.first..vnes.last

Delete from multab where sno:=vnes(i);

Return sno bulk collect into vnes1;

DOPL(vnes.count);

End;

**Using save exception (10g):-**

* It filters outs the error records and allows the other records to process, instead of cancelling all the records.
* Save exception saves the errors and we can retrieve the error by using ‘%bulk exception.’

Declare

Type nes is table of varchar2(10);

Vnes nes:=(‘x’,’a’,’y’,’b’,’z’);

Begin

For all I in vnes.first..vnes.last

Save exception

Insert into contab values(values (i));

Exception

When others then

For I in sql%bulk\_exceptions.count loop

DOPL(sql%bulk\_exceptions(i).error\_index||’ ‘||sql%bulk\_exceptions(i).error\_code);

End loop;

End;

* **Error-code displays the error code without ‘-sign’**

**Indeces(index/subscript) of values of:-**

Eg for “values of” clause:-

* Here we are comparing one collection values with another collection of indices values.
* Declare

Type nes is table of varchar2(10);

Vnes nes:=nes(‘a’,’b’,’c’,’d’,’e’,’f’);

Type nes1 is table of pls\_integer;

Vnes1 nes1:=nes1(1,3,5);

Begin

For all I in vnes values of vnes1

Insert into contab values of vnes(i);

End;

**Output:**

|  |
| --- |
| **Loc** |
| A  C  C |

**Eg for “indecies of” clause:-**

Declare

Type nes is table of varchar2(10);

Vnes nes:=nes(‘a’,’b’,’c’,’d’,’e’,’f’);

Vnes1 nes1:=nes1(1,3,5);

Begin

For all I in vnes indices of vnes1

Insert into contab values(vnes(i));

End;

**Output:-**

|  |
| --- |
| **Loc** |
| **A**  **B**  **C** |

**Table function:-**

* It is a function which acts as same to that of table to SQL engine with the help of table operator called ‘table’.
* We provide table function in from clause of a select statement.
* SQL engine can’t identify it as a function.

**Note:-**

* Table functions have to always return data from collections only.
* But we can’t provide functions in from clause.

**Eg:-**

Create or replace function colfun return nes is

Vnes nes:=nes(‘a’,’b’,’c’,’d’,’e’,’f’)

Begin

Return vnes;  **// here vnes always collection.**

End;

Select \* from table(col\_fun);

Column\_val // it internally pseudo column generates, introduced from 10g

A

B

C

D

E

F

* Table function data will be displayed under a column name called column\_val which is a pseudo column.

**Dynamic SQl and Dynamic PL/SQL:-**

* Dynamic SQL statements are the statements which are constructed (or) get framed (or) build up at the time of execution.
* When an SQL statement get framed before of compilation time those are called ‘static SQL’.
* Creation of PL/SQL program at runtime is called as “dynamic PL/SQL”.
* To execute dynamic SQL and PL/SQL we use

1. Execute immediate(statement).
2. DBMS\_SQL(packages)

**Note:-**Dynamic SQL statement falls pray, to SQL injections. We can also execute immediate to execute DDL statements in the program which is normally not possible.

**Syntax for execute immediate:-**

Execute immediate ‘string’ [into {var1,var2..}]

[using[in/out/inout]] bind\_variable,..];

**Eg for execute immediate:-**

Declare

V\_create varchar2(100):= ‘create table etab(sno number);

V number(5):=10;

Vsno number(5);

Begin

Execute immediate v\_create;

Execute immediate ‘insert into etab values(:1)’ usingv;

Execute immediate ‘select sno from etab where sno:=2’ into vsno using v;

Execute immediate ‘begin null; end;’;

Execute immediate ‘drop table etab;

DOPL(vsno);

End;

* At runtime if you want to drop the database object, we prefer following program.

SQL>create or replace procedure p(X varchar2) is

Begin

Execute immediate ‘drop table’||x;

End p;

>exec p(‘stab’);

* The above program is sensitive for SQL injection but by using bind arguments we’ll safeguard the above program.

**Using bind arguments will also enhance the performance:-**

Create or replace procedure p(X varchar2) is

V varchar2(10);’begin

V:=x;

Execute immediate ‘DROP table :a’ using v;

End p;

**SQL injection:-(**safe guarding PL/SQL code from SQL injection)

* Reducing the surface attack is a preferable step to safe guard the PL/SQL code from SQL injection.
* We will reduce the attacking surface in two ways

1. Revoking unnecessary, unintended and excess privileges from user.

2. defining a PL/SQL program with invoker write (instead of definer write).

* Grant select on stab to u2;
* Select \* from u1.stab;
* Insert into u1.stab values() **//invalid**

**Defining a program with definer write:-**

Create or replace procedure p is v number(10):=10;

Begin

Insert into u1.stab values(10);

Select sno into v from u1.stab where sno=v;

End p;

* Grant execute on p to u2;
* Execute u1.p
* Select \* from u1.stab;

**Defining a program with invoker write:-**

Create or replace procedure p authid current user is v number(10):=10;

Begin

Insert into u1.stab values(v);

Select sno into v from u1.stab where sno=v;

End p;

* Grant execute on p to u2;
* Exec u1.p //insufficient privilege
* To deduce the SQL injection (or) to immunize the PL/SQL code from SQL injection better avoid dynamic SQL statements.
* Following is an eg of which specifies need to avoid dynamic SQL statements.

Create or replace procedure p(X-name varchar2) is

Vrec sys\_refcursor;

V\_name varchar2(10);

V\_deptno number(5);

Begin

Open vrec for ‘select ename,deptno from emp where enmae=’||x\_name;

**// here ‘vrec’ as a variable**

Loop

Fetch vrec into v\_name, v\_deptno;

DOPL(v\_name||’ ‘||v\_deptno);

Exit when vrec%not found;

End loop;

Close vrec;

End;

**Calling environment:-**

**SQL>**exec p(‘ ‘ ‘king’ ‘ ‘);

King 10

Exec p(‘null union select ename, sal from emp’);

* Here we have a chance for attacking malicious data so, to restrict the SQL injection convert the dynamic SQL into static as shown in following eg.

Create or replace procedure p(X-name varchar2) is

Vrec sys\_refcursor;

V\_name varchar2(10);

V\_deptno number(5);

Begin

Open vrec for select ename, deptno from emp where ename=x\_name;

Loop

Fetch vrec into v\_name, v\_deptno;

DOPL(v\_name||’ ‘||v\_deptno);

Exit when vrec%not found;

End loop;

Close vrec;

End;

* Exec p(‘king’);

King 10

* Exec p(‘null union select ename,sal from emp); (here getting unintended data is not possible).

**Note:-**

* When we know all the SQl identifiers (column, table names) before of fraiming a statement, in that case only it is possible to convert the dynamic SQL statements into static SQl which means it is not possible to convert all the dynamic SQL statements static if this is the case then mien/mise the attacking of SQL injection by using “bind argument” as shown in the following snippet (or) eg.
* In addition with protectiong the PL/SQL code from SQL injection it also possible to enhance the performance immeyely(more effectively).
* Using bind arguments in dynamic SQL will avoid hard parsing(buffer myss) and promote soft parsing(buffer hit).
* For the above eg just change

Begin

Open vrec for ‘select enaame, deptno from emp where ename=:’ using x\_name;

Loop

**// remaining structure is same.**

* Exec p(‘king’);
* Exec p(‘null union select ename,sal from emp’);
* Finally we will reduce the SQL injection by sanitizing the user inputs with the help of an APL i.e DBMS\_assert(package).
* While tunning the PL/SQL code avoid implicit conversion which ia a burden to the ename as server as such in the following.

**Eg:-**

Declare

Stime number:=0;

Etime number:=0;

V varchar2(10);

Begin

Stime:=dbms\_utility.get\_time;

For I in 1..1000 loop

V:=I;

End loop;

Etime:=dbms\_utility.get\_time;

DOPL(edate\_sdate);

End;

Declare

Stime number:=0;

Etime number:=0;

V number(5);

Begin

Stime:=dbms\_utility.get\_time;

For I in 1..1000 loop

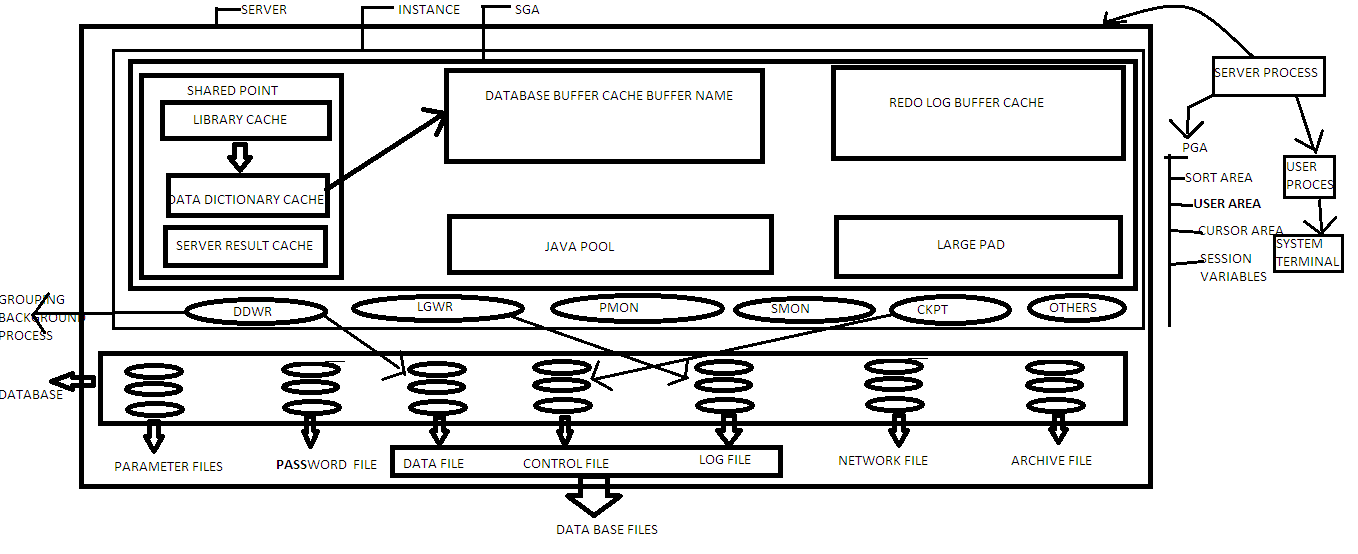
V:=i;

End loop;

Ctime:=dbms\_utility.get\_time;

DOPL(edate-sdate);

end;

**cursor eg:-**

declare

cursor c is select \* from emp;//here just defines the definition c.

vrow c%rowtype;

begin

open c;//here select stmt get executes and opens the memory area and gets data.

Loop

Fetch c into vrow;//here fetching will happen.

DOPL(vrow.ename);

If c%notfound then //(1)

Exit;

End if;

End loop;

Close c;

End;

In case of (1) we can write

If c%rowcount=3 then

Exit;

End if;

* If you write like this we can stop the program after 3rd record. In this case for loop is not prefereable, simple loop is prefereable.

**Paramatarised cursor eg:-**

Declare

Cursor c(x number) is select \* from emp where deptno:=x;

Vrow emp%rowtype;

Vno number(5):=&n;

Vnol number(5):=&m;

Begin

Open c(vno);

Loop

Fetch c into vrow;

DOPL(vrow.ename)

Exit when c%not found;

End loop;

Close c;

Open c(vnol);

Loop

Fetch c into vrow;

DOPL(vrow.ename);

Exit when c%not found;

End loop;

Close c;

* Before ‘open c’:

% is open -F

% found –E

% notfound -E

% rowcount -E

* After ‘open c’:

% is open -T

% found -T

% notfound -F

% rowcount -0

* After fetch: (1st fetch)

% is open -T

% found -1

% notfound -F

% rowcount -1

* After close c:

% is open -F

% found -E

% notfound -E

% rowcount -E

We can use cursor variable as a parameter in stored procedures

* Create or replace function f return sys\_refcursor is vrec sys\_refcursor;

Begin

Open vrec for select \* from emp;

Return vrec;//return statement

End f;

* We can also use sys\_refcursor datatype for function return value (or) using cursor variable as a return value in function.
* Calling:-

Var refvar refcursor;

Exec :refvar:=f;

Print refvar;//emp data:14records.

* Create or replace function f(vrec out sys\_refcursor) return Boolean

Begin

Open vrec for select \* from emp;

Close vrec;

Return true;

End;

Declare

Vrec1 sys\_refcursor;

V Boolean;

Vrow emp%rowtype;

Begin

V:=f(vrec1);

Loop

Fetch vrec1 into vrow;

DOPL(vrow.ename);

Exit when vrec1%not found;

End loop;

End;

* Create or replace function f(vrec out sys\_refcursor) return Boolean is

Vrow emp%rowtype;

Begin

Open vrec for select \* from emp;

Fetch vrec into vrow;

Return true;

End f;

Declare

Vrec1 sys\_refcursor;

V Boolean;

Vrow emp%rowtype;

Begin

V:=f(vrec1);

Loop

Fetch vrec1 into vrow;

DOPL(vrow.ename);

Exit when vrec1%not found;

End loop;

End;

**Output:-** It will be 13 records:

Black

Clak

.

.

.

* Here points the second record in emp.
* Create or replace function f return sys\_refcursor is

Vrec sys\_refcursor;

Vrec1 sys\_refcursor;

Begin

Open vrec for select \* from emp;

Vrec1:=vrec;

Close vrec; //here ‘close vrec’ closes both the cursors.

Return vrec1;

End f;

Declare

Vrec2 sys\_refcursor;

Vrow emp%rowtype;

Begin

Vrec2:=f;

Loop

Fetch vrec2 into vrow;

DOPL(vrow.ename);

Exit when vrec2%notfound;

End loop;

End;

**Analyzing PL/SQL code:-**

* By using ‘format-call-stack” procedure we will sequentially trace out the program calls so , to analyse the coading as shown in the following eg. (or) snippet.
* Create or replace procedure p is

Begin

DOPL(‘dbms\_utility.format\_call\_stack’);

End;

Create or replace procedure p1 is

Begin

P;

End p1;

Create or replace procedure p2 is

Begin

P1;

End p2;

Procedure created.

Begin

P2;

End;

PL/SQL call stack

|  |  |
| --- | --- |
| **Lienumber** | **Object** |
| 3 | Procedure APPs-p |
| 3 | Procedure |
| 3 | Procedure apps.p2 |
| 2 | Anonymous block |

**Atomics of program (or) PL/SQL program:-**

Identifiers->literals->delimiters->comments

Identifiers are also derived as:

Constants, variables, packages, cursors, exceptions.

* If program is length at that time we have to know where calling happened to know the line numbers for easy identification we use above package.

**/\***

**…….**

**………**

**\*/**

**Note:-**

If you want to execute to multiple lines we give like this.

**/\***

**……**

**……..**

**/\***

**…..**

**……**

**\*/**

**…….**

**….....**

**\*/**

**Note:-**

**We can’t nest comment in comment.**

**Instead of trigger:-**

* This trigger is for views, to perform DML operations on complex views will not be allowed directly, but by using instead of trigger it is passible.
* This trigger is always a rowlevel trigger even if you won’t mention the “for each row” clause.
* Defining instead of trigger on tables ends with an error.
* Create table vtabl(sno number(5));
* Create table vtab2(loc varchar2(10);
* Create view comview as select \* from vtab1, vtab2;
* Insert into comview values(10,’hyd’);

**Error:**

Can not modify a column which maps to a non key preserved data.

Create or replace triggr comtrig

Instead of insert on comview;

Begin

Insert into vtab1 values(:new.sno);

Insert into vtab2 values(:new.loc);

End;

Insert into comview values(10,’hyd’);//here insertion is successfully happening on view.

**Note:-**DML operation can’t perform on complex views. But as in the above case trigger is inserting the values before insertion.

**Mutating error:-**

* We get this error when a row level trigger attempts to read or write the table from which it was raised.
* You won’t get this error in statement level.
* Create or replace trigger muttrig Before update on vtab1

For each row

Declare

Vno number95);

Begin

Select sno into vno from vtab1 where sno=30;

End;

* Update vtab1 set sno=50;

Error: table apps.vtab1 is mutating, trigger/function may not see it.

* To Avoid Mutating Errors We Use “Pragma Atonomous Transaction” (or) we can also use “statement level trigger”.