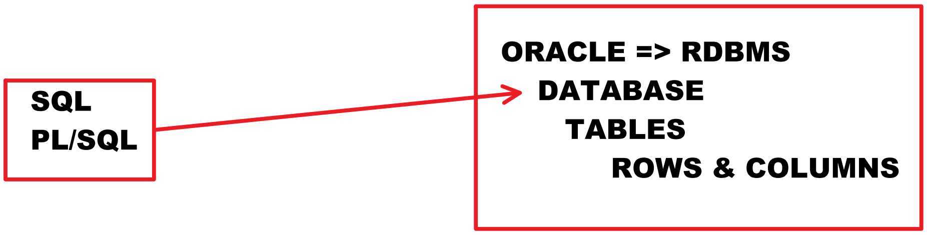
PL/SQL

Thursday, April 18, 2024 10:00 AM



SQL
PL/SQL

ORACLE => RDBMS
DATABASE
TABLES
ROWS & COLUMNS

PL/SQL:

- **PL => Procedural Language.**
- **SQL => Structured Query language.**

- **It is a programming language.**

- **In this language, by developing PL/SQL programs we can communicate with ORACLE DB.**

- **is extension of SQL.**

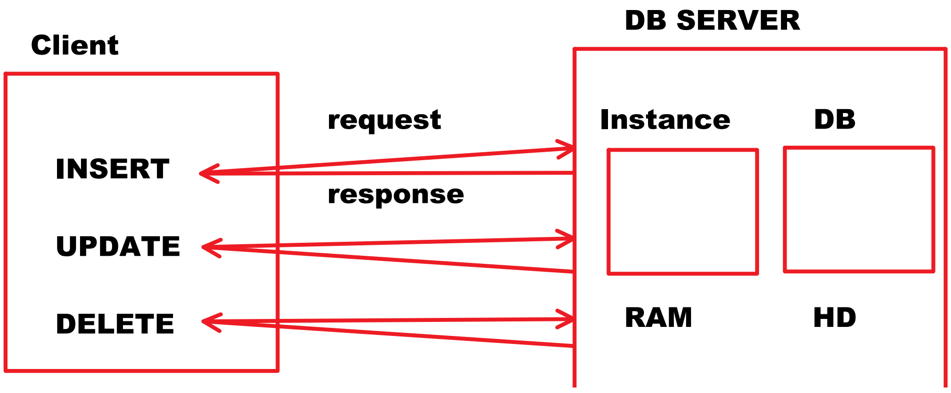
- **PL/SQL = SQL + Programming.**

- **All SQL queries can be written as statements in PL/SQL program.**

Advantages:

- **improves the performance.**
- **provides conditional control structures.**
- **provides looping control structures.**
- **provides Exception handling.**
- **provides security.**
- **provides reusability.**

improves the performance:



DB SERVER

Client

request

Instance

DB

INSERT

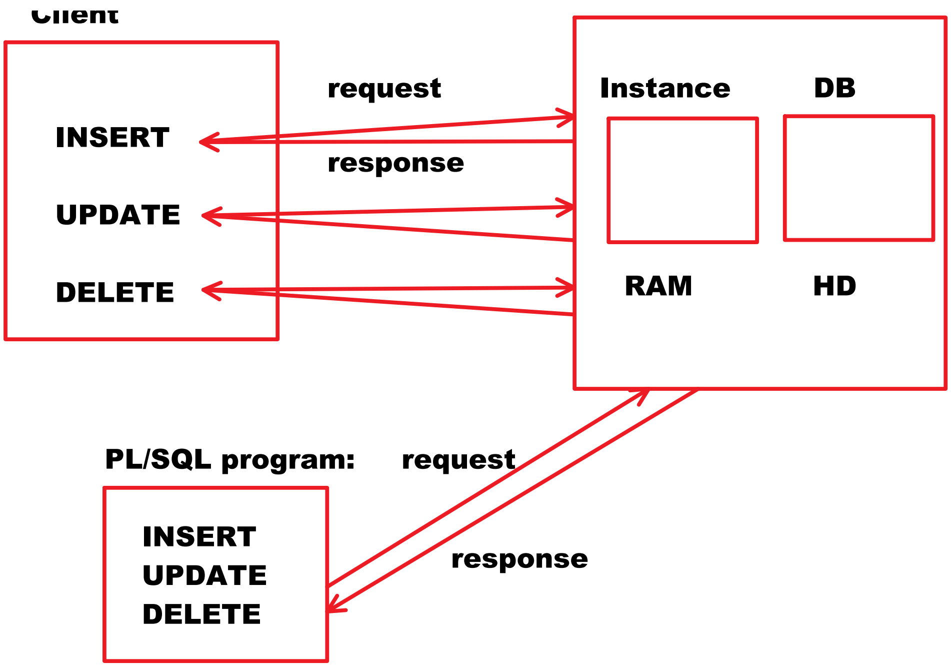
response

UPDATE

DELETE

RAM

HD



In PL/SQL program we can group the SQL queries and submit as one request. it reduces no of requests and responses. then automatically performance will be improved.

Types of Blocks:

2 types:

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

Anonymous Block:

- **A block without name is called "Anonymous Block".**

Named Block:

- **A block with the name is called "Named block".**
- **Examples: procedures, functions, packages and triggers**

Anonymous Block:

```
BEGIN
   --Statements
END;
```

Named Block:

```
CREATE PROCEDURE demo AS
BEGIN
   --Statements
END
```

```
    --Statements
END;
```

Block

```
BEGIN
    --Statements
END;
```

Block

Syntax of Anonymous Block:

```
DECLARE
    --declare the variables
BEGIN
     --Executable Statements
END;
/
```

Declaration part [optional]

Execution part

```
In C,    printf("hello");

In Java,  System.out.println("hello");

In PL/SQL, dbms_output.put_line('hello');
```

```
CREATE PACKAGE dbms_output

PROCEDURE put_line(...) AS
BEGIN
    --statements
END;
```

dbms_output.put_line():

- 'put_line()' is a packaged procedure.
- it is defined in 'dbms_output' package.
- put_line() procedure is used to print the data on screen.

Syntax to call packaged procedure:

```
<package_name>.<procedure_name>(<arguments_list>)
```

Example:

```
dbms_output.put_line('hello');       => procedure call
```

Note:

- **SQL is not case sensitive language.**
- **PL/SQL is not case sensitive language.**
- **In PL/SQL program, every statement ends with ; [semicolon].**

Program to print hello on screen:

Developing PL/SQL program:

```
BEGIN
   dbms_output.put_line('HELLO');
END;
/
```

- **Type above program in any Text Editor like notepad, edit plus ...etc**

- **Save it in "D:" Drive, "batch9am" Folder with the name "HelloDemo.sql".**

Compiling and running PL/SQL program:

Syntax to compile PL/SQL program:

```
 @<path>
```

- **Open SQL PLUS.**
- **login as user**

```
SQL> SET SERVEROUTPUT ON

SQL> @d:\batch9am\HelloDemo.sql
Output:
HELLO
```

Note:
By default, SERVEROUTPUT is OFF.
If it is OFF, messages cannot be sent to output.
To send messages to output, we must set SERVEROUTPUT as ON.

setting serveroutput as ON:

```
SQL> SET SERVEROUTPUT ON
```

Data Types in PL/SQL:

Character Related	**Char(n)**
	Varchar2(n)
	String(n) PL/SQL only
	LONG
	CLOB
	
	nChar(n)
	nVarchar2(n)
	nCLOB

Integer related	**NUMBER(p)** **INTEGER** **INT** **BINARY_INTEGER PL/SQL only** **PLS_INTEGER PL/SQL only**
Floating Point related	**NUMBER(p,s)** **FLOAT** **BINARY_FLOAT** **BINARY_DOUBLE**
Date & Time related	**DATE** **TIMESTAMP**
Binary related	**BFILE** **BLOB**
Boolean related	**BOOLEAN** **[till ORACLE 21C, PL/SQL only]** **[from ORACLE 23C, SQL also]**
Attribute related	**%TYPE [PL/SQL only]** **%ROWTYPE [PL/SQL only]**
Cursor related	**SYS_REFCURSOR [PL/SQL only]**
Exception related	**EXCEPTION**

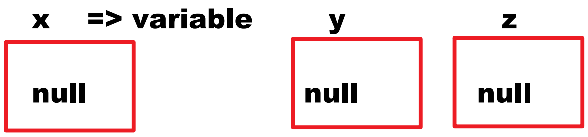
Declare the Variable:

Syntax:

<variable> <data_type>;

Examples:

```
x INT;
y VARCHAR2(10);
z DATE;
```



Note:

Variable:

- It is an Identifier.
- It is name of storage location
- In this location, it holds a value
- A variable can hold one value at a time

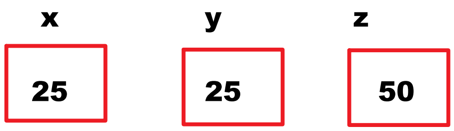
- **Declare variables in DECLARE section [declaration part]**

Assigning value to variable:

Syntax:

<variable> := <constant / variable / expression>;

```
x INT;
y INT;
z INT;
```



Assignment Operator :=

```
x := 25;
y := x;
z := x+y;
```

Printing data:

```
dbms_output.put_line(x);
```

Reading data:

```
x := &x;
```

Output:

enter value for x: 25

```
a := &firstnum;
```

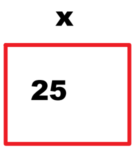
Output:
enter value for firstnum: 500

Initializing variable:
Initialization means, giving value at the time of declaration

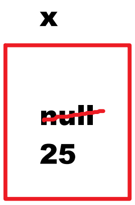
Example:

```
x INT:=25;
```

```
x INT:=25;
--initialization
```


x
25

```
x INT;   --declare
x:=25;   --asign
```


x
~~null~~
25

DECLARE	**x INT;**
ASSIGN	**x := 25;**
PRINT	**dbms_output.put_line(x);**
READ	**x := &x;**
Initialize	**x INT := 25;**

Program to add 2 numbers:

x y
10 20

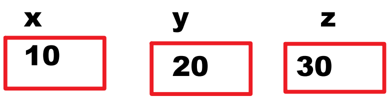
z
10+20 = 30

declare x,y,z as NUMBER

assign 10 value to x
assign 20 value to y

calculate x+y and store it in z

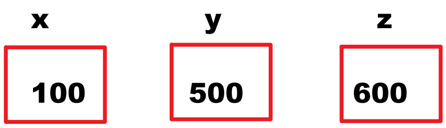
print z


x y z
10 20 30

DECLARE

```
   x NUMBER(4);
   y NUMBER(4);
   z NUMBER(4);
BEGIN
   x := 10;
   y := 20;

   z := x+y;

   dbms_output.put_line('sum=' || z);
   dbms_output.put_line('sum of ' || x || ' and ' || y || ' is ' || z);
END;
/
        Output:
           sum=30
           sum of 10 and 20 is 30
```

Program to add 2 numbers. Read those 2 numbers at runtime:



```
DECLARE
   x NUMBER(4);
   y NUMBER(4);
   z NUMBER(4);
BEGIN
   x := &x;
   y := &y;

   z := x+y;

   dbms_output.put_line('sum=' || z);
END;
/

Output:
SQL> @d:\batch9am\ReadDemo.sql
Enter value for x: 50
old   6:        x := &x;
new   6:        x := 50;
Enter value for y: 10
old   7:        y := &y;
new   7:        y := 10;
sum=60

SQL> SET VERIFY OFF

SQL> /
```

```
Enter value for x: 20
Enter value for y: 30
sum=50
```

Note:
to avoid of displaying old and new parameters we need to SET VERIFY as OFF.

to set verify off:

```
SQL> SET VERIFY OFF
```

Using SQL commands in PL/SQL:

- DRL,DML,TCL commands can be used directly in PL/SQL program.
- DDL,DCL commands cannot be used directly in PL/SQL. To use them, we use DYNAMIC SQL.

Using SELECT Command in PL/SQL:

Syntax:

```
SELECT <columns_list>/* INTO <variables_list>
FROM <table_name>
WHERE <condition>;
```

Examples:

```
SELECT ename, sal  INTO x,y
FROM emp
WHERE empno=7499;
```

EMPNO	ENAME	SAL	HIREDATE
7369	SMITH	800	..
7499	ALLEN	1600	23-AUG-1981
7521	WARD	2000	..

x	y
ALLEN	1600

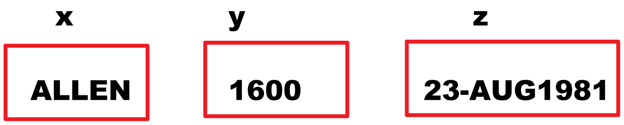
```
SELECT ename, sal, hiredate INTO x,y,z
FROM emp
WHERE empno=7499;
```

select the data and copy into variables

x y z

using these variables we work with table data

```
WHERE empno=7499;
```



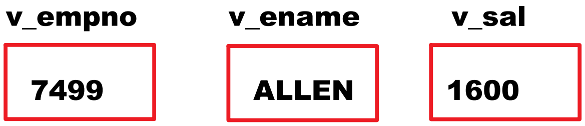
using these variables
we work with table data
in program

column names can be used
in SQL commands only

Example on uisng SELECT in PL/SQL:

```
enter value for empno: 7499
ALLEN     1600

enter value for empno: 7934
MILLER    3000
```

- declare the variables v_empno, v_ename, v_sal
- read empno
- select given empno's data and copy into variables
- print the data



```
DECLARE
   v_empno NUMBER(4);
   v_ename VARCHAR2(10);
   v_sal NUMBER(7,2);
BEGIN
   v_empno := &empno;

   SELECT ename, sal INTO v_ename, v_sal
   FROM emp WHERE empno=v_empno;

   dbms_output.put_line(v_ename || '   ' || v_sal);
END;
/
```

EMP

EMPNO	ENAME	SAL
7369	SMITH	800
7499	ALLEN	1600
7521	WARD	2000

```
Output:
enter value for empno: 7499
7499      1600
```

Assignment:

ACCOUNTS

ACNO	NAME	BALANCE

1001	A	50000
1002	B	80000
1003	C	20000

display the account balance of given acno:

enter value for acno: 1002
account balance is: 80000

EMP TABLE
EMPNO NUMBER(4)

7369
7499
7521

v_empno NUMBER(2) => max => 99

problem-1:
field sizes are mismatching

v_empno DATE;

problem-2:
data types are mismatching

%TYPE:

- **is attribute related data type.**
- **it is used to declare a variable with table column's type.**
- **it avoids mismatch between field sizes of table column and variable.**
- **it avoids mismatch between data types of table column and variable.**

Syntax:

```
<variable> <table_name>.<column>%TYPE;
```

Example:

```
v_empno EMP.EMPNO%TYPE;
```

EMP table's EMPNO column's data type will be taken as "v_empno" variable's data type

```
v_ename EMP.ENAME%TYPE;
```

Program to demonstrate %TYPE:

display the emp record of given empno:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_ename EMP.ENAME%TYPE;
   v_sal EMP.SAL%TYPE;
BEGIN
   v_empno := &empno;

   SELECT ename, sal INTO v_ename, v_sal FROM emp
   WHERE empno=v_empno;

   dbms_output.put_line(v_ename || '    ' || v_sal);
END;
/
```

Output:

```
enter value for empno: 7499
7499      1600
```

%ROWTYPE:

- **it is attribute related data type.**
- **it is used to hold entire row of a table.**
- **A %ROWTYPE variable can hold only 1 row at a time.**
- **It reduces no of variables.**

Syntax:

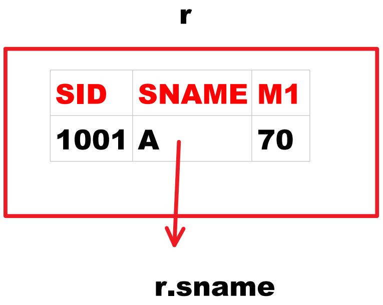
```
<variable> <TABLE_NAME>%ROWTYPE;
```

Example:

```
r STUDENT%ROWTYPE;
```

STUDENT

SID	SNAME	M1
1001	A	70
1002	B	80
1003	C	66
1004	D	45
1005	E	77



r

SID	SNAME	M1
1001	A	70

r.sname

```
SELECT * INTO r FROM student
WHERE sid=1001;

dbms_output.put_line(r.ename);  --A
dbms_output.put_line(r.m1);       --70
```

Example on %ROWTYPE:

Program to display dept details of given deptno:

```
DECLARE
   v_deptno DEPT.DEPTNO%TYPE;
   r DEPT%ROWTYPE;
BEGIN
   v_deptno := &deptno;

   SELECT * INTO r FROM dept WHERE deptno=v_deptno;

   dbms_output.put_line(r.dname || '    ' || r.loc);
END;
/
```

v_Deptno

30

r

deptno	dname	loc
30	SALES	CHICAGO

Output:

```
enter vaue for deptno: 30
SALES     CHICAGO
```

Display emp record of given empno:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   r EMP%ROWTYPE;
BEGIN
   v_empno := &empno;

   SELECT * INTO r FROM emp
   WHERE empno=v_empno;

   dbms_output.put_line(r.ename || '    ' || r.sal);
END;
/
```

%TYPE => to declare a variable with table column's type

%ROWTYPE => to hold a table row

Using UPDATE command in PL/SQL:

Program to increase salary of given empno with given amount:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_amount FLOAT;
BEGIN
   v_empno := &empno;
   v_amount := &amount;

   UPDATE emp SET sal=sal+v_amount
   WHERE empno=v_empno;

   COMMIT;

   dbms_output.put_line('sal increased..');
END;
/
```

v_empno	v_amount
7934	2000

EMP

EMPNO	ENAME	SAL
7934	MILLER	~~1300~~ 3300
..	..	..

Output:

```
Enter value for empno: 7934
Enter value for amount: 2000
sal increased..
```

Using DELETE command in PL/SQL:

Program to delete emp record of given empno:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
BEGIN
   v_empno := &empno;   --7788

   DELETE FROM emp WHERE empno=v_empno;

   COMMIT;

   dbms_output.put_line('record deleted..');
END;
/
```

v_empno
7788

Output:

```
Enter value for empno: 7788
record deleted..
```

Using INSERT command in PL/SQL:

STUDENT

SID	SNAME	M1
1001	A	70

```
CREATE TABLE student
(
sid NUMBER(4),
sname VARCHAR2(10),
m1 NUMBER(3)
);
```

Program to insert student record into table:

```
DECLARE
   r STUDENT%ROWTYPE;
BEGIN
   r.sid := &sid;
   r.sname := '&sname';
   r.m1 := &m1;

   INSERT INTO student VALUES(r.sid, r.sname, r.m1);
   COMMIT;

   dbms_output.put_line("record inserted..);
END;
/
```

(or)

```
BEGIN
   INSERT INTO student VALUES(&sid, '&sname', &m1);
   COMMIT;

   dbms_output.put_line("record inserted..);
END;
/
```

Output:

```
Enter value for sid: 1001
```

```
   Enter value for sname: A
   Enter value for m1: 70
   record inserted..
```

Program to find experience of given empno:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_hiredate DATE;
   v_exp INT;
BEGIN
   v_empno := &empno;   --7369

   SELECT hiredate INTO v_hiredate FROM emp
   WHERE empno=v_empno;

   v_exp := (sysdate-v_hiredate)/365;

   dbms_output.put_line('experience=' || v_exp);
END;
/
```

data types

declare	x INT;
assign	x:=30;
print	dbms_output.put_line(x);
read	x:=&x;
initialize	x INT:=30;

SELECT

INSERT

UPDATE

DELETE

%TYPE

%ROWTYPE

Control Structures

Monday, April 22, 2024 10:15 AM

max marks: 100
min marks: 40

```
DECLARE
   m INT:=30;
BEGIN
   dbms_output.put_line('PASS');

   dbms_output.put_line('FAIL');
END;
/
Output:
PASS
FAIL
```

```
DECLARE
   m INT:=30;
BEGIN
   IF m>=40 THEN
      dbms_output.put_line('PASS');
   ELSE
      dbms_output.put_line('FAIL');
   END IF;
END;
/
Output:
FAIL
```

Control Structures:

- **Control Structure is used to control the flow of execution of statements.**
- **Normally, Program gets executed sequentially. To change sequential execution, to transfer the control to our desired location we use Control Structures.**

PL/SQL provides following Control Structures:

Conditional	**IF .. THEN**
	IF .. THEN .. ELSE
	IF .. THEN .. ELSIF
	Nested If
	CASE

Looping	**WHILE** **FOR** **SIMPLE LOOP**

Jumping	**GOTO** **EXIT** **EXIT WHEN** **CONTINUE** **RETURN**

Conditional Control Structures:

Conditional Control Structure executes the statements based on conditions.

PL/SQL provides following Conditional Control Structures:

- **IF .. THEN**
- **IF .. THEN .. ELSE**
- **IF .. THEN .. ELSIF**
- **NESTED IF**
- **CASE**

IF .. THEN:

Syntax:

```
IF <condition> THEN

   --Statements                      --condition => TRUE

END IF;
```

The statements in "IF .. THEN" get executed when the condition is TRUE.

TO perform an action based on condition we use "IF .. THEN"

Example on IF .. THEN:

Program to delete emp record of given empno.
If emp experience is more than 42 then only delete emp

record.

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_hiredate DATE;
   v_exp INT;
BEGIN
   v_empno := &empno;

   SELECT hiredate INTO v_hiredate FROM emp
   WHERE empno=v_empno;

   v_exp := (sysdate-v_hiredate)/365;
   dbms_output.put_line('experience=' || v_exp);

   IF v_exp>42 THEN
      DELETE FROM emp WHERE empno=v_empno;
      COMMIT;
      dbms_output.put_line('record deleted..');
   END IF;
END;
/
```

```
Output-1:
Enter value for empno: 7876
experience=41

Output-2:
Enter value for empno: 7566
experience=43
record deleted..
```

IF .. THEN .. ELSE:

Syntax:

```
IF <condition> THEN
   --Statements                  --Condn => TRUE
ELSE
   --Statements                   --Condn => FALSE
END IF;
```

```
ELSE
   --Statements               --Condn => FALSE
END IF;
```

- The statements in "IF .. THEN" get executed when condition is TRUE.
- The Statements in "ELSE" get executed when condition is FALSE.
- To perform any 1 of 2 actions we use "IF .. THEN..ELSE".

Examples on "IF .. THEN .. ELSE":

Program to increase salary of given empno based on job as following:
if job is manager then increase 20% on sal
for others, increase 10% on sal

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_job EMP.JOB%TYPE;
   v_per FLOAT;
BEGIN
   v_empno := &empno;

   SELECT job INTO v_job FROM emp
   WHERE empno=v_empno;

   IF v_job='MANAGER' THEN
      v_per := 20;
   ELSE
      v_per := 10;
   END IF;

   UPDATE emp SET sal=sal+sal*v_per/100
   WHERE empno=v_empno;

   COMMIT;
```

```
    dbms_output.put_line('job is ' || v_job);
    dbms_output.put_line(v_per || '% on sal increased..');
END;
/
```

```
Output-1:
Enter value for empno: 7698
job is MANAGER
20% on sal increased..

Output-2:
Enter value for empno: 7934
job is CLERK
10% on sal increased..
```

IF .. THEN .. ELSIF:

Syntax:

```
IF <condn1> THEN
   --Statements
ELSIF <condn2> THEN
   --Statements
.
.
ELSE
   --Statements
END IF;
```

- **The statements in IF .. THEN .. ELSIF get executed when corresponding condition is TRUE.**
- **When all conditions are FALSE, it executes ELSE block statements.**
- **Writing ELSE block is optional.**

- **To perform any 1 of more than 2 actions we use "IF .. THEN .. ELSIF".**

Example on IF .. THEN .. ELSIF:

Program to increase salary of given empno based on job as following:

if job is MANAGER => increase 20% on sal

```
            CLERK                                  15%
            others                                 10%
```

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_job EMP.JOB%TYPE;
   v_per FLOAT;
BEGIN
   v_empno := &empno;

   SELECT job INTO v_job FROM emp
   WHERE empno=v_empno;

   IF v_job='MANAGER' THEN
      v_per := 20;
   ELSIF v_job='CLERK' THEN
      v_per := 15;
   ELSE
      v_per := 10;
   END IF;

   UPDATE emp SET sal=sal+sal*v_per/100
   WHERE empno=v_empno;

   COMMIT;

   dbms_output.put_line('job is ' || v_job);
   dbms_output.put_line(v_per || '% on sal increased..');
END;
/
```

Output-1:

```
Enter value for empno: 7698
job is MANAGER
20% on sal increased..
```

Output-2:

```
Enter value for empno: 7934
job is CLERK
15% on sal increased..
```

```
Output-3:
Enter value for empno: 7499
job is SALESMAN
10% on sal increased..
```

Assignment:

```
Program to increase salary of given empno
based on deptno as following:
if deptno 10 => increase 10% on sal
            20                  20%
            30                  15%
            others               5%
```

Nested If:

Writing "IF" in another "IF" is called "NESTED IF".

Syntax:

```
IF <condition1> THEN

   IF <condition2> THEN
         --Statements            --condn1, condn2 => TRUE
   END IF;

END IF;
```

The statements in INNER IF get executed when outer condition and inner condition are TRUE.

Example on NESTED IF:

STUDENT

SID	SNAME	M1	M2	M3
1001	A	70	90	80
1002	B	60	30	75

RESULT

sid	total	avrg	result

Program to calculate total, average and result of given sid and insert these values in RESULT table:
max marks: 100 for each subject
min marks: 40 for pass in each subject
in any subject if marks are <40 then result is FAIL
if PASS check avrg.
if avrg is 60 or more => FIRST
if avrg is b/w 50 to 59 => SECOND
if avrg b/w 40 to 49 => THIRD

```
CREATE TABLE student
(
sid NUMBER(4),
sname VARCHAR2(10),
m1 NUMBER(3),
m2 NUMBER(3),
m3 NUMBER(3)
);

INSERT INTO student VALUES(1001,'A',70,90,80);
INSERT INTO student VALUES(1002,'B',60,30,75);
COMMIT;


CREATE TABLE result
(
sid NUMBER(4),
total NUMBER(3),
avrg NUMBER(5,2),
result VARCHAR2(10)
);
```

v_sid

1001

enter .. sid: 1001

r1

SID	SNAME	M1	M2	M3
1001	A	70	90	80

r2

sid	total	avrg	result
	240	80	FIRST

RESULT

sid	total	avrg	result
v_sid	r2.total	r2.avrg	r2.result

Program:

```
DECLARE
   v_sid STUDENT.SID%TYPE;
   r1 STUDENT%ROWTYPE;
   r2 RESULT%ROWTYPE;
BEGIN
   v_sid := &sid;

   SELECT * INTO r1 FROM student WHERE sid=v_sid;

   r2.total := r1.m1+r1.m2+r1.m3;
   r2.avrg := r2.total/3;

   IF r1.m1>=40 AND r1.m2>=40 AND r1.m3>=40
   THEN
      IF r2.avrg>=60 THEN
         r2.result := 'FIRST';
      ELSIF r2.avrg>=50 THEN
         r2.result := 'SECOND';
      ELSE
         r2.result := 'THIRD';
      END IF;
   ELSE
      r2.result := 'FAIL';
   END IF;

   INSERT INTO result VALUES(v_sid, r2.total,
   r2.avrg, r2.result);

   COMMIT;

   dbms_output.put_line('result stored in RESULT
   table');
END;
/
```

Output:

```
Enter value for sid: 1001
result stored in RESULT table
```

CASE:

2 ways:

- **Simple CASE [same as switch in JAVA]**
- **Searched CASE [same as if else if in JAVA]**

Simple CASE:
It can check equality condition only

Searched CASE:
It can check any condition

Syntax of Simple CASE:

```
CASE <expression>
WHEN <constant1> THEN
    --Statements
WHEN <constant2> THEN
    --Statements
.
.
ELSE
    --Statements
END CASE;
```

Example on Simple CASE:

Program to check whether the given number is EVEN or ODD:

even	2,4,6,8, ...	divide with 2	remainder 0
odd	1,3,5,7, ...	divide with 2	remainder 1

```
DECLARE
    n INT;
BEGIN
```

```
        n := &n;

        CASE mod(n,2)
            WHEN 0 THEN
                dbms_output.put_line('EVEN');
            WHEN 1 THEN
                dbms_output.put_line('ODD');
        END CASE;

    END;
    /
```

Assignment:

Assignment:

Program to increase salary of given empno based on deptno as following:

```
if deptno 10 => increase 10% on sal
                20                    20%
                30                    15%
                others                 5%
write it with simple CASE
```

Searched CASE:

Syntax:

```
CASE
WHEN <condition1> THEN
    --Statements
WHEN <condition2> THEN
    --Statements
.
.
ELSE
    --statements
END CASE;
```

```
    --statements
 END CASE;
```

Program to check whether the given number is +ve or -ve or zero:

1,2,3,4,5,.....	+ve	>0
-1,-2,-3, ...	-ve	<0

```
DECLARE
   n INT;
BEGIN
   n := &n;

   CASE
   WHEN n>0 THEN
      dbms_output.put_line('+VE');
   WHEN n<0 THEN
      dbms_output.put_line('-VE');
   ELSE
      dbms_output.put_line('ZERO');
   END CASE;

END;
/
```

Looping Control Structures:

Looping Control Structure is execute the statements repeatedly.

PL/SQL provides 3 Looping Control Structures:

- **WHILE**
- **SIMPLE LOOP**
- **FOR**

WHILE:

Syntax:

```
WHILE <condition>
LOOP
    --Statements
END LOOP;
```

The statements in WHILE get executed as long as the condition is TRUE.
When the condition is FALSE, it terminates the LOOP.

Example on WHILE:

Program to print numbers from 1 to 4:

```
i
1
2
3
4
```

```
i:=1;

dbms_output.put_line(i);  --1
i:=i+1;   --i=2

dbms_output.put_line(i);  --2
i:=i+1;  --i=3

dbms_output.put_line(i); --3
i:=i+1;    --i=4

dbms_output.put_line(i); --4
```

```
i
1 2  3  4
```

```
i:=1
WHILE i<=4
LOOP
    dbms_output.put_line(i);
    i:=i+1;
END LOOP;
```

Program:

```
DECLARE
    i INT;
BEGIN
    i:=1;

    WHILE i<=4
    LOOP
        dbms_output.put_line(i);
        i:=i+1;
    END LOOP;
END;
/
```

Simple Loop:

Syntax:

```
LOOP
    --Statements
    EXIT WHEN <condition>;    /   EXIT;
END LOOP;
```

Example on Simple Loop:

Program to print numbers from 1 to 4:

```
 i

 1
 2
 3
 4
```

```
DECLARE
    i INT;
BEGIN
    i:=1;

    LOOP
        dbms_output.put_line(i);
        EXIT WHEN i=4;
        i:=i+1;
    END LOOP;
END;
/
```

```
EXIT WHEN i=4;
```

=

```
IF i=4 THEN
    EXIT;
END IF;
```

EXIT WHEN:

- It is a jumping control structure.
- It is used to terminate the loop in the middle of execution.
- It can be used in LOOP only.

EXIT:

- It is a jumping control structure.
- It is used to terminate the loop in the middle of execution.
- It can be used in LOOP only.

```
BEGIN
    dbms_output.put_line('HI');
    EXIT;
    dbms_output.put_line('BYE');
END;
/
```

Output:

```
ERROR: EXIT can be used inside of a loop only
```

For:

Syntax:

```
FOR <variable> IN [REVERSE] <lower> .. <upper>
LOOP
    --Statements
END LOOP;
```

Example on FOR loop:

Program to print numbers from 1 to 4:

i

```
1
2
3
4
```

```
BEGIN
   FOR i IN 1 .. 4
   LOOP
      dbms_output.put_line(i);
   END LOOP;
END;
/
```

Note:

In FOR LOOP,

- **we have no need to declare loop variable.**
  implicitly it will be declared as NUMBER type.

- **Loop variable is read-only variable.**

Example:

```
BEGIN
   FOR i IN 1 .. 20
   LOOP
      i:=10;
      dbms_output.put_line(i);
   END LOOP;
END;
/
```

Output:

ERROR: i cannot be used as assignment target

i => read-only variable

- **loop variable scope is limited to loop only.**

Example:

```
BEGIN
   FOR i IN 1..10
   LOOP
      dbms_output.put_line(i);
   END LOOP;

    dbms_output.put_line(i);
END;
```

```
/
```

Output:

ERROR: i must be declared

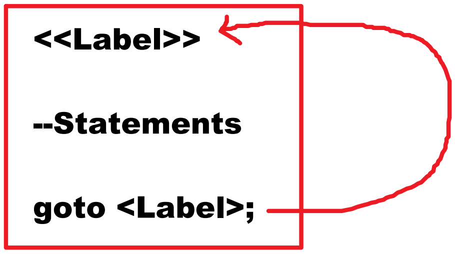
Program to print numbers from 4 to 1:

```
BEGIN
   FOR i IN REVERSE 1 .. 4
   LOOP
      dbms_output.put_line(i);
   END LOOP;
END;
/
```

GOTO:

When GOTO statement is executed execution jumps to specified label.

Syntax:



```
<<Label>>

--Statements

goto <Label>;
```

Example on GOTO:

Program to print numbers from 1 to 4:

```
i
1
2
3
4
```

```
DECLARE
   i INT;
BEGIN
   i:=1;

   <<abc>>

      dbms_output.put_line(i);
      i:=i+1;

   IF i<=4 THEN
      goto abc;
   END IF;

END;
/
```

Continue:

- **It can be used in LOOP only.**
- **It is used to skip current iteration and continue the next iteration.**

Example:

Program to print numbers from 1 to 10 except 7:

```
BEGIN
   FOR i IN 1 .. 10
   LOOP
      IF i=7 THEN
         CONTINUE;
      END IF;
      dbms_output.put_line(i);
   END LOOP;
END;
/
```

```
1
2
3
4
5
6
8
9
10
```

CURSORS

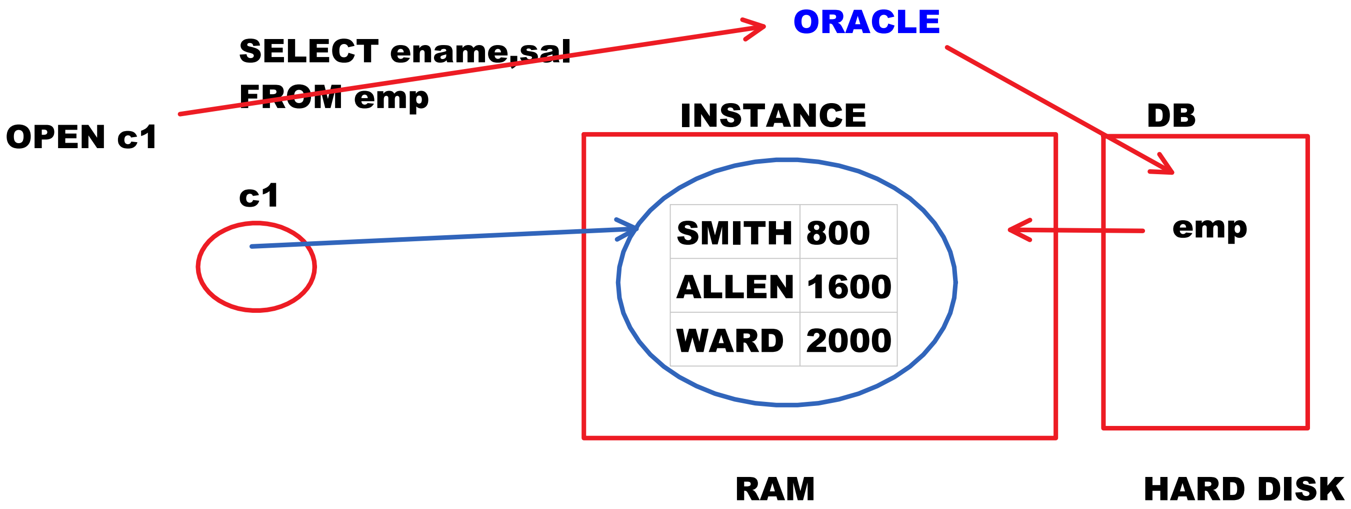
Thursday, April 25, 2024 9:43 AM

GOAL:

CURSOR is used to hold multiple rows and process them one by one.

to hold 1 column value we use %TYPE
to hold 1 row we use %ROW TYPE
to hold multiple rows we use CURSOR

Note:
Every CURSOR is associated with SELECT query.



SMITH	800
ALLEN	1600
WARD	2000

CURSOR:

- **CURSOR is a POINTER to memory location which is in INSTANCE [RAM]. This memory location has multiple rows.**

- **CURSOR is used to hold multiple rows and process them one by one.**

- **To execute any DRL or DML command CURSOR is**

required.

Steps to use CURSOR:

4 steps:

- **DECLARE**
- **OPEN**
- **FETCH**
- **CLOSE**

DECLARING CURSOR:

Syntax:

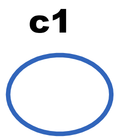
```
CURSOR <cursor_name> IS <SELECT query>;
```

Example:

```
CURSOR c1 IS SELECT ename,sal FROM emp;
```

When we declare the cursor,

- **Cursor variable will be created**
- **SELECT query will be identified**



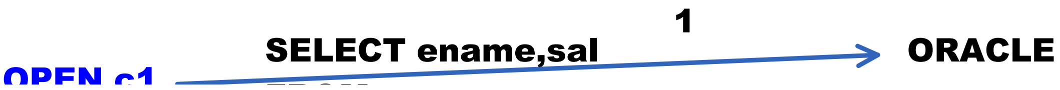
OPENING CURSOR:

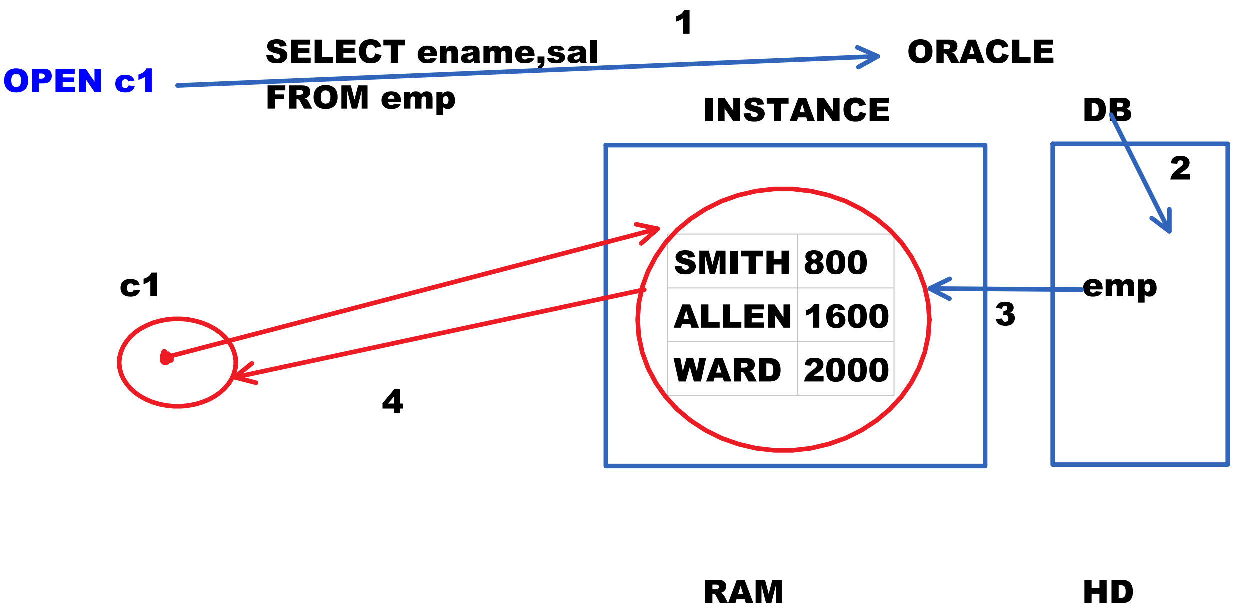
Syntax:

```
OPEN <cursor_name>;
```

Example:

```
OPEN c1;
```





SMITH	800
ALLEN	1600
WARD	2000

When CURSOR is opened,
1. SELECT query will be submitted to ORACLE
2. ORACLE goes to DB.
3. Selects the data and copies into INSTANCE.
4. This memory location address will be given to CURSOR

Now CURSOR has that memory location address.
Now CURSOR has multiple rows.

FETCHING RECORDS FROM CURSOR:

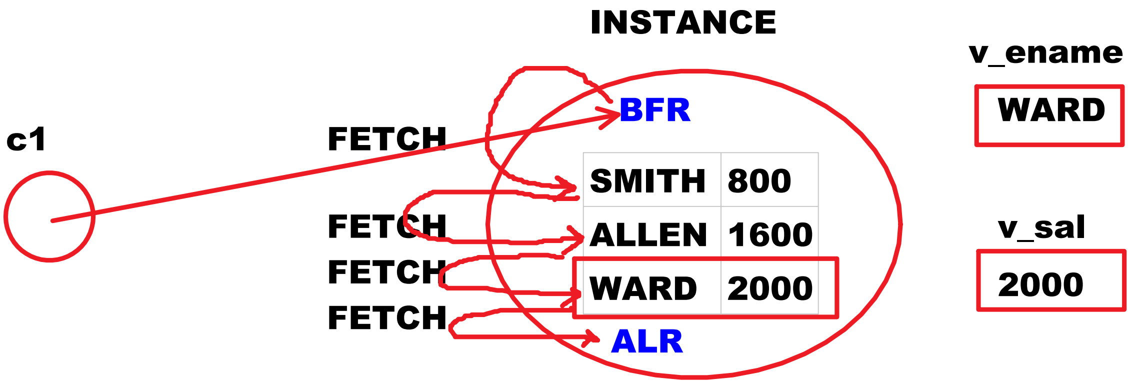
Syntax:

```
FETCH <cursor_name> INTO <variables_list>;
```

Example:

```
FETCH c1 INTO v_ename, v_sal;
```

- **When FETCH statement is executed it fetches next row.**
- **One FETCH statement can FETCH 1 row only.**
- **To FETCH multiple rows write FETCH statement in LOOP.**



BFR => Before First Row
ALR => After Last Row

When FETCH
is unsuccessful
stop FETCHING

CLOSING CURSOR:

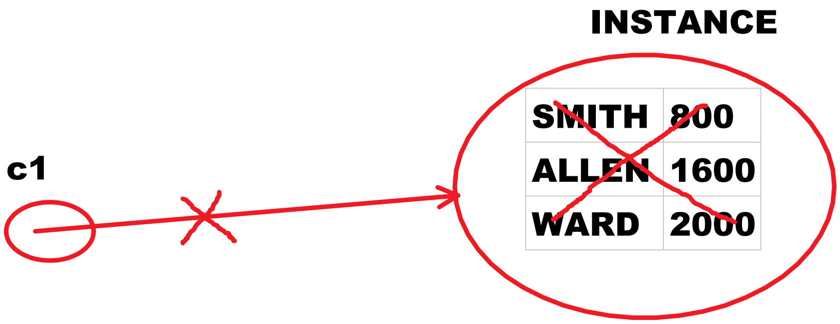
Syntax:

```
CLOSE <cursor_name>;
```

Example:

```
CLOSE c1;
```

When CURSOR is CLOSE, memory will be cleared and reference to memory location will be gone.



CURSOR ATTRIBUTES:

%FOUND

%NOTFOUND
%ROWCOUNT
%ISOPEN

Syntax:

```
<cursor_name><attribute_name>
```

Example:

```
c1%FOUND
c1%NOTFOUND
c1%ROWCOUNT
c1%ISOPEN
```

%FOUND:

- It holds boolean value [true or false].
- If fetch is successful then it holds TRUE
- If fetch is unsuccessful then it holds FALSE

%NOTFOUND:

- It holds boolean value [true or false].
- If fetch is successful then it holds FALSE
- If fetch is unsuccessful then it holds TRUE

%ROWCOUNT:

- It's default value is 0
- If fetch is successful, ROWCOUNT value will be incremented by 1.

%ISOPEN:

- If cursor is opened then it holds TRUE.
- If cursor is not opened then it holds FALSE.

Write a program to print all emp records:

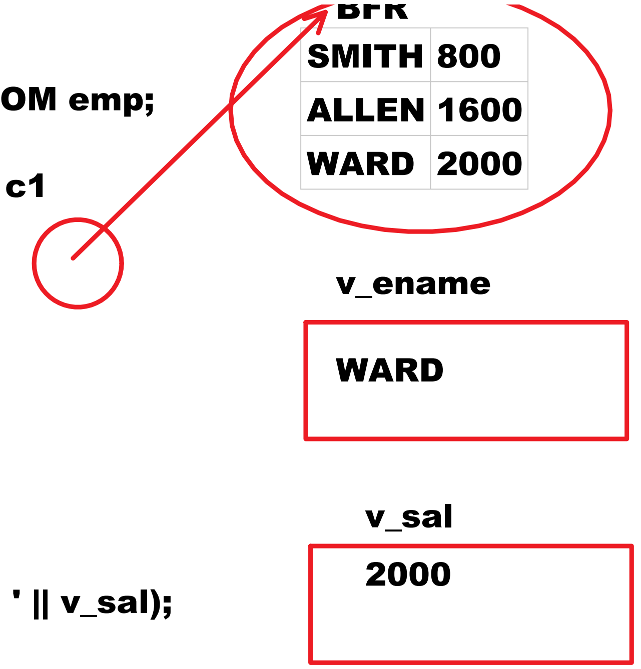
```
DECLARE
   CURSOR c1 IS SELECT ename,sal FROM emp;
```

INSTANCE

BFR

SMITH	800
ALLEN	1600

```
DECLARE
   CURSOR c1 IS SELECT ename,sal FROM emp;
   v_ename EMP.ENAME%TYPE;
   v_sal EMP.SAL%TYPE;
BEGIN
   OPEN c1;

   LOOP
      FETCH c1 INTO v_ename, v_sal;

      EXIT WHEN c1%NOTFOUND;

      dbms_output.put_line(v_ename || '   ' || v_sal);
   END LOOP;

   CLOSE c1;
END;
/
```



SMITH	800
ALLEN	1600
WARD	2000

Program to increase salary of all emps according to hike table percentages:

EMPLOYEE

EMPNO	ENAME	SAL
1001	A	5000
1002	B	3000
1003	C	7000

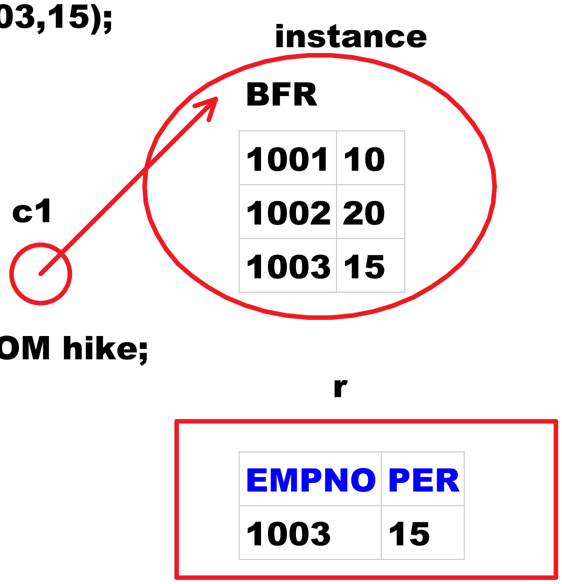
HIKE

EMPNO	PER
1001	10
1002	20
1003	15

```
create table employee
(
empno NUMBER(4),
ename VARCHAR2(10),
sal NUMBER(8,2)
);

INSERT INTO employee VALUES(1001,'A',5000);
INSERT INTO employee VALUES(1002,'B',3000);
INSERT INTO employee VALUES(1003,'C',7000);
```

```
COMMIT;

create table hike
(
empno NUMBER(4),
per NUMBER(2)
);

INSERT INTO hike VALUES(1001,10);
INSERT INTO hike VALUES(1002,20);
INSERT INTO hike VALUES(1003,15);
COMMIT;
```

Program:



1001	10
1002	20
1003	15

EMPNO	PER
1003	15

```
DECLARE
   CURSOR c1 IS SELECT * FROM hike;
   r HIKE%ROWTYPE;
BEGIN
   OPEN c1;

   LOOP
      FETCH c1 INTO r;

      EXIT WHEN c1%notfound;

      UPDATE employee SET sal=sal+sal*r.per/100
      WHERE empno=r.empno;
   END LOOP;

   COMMIT;
   dbms_output.put_line(c1%rowcount || ' rows updated..');

   CLOSE c1;
END;
/
```

Program to calculate total, avrg and result of all students and insert those values into RESULT table:

STUDENT

SID	SNAME	M1	M2	M3
1001	A	70	90	80
1002	B	60	30	75

RESULT

sid	total	avrg	result

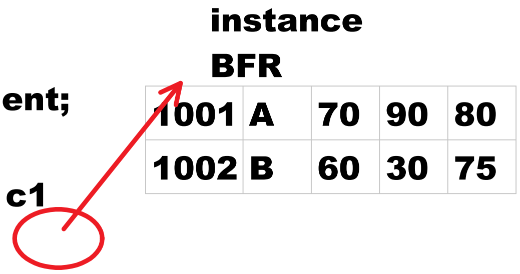
```
CREATE TABLE student
(
sid NUMBER(4),
sname VARCHAR2(10),
m1 NUMBER(3),
m2 NUMBER(3),
m3 NUMBER(3)
);

INSERT INTO student VALUES(1001,'A',70,90,80);
INSERT INTO student VALUES(1002,'B',60,30,75);
COMMIT;


CREATE TABLE result
(
sid NUMBER(4),
total NUMBER(3),
avrg NUMBER(5,2),
result VARCHAR2(10)
);
```

Program:

instance
BFR

```
 DECLARE
    CURSOR c1 IS SELECT * FROM student;
```

```
DECLARE
    CURSOR c1 IS SELECT * FROM student;
    r1 STUDENT%ROWTYPE;
    r2 RESULT%ROWTYPE;
BEGIN
    OPEN c1;

    LOOP
        FETCH c1 INTO r1;

        EXIT WHEN c1%NOTFOUND;

        r2.total:=r1.m1+r1.m2+r1.m3;
        r2.avrg:=r2.total/3;

        IF r1.m1>=40 AND r1.m2>=40 AND r1.m3>=40
        THEN
            r2.result:='PASS';
        ELSE
            r2.result:='FAIL';
        END IF;

        INSERT INTO result VALUES(r1.sid, r2.total,
        r2.avrg, r2.result);
    END LOOP;

    COMMIT;

    dbms_output.put_line('result stored in RESULT
    table..');

    CLOSE c1;
END;
/
```



1001	A	70	90	80
1002	B	60	30	75

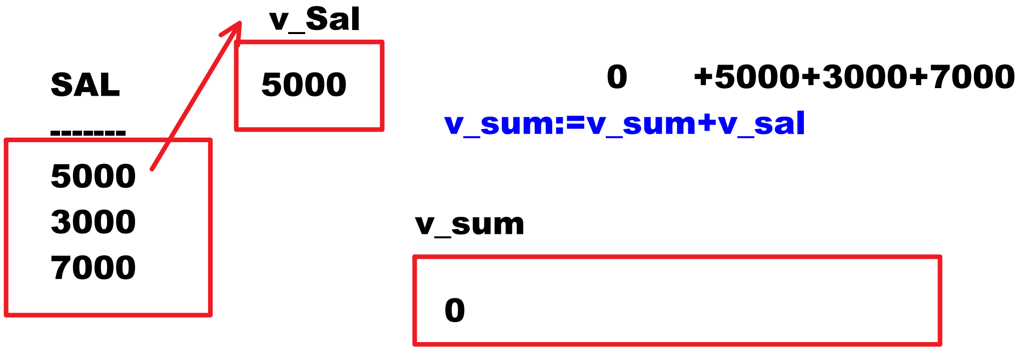
r1

SID	SNAME	M1	M2	M3
1001	A	70	90	80

r2

sid	total	avrg	result
	240	80	PASS

Program to find sum of salaries of all emps using CURSOR:



SAL

5000

3000

7000

v_Sal

5000

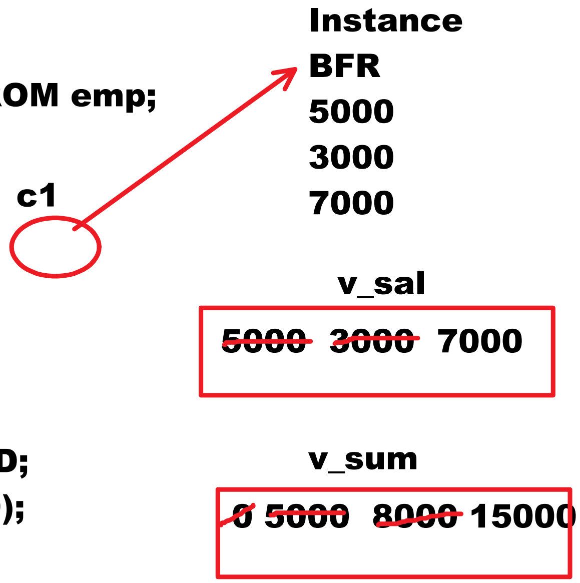
0 +5000+3000+7000

v_sum:=v_sum+v_sal

v_sum

0

```
DECLARE
   CURSOR c1 IS SELECT sal FROM emp;
   v_sal EMP.SAL%TYPE;
   v_sum NUMBER:=0;
BEGIN
   OPEN c1;

   LOOP
      FETCH c1 INTO v_sal;
      EXIT WHEN c1%NOTFOUND;
      v_sum:=v_sum+NVL(v_sal,0);
   END LOOP;

   dbms_output.put_line('sum of salaries=' || v_sum);

   CLOSE c1;
END;
/
```



c1

Instance

BFR

5000

3000

7000

v_sal

~~5000~~ ~~3000~~ 7000

v_sum

~~0~~ ~~5000~~ ~~8000~~ 15000

Cursor For Loop:

```
FOR <variable> IN <cursor_name>
LOOP
   --Statements
END LOOP;
```

- **If we use CURSOR FOR LOOP, We have no need to open, fetch and close the cursor. All these 3 actions will be done implicitly.**
- **We have no need to declare CURSOR FOR LOOP variable. Implicitly it will be declared as %ROWTYPE variable.**

Example on Cursor For Loop:

Display all emp records:

```
DECLARE
   CURSOR c1 IS SELECT * FROM emp;
BEGIN
   FOR r IN c1
   LOOP
      dbms_output.put_line(r.ename || '    ' || r.sal);
   END LOOP;
END;
/
```

Inline Cursor:

- **If SELECT QUERY is specified in CURSOR FOR LOOP then it is called "Inline Cursor".**

Syntax:

```
FOR <variable> IN (<SELECT QUERY>)
LOOP
   --Statements
END LOOP;
```

Example on Inline Cursor:

Display all emp records:

```
BEGIN
   FOR r IN (SELECT * FROM emp)
   LOOP
      dbms_output.put_line(r.ename || '    ' || r.sal);
   END LOOP;
END;
/
```

Display all emp records using WHILE loop:

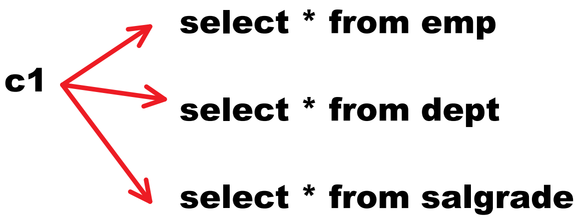
```
 DECLARE
    CURSOR c1 IS SELECT * FROM emp;
    r EMP%ROWTYPE;
 BEGIN
    OPEN c1;

    FETCH c1 INTO r;

    WHILE c1%found
    LOOP
       dbms_output.put_line(r.ename || '    ' || r.sal);
       FETCH c1 INTO r;
    END LOOP;

    CLOSE c1;
 END;
 /
```

Ref Cursor:

Simple Cursor:

```
CURSOR c1 IS SELECT * FROM emp;
CURSOR c2 IS SELECT * FROM dept;
CURSOR c3 IS SELECT * FROM salgrade;
```

Ref Cursor:



c1

select * from emp

select * from dept

select * from salgrade

- **in Simple Cursor, one cursor can be used for one SELECT QUERY only. Here Select query is fixed. It cannot be changed.**
  WHERE AS
  in Ref Cursor, same cursor can be used for multiple select queries. Here, select query is not fixed, it can be changed.

- **It has data type. i.e: SYS_REFCURSOR**

- **It can be used as procedure parameter.**

DECLARING REF CURSOR:

Syntax:

```
<cursor_name> SYS_REFCURSOR;
```

Example:

```
c1 SYS_REFCURSOR;
```

OPENING REF CURSOR:

Syntax:

```
OPEN <cursor_name> FOR <select query>;
```

Examples:

```
OPEN c1 FOR SELECT * FROM emp;
.
.
OPEN c1 FOR SELECT * FROM dept;
```

Example on REF CURSOR:

Display all emp table records. then display all dept table records:

```
DECLARE
   c1 SYS_REFCURSOR;
   r1 EMP%ROWTYPE;
   r2 DEPT%ROWTYPE;
BEGIN
   OPEN c1 FOR SELECT * FROM emp;

   LOOP
      FETCH c1 INTO r1;
      EXIT WHEN c1%NOT FOUND;
      dbms_output.put_line(r1.ename || '    ' || r1.sal);
   END LOOP;

   CLOSE c1;

   OPEN c1 FOR SELECT * FROM dept;

   LOOP
      FETCH c1 INTO r2;
      EXIT WHEN c1%notfound;
      dbms_output.put_line(r2.deptno ||  '     ' || r2.dname);
   END LOOP;

   CLOSE c1;
END;
```

```
/
```

Differences between Simple Cursor and Ref Cursor:

Simple Cursor	Ref Cursor
• One cursor can be used for one select query only.	• Same cursor can be used for multiple select queries.
• it is static [fixed].	• It is dynamic [can be changed]
• It has no data type	• it has data type. i.e: sys_refcursor
• it cannot used as procedure parameter. because, it has no data type.	• it can be used as procedure parameter. because, it has data type.
• In simple cursor, select query will be specified at the time of declaring cursor.	• In Ref Cursor, select query will be specified at the time of opening cursor.

Parameterized Cursor:

- A cursor which is declared using parameter is called "Parameterized Cursor".
- At the time of opening cursor we pass value to parameter.

Example:

```
CURSOR c1(n NUMBER) IS SELECT * FROM emp WHERE deptno=n;

                                                     n=30

OPEN c1(30)
```

Example on Parameterized Cursor:

Display specific dept emp records:

```
DECLARE
   CURSOR c1(n NUMBER) IS SELECT * FROM emp
   WHERE deptno=n;
   r EMp%ROWTYPE;
BEGIN
   OPEN c1(30);

   LOOP
      FETCH c1 INTO r;
      EXIT WHEN c1%notfound;
      dbms_output.put_line(r.ename || '   ' || r.sal || '   ' || r.deptno);
   END LOOP;

   CLOSE c1;
END;
/
```

Types of Cursors:

2 types:

- Implicit Cursor
- Explicit Cursor
  - simple cursor
  - ref cursor

Implicit Cursor:

- To execute any DRL or DML command implicitly ORACLE uses a cursor. This is called "Implicit Cursor".

- this implicit cursor name is: SQL

- To use cursor attributes on implicit cursor we can write as following:
  SQL%FOUND
  SQL%NOTFOUND
  SQL%ROWCOUNT
  SQL%ISOPEN

Explicit Cursor:
A cursor which is defined by user is called "Explicit Cursor".

Example on Implicit Cursor:

Program to increase 1000 rupees salary to all emps:

```
BEGIN
   UPDATE emp SET sal=sal+1000;
   dbms_output.put_line(SQL%ROWCOUNT || ' rows updated..');
   COMMIT;
END;
/
```

Program to increase salary of given empno with given amount:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_amount NUMBER;
BEGIN
   v_empno:=&empno;
   v_amount:=&amount;

   UPDATE emp SET sal=sal+v_amount
   WHERE empno=v_empno;

   IF sql%notfound THEN
      dbms_output.put_line('emp not existed');
   ELSE
      COMMIT;
      dbms_output.put_line('sal increased...');
   END IF;
END;
/
```

Output-1:

```
Enter value for empno: 7369
Enter value for amount: 1000
sal increased...

Output-2:
Enter value for empno: 9001
Enter value for amount: 2000
emp not existed
```

CURSOR:

CURSOR is a pointer to memory location which is in instance.

To hold multiple rows and process them one by one we use CURSOR.

to use CURSOR follow 4 steps:

- **declare**
- **open**
- **fetch**
- **close**

cursor for loop:

no need to open, fetch, close

inline cursor:

if select query specified in cursor for loop

parameterized cursor:

cursor with parameter

ref cursor:

same cursor can be used for multiple select queries.

types of cursors:

2 types:

- **implicit cursor**

- **explicit cursor**
    - **simple cursor**
    - **ref cursor**

Exception Handling

Monday, April 29, 2024 9:20 AM

Exception	**Run Time Error => problem: abnormal termination**
Exception handling	**The way of handling run time errors => Solution**

Types of Errors:

3 types:

- **Compile Time Errors**
- **Run Time Errors**
- **Logical Errors**

Compile time Errors:

- **These errors occur at compile time.**
- **These errors occur due to syntax mistakes.**

Examples:

missing ;
missing end if
missing end loop
missing '
missing)

Run Time Errors:

- **These errors occur during program execution.**
- **These errors occur due to many reasons like:**
  - **when we try to divide with 0**
  - **when record is not found**
  - **when we insert duplicate value or null in PK**
  - **when check constraint violated**
  - **wrong input**
  - **when size is exceeded**
  - **when we fetch for record without opening cursor**

```
100 lines

...............
.................
...............
20/0                50th line
................
....................
```

Logical Errors:

- **With Logical Error, we get wrong results.**
- **programmer must write correct logic.**

  Example:
  withdraw => balance:=balance+amount

Exception:

- **Exception means Run Time Error.**
- **It is a problem.**
- **When Run Time Error occurs our application will be closed in the middle of execution. So, abnormal termination will occur due to run time error.**
- **With this user cannot continue his work.**
- **We may loss the data when transaction is stopped in middle of execution.**
- **We may get wrong results.**

That's why we must handle run time errors.

Exception Handling:

- **The way of handling Run Time Errors is called "Exception Handling".**
- **For Exception Handling add "EXCEPTION" block**

and define handling code in it.

- **With Exception handling abnormal termination will be converted to normal termination.**

Syntax of exception Handling:

```
DECLARE
     --declare the variables
BEGIN
    --executable statements

    EXCEPTION
        WHEN <exception_name> THEN
            --Handling Code
        WHEN <exception_name> THEN
            --Handling Code
        .
        .
END;
/
```

Program on Exception Handling:

Program to divide 2 numbers:

```
DECLARE
    x NUMBER(4);
    y NUMBER(4);
    z NUMBER(4);
BEGIN
    x := &x;
    y := &y;

    z := x/y;

    dbms_output.put_line('z=' || z);
```

```
   EXCEPTION
      WHEN zero_divide THEN
         dbms_output.put_line('cannot divide with zero');
      WHEN value_error THEN
         dbms_output.put_line('wrong input or size is exceeded');
      WHEN others THEN
         dbms_output.put_line('something went wrong..');
END;
/
```

```
Output-1:
Enter value for x: 20
Enter value for y: 2
z=10

Output-2:
Enter value for x: 20
Enter value for y: 0
cannot divide with zero

Output-3:
Enter value for x: 123456
Enter value for y: 2
wrong input or size is exceeded


Output-4:
Enter value for x: 'raju'
Enter value for y: 2
wrong input or size is exceeded
```

Types of Exceptions:

2 types:

- **Built-In Exception**
- **User-Defined Exception**

Built-In Exception:

- **The exception which is already defined by ORACLE DEVELOPERS is called "Built-In Exception".**
- **It will be raised implicitly by ORACLE.**

Examples:

- **zero_divide**
- **value_error**
- **no_data_found**
- **dup_val_on_index**
- **too_many_rows**
- **invalid_cursor**
- **cursor_aready_open**

zero_divide:
when we try to divide with 0 then zero_divide exception will be raised.

value_error:
when we give wrong input or when size is exceeeded then value_error exception will be raised.

no_data_found:
when record is not found in table then no_data_found exception will be raised.

Example on no_data_found:

Program to display emp record of given empno:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   r EMP%ROWTYPE;
BEGIN
   v_empno := &empno;

   SELECT * INTO r FROM emp WHERE
   empno=v_empno;

   dbms_output.put_line(r.ename || '   ' || r.sal);

   EXCEPTION
      WHEN no_data_found THEN
         dbms_output.put_line('no employee existed
         with this empno..');
END;
/
```

```
Output-1:
Enter value for empno: 7934
MILLER   6174.5
```

```
Output-2:
Enter value for empno: 1234
no employee existed with this empno..
```

Dup_Val_On_Index:
When we try to insert duplicate value in PRIMARY KEY then dup_val_on_index exception will be raised.

Example on dup_val_on_index:

Program to insert student record into student table:

```
CREATE TABLE student
(
sid NUMBER(4) PRIMARY KEY,
sname VARCHAR2(10)
);

BEGIN
   INSERT INTO student VALUES(&sid, '&sname');
   COMMIT;
   dbms_output.put_line('record inserted..');

   EXCEPTION
      WHEN dup_val_on_index THEN
         dbms_output.put_line('sid must be unique..');
END;
/
```

```
Output-1:
Enter value for sid: 5
Enter value for sname: E
record inserted..
```

```
Output-2:
Enter value for sid: 5
Enter value for sname: F
sid must be unique..
```

Too_Many_Rows:
If SELECT QUERY selects multiple rows then Too_Many_Rows Exception will be raised.

Example on Too_many_Rows:

Program to display emp records based on given job:

```
DECLARE
   v_job EMP.JOB%TYPE;
   r EMP%ROWTYPE;
BEGIN
   v_job := '&job';

   SELECT * INTO r FROM emp WHERE job=v_job;

   dbms_output.put_line(r.ename || '   ' || r.job || '    ' || r.sal);

   EXCEPTION
      WHEN too_many_rows THEN
         dbms_output.put_line('many emps are there..');
END;
/
```

```
output-1:
Enter value for job: PRESIDENT
KING   PRESIDENT    7000


output-2:
Enter value for job: CLERK
many emps are there..
```

Invalid_Cursor:
When we try to fetch for the record without opening cursor then Invalid_Cursor Exception will be raised.

Example on Invalid_Cursor:

Display all emp records:

```
DECLARE
   CURSOR c1 IS SELECT * FROM emp;
   r EMP%ROWTYPE;
BEGIN
   LOOP
      FETCH c1 INTO r;

      EXIT WHEN c1%notfound;

      dbms_output.put_line(r.ename || '    ' || r.sal);
   END LOOP;

   CLOSE c1;

   EXCEPTION
      WHEN invalid_cursor THEN
         dbms_output.put_line('cursor not opened..');
END;
/
```

output:

```
cursor not opened..
```

Cursor_Already_Open:
If we try to open opened cursor then Cursor_Already_Open exception will be raised.

Example on Cursor_Already_Open:

```
DECLARE
```

```
    CURSOR c1 IS SELECT * FROM emp;
    r EMP%ROWTYPE;
BEGIN
    OPEN c1;

    OPEN c1;

    LOOP
        FETCH c1 INTO r;

        EXIT WHEN c1%notfound;

        dbms_output.put_line(r.ename || '    ' || r.sal);
    END LOOP;

    CLOSE c1;

    EXCEPTION
        WHEN cursor_already_open THEN
            dbms_output.put_line('cursor already opened..');
END;
/
```

Output:

```
cursor already opened..
```

User-Defined Exception:

- We can define our own exceptions. these are called "User-Defined Exceptions".

- It will be raised explicitly.

Examples:

one_divide
sunday_not_allow
xyz

Built-In Exception:

follow 1 step:

- **Handle**

User-Defined Exception:

follow 3 steps:

- **Declare Exception**
- **Raise the Exception**
- **Handle the Exception**

Declaring exception:

Syntax:

```
<exception_name> EXCEPTION;
```

Examples:

one_divide EXCEPTION;
xyz EXCEPTION;

EXCEPTION is the data type.
It is used to declare the exception names.

Raising Exception:

Syntax:

```
RAISE <exception_name>;
```

Examples:

RAISE one_divide;
RAISE xyz;

RAISE is the keyword.
It is used to raise the exception explicitly.

Example on User-Defined exception:

Program to divide 2 numbers.
if denominator is 0, ORACLE raises run time error. handle it.
if denominator is 1, raise the exception and handle it.

```
DECLARE
    x NUMBER(4);
    y NUMBER(4);
    z NUMBER(4);
    one_divide EXCEPTION;
BEGIN
    x:=&x;
    y:=&y;

    IF y=1 THEN
        RAISE one_divide;
    END IF;

    z:=x/y;

    dbms_output.put_line('z=' || z);

    EXCEPTION
        WHEN zero_divide THEN
            dbms_output.put_line('you cannot divide with 0');
        WHEN one_divide THEN
            dbms_output.put_line('do not enter y as 1');
END;
/
```

```
Output:
Enter value for x: 10
Enter value for y: 1
do not enter y as 1
```

Program to increase salary of given empno with given amount. If Sunday,

raise the exception and handle it:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_amount NUMBER;
   sunday_not_allow EXCEPTION;
BEGIN
   v_empno:=&empno;
   v_amount:=&amount;

   IF to_char(sysdate,'dy')='sun' THEN
      RAISE sunday_not_allow;
   END IF;


   UPDATE emp SET sal=sal+v_amount
   WHERE empno=v_empno;

   COMMIT;

   dbms_output.put_line('sal increased..');

   EXCEPTION
      WHEN sunday_not_allow THEN
         dbms_output.put_line('you cannot increase sal on sunday');
END;
/
```

Note:

- **We can raise the error using 2 ways. they are:**
  - **Using RAISE keyword**
  - **Using RAISE_APPLICATION_ERROR() procedure**

RAISE_APPLICATION_ERROR():

- **it is a procedure.**
- **it is used to raise the run time error explicitly.**
- **Using it, we can raise the error with our own code and message.**

Syntax:

Raise_Application_Error(<user_defined error_code>, <error_message>)

<user_defined error_code>	**valid range: -20000 to -20999**

Example:

```
Raise_Application_Error(-20050,'cannot divide with 1');
```

Output:

```
ORA-20050: cannot divide with 1
```

Example on RAISE_APPLICATION_ERROR():

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_amount NUMBER;
   sunday_not_allow EXCEPTION;
BEGIN
   v_empno:=&empno;
   v_amount:=&amount;

   IF to_char(sysdate,'dy')='sun' THEN
      RAISE_APPLICATION_ERROR(-20050,'you cannot update on sunday');
   END IF;


   UPDATE emp SET sal=sal+v_amount
   WHERE empno=v_empno;

   COMMIT;

   dbms_output.put_line('sal increased..');

END;
/
```

Output [from mon to sat]:

```
Enter value for empno: 7934
Enter value for amount: 1000
sal increased..
```

```
Output [on Sunday]:
Enter value for empno: 7934
Enter value for amount: 1000
ERROR at line 1:
ORA-20050: you cannot update on sunday
```

Example:

```
DECLARE
     x NUMBER(4);
     y NUMBER(4);
     z NUMBER(4);
BEGIN
    x:=&x;
    y:=&y;

    IF y=1 THEN
       Raise_Application_Error(-20070,'denominator cannot be 1');
    END IF;

    z:=x/y;

    dbms_output.put_line('z=' || z);
END;
/
```

```
Output:
Enter value for x: 10
Enter value for y: 1
ERROR at line 1:
ORA-20070: denominator cannot be 1
```

Differences b/w RAISE and RAISE_APPLICATION_ERROR():

RAISE	RAISE_APPLICATION_ERROR()
• it is a keyword	• it is a procedure
• it raises the exception using name	• it raises the exception using error code.

PRAGMA EXCEPTION_INIT():

ERROR CODE	**-1476**
ERROR MESSAGE	**divisor is equal to zero**
ERROR NAME	**zero_divide**

ERROR CODE	**-1**
ERROR MESSAGE	**unique constraint violated**
ERROR NAME	**dup_val_on_index**

ERROR CODE	**-2290**
ERROR MESSAGE	**check constraint violated**
ERROR NAME	- **[no error name]**

ERROR CODE	**-1400**
ERROR MESSAGE	**cannot insert NULL**
ERROR NAME	- **[no error name]**

Syntax:

```
pragma exception_init(<user_defined_exception_name>,
<built-in error_code>)
```

Example:

```
   check_violate EXCEPTION;
   pragma exception_init(check_violate, -2290);

   cannot_insert_null EXCEPTION;
   pragma exception_init(cannot_insert_null, -1400);
```

- **Some errors have names.**
  Some errors does not have names.

- **To handle Run Time Error in EXCEPTION block name is required.**

- **To define name for unnamed exception we use "PRAGMA EXCEPTION_INIT().**

- **It is compiler directive. It is command to compiler.**
- **directive => command / instruction**
- **it instructs that before compiling PL/SQL program first execute this line.**

Example on Pragma_Exception_Init():

Program to insert student record into student table:

```
create table student
(
sid number(4) primary key,
sname varchar2(10),
m1 number(3) check(m1 between 0 and 100)
);
```

```
DECLARE
   check_violate EXCEPTION;
   PRAGMA EXCEPTION_INIT(check_violate, -2290);
BEGIN
```

```
    INSERT INTO student VALUES(&sid, '&sname', &m1);
    COMMIT;
    dbms_output.put_line('record inserted');

    EXCEPTION
       WHEN dup_val_on_index THEN
          dbms_output.put_line('PK does not accept dups');
       WHEN check_violate THEN
          dbms_output.put_line('marks must be b/w 0 to 100');
 END;
 /
```

exception	**Run Time Error** **problem: abnormal termination** **we may loss the data** **we may wrong results**
exception handling	**the way of handling run time errors** **add exception block to handle RTE**

Types of exceptions:

2 types:

built-in exception:
zero_divide
no_data_found
too_many_rows
dup_val_on_index
invalid_cursor
cursor_already_open

user-defined exception:

RAISE keyword
RAISE_APPLICATION_ERROR() procedure

pragma exception_init():

to define name to unnamed exception

STORED PROCEDURES

Wednesday, May 1, 2024 9:55 AM

Procedure:

- **Procedure is one ORACLE DB Object.**
- **Procedure is a named block of statements that gets executed on calling.**
- **Procedure can be also called as Sub Program.**

Types of Procedures:

2 types:

- **Stored Procedure**
- **Packaged Procedure**

Stored procedure:

- **if procedure is defined in SCHEMA then it is called "Stored procedure".**

Example:

```
SCHEMA c##batch9am
    PROCEDURE withdraw      => Stored procedure
```

Packaged procedure:

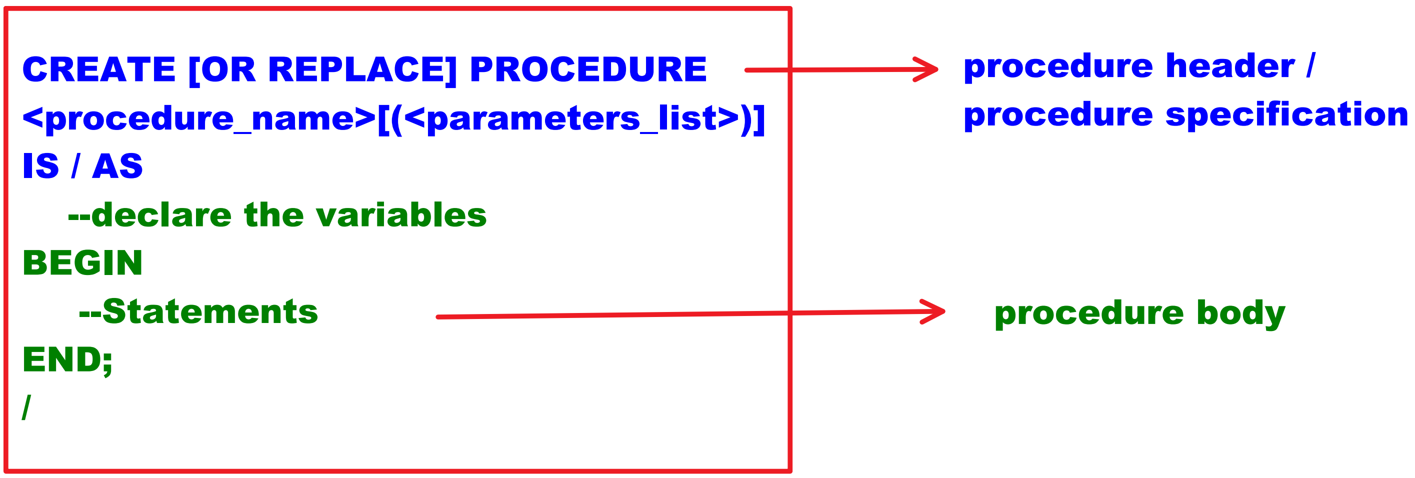
- **if procedure is defined in PACKAGE then it is called "Packaged procedure".**

Example:

```
SCHEMA c##batch9am
    PACKAGE bank
        PROCEDURE withdraw      => Packaged procedure
```

Syntax to define Stored procedure:

Syntax to define Stored procedure:



```
CREATE [OR REPLACE] PROCEDURE
<procedure_name>[(<parameters_list>)]
IS / AS
    --declare the variables
BEGIN
     --Statements
END;
/
```

procedure header / procedure specification

procedure body

Example on defining procedure:

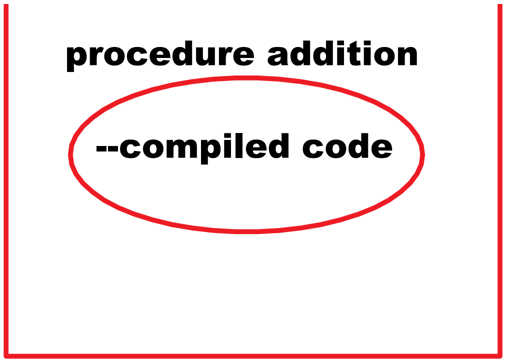
```
CREATE OR REPLACE PROCEDURE
addition(x NUMBER, y NUMBER)
AS
    z NUMBER(4);
BEGIN
    z:=x+y;
    dbms_output.put_line('sum=' || z);
END;
/
```

- write above code in text editor like notepad, edit plus.
- save it in d: drive, batch9am folder, with the name ProcedureDemo.sql

- open SQL PLUS
- log in as user

```
SQL> @d:\batch9am\ProcedureDemo.sql
Output:
procedure created.
```

Note:
when we call procedure
ORACLE runs compiled code.



Calling Stored Procedure:

3 ways:

- **From SQL prompt**
- **From PL/SQL program**
- **From Programming Languages [Java, C#, Python]**

Calling From SQL prompt:

- **To call procedure from SQL prompt we use EXEC[UTE] command.**

```
SQL> EXEC addition(5,4);
Output:
sum=9
```

Calling from PL/SQL program:

```
DECLARE
    a NUMBER(4);
    b NUMBER(4);
BEGIN
   a := &a;
   b := &b;

   addition(a,b);                --procedure call
END;
```

```
/
Output:
enter .. a: 5
enter .. b: 4
sum=9
```

Note:
to see errors of procedure write following command:
SQL> SHOW ERRORS

Parameter:

- **parameter is a local variable that is declared in procedure header.**

Syntax:

```
<parameter_name> [<parameter_mode>] <parameter_data_type>
```

Examples:

```
x IN NUMBER
y OUT NUMBER
z IN OUT NUMBER
```

Parameter Modes:

There are 3 parameter modes. They are:

- **IN**
- **OUT**
- **IN OUT**

IN:

- **it is default one.**
- **IN parameter captures input.**
- **It is used to bring value into procedure from out of procedure.**
- **It is read-only parameter.**

- ○ **In procedure call, it can be constant or variable.**

OUT:

- ○ **OUT parameters sends output.**
- ○ **It is used to send the result out of the procedure.**
- ○ **It is read-write parameter.**
- ○ **In procedure call, it must be variable only.**

IN OUT:

- ○ **Same parameter can take input and send output.**
- ○ **It is read-write parameter.**
- ○ **In procedure call, it must be variable only.**

Example on OUT parameter:

Define a procedure to add 2 numbers and send the result out of procedure:

```
CREATE OR REPLACE PROCEDURE
addition(x IN NUMBER, y IN NUMBER, z OUT NUMBER)
AS
BEGIN
    z := x+y;
END;
/
```

Calling from SQL prompt:

```
SQL> VARIABLE a NUMBER
SQL> EXEC addition(10,5,:a);
SQL> PRINT a
Output:
15
```

Note:

Bind variable:

- ▪ **The variable which is declared at SQL prompt is called**

"Bind variable".

- To write data into bind variable we use bind operator :
- To print bin variable data use "PRINT" command.

Syntax:

```
VAR[IABLE] <variable> <data_type>
```

Calling from PL/SQL program [main program]:

```
DECLARE
   a NUMBER(4);
   b NUMBER(4);
   c NUMBER(4);
BEGIN
   a := &a;
   b := &b;

   addition(a,b,c);

   dbms_output.put_line('sum=' || c);
END;
/
```

Define a procedure to increase salary of specific employee:

```
CREATE OR REPLACE PROCEDURE
update_salary(p_empno IN NUMBER, p_amount IN NUMBER)
AS
BEGIN
   UPDATE emp SET sal=sal+p_amount
   WHERE empno=p_empno;

   COMMIT;

   dbms_output.put_line('sal increased..');
END;
/
```

Calling:

```
EXEC update_salary(7369,1000);
```

Output:

```
sal increased..
```

Define a procedure to increase salary of specific employee. After increment send increased salary out of the procedure:

```
CREATE OR REPLACE PROCEDURE
update_Salary(p_empno IN NUMBER, p_amount IN NUMBER,
p_sal OUT NUMBER)
AS
BEGIN
    UPDATE emp SET sal=sal+p_amount
    WHERE empno=p_empno;

    COMMIT;

    dbms_output.put_line('sal increased..');

    SELECT sal INTO p_sal FROM emp WHERE empno=p_empno;
END;
/
```

Calling:

```
SQL> VAR s NUMBER
SQL> EXEC update_salary(7369,1000,:s);
sal increased..
SQL> PRINT s
```

```
CREATE OR REPLACE PROCEDURE
addition(x NUMBER, y NUMBER)
AS
```

→ x,y => formal parameters

x,y => formal parameters

```
AS
   z NUMBER(4);
BEGIN
   z:=x+y;
   dbms_output.put_line('sum=' || z);
END;
/
```

```
EXEC addition(2,3);
```

→ 2,3 => actual parameters

Formal parameter:
A parameter which is declared in procedure header is called "Formal parameter".

Actual parameter:
A parameter which is in procedure call is called "Actual parameter"

Parameter mapping techniques /
Parameter association techniques /
Parameter notations:

There are 3 parameter mapping techniques. They are:

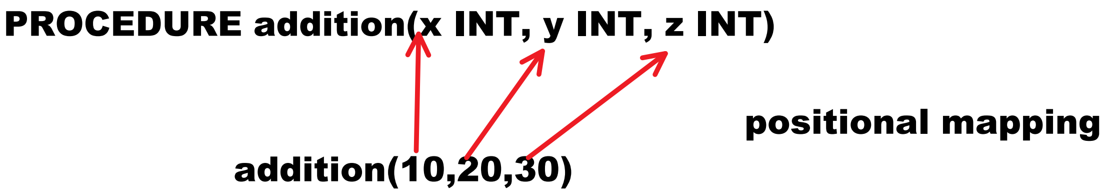
- **Positional mapping**
- **Named mapping**
- **Mixed mapping**

Positional mapping:
In positional mapping, actual parameters will be mapped with formal parameters based on positions.

Example:

```
PROCEDURE addition(x INT, y INT, z INT)
```

Example:

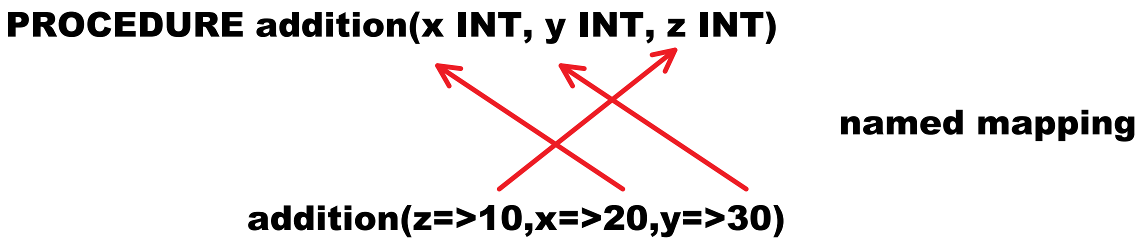


```
PROCEDURE addition(x INT, y INT, z INT)

addition(10,20,30)
```

positional mapping

named mapping:
In named mapping, actual parameters will be mapped with formal parameters based on names.

Example:

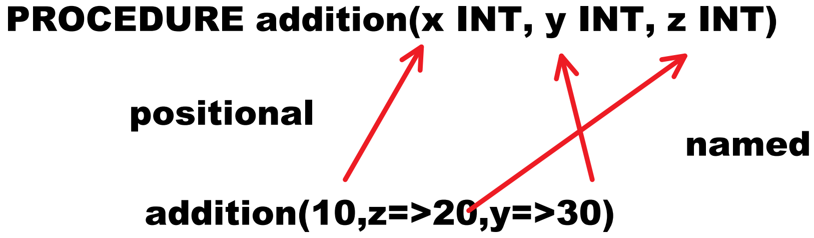


```
PROCEDURE addition(x INT, y INT, z INT)

addition(z=>10,x=>20,y=>30)
```

named mapping

mixed mapping:
In mixed mapping, actual parameters will be mapped with formal parameters based on positions and names.

Example:



```
PROCEDURE addition(x INT, y INT, z INT)

addition(10,z=>20,y=>30)
```

positional

named

addition(z=>10,20,30) => ERROR

Define a procedure to add 3 numbers:

```
CREATE OR REPLACE PROCEDURE
addition(x NUMBER, y NUMBER, z NUMBER)
AS
BEGIN
    dbms_output.put_line('sum=' || (x+y+z));
    dbms_output.put_line('x=' || x);
    dbms_output.put_line('y=' || y);
    dbms_output.put_line('z=' || z);
END;
/
```

```
SQL> EXEC addition(10,20,30);
Output:
sum=60
x=10
y=20
z=30
```

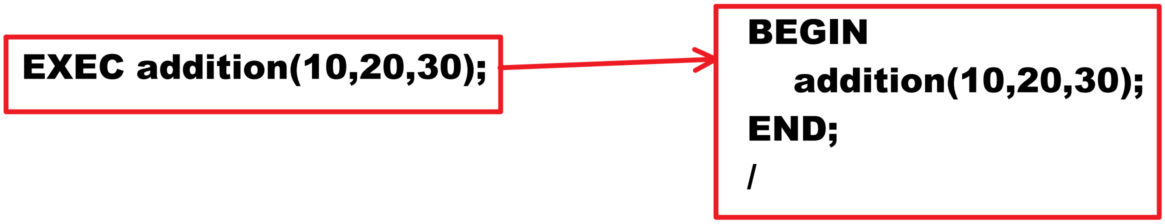
```
SQL> EXEC addition(z=>10,x=>20,y=>30);
Output:
sum=60
x=20
y=30
z=10
```

```
SQL> EXEC addition(10,z=>20,y=>30);
Output:
sum=60
x=10
y=30
z=20
```

Note:

```
BEGIN
```

Note:



```
EXEC addition(10,20,30);
```

```
BEGIN
    addition(10,20,30);
END;
/
```

Example on IN OUT:

Define a procedure to find square of a number:

```
CREATE OR REPLACE PROCEDURE
square(x IN OUT NUMBER)
AS
BEGIN
    x:=x*x;
END;
/
```

Calling:

```
SQL> VAR a NUMBER
SQL> EXEC :a := 3;

SQL> PRINT a
Output: 3

SQL> EXEC square(:a);

SQL> PRINT a
Output: 9
```

Example:

ACCOUNTS

ACNO	NAME	BALANCE
1001	A	70000

1002	B	60000

```
CREATE TABLE accounts
(
acno NUMBER(4),
name VARCHAR2(10),
balance NUMBER(9,2)
);

INSERT INTO accounts VALUES(1001,'A',70000);
INSERT INTO accounts VALUES(1002,'B',60000);
COMMIT;
```

Define a procedure to perform withdraw operation:

```
CREATE OR REPLACE PROCEDURE
withdraw(p_acno IN NUMBER, p_amount IN NUMBER)
AS
   v_balance ACCOUNTS.BALANCE%TYPE;
BEGIN
   SELECT balance INTO v_balance FROM accounts
   WHERE acno=p_acno;

   IF p_amount>v_balance THEN
       raise_application_error(-20050,'Insufficient Balance');
   END IF;

   UPDATE accounts SET balance=balance-p_amount
   WHERE acno=p_acno;

   COMMIT;

   dbms_output.put_line('successful withdrawl');
END;
/
```

Calling:

```
SQL> EXEC withdraw(1001,90000);
Output:
ERROR at line 1:
ORA-20050: Insufficient Balance

SQL> EXEC withdraw(1001,20000);
Output:
successful withdrawl
```

Example:

ACCOUNTS

ACNO	NAME	BALANCE
1001	A	50000
1002	B	60000

define a procedure to perform deposit operation:

```
CREATE OR REPLACE PROCEDURE
deposit(p_acno IN NUMBER, p_amount IN NUMBER)
AS
BEGIN
   UPDATE accounts SET balance=balance+p_amount
   WHERE acno=p_acno;

   COMMIT;

   dbms_output.put_line('amount deposited successfully..');
END;
/
```

Calling:

```
SQL> EXEC deposit(1001,30000);
Output:
amount deposited successfully..
```

Example:

ACCOUNTS

ACNO	NAME	BALANCE
1001	A	80000
1002	B	60000

Define a procedure to perform deposit operation,
after deposit, send current balance out of the procedure:

```
CREATE OR REPLACE PROCEDURE
deposit(p_acno IN NUMBER, p_amount IN NUMBER,
p_balance OUT NUMBER)
AS
BEGIN
   UPDATE accounts SET balance=balance+p_amount
   WHERE acno=p_acno;

   COMMIT;

   dbms_output.put_line('transaction successful..');

   SELECT balance INTO p_balance FROM accounts
   WHERE acno=p_acno;
END;
/
```

Calling:

```
SQL> VAR b NUMBER

SQL> EXEC deposit(1001,20000,:b);
```

Output:

```
transaction successful..
```

```
SQL> PRINT b
```

Assignment:

Define a procedure to perform fund transfer operation:

ACCOUNTS

ACNO	NAME	BALANCE
1001	A	100000
1002	B	60000

1001	A	50000
1002	B	110000

transfer 50000 amount from 1001 to 1002

```
EXEC fund_transfer(1001, 1002, 50000)
```

```
CREATE OR REPLACE PROCEDURE
fund_transfer(p_from NUMBER, p_to NUMBER, p_amount NUMBER)
AS
BEGIN
   --check sufficient funds available p_FROM

   --if not raise error

   -- if sufficient funds available, perform fund transfer
   --update from account balance
   --update to account balance
   --commit
   --display message
END;
/
```

user_procedures
user_source

user_procedures:

- it is a system table / readymade table
- It maintains all procedures information

To see all procedures list:

```
DESC user_procedures

SELECT object_name, object_type
FROM user_procedures
WHERE object_type='PROCEDURE';
```

user_source:

- it is a system table / readymade table
- It maintains all procedures information including code.

TO see all procedures list:

```
DESC user_source

SELECT DISTINCT name, type
FROM user_source
WHERE type='PROCEDURE;
```

To see procedure's code:

```
SELECT text
FROM user_source
WHERE name='WITHDRAW';
```

Dropping procedure:

Syntax:

```
DROP PROCEDURE <name>;
```

Example:

```
DROP PROCEDURE square;
```

To give permission on procedure to other user:

```
GRANT execute ON addition TO c##userabc;
```

login as c##userabc:

```
SQL> EXEC c##batch9am.addition(10,20,30);
```

Stored Functions

Friday, May 3, 2024 10:25 AM

Function:

- **Function is one ORACLE DB Object.**

- **Function is a named block of statements that gets executed on calling.**

- **A function can be also called as "Sub Program".**

Types of Functions:

2 types:

- **Stored Function**
- **Packaged Function**

Stored Function:

- **If function is defined in SCHEMA then it is called "Stored Function".**

  Example:

  SCHEMA c##batch9am
  FUNCTION check_balance => stored function

Packaged Function:

- **If function is defined in PACKAGE then it is called "Packaged Function".**

Example:

SCHEMA c##batch9am
PACKAGE bank
FUNCTION check_balance   => packaged function

Note:

- **To perform DML operations define PROCEDURE**
- **To perform calculations or FETCH (Select) operations define FUNCTION**

Example:

```
opening account  => INSERT  => PROCEDURE
withdraw         => UPDATE => PROCEDURE
deposit          => UPDATE => PROCEDURE
closing account  => DELETE  => PROCEDURE

checking balance        => SELECT => FUNCTION
calculate experience   => calc       => FUNCTION
transaction statement => SELECT => FUNCTION
```

Syntax to define Stored Function:

```
CREATE [OR REPLACE] FUNCTION                     → Header
<name>[(<paramters_list>)] RETURN <type>
IS/AS
    --declare the variables
BEGIN
    --Statements                                 → Body
    RETURN <expression>;
END;
/
```

```
/
```

Note:

- **Every function returns the value.**
- **here, returning value is mandatory.**
- **A function can return 1 value only.**
- **In Function, declare all parameters are IN parameters only. Don't declare OUT parameters.**

Example on Stored Function:

Define a function to multiply 2 numbers:

```
CREATE OR REPLACE FUNCTION
product(x NUMBER, y NUMBER) RETURN
NUMBER
AS
   z NUMBER(4);
BEGIN
   z := x*y;

   RETURN z;
END;
/
```

Calling a function:

3 ways:

- **From SQL prompt**
- **From PL/SQL program [main program]**
- **From Programming Languages**

- **From SQL prompt:**

```
SQL> SELECT product(2,3) FROM dual;
Output: 6
```

From PL/SQL program:

```
DECLARE
    a NUMBER(4);
    b NUMBER(4);
    c NUMBER(4);
BEGIN
    a := &a;
    b := &b;

    c := product(a,b);    --function call

    dbms_output.put_line('product=' || c);
END;
/
```

Define a function to calculate experience of an employee:

```
CREATE OR REPLACE FUNCTION
experience(p_empno NUMBER) RETURN NUMBER
```

```
AS
   v_hiredate DATE;
   v_exp INT;
BEGIN
   SELECT hiredate INTO v_hiredate FROM emp
   WHERE empno=p_empno;

   v_exp := TRUNC((sysdate-v_hiredate)/365);

   RETURN v_exp;
END;
/
```

Calling:

```
SQL> SELECT experience(7369) FROM dual;
Output: 43
```

Display all emp names in lower case, hiredates and calculate experience of all emps:

```
   SELECT empno, lower(ename) AS ename,
   hiredate, experience(empno) As experience
   FROM emp;
```

Define a function to display emp records of specific dept:

```
CREATE OR REPLACE FUNCTION
getdept(p_deptno NUMBER) RETURN SYS_REFCURSOR
AS
   c1 SYS_REFCURSOR;
BEGIN
   OPEN c1 FOR SELECT * FROM emp WHERE
   deptno=p_deptno;
   RETURN c1;
END;
/
```

calling:

```
SQL> SELECT getdept(10) FROM dual;
```

Output: prints 10th dept records

CALLING FROM PL/SQL PROGRAM:

```
DECLARE
      C1 SYS_REFCURSOR;
      R EMP%ROWTYPE;
BEGIN
      C1 := GETDEPT(10);

     LOOP
    FETCH C1 INTO R;
    EXIT WHEN C1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(R.ENAME || '    ' || R.DEPTNO);
     END LOOP;
    CLOSE C1;
END;
/
```

display top n salaried emp records:

```
CREATE OR REPLACE FUNCTION
gettopn(n NUMBER) RETURN SYS_REFCURSOR
AS
   c1 SYS_REFCURSOR;
```

```
BEGIN
    OPEN c1 FOR SELECT * FROM (SELECT ename, sal,
    dense_rank() over(order by sal desc) as rank
    FROM emp) WHERE rank<=n;

    RETURN c1;
END;
/
```

calling:

```
select gettopn(3) from dual;
```

Output: display top 3 salaried emp records

Example:

ACCOUNTS

ACNO	NAME	BALANCE
1001	A	100000
1002	B	60000

Define a function to check the balance:

```
CREATE OR REPLACE FUNCTION
check_balance(p_acno ACCOUNTS.ACNO%TYPE) RETURN
NUMBER
AS
    v_balance ACCOUNTS.BALANCE%TYPE;
BEGIN
    SELECT balance INTO v_balance FROM accounts
    WHERE acno=p_acno;

    RETURN v_balance;
END;
/
```

calling:

```
SELECT check_balance(1001) FROM dual;
Output: 100000
```

Note:

There are 2 types of sub programs. they are:

- PROCEDURE
- FUNCTION

Advantages of Sub Program:

- improves the performance.
  Sub program holds compiled code.
  it saves compilation time. So, improves the performance.

- provides reusability

- reduces length of code

- provides security

- better maintenance

- improves the understandability

user_procedures
user_source

user_procedures:

- **is a system table**
- **it maintains all functions info**

user_source:

- **is a system table**
- **it maintains all functions info including code**

to see functions list:

```
SELECT object_name, object_type
FROM user_procedures
WHERE object_type='FUNCTION';

SELECT DISTINCT name, type
FROM user_source
WHERE type='FUNCTION';
```

to see function's code:

```
SELECT text
FROM user_source
WHERE name='CHECK_BALANCE';
```

Dropping Function:

Syntax:

```
DROP FUNCTION <name>;
```

Example:

```
DROP FUNCTION getdept;
```

To give permission on procedure to other user:

```
GRANT execute ON product TO c##userabc;
```

login as c##userabc:

```
SQL> SELECT c##batch9am.product(5,4) FROM dual;
Output: 20
```

PACKAGES

Saturday, May 4, 2024 10:22 AM

PACKAGES:

- PACKAGE is one ORACLE DB Object.

- PACKAGE is a collection of procedures, functions, variables, types, exceptions and cursors.

Creating Package:
To create the package follow 2 steps:

- Package Specification
- Package Body

Package Specification:
in this, we declare procedures, functions, global variables … etc.

Syntax:

```
CREATE [OR REPLACE] PACKAGE <name>
IS/AS
   --declare the procedures, functions, global variables
END;
/
```

Package body:

- In this, we define body of procedures and functions.

Syntax:

```
CREATE [OR REPLACE] PACKAGE BODY <name>
IS/AS
   --define the procedures, functions
END;
/
```

Example on defining package:

```
    PACKAGE math

        PROCEDURE addition
        FUNCTION product
```

```
--package specification

CREATE OR REPLACE PACKAGE math
AS
    PROCEDURE addition(x NUMBER, y NUMBER);
    FUNCTION product(x NUMBER, y NUMBER) RETURN
    NUMBER;
END;
/



--package body

CREATE OR REPLACE PACKAGE BODY math
AS
    PROCEDURE addition(x NUMBER, y NUMBER)
    AS
    BEGIN
        dbms_output.put_line('sum=' || (x+y));
    END addition;

    FUNCTION product(x NUMBER, y NUMBER) RETURN
    NUMBER
    AS
    BEGIN
        RETURN x*y;
    END product;
END;
/
```

Syntax to call packaged procedure:

```
    <package_name>.<procedure_name>(<arguments>)
```

Syntax to call packaged function:

```
    <package_name>.<function_name>(<arguments>)
```

Calling from SQL prompt:

```
SQL>EXEC math.addition(2,3);
Output: sum=5

SQL> SELECT math.product(2,3) FROM dual;
Output: 6
```

Calling from PL/SQL program:

```
    DECLARE
        a NUMBER;
        b NUMBER;
        c NUMBER;
    BEGIN
        a := &a;
        b := &b;

        math.addition(a,b);

        c := math.product(a,b);
        dbms_output.put_line('product=' || c);
    END;
    /
```

Example:

PACKAGE HR

PROCEDURE HIRE
PROCEDURE FIRE
PROCEDURE HIKE

FUNCTION experience

PROCEDURE HIKE

FUNCTION experience

```
--Package Specification

CREATE OR REPLACE PACKAGE HR
AS
   PROCEDURE hire(p_empno NUMBER, p_ename VARCHAR2);
   PROCEDURE fire(p_empno NUMBER);
   PROCEDURE hike(p_empno NUMBER, p_amount NUMBER);

   FUNCTION experience(p_empno NUMBER) RETURN NUMBER;
END;
/

--Package body

CREATE OR REPLACE PACKAGE BODY HR
AS
   PROCEDURE hire(p_empno NUMBER, p_ename VARCHAR2)
   AS
   BEGIN
      INSERT INTO emp(empno,ename) VALUES(p_empno,p_ename);
      COMMIT;
      dbms_output.put_line('record inserted..');
   END hire;

   PROCEDURE fire(p_empno NUMBER)
   AS
   BEGIN
      DELETE FROM emp WHERE empno=p_empno;
      COMMIT;
      dbms_output.put_line('record deleted..');
   END fire;

   PROCEDURE hike(p_empno NUMBER, p_amount NUMBER)
   AS
   BEGIN
      UPDATE emp SET sal=sal+p_amount WHERE empno=p_empno;
      COMMIT;
      dbms_output.put_line('sal increased..');
   END hike;

   FUNCTION experience(p_empno NUMBER) RETURN NUMBER
   AS
      v_hiredate DATE;
   BEGIN
      SELECT hiredate INTO v_hiredate FROM emp
      WHERE empno=p_empno;

      RETURN TRUNC((sysdate-v_hiredate)/365);
   END experience;
END;
/
```

calling:

```
SQL> EXEC hr.hire(1001,'A');
output:
record inserted..

SQL> EXEC hr.hike(7369,1000);
output:
sal increased..

SQL> EXEC hr.fire(1001);
output:
record deleted..

SQL> SELECT hr.experience(7369) from dual;
output:
HR.EXPERIENCE(7369)
------------------
                43

SQL> select empno, ename, hr.experience(empno) from emp;
    output:
    EMPNO ENAME      HR.EXPERIENCE(EMPNO)
    ---------------------------------------------------------------------------
     7369 SMITH                       43
     7499 ALLEN                       43
     7521 WARD                        43
```

Assignment:

Accounts

ACNO	NAME	BALANCE
1001	A	70000
1002	B	50000

PACKAGE bank

insert => PROCEDURE opening_account
delete => PROCEDURE closing_account
update => PROCEDURE withdraw
update => PROCEDURE deposit
select => FUNCTION check_balance

Advantages:

- **We can group related procedures and functions.**
- **improves the performance.**
  number of travels to DB can be reduced using PACKAGE. So, performance will be improved.
- **provides reusability**
- **decreases length of code**
- **provides security**
- **better maintenance**
- **Packaged procedures or Packaged functions can be overloaded. WHERE AS stored procedures or stored functions cannot be overloaded.**

- **We can declare global variables.**

- **We can make members as public or private.**

OVERLOADING:

- **defining multiple procedures or functions with same name and different signatures is called "Overloading".**

PACKAGE p1

PROCEDURE demo **change in number of parameters**
PROCEDURE demo(x int)

PROCEDURE demo(x int, y varchar2)
PROCEDURE demo(x date, y char) **change in data types**

PROCEDURE demo(x int, y varchar2)
PROCEDURE demo(x varchar2, y int) **change in order of parameters**

different signature means,

- **change in no of parameters**
- **change in data types**
- **change in order of parameters**

- **Packaged procedures or Packaged functions can be overloaded. WHERE AS stored procedures or stored functions cannot be overloaded.**

Example on OVERLOADING:

PACKAGE OLDEMO

FUNCTION addition(x NUMBER, y NUMBER)

```
FUNCTION addition(x NUMBER, y NUMBER)
FUNCTION addition(x NUMBER, y NUMBER, z NUMBER)
```

```
--PACKAGE SPECIFICATION

CREATE OR REPLACE PACKAGE OLDEMO
AS
   x INT := 500;     --global variable
   FUNCTION addition(x NUMBER, y NUMBER) RETURN NUMBER;
   FUNCTION addition(x NUMBER, y NUMBER, z NUMBER) RETURN NUMBER;
END;
/



--package body

CREATE OR REPLACE PACKAGE BODY OLDEMO
AS
   FUNCTION addition(x NUMBER, y NUMBER) RETURN NUMBER
   AS
   BEGIN
      RETURN x+y;
   END addition;

   FUNCTION addition(x NUMBER, y NUMBER, z NUMBER) RETURN NUMBER
   AS
   BEGIN
      RETURN x+y+z;
   END addition;
END;
/
```

calling:

```
SELECT OLDEMO.addition(1,2) FROM dual;   --3

SELECT OLDEMO.addition(1,2,3) FROM dual;   --6
```

```
EXEC dbms_output.put_line(OLDEMO.x);        --500
```

program:

```
DECLARE
   a INT;
BEGIN
   a := 20;

   dbms_output.put_line('sum=' || (OLDEMO.x+a));
END;
/
```

Output:

```
sum=520
```

Example:

PACKAGE SPECIFICATION demo

PROCEDURE p2
PROCEDURE p3

PACKAGE BODY demo

PROCEDURE p1
PROCEDURE p2
PROCEDURE p3

Note:
The members which are declared in PACKAGE SPECIFICATION are called public members.

The members which are defined in PACKAGE BODY but not declared in PAKCAGE SPECIFICATION are called private members

```
p2, p3 => public member
p1     => private member
```

Example on public and private:

PACKAGE SPECIFICATION demo

PROCEDURE p2
PROCEDURE p3

PACKAGE BODY demo

PROCEDURE p1
PROCEDURE p2
PROCEDURE p3

```
--package specification

CREATE OR REPLACE PACKAGE demo
AS
    PROCEDURE p2;
    PROCEDURE p3;
END;
/

--package body

CREATE OR REPLACE PACKAGE BODY demo
AS
    PROCEDURE p1 AS
    BEGIN
        dbms_output.put_line('p1 called');
    END p1;

    PROCEDURE p2 AS
    BEGIN
        p1;
        dbms_output.put_line('p2 called');
    END p2;

        PROCEDURE p3 AS
    BEGIN
        p1;
        dbms_output.put_line('p3 called');
    END p3;
END;
/
```

calling:

```
EXEC demo.p2
```

Output:

```
p1 called
p2 called
```

```
EXEC demo.p3
```

Output:

```
p1 called
p3 called
```

```
EXEC demo.p1
```

Output:

```
ERROR: p1 is private member. private
member can be used with in package only.
```

user_procedures
user_source

user_procedures:

- **it is a system table**
- **it maintains all packages information**

user_Source:

- **it is a system table**
- **it maintains all packages information including code**

to see packages info:

```
column object_name format a10

SELECT object_name, procedure_name, object_type
FROM user_procedures
WHERE object_type='PACKAGE';
```

(or)

```
SELECT DISTINCT name, type
```

```
  FROM user_source
  WHERE type='PACKAGE';
```

to see package code:

```
   SELECT text
   FROM user_source
   WHERE name='HR';
```

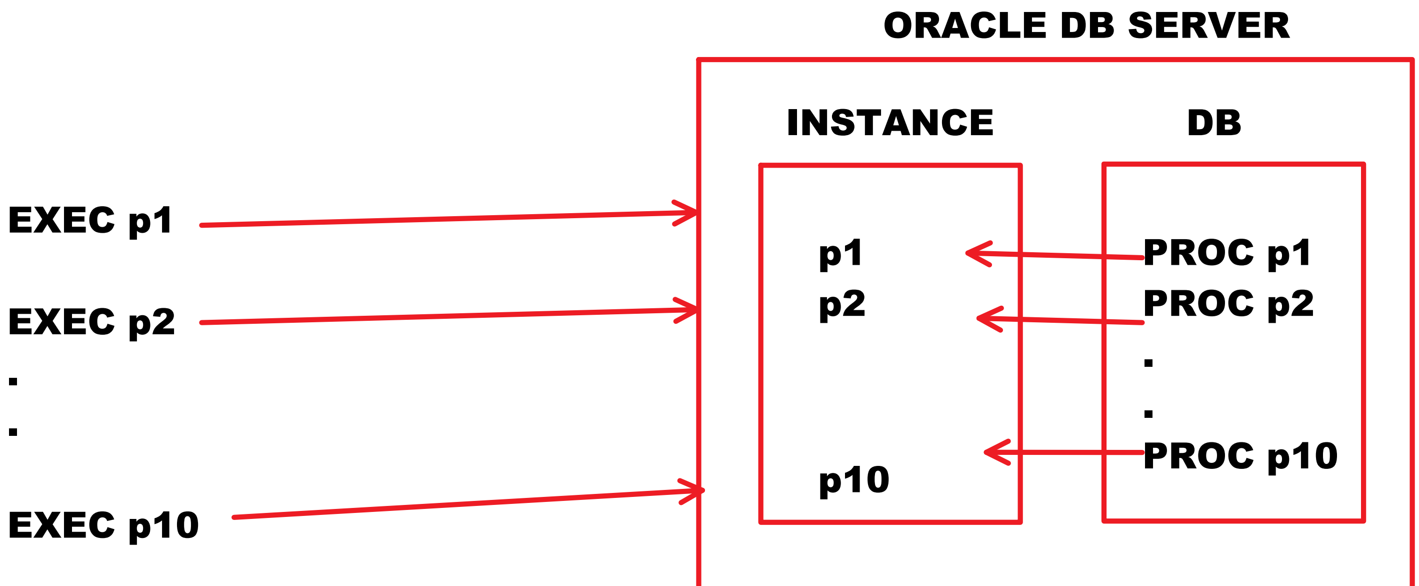
Dropping package:

```
   Syntax:
      DROP PACKAGE <name>;

   Example:
      DROP PACKAGE hr;
      --package specification and body will be dropped
```

Dropping package body only:

```
   SYNTAX:
      DROP PACKAGE BODY <name;

   Example:
      DROP PACKAGE BODY demo;
```

Monday, May 6, 2024 9:50 AM



ORACLE DB SERVER

INSTANCE

DB

EXEC p1

EXEC p2

.

.

EXEC p10

p1

p2

p10

PROC p1

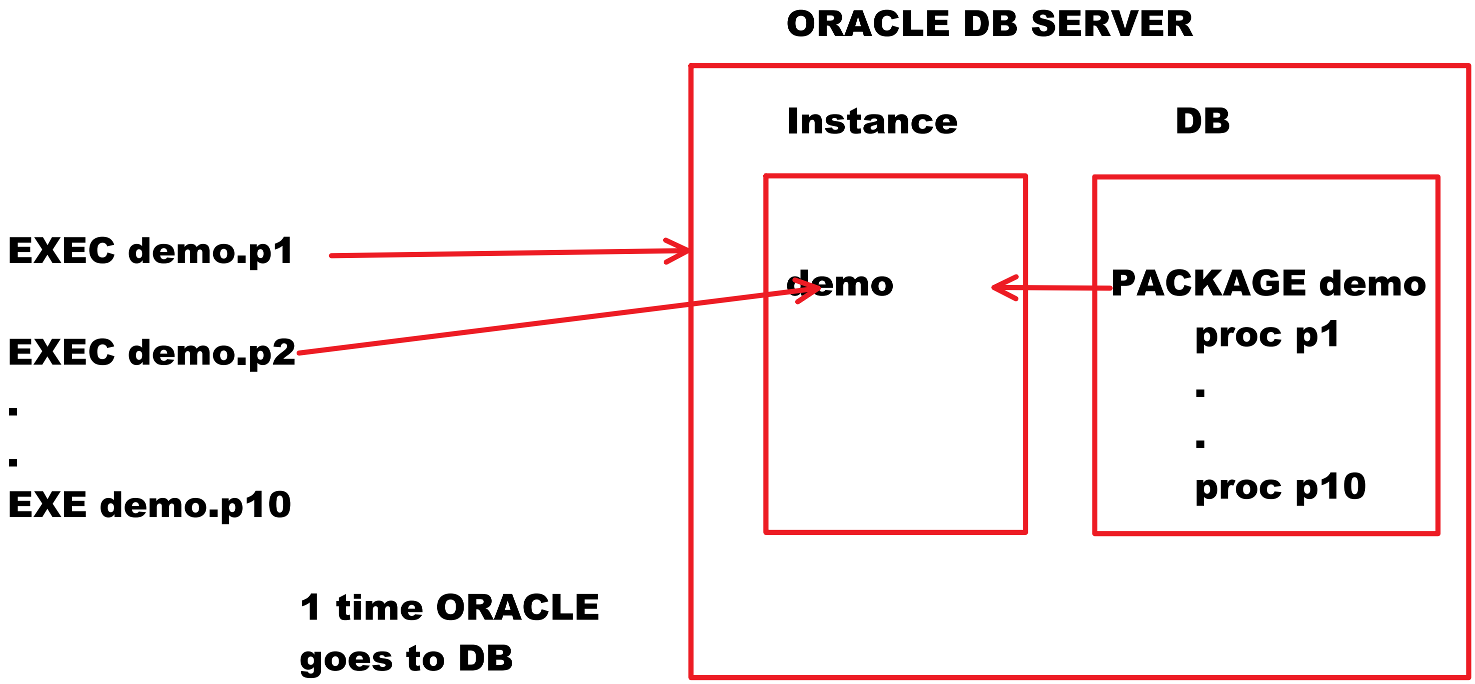
PROC p2

.

.

PROC p10

10 times ORACLE
goes to DB



ORACLE DB SERVER

Instance

DB

EXEC demo.p1

EXEC demo.p2

.

.

EXE demo.p10

demo

PACKAGE demo
proc p1
.
.
proc p10

1 time ORACLE
goes to DB
and loads entire
package into INSTANCE

COLLECTIONS

Tuesday, May 7, 2024 9:18 AM

COLLECTION:

- **COLLECTION is a set of elements of type.**

Examples:

x

20	x(1)
80	x(2)
45	x(3)
77	x(4)

NUMBER type

d

ACCOUNTS	d(1)
HR	d(2)
RESEARCH	d(3)
OPERATIONS	d(4)

VARCHAR2 type

e

EMPNO	ENAME	SAL
1001	A	6000

e(1)

EMPNO	ENAME	SAL
1002	B	9000

e(2)

EMPNO	ENAME	SAL
1003	C	7000

e(3)

employee%rowtype

GOAL:
CURSOR is used to hold multiple rows
COLLECTION is used to hold multiple rows

CURSOR has some drawbacks. To avoid those drawbacks we use COLLECTION.

Types of Collections:

3 types:

- **Associative Array / Index By Table / PL_SQL Table**
- **Nested Table**
- **V-Array**

Associative Array:

- **Associative Array is a table of 2 columns. they are:**
  - **INDEX**
  - **ELEMENT**

Example:

x

INDEX	ELEMENT
1	70
2	40
3	90

x(1) => 70

a

INDEX	ELEMENT
HYD	800000
BLR	1000000
DLH	900000

a('HYD') => 800000

Creating Associative Array:

2 steps:

- **define our own associative array data type**
- **declare variable for it**

define our own associative array data type:

Syntax:

```
TYPE <type_name> IS TABLE OF <element_type>
INDEX BY <index_type>;
```

Note:
For Associative array, data type is not ready.
So, define our own data type and declare variable for it.

Example:

```
TYPE num_array IS TABLE OF NUMBER(2)
INDEX BY binary_integer;
```

x

INDEX	ELEMENT
1	70
2	40
3	90

Note:
If INDEX is number type, use binary_integer

or pls_integer.

- **declare variable for our own associative array type:**

Syntax:
<variable> <data_type>;

Example:
x NUM_ARRAY;

x

INDEX	ELEMENT
1	50
2	90
3	30

Note:

- **when we define our own data type, implicitly one function will be created with data type name. this special function is called "collection constructor".**

x := num_array(50,90,30);

x(1) => 50
x(2) => 90
x(3) => 30

num_array is collection constructor
it is used to initialize the collection

Collection members:

FIRST	first index	x.first
LAST	last index	x.last
NEXT	next index	x.next(2) =>next index of 2 => 3
PRIOR	previous index	x.prior(2) => prev index of 2 => 1

Example:
Create an Associative Array to hold number type elements and indexes as following:

x

INDEX	ELEMENT
1	50
2	90

3	30
4	88
5	75

x

INDEX	ELEMENT
1	50
2	90
3	30
4	88
5	75

```
DECLARE
   TYPE num_array IS TABLE OF number(2)
   INDEX BY binary_integer;

   x NUM_ARRAY;
BEGIN

   x :=    num_array(50,90,30,88,75);   --oracle 21c only

   /*     x(1) := 50;
          x(2) := 90;
          x(3) := 30;
          x(4) := 88;
          x(5) := 75;    */      --lesser than oracle 21c


   dbms_output.put_line('first element=' || x(1));
   dbms_output.put_line('first index=' || x.first);
   dbms_output.put_line('last index=' || x.last);

   dbms_output.put_line('next index of 2=' || x.next(2));
   dbms_output.put_line('prev index of 2=' || x.prior(2));


   dbms_output.put_line('all elements are:');
   FOR i IN x.first .. x.last
   LOOP
   dbms_output.put_line(x(i));
   END LOOP;

END;
/
```

Example:
Create an associative array to hold dept names of dept table:

d

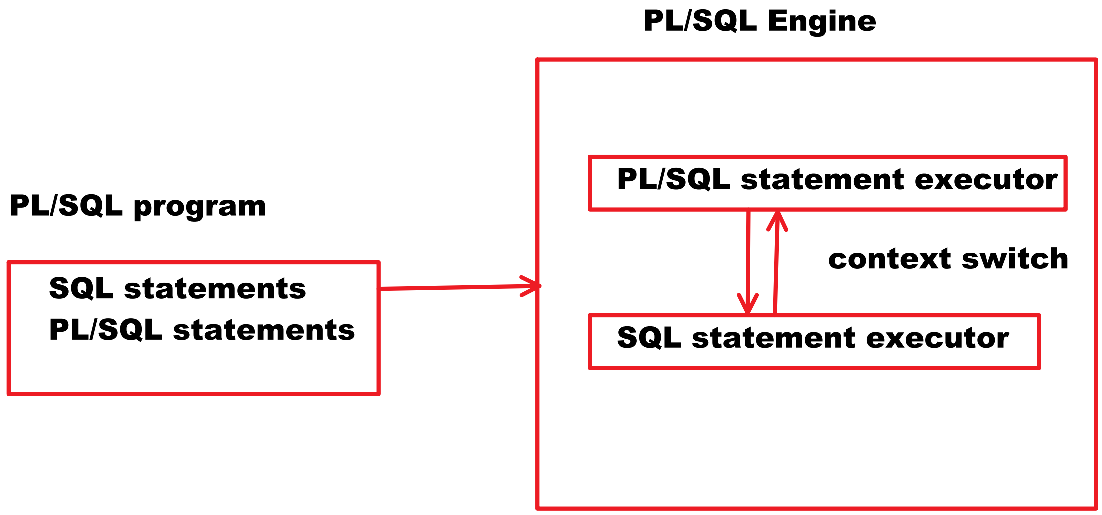
INDEX	ELEMENT
1	ACCOUNTING

2	RESEARCH
3	SALES
4	OPERATIONS

```
DECLARE
   TYPE dept_array IS TABLE OF varchar2(10)
   INDEX BY binary_integer;

   d DEPT_ARRAY;
BEGIN
   SELECT dname INTO d(1) FROM dept WHERE deptno=10;
   SELECT dname INTO d(2) FROM dept WHERE deptno=20;
   SELECT dname INTO d(3) FROM dept WHERE deptno=30;
   SELECT dname INTO d(4) FROM dept WHERE deptno=40;

   FOR i IN d.first .. d.last
   LOOP
      dbms_output.put_line(d(i));
   END LOOP;
END;
/
```

```
Output:
ACCOUNTING
RESEARCH
SALES
OPERATIONS
```

Above program degrades the performance.



Above program degrades the performance.

PL/SQL Engine contains PL/SQL statement executor and SQL statement executor.

PL/SQL statement executor can execute PL/SQL statements only. If any SQL statement found, it submits it to SQL statement executor. This result will be submitted to PL/SQL statement executor.

Travelling from PL/SQL statement executor to SQL statement executor and SQL statement executor to PL/SQL statement executor is called one "CONTEXT SWITCH".

In above program, 4 context switches will occur to execute 4 SELECT commands.

If no of context switches are increased then performance will be degraded.

TO improve the performance we use BULK COLLECT

BULK COLLECT:

- **It is used to collect entire data with single context switch.**

- **It reduces number of context switches. So, it improves the performance.**

Above program write as following to improve the performance:

```
DECLARE
   TYPE dept_array IS TABLE OF varchar2(10)
   INDEX BY binary_integer;

   d DEPT_ARRAY;
BEGIN
   SELECT dname BULK COLLECT INTO d FROM dept;

   FOR i IN d.first .. d.last
   LOOP
      dbms_output.put_line(d(i));
   END LOOP;
END;
```

```
/
```

Program to print all emp records. Create an associative array. hold all emp records in associative array. and print them:

e

INDEX	ELEMENT
1	empno ename sal 1001 A 7000
2	empno ename sal 1002 B 5000
3	empno ename sal 1003 C 6000

```
DECLARE
    TYPE emp_array IS TABLE OF emp%rowtype
    INDEX BY binary_integer;

    e EMP_ARRAY;
BEGIN
    SELECT * BULK COLLECT INTO e FROM emp;

    FOR i IN e.first .. e.last
    LOOP
        dbms_output.put_line(e(i).ename || '    ' || e(i).sal);
    END LOOP;
END;
/
```

Program to increase salary of all employees according to HIKE table percentages. Create an Associative Array. Hold All HIKE table records in it.

Using it, update salary in employee table:

EMPLOYEE

EMPNO	ENAME	SAL
1001	A	5000
1002	B	3000
1003	C	7000

HIKE

EMPNO	PER
1001	10
1002	20
1003	15

```
create table employee
(
empno NUMBER(4),
ename VARCHAR2(10),
sal NUMBER(8,2)
);

INSERT INTO employee VALUES(1001,'A',5000);
INSERT INTO employee VALUES(1002,'B',3000);
INSERT INTO employee VALUES(1003,'C',7000);
COMMIT;

create table hike
(
empno NUMBER(4),
per NUMBER(2)
);

INSERT INTO hike VALUES(1001,10);
INSERT INTO hike VALUES(1002,20);
INSERT INTO hike VALUES(1003,15);
COMMIT;
```

h

INDEX	ELEMENT
1	EMPNO PER 1001 10
2	EMPNO PER 1002 20
3	EMPNO PER 1003 15

```
DECLARE
   TYPE hike_array IS TABLE OF hike%rowtype
   INDEX BY binary_integer;

   h HIKE_ARRAY;
BEGIN
   SELECT * BULK COLLECT INTO h FROM hike;

   FOR i IN h.first .. h.last
   LOOP
      UPDATE employee SET sal=sal+sal*h(i).per/100
      WHERE empno=h(i).empno;
   END LOOP;

   COMMIT;

   dbms_output.put_line('sal increased to all emps..');
END;
/
```

Above program degrades the performance.

FOR LOOP will be executed by PL/SQL statement executor.
UPDATE command will be executed by SQL statement executor.
To increase salary to 5 emps 5 context switches will occur.

If number of context switches are increased then performance will be degraded.
To improve performance we use BULK BIND.

BULK BIND:

- **BULK BIND is used to execute BULK UPDATE / BULK INSERT / BULK DELETE commands.**

- **With single context switch BULK UPDATE / BULK INSERT / BULK DELETE commands get executed.**

- **it reduces number of context switches and improves the performance.**

- **For BULK BIND we use "FORALL LOOP".**

Syntax of FORALL:

```
FORALL <variable> IN <lower> .. <upper>
    --DML statement
```

```
DECLARE
    TYPE hike_array IS TABLE OF hike%rowtype
    INDEX BY binary_integer;
    h HIKE_ARRAY;
BEGIN
    SELECT * BULK COLLECT INTO h FROM hike;

    FORALL i IN h.first .. h.last
        UPDATE employee SET sal=sal+sal*h(i).per/100
        WHERE empno=h(i).empno;

    COMMIT;
    dbms_output.put_line('sal increased to all emps..');
END;
/
```

Nested Table:

- nested table is a table of 1 column.
- here, INDEX can be number type only. that is why no need to maintain INDEX separately.

Example:

x

ELEMENT	
50	x(1)
90	x(2)
30	.
88	.
75	.

Creating Nested Table:

follow 2 steps:

- **create our own nested table data type**
- **declare variable for it.**

- **create our own nested table data type:**

Syntax:

```
TYPE <name> IS TABLE OF <element_type>;
```

Example:

```
TYPE num_array IS TABLE OF NUMBER(4);
```

declare variable for it:

Syntax:

```
<variable> <data_type>;
```

Example:

```
x NUM_ARRAY;
```

Program to create a nested table. hold number type elements in it as following:

x

ELEMENT
50
90
30
88
75

```
DECLARE
   TYPE num_array IS TABLE OF number(2);


   x NUM_ARRAY;
BEGIN
```

```
    x :=     num_array(50,90,30,88,75);

    FOR i IN x.first .. x.last
    LOOP
    dbms_output.put_line(x(i));
    END LOOP;
END;
/
```

V-ARRAY:

- **V-ARRAY => Variable Size Array**

- **it is a same as nested table.**
- **we must specify the size.**
- **in v-array we can store limited number of elements**

Creating V-Array:

2 steps:

- ○ **Define our own V-ARRAY data type**
- ○ **Declare variable for it**

Define our own V-ARRAY data type:

Syntax:

```
TYPE <name> IS VARRAY(<Size>) OF <element_type>;
```

Example:

```
TYPE num_array IS VARRAY(10) OF number(2);
```

Declaring Variable for it:

Syntax:

```
<variable> <data_type>;
```

Example:

```
x NUM_ARRAY;
```

Example on V-ARRAY:

```
DECLARE
```

```
      TYPE num_array IS VARRAY(10) OF number(2);


      x NUM_ARRAY;
   BEGIN
      x :=    num_array(50,90,30,88,75);

      FOR i IN x.first .. x.last
      LOOP
      dbms_output.put_line(x(i));
      END LOOP;
   END;
   /
```

Print all emp records using nested table:

```
DECLARE
   TYPE emp_array IS TABLE OF emp%rowtype;


   e EMP_ARRAY;
BEGIN
   SELECT * BULK COLLECT INTO e FROM emp;

   FOR i IN e.first .. e.last
   LOOP
      dbms_output.put_line(e(i).ename || '    ' || e(i).sal);
   END LOOP;
END;
/
```

Print all emp records using V-Array:

```
DECLARE
   TYPE emp_array IS VARRAY(20) OF emp%rowtype;

   e EMP_ARRAY;
BEGIN
   SELECT * BULK COLLECT INTO e FROM emp;

   FOR i IN e.first .. e.last
   LOOP
```

```
        dbms_output.put_line(e(i).ename || '    ' || e(i).sal);
    END LOOP;
END;
/
```

Differences b/w Associative Array, nested Table and V-Array:

COLLECTION	INDEX	NO OF ELEMENTS	DENSE OR SPARSE
Associative Array	NUMBER VARCHAR2	unlimited	DENSE or SPARSE
Nested table	NUMBER	unlimited	stars as DENSE it can become SPARSE
V-Array	NUMBER	limited	DENSE

DENSE	no gaps

INDEX
1
2
3
4

SPARSE

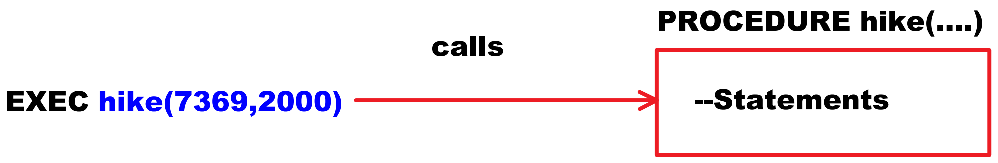
INDEX
10
20
65
91

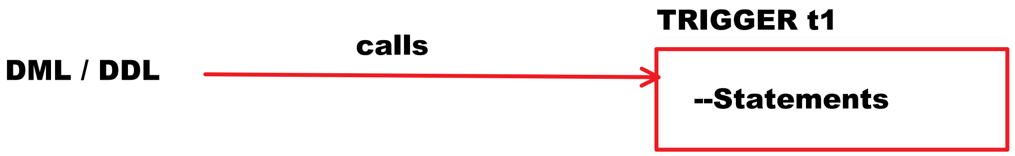
Differences b/w CURSOR and COLLECTION:

CURSOR	COLLECTION
• fetches row by row	• fetches all rows at a time and copies into collection
• can move forward only	• can move in any direction
• supports to Sequential Accessing	• supports to Random Accessing
• Slower	• Faster

TRIGGERS

Thursday, May 9, 2024 9:17 AM





TRIGGER:

- **TRIGGER is one ORACLE DB Object.**
- **TRIGGER is a named block of statements that gets executed automatically when we submit DML or DDL command.**
- **When we submit DML or DDL command implicitly ORACLE calls the TRIGGER.**
- **For procedure execution explicit call is required. But, for trigger execution explicit call is not required.**

Note:
To perform DMLs, define PROCEDURE.
To control the DMLs, define TRIGGER.

Trigger can be used for following purposes:

- **To control the DMLs**

Examples:

Don't allow DMLs on Sunday

Don't allow DMLs before and after office timings

10AM to 4 PM

- **To audit the tables or databases.**

  Example:
  which user
  on which date
  at which time
  which operations
  what was old data
  what is new data

  above things can be recorded in another table => auditing

- **To implement our own business rules [constraints].**

  Example:
  don't allow user to decrease the salary

Types of Triggers:

3 Types:

- **Table Level Trigger / DML Trigger**
  - **Statement Level Trigger**
  - **Row Level Trigger**
- **Schema Level Trigger / DDL Trigger / System Trigger**
- **Database Level Trigger / DDL Trigger / System Trigger**

Table Level Trigger:

- **If a trigger is created on table then it is called "Table Level Trigger".**
- **There are 2 types of Table Level Triggers. They are:**
  - **Statement Level Trigger**
  - **Row Level Trigger**

Statement Level Trigger:
In this, Trigger gets executed once for 1 DML statement.

Row Level Trigger:
In this, Trigger gets executed once for every row affected by DML.

Example:

```
UPDATE emp SET sal=sal+1000
WHERE job='MANAGER';
Output:
3 rows updated.
```

Statement Level Trigger	**1 time**
Row Level Trigger	**3 times**

```
UPDATE emp SET sal=sal+1000;
Output:
13 rows updated
```

Statement Level Trigger	**1 time**
Row Level Trigger	**13 times**

Syntax of Table Level Trigger:

```
CREATE [OR REPLACE] TRIGGER <name>
BEFORE/AFTER <DMLs list>
ON <table_name>
[FOR EACH ROW]
DECLARE
   --declare the variables
BEGIN
   --statements
END;
/
```

Trigger Header /
Trigger Specification

Trigger Body

Before Trigger:

- **First Trigger gets executed.**
- **Then DML operation will be performed.**

After Trigger:

- **First DML operation will be performed.**
- **Then Trigger gets executed.**

Program to demonstrate Statement Level Trigger:

```
CREATE OR REPLACE TRIGGER t1
AFTER insert or update or delete
ON emp
BEGIN
   dbms_output.put_line('stmt level trigger executed');
END;
/

Testing:
   UPDATE emp SET sal=sal+1000;
   Output:
   13 rows updated

   stmt level trigger executed


   UPDATE emp SET sal=sal+1000 WHERE job='MANAGER';
   Output:
   3 rows updated

   stmt level trigger executed
```

Program to demonstrate Row Level Trigger:

```
CREATE OR REPLACE TRIGGER t2
AFTER insert or update or delete
ON emp
FOR EACH ROW
BEGIN
   dbms_output.put_line('row level trigger executed');
END;
/


Testing:
   UPDATE emp SET sal=sal+1000;
   Output:
   13 rows updated

   row level trigger executed
   row level trigger executed
   .
   .
                                 13 times
```

```
UPDATE emp SET sal=sal+1000 WHERE job='MANAGER';
Output:
3 rows updated

row level trigger executed
row level trigger executed
row level trigger executed
```

Disabling and Enabling Trigger:

Syntax:

```
ALTER TRIGGER <name> DISABLE/ENABLE;
```

Examples:

```
ALTER TRIGGER t1 DISABLE;      --temporarily t1 will not work

ALTER TRIGGER t1 ENABLE;       --again trigger will work
```

Dropping Trigger:

Syntax:

```
DROP TRIGGER <name>;
```

Example:

```
DROP TRIGGER t1;
```

Define a trigger to don't allow the user to perform DMLs on SUNDAY:

```
CREATE OR REPLACE TRIGGER t3
BEFORE insert or update or delete
ON emp
BEGIN
```

```
   IF to_char(sysdate,'DY')='SUN' THEN
      raise_application_error(-20050,'you cannot update on SUNDAY');
   END IF;
END;
/

Testing:
Mon-Sat:
UPDATE emp SET sal=sal+1000;
Output:
13 rows updated.

On Sunday:
UPDATE emp SET sal=sal+1000;
Output:
ERROR:
ORA-20050: you cannot update on SUNDAY
```

Define a trigger not to allow the user to perform DMLs before or after office timings [office timings: 10AM to 4PM]

```
   CREATE OR REPLACE TRIGGER t4
   BEFORE insert or update or delete
   ON emp
   DECLARE
      h INT;
   BEGIN
      h := to_char(sysdate,'HH24');

      IF h NOT BETWEEN 11 AND 15 THEN
         raise_application_error(-20070,'DMLs allowed b/w 10AM to 4PM only');
      END IF;
   END;
   /

   Testing:
      b/w 10AM to 3.59 PM:

         UPDATE emp SET sal=sal+1000;
         Output:
         13 rows updated

      before 10AM or after 4PM:

         UPDATE emp SET sal=sal+1000;
```

```
Output:
ERROR:
ORA-20070: DMLs allowed b/w 10AM to 4PM only
```

Define a trigger not to allow user to update empno:

BEFORE update OF empno	user cannot update empno
BEFORE update OF empno,ename	user cannot update empno,ename

```
CREATE OR REPLACE TRIGGER t5
BEFORE update OF empno
ON emp
BEGIN
   raise_application_error(-20050,'you cannot update empno');
END;
/

Testing:
update emp set empno=5001
where empno=7369;
Output:
ERROR:
ORA-20050: you cannot update empno
```

:NEW and :OLD:

- :NEW and :OLD are bind variables.
- These are built-in variables.
- these are %ROWTYPE variables.

- :NEW holds new row.
- :OLD holds old row.

- These can be used in row level trigger only.
  These cannot be used in statement level trigger.

DML	:NEW	:OLD
INSERT	new row	null

DELETE	null	old row
UPDATE	new row	old row

Example:

EMPLOYEE

EMPNO	ENAME	SAL
1001	A	6000
1002	B	4000
1003	C	8000

```
INSERT INTO employee VALUES(1004,'D',9000);
```

:NEW

EMPNO	ENAME	SAL
1004	D	9000

:OLD

EMPNO	ENAME	SAL
null	null	null

```
DELETE FROM employee WHERE empno=1001;
```

:NEW

EMPNO	ENAME	SAL
null	null	null

:OLD

EMPNO	ENAME	SAL
1001	A	6000

```
UPDATE employee SET sal=sal+1000
WHERE empno=1002;
```

:NEW

EMPNO	ENAME	SAL
1002	B	5000

:OLD

EMPNO	ENAME	SAL
1002	B	4000

Example:

Define a trigger to record deleted records in emp_resign table:

EMP_RESIGN **DOR => date of resign**

EMPNO	ENAME	JOB	SAL	DOR

```
   CREATE TABLE emp_resign
   (
   empno NUMBER(4),
   ename VARCHAR2(10),
   job VARCHAR2(10),
   sal NUMBER(7,2),
   DOR DATE
   );
```

```
CREATE OR REPLACE TRIGGER t10
AFTER delete
ON emp
FOR EACH ROW
BEGIN
   INSERT INTO emp_resign
   VALUES(:old.empno, :old.ename, :old.job, :old.sal, sysdate);
END;
/
```

Testing:

```
delete from  emp
where empno=7876;

commit;

select * from emp_resign;
```

output:

```
empno            ename           .........................
-------------    ----------
7876             ADAMS          .........................
```

Define a trigger to audit emp table:

```
   CREATE TABLE EMP_AUDIT
   (
   UNAME VARCHAR2(20),
   OP_DATE_TIME TIMESTAMP,
   OP_TYPE varchar2(10),
 OLD_EMPNO NUMBER(4),
 OLD_ENAME VARCHAR2(10),
 OLD_SAL NUMBER(7,2),
 NEW_EMPNO NUMBER(4),
 NEW_ENAME VARCHAR2(10),
 NEW_SAL NUMBER(7,2)
 );
```

EMP_AUDIT

uname	op_date	op_type	old_empno	old_ename	old_sal	new_empno	new_ename	new_sal
user	systimestamp	op	:old.empno	:old.ename	:old.sal	:new.empno	:new.ename	:new.sal

```
CREATE OR REPLACE TRIGGER t11
AFTER insert or delete or update
ON emp
FOR EACH ROW
DECLARE
   op STRING(10);
BEGIN
   IF inserting THEN
      op:='INSERT';
   ELSIF updating THEN
      op:='UPDATE';
   ELSIF deleting THEN
      op:='DELETE';
   END IF;

   INSERT INTO emp_audit
   VALUES(user, systimestamp, op,
   :old.empno, :old.ename, :old.sal,
   :new.empno, :new.ename, :new.sal);
END;
/

testing:
insert into emp(empno,ename,sal)
values(9001,'AA',10000);

commit;

select * from emp_audit;
```

Example:

Define a trigger to don't allow the user to decrease the salary:

:OLD

empno	ename	sal
7902	FORD	11000

:NEW

empno	ename	sal
7902	FORD	9000

```
CREATE OR REPLACE TRIGGER t12
BEFORE update
ON emp
FOR EACH ROW
BEGIN
   IF :new.sal<:old.sal THEN
      raise_application_error(-20080,'you cannot decrease the sal');
   END IF;
END;
/
```

testing:

```
update emp set sal=sal-1000;
```

Output:

```
ERROR:
ORA-20080: you cannot decrease the sal
```

Schema Level Trigger:

- **If trigger is defined on SCHEMA then it is called "Schema Level Trigger".**
- **DBA defines it.**
- **It can be also called as DDL TRIGGER / SYSTEM TRIGGER.**

Syntax:

```
CREATE OR REPLACE TRIGGER <name>
BEFORE/AFTER <DDLs_list>
ON <user_name>.SCHEMA
DECLARE
   --declare the variables
BEGIN
   --statements
END;
/
```

Example:
Define a trigger to don't allow the c##batch9am user to drop any DB Object:

Login as DBA:

```
CREATE OR REPLACE TRIGGER st1
BEFORE drop
ON c##batch9am.SCHEMA
BEGIN
    raise_application_error(-20060,'you cannot drop any DB obj');
END;
/
```

Login as c##batch9am:

```
DROP TABLE emp;
Output:
ERROR:
ORA-20060: you cannot drop any Db obj

DROP PROCEDURE withdraw;
Output:
ERROR:
ORA-20060: you cannot drop any Db obj
```

System Variable	Purpose
ora_dict_obj_type	it holds object type Examples: TABLE, TRIGGER, PROCEDURE, VIEW
ora_dict_obj_name	it holds object name Examples: EMP, WITHDRAW, T1
ora_login_user	it holds current user name Example: C##BATCH9AM
ora_sysevent	it holds user action Examples: DROP, ALTER, TRUNCATE

Define a trigger to don't allow the c##batch9am user to drop table:

login as DBA:

```
CREATE OR REPLACE TRIGGER st1
BEFORE drop
ON c##batch9am.SCHEMA
BEGIN
    IF ora_dict_obj_type='TABLE' THEN
        raise_application_error(-20060,'you cannot drop table');
    END IF;
END;
/
```

Testing:
log in as c##batch9am:
drop table emp;
output:
error

Database Level Trigger:

- **If trigger is defined on DATABASE then it is called "Database Level Trigger".**
- **DBA defines it.**
- **It can be also called as DDL TRIGGER / SYSTEM TRIGGER.**

Syntax:

```
CREATE OR REPLACE TRIGGER <name>
BEFORE/AFTER <DDLs_list>
ON database
DECLARE
    --declare the variables
BEGIN
    --statements
END;
/
```

```
  END;
  /
```

Example on database level trigger:

define a trigger to don't allow c##batch9am, c##batch2pm, c##userA to drop any db object:

login as DBA:

```
CREATE OR REPLACE TRIGGER dt1
BEFORE drop
ON database
BEGIN
   IF user IN('C##BATCH9AM', 'C##BATCH2PM', 'C##USERA') THEN
      raise_application_error(-20060,'you cannot drop any db object');
   END IF;
END;
/
```

testing:

c##userA:

```
drop table t1;
output:
error:
ora-20060: you cannot drop any db object
```

user_triggers:

- it is a system table
- it maintains all triggers info

to see trigger info:

```
SELECT trigger_name, trigger_type, triggering_event, table_name
FROM user_triggers;
```

to see trigger code:

```
SELECT text
```

```
FROM user_source
WHERE name='T3';
```

Working with LOBs

Saturday, May 11, 2024 9:18 AM

Binary Related Data Types:

2 Types:

- **BFILE**
- **BLOB**

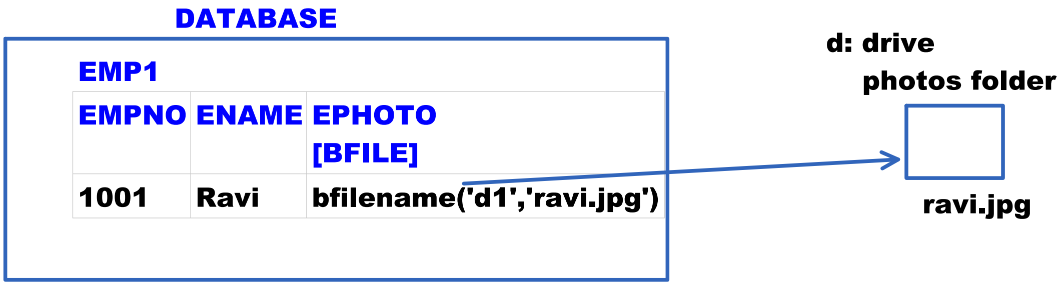
BFILE:

- **It is used to maintain multimedia objects like images, audios, videos, documents and animations.**
- **It is a pointer to multimedia object. It means, it maintains path of multimedia object.**
- **it can be also called as "External Large Object".**
- **It is not secured.**
- **max size: 4GB**

Example:

directory object d1

d1 => d:\photos



EMPNO	ENAME	EPHOTO [BFILE]
1001	Ravi	bfilename('d1','ravi.jpg')

Note:

- **bfilename() function is used to maintain multimedia object's path.**

  syntax:

  bfilename(<directory_object>, <file_name>)

directory_object:

- **directory object is a pointer to specific folder.**

Creating directory object:

Syntax:

```
CREATE DIRECTORY <name> AS <folder_path>;
```

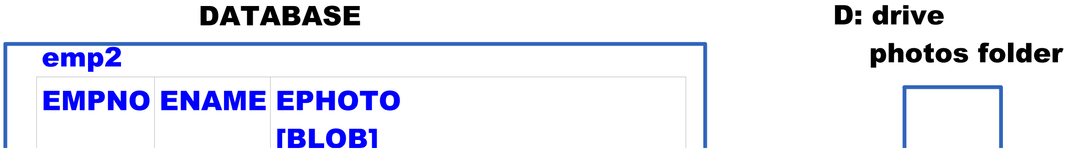
Example on BFILE:

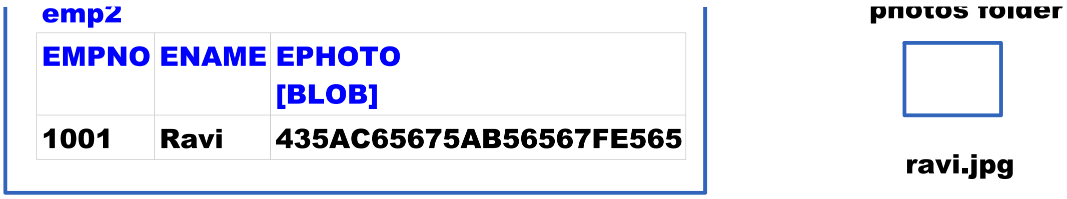
```
login as DBA:
    username: system
    password: nareshit

    CREATE DIRECTORY d1 AS 'D:\photos';

    GRANT read, write
    ON DIRECTORY d1
    TO c##batch9am;


login as c##batch9am:

    CREATE TABLE emp1
    (
    empno NUMBER(4),
    ename VARCHAR2(10),
    ephoto BFILE
    );

    INSERT INTO emp1
    VALUES(1001,'Ellison',bfilename('D1','Ellison.jpg'));

    COMMIT;
```

BLOB:

- **BLOB => Binary Large Object**
- **It is used to maintain multimedia objects like images, audios, videos ...etc.**
- **It is used to maintain multimedia object inside of table.**
- **It can be also called as "Internal Large Object".**
- **It is secured.**
- **max size: 4 GB**

Example:




emp2

EMPNO	ENAME	EPHOTO [BLOB]
1001	Ravi	435AC65675AB56567FE565

Example on BLOB:

```
CREATE TABLE emp2
(
empno NUMBER(4),
ename VARCHAR2(10),
ephoto BLOB
);

INSERT INTO emp2
VALUES(1001,'Ellison', empty_blob());
```

Define a procedure to update the photo:

```
EXEC update_photo(1001,'Ellison.jpg');

CREATE OR REPLACE PROCEDURE
update_photo(p_empno NUMBER, p_fname VARCHAR2)
AS
    s BFILE;                     --to hold image path
    t BLOB;                      --to hold image binary data
    length NUMBER;
BEGIN
    s := bfilename('D1',p_fname);          -- s => ellison image path

    SELECT ephoto INTO t FROM emp2
    WHERE empno=p_empno FOR UPDATE;     -- lock the record

    dbms_lob.open(s, dbms_lob.lob_readonly);     --s image opens in read mode

    length := dbms_lob.getlength(s);        --find image size 6638

    dbms_lob.LoadFromFile(t,s,length);       --reads 6638 bytes data from  s image and writes into t

    UPDATE emp2 SET ephoto=t WHERE empno=p_empno;

    COMMIT;

    dbms_output.put_line('image saved');
END;
/
```

DYNAMIC SQL

Saturday, May 11, 2024 10:36 AM

DYNAMIC SQL:

- **DRL, DML, TCL commands can be used directly in PL/SQL.**
- **DDL, DCL commands cannot be used directly in PL/SQL.
  to use them, we use DYNAMIC SQL.**
- **DYNAMIC SQL is used to execute DYNAMIC QUERIES.**
- **The query which is generated at runtime is called Dynamic Query.**

Static Query:

DROP TABLE emp;

Dynamic Query:

'DROP TABLE ' || n

if n='emp'
DROP TABLE emp

if n='dept'
DROP TABLE dept

EXECUTE IMMEDIATE 'DROP TABLE ' || n;

Note:

- **EXECUTE IMMEDIATE command is used to execute DYNAMIC QUERY.**

- **Submit DYNAMIC QUERY as string to EXECUTE IMMEDIATE command.**

Example program to demonstrate DYNAMIC SQL:

Define a procedure to drop the table:

```
CREATE OR REPLACE PROCEDURE
drop_table(n VARCHAR2)
AS
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE ' || n;
   dbms_output.put_line(n || ' table dropped');
END;
/
```

calling:

```
SQL> EXEC drop_table('salgrade');
```

Output:

```
salgrade table dropped
```