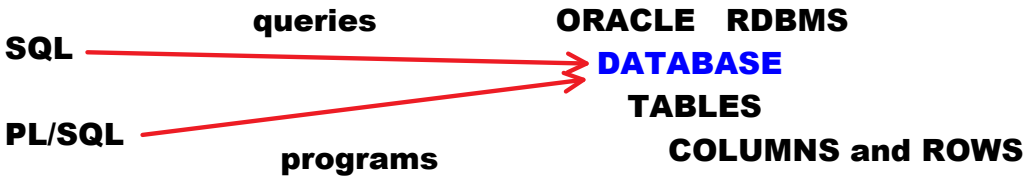
PL/SQL

Thursday, June 6, 2024 7:01 PM



SQL:

- **SQL => Structured Query Language**
- **it is a query language**
- **it is non-procedural language. [no programs]**
- **just we write queries to communicate with ORACLE DB**

PL/SQL:

- **PL/SQL => Procedural Language / Structured Query Language**
- **it is a programming language**
- **it is procedural language**
- **In this, we develop the programs to communicate with ORACLE DB.**

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

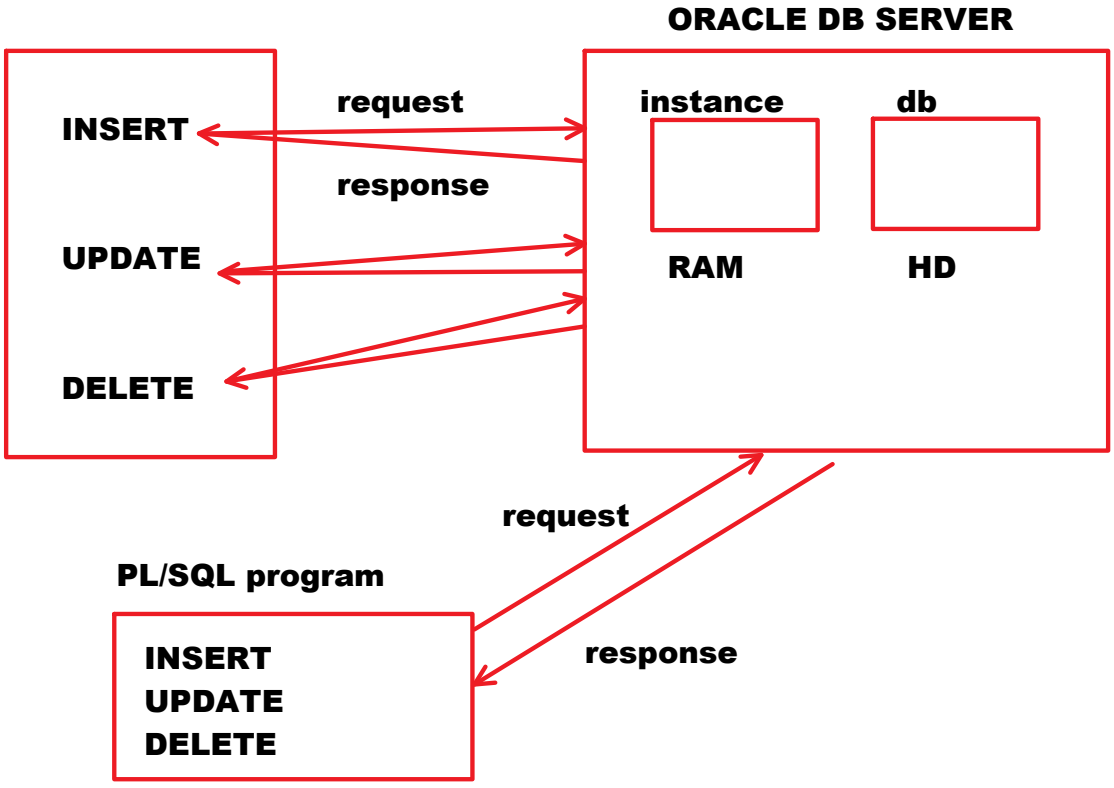
- **PL/SQL is extension of SQL**

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

Advantages:

- **improves the performance.**
- **provides control structures.**
- **provides exception handling.**
- **provides reusability.**
- **provides security.**

improves the performance:



We can group SQL queries in PL/SQL program and we can submit as 1 request. it decreases number of requests and responses. So, performance will be improved

provides control structures:

- PL/SQL provides conditional control structures like IF .. THEN, IF .. THEN .. ELSE.
- it provides looping control structures like FOR, WHILE, SIMPLE LOOP.

provides exception handling:

exception => runtime error
exception handling => the way of handling runtime errors

provides reusability:
we define procedure or function or package only once.
But, we can use it for any number of times by calling.

provides security:
only authorized users can call our procedures or functions or packages.

Types of Blocks:

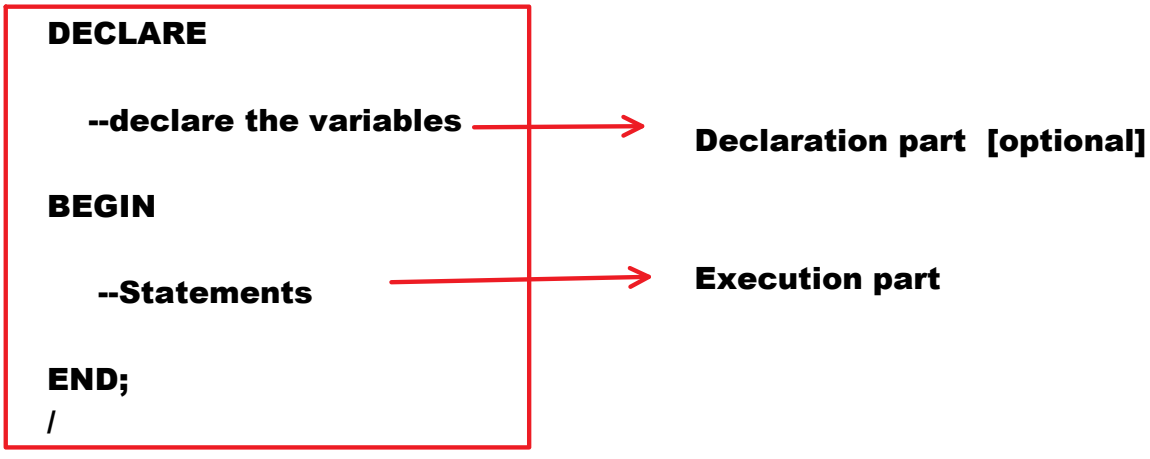
2 types:

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

Anonymous Block:
A block without name is called Anonymous block.

Named Block:
A block with name is called Named Block.
Examples:
procedures, functions, packages, triggers

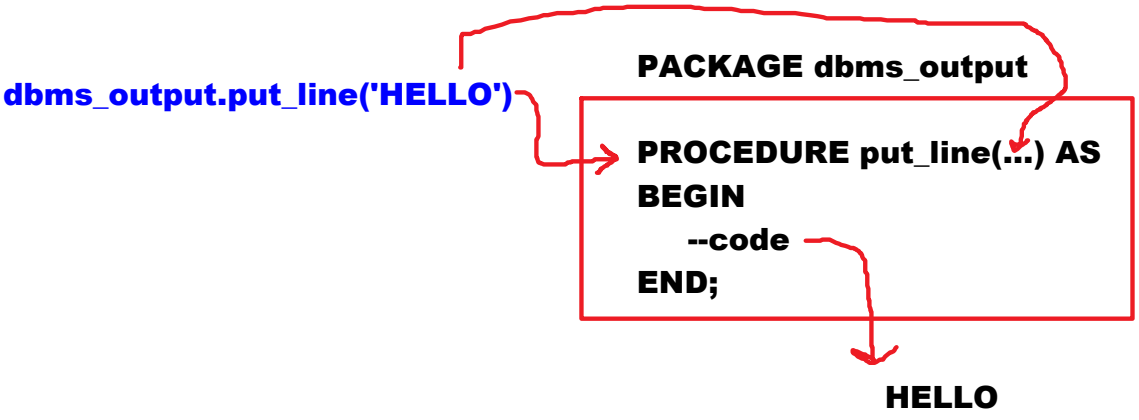
Syntax of Anonymous Block:



In C: printf("HELLO"); --function call

In Java: System.out.println("HELLO"); --method call

In PL/SQL: dbms_output.put_line('HELLO'); --procedure call



put_line():

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

Syntax to call packaged procedure:

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

Example:

```
dbms_output.put_line('HELLO');
```

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, notepad++.
- save it in D: Drive, batch6pm folder with the name HelloDemo.sql

Compiling and Running PL/SQL program:

- **open SQL PLUS**
- **login as user**

SQL> SET SERVEROUTPUT ON

SQL> @D:\batch6pm\HelloDemo.sql
Output:
HELLO

Note:
Syntax of compiling PL/SQL program:
@<path_of_file>

Syntax to run PL/SQL program:
/

Programming Basics:

data types
declare
assign
print
read
initialize

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 pls_integer PL/SQL only binary_integer PL/SQL only
floating point related	Number(p,s) float binary_float binary_double
Date & Time Related	Date Timestamp
Binary Related	BFILE BLOB
Attribute Related [pl/sql only]	%TYPE %ROWTYPE
Cursor related [pl/sql only]	SYS_REFCURSOR
Exception related [pl/sql only]	EXCEPTION

Variable:

- **Variable is an Identifier.**
- **Variable is a name of storage location.**
- **To hold the data variable is required.**
- **A variable can hold only 1 value at a time.**

Declaring variable:

Syntax:

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

Example:

```
x NUMBER(4);
y VARCHAR2(10);
z DATE;
```

Assigning value:

Assignment operator	:=

Syntax:

```
<variable> := <value>;
```

Example:

```
x := 25;
y := 'RAJU';
z := to_date('25-DEC-2023');
```

Printing data:

Example:

```
dbms_output.put_line(x);    --prints 25
dbms_output.put_line(y);    --prints RAJU
```

Reading data:

Example:

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

Output:

```
enter value for x: 25
```

Initializing variable:

if value is given at the time of declaration then it is called "Initialization".

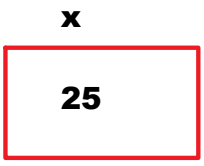
Syntax:

<variable> <data_type> := <value>;

x INT; --declare

x := 25; --assign

x INT := 25; --initialization



DECLARE	**x INT;**
ASSIGN	**x := 30;**
PRINT	**dbms_output.put_line(x);**
READ	**x := &x;**
INITIALIZE	**x INT := 30;**

Program to add 2 numbers:

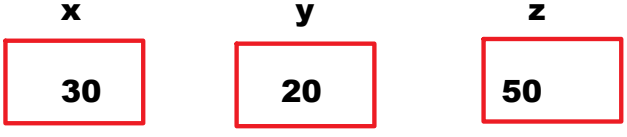
30 20

30+20 = 50

- **declare x,y,z as number type**
- **assign 30 to x**
- **assign 20 value to y**
- **calculate x+y and store it in z**
- **print z**

Program:

```
DECLARE
   x NUMBER(4);
   y NUMBER(4);
   z NUMBER(4);
```



```
BEGIN
    x := 30;
    y := 20;

    z := x+y;

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

Output:
sum=50

Program to add 2 numbers. read 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;
/
```

x	y	z
5	4	9

```
SQL> @d:\batch6pm\ReadDemo.sql
Output:
Enter value for x: 5
old   6:        x := &x;
new   6:         x := 5;
Enter value for y: 4
old   7:        y := &y;
new   7:         y := 4;
sum=9
```

To avoid old and new parameters we need to set VERIFY as OFF.

```
SQL>  SET VERIFY OFF

SQL> @d:\batch6pm\ReadDemo.sql
Output:
```

```
Enter value for x: 5
Enter value for y: 4
sum=9
```

Using SQL commands in PL/SQL program:

- DRL, DML, TCL commands can be used directly in PL/SQL program.
- DDL, DCL commands cannot be used directly in PL/SQL program.
  If we want 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
```

Example:

x	y
ALLEN	1600

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

INTO clause copies selected data to corresponding variables.

Program to display emp record of given empno:

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

v_empno	v_ename	v_sal
7900	JAMES	950

```
    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 .. empno: 7900
JAMES   950
```

Problem-1:
mismatch may be there between table column field size and variable size.

EMP TABLE
EMPNO NUMBER(4)

VARIABLE
v_empno NUMBER(2)

mismatch

Problem-2:
mismatch may be there between table column data type and variable data type.

EMP TABLE
EMPNO NUMBER(4)

VARIABLE
v_empno DATE

mismatch

To solve above problems, we use %TYPE.

%TYPE:

- **it is attribute related data type.**

- **it is used to declare the variable with table column's data type and field size.**

- **it avoids mismatch b/w field sizes of table column and variable.**
- **it avoids mismatch b/w data types of table column and variable.**

Syntax:

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

Examples:

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

this statement instructs that, take emp table's empno column's data type as v_empno variable's data type.

```
v_ename EMP.ENAME%TYPE;

v_sal EMP.SAL%TYPE;
```

Example on %type:

program to display 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;
/
```

Example:
Program to display accout balance of given acno:

ACCOUNTS

ACNO	NAME	BALANCE
1234	A	80000
1235	B	50000

```
create table accounts
(
acno number(4),
name varchar2(10),
balance number(9,2)
);

insert into accounts values(1234,'A',80000);
insert into accounts values(1235,'B',50000);
COMMIT;
```

Program:

```
DECLARE
   v_acno ACCOUNTS.ACNO%TYPE;
   v_balance ACCOUNTS.BALANCE%TYPE;
BEGIN
   v_acno := &acno;

   SELECT balance INTO v_balance FROM accounts
   WHERE acno=v_acno;

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

Output:

```
Enter value for acno: 1234
balance=80000
```

Program to find experience of given empno:

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

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

v_empno	v_hiredate	v_exp
7499	20-FEB-81	43

Output:

```
enter ... empno: 7499
experience=43 years
```

%ROWTYPE:

- It is attribute related data type.
- It is used to hold entire row of a table.
- This data type variable can hold 1 row at a time.
- It decreases number of variables.

Syntax:

```
<variable> <table_name>%rowtype;
```

Example:

```
r1 EMP%ROWTYPE;
```

r1

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	..	..	..	1600	..	..

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7499	ALLEN	..	..	..	1600	..	..

r1.ename

r1.job

r1.sal

r2 DEPT%ROWTYPE;

r2

DEPTNO	DNAME	LOC
20	RESEARCH	DALAS

r2.dname

r2.loc

Program to display the 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.job || '    ' || r.sal);
END;
/
```

v_empno

7499

r

empno	ename	job	mgr	hiredate	sal	comm	deptno
7499	ALLEN	..	..	..	1600	..	..

Using UPDATE command in PL/SQL:

Program to increase salary of given empno with given amount:

```
DECLARE                                   v_empno      v_amount
```

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
7900	2000

```
enter .. empno: 7900
enter .. 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;

   DELETE FROM emp WHERE empno=v_empno;

   COMMIT;

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

Using INSERT command in PL/SQL:

STUDENT

SID	SNAME	M1

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

Program to insert student record into student 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;
/
```

Control Structures

Monday, June 10, 2024 7:32 PM

Control Structures:

- **Control Structure is used to control the flow of execution of program.**
- **To change sequential execution and 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
```

Syntax:

```
IF <condition> THEN
    --Statements
END IF;
```

condition -> TRUE

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

Example:

program to delete emp record of given empno.
If experience is more than 42 years then only delete the 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 := TRUNC((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;
/
```

v_empno	v_hiredate	v_exp
7782	....	43

Output:

```
enter .. empno: 7782
experience=43
record deleted..
```

IF .. THEN .. ELSE:

Syntax:

```
IF <condition> THEN
   --Statements
ELSE
   --Statements
```

condition => TRUE

condition => FALSE

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

The statements in IF..THEN block get executed when the condition is TRUE.
The statements in ELSE block get executed when condition is FALSE.

Example:
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;     --7499

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

Output-1:

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

Output-2:

```
Enter value for empno: 7499
job=SALESMAN
10% on sal increased..
```

Example:

increase salary of given empno with given amount.
after increment if salary is more than 10000 then
cancel it:

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

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

   SELECT sal INTO v_sal FROM emp
   WHERE empno=v_empno;

   IF v_sal>10000 THEN
      ROLLBACK;
      dbms_output.put_line('rolled back');
   ELSE
      COMMIT;
      dbms_output.put_line('committed..');
   END IF;

END;
/
```

Assignment:
Program to increase salary of given empno based on deptno as following:
if emp is working in deptno 10 then increase 15% on sal
for others, increase 10% in sal

IF .. THEN .. ELSIF:

Syntax:

```
IF <condition1> THEN
   --Statements
ELSIF <condition2> THEN
   --Statements
ELSIF <condition3> THEN
   --Statements
.
.
ELSE
   --Statements
END IF;
```

conditon1 => T

c1 => F, c2 => T

c1,c2 => F, c3 => T

c1,c2,c3, ... => F

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

defining ELSE block is optional.

Example on IF .. THEN .. ELSIF:

Program to increase salary of given empno based on job as following:
if job is MANAGER then increase 20% on sal
CLERK 10%
others 5%

```
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 := 10;
   ELSE
      v_per := 5;
   END IF;
```

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

   COMMIT;

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

Assignment:

program to increase salary of given empno based on deptno.
if deptno is 10 then increase 10% on sal

20	20%
30	15%
others	5%

CASE:

CASE control structure can be used in 2 ways. They are:

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

Simple CASE:
it can check equality condition only

Searched CASE:
it can check any condition

Syntax of Simple CASE:

Syntax:

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

```
ELSE
   --Statements
END CASE;
```

Example on Simple Case:

program to check whether the given number is even or odd:

2,4,6,8, ...	divide with 2	remainder 0
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;
/
```

Searched CASE:

Syntax:

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

Example:

Program to check whether the given number is +ve or -ve or 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;
/
```

Assignment:

program to increase salary of given empno based on deptno.
use simple case:
if deptno is 10 then 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
   END IF;
END IF;
```

condition1, conditon2 => TRUE

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

Program to find total marks, avrg marks and result of given student id. And store these values in RESULT table:
max marks: 100
min marks: 40
if marks are <40 in any subject then result is fail.
if pass, check average.
if avrg is 60 or more => FIRST
if avrg is b/w 50 to 59 => SECOND
if avrg b/w 40 to 49 => THIRD

STUDENT

SID	SNAME	M1	M2	M3
1001	A	70	60	80
1002	B	55	30	64

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',70,60,80);
INSERT INTO student VALUES(1002,'B',55,30,64);
COMMIT;

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

Program:

```
DECLARE
   v_sid STUDENT.SID%TYPE;
   r1 STUDENT%ROWTYPE;
   r2 RESULT%ROWTYPE;
BEGIN
   v_sid := &sid;    --1001

   SELECT * INTO r1 FROM student WHERE sid=v_sid;
```

v_sid

1001

r1

SID	SNAME	M1	M2	M3
1001	A	70	60	80

```
   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(r1.sid, r2.total, r2.avrg, r2.result);
   COMMIT;

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

r2

SID	TOTAL	AVRG	RESULT
	210	70	FIRST

Looping Control Structures:

Looping Control Structure is used to execute the statements repeatedly.

```
WHILE 100000 times
LOOP
   d_o.p_l('hello');
END LOOP;
```

1 lkh stmts

```
d_o.p_l('hello');
d_o.p_l('hello');
d_o.p_l('hello');
.
.
d_o.p_l('hello');
d_o.p_l('hello');
d_o.p_l('hello');
```

100000 hellos

```
hello
hello
hello
.
.
hello
hello
hello
```

PL/SQL provides following Looping Control Structures:

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

WHILE:

Syntax:

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

The statements in WHILE loop get executed as long as the condition is TRUE.

Example:

program to print numbers from 1 to 4:

```
i
1
2
3
4
```

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

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

END;
/
```

```
i
1  2  3  4  5
```

```
i<=4
--------
1<=4   T      1
2<=4   T      2
3<=4   T      3
4<=4   T      4
5<=4   F
```

Simple Loop:

Syntax:

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

Example:

Program to print numbers from 1 to 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:

- 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.

Syntax:

```
EXIT WHEN <condition>;
```

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.

Syntax:

```
EXIT;
```

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

Output:

ERROR: EXIT must appear inside of loop

FOR:

Syntax:

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

Example:

Program to print numbers from 1 to 4:

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

```
i
1
2
3
4
```

Note:

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

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

```
BEGIN
   FOR i IN 1 .. 10
   LOOP
      i:=5;
      dbms_output.put_line(i);
   END LOOP;
END;
/
```

Output:

ERROR: i cannot be used as assignment target

- **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;
/
```

Note:
Till ORACLE 19c, step value will be always 1
From ORACLE 21C, we can specify step value using BY keyword

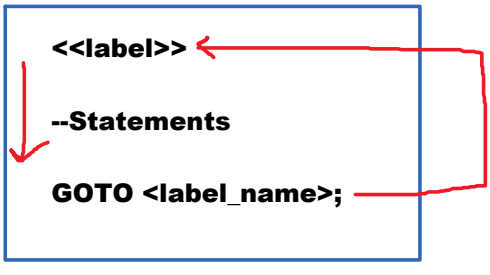
Program to print even numbers b/w 1 to 20:

```
   BEGIN
      FOR i IN 2 .. 20 BY 2
      LOOP
         dbms_output.put_line(i);
      END LOOP;
   END;
   /
```

GOTO:

- **when GOTO statement is executed, execution jumps to specified label.**

Syntax:



Example:
Program to print numbers from 1 to 4:

```
DECLARE
   i INT;
BEGIN
   i :=  1;
   <<xyz>>
      dbms_output.put_line(i);
      i:=i+1;
   IF i<=4 THEN
      GOTO xyz;
   END IF;
END;
/
```

Continue:

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

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;
/
```

CURSORS

Thursday, June 13, 2024 7:10 PM

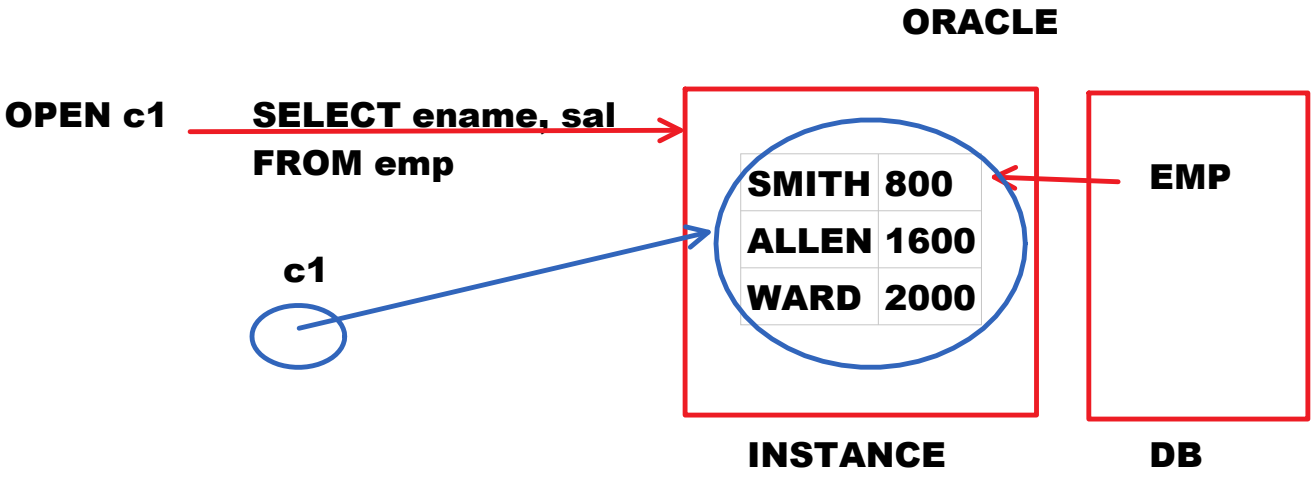
CURSOR:

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 %ROWTYPE.
To hold multiple rows we use CURSOR.

Every CURSOR is associated with SELECT QUERY.



- **CURSOR is a pointer to memory location which is INSTANCE.**
- **This memory location has multiple rows.**
- **To hold multiple rows and process them one by one we use CURSOR.**

Steps to use CURSOR:

4 steps:

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

DECLARING CURSOR:

Syntax:

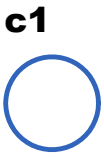
CURSOR <cursor_name> IS <SELECT QUERY>;

Example:

CURSOR c1 IS SELECT ename, sal FROM emp;

When CURSOR is declared,

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



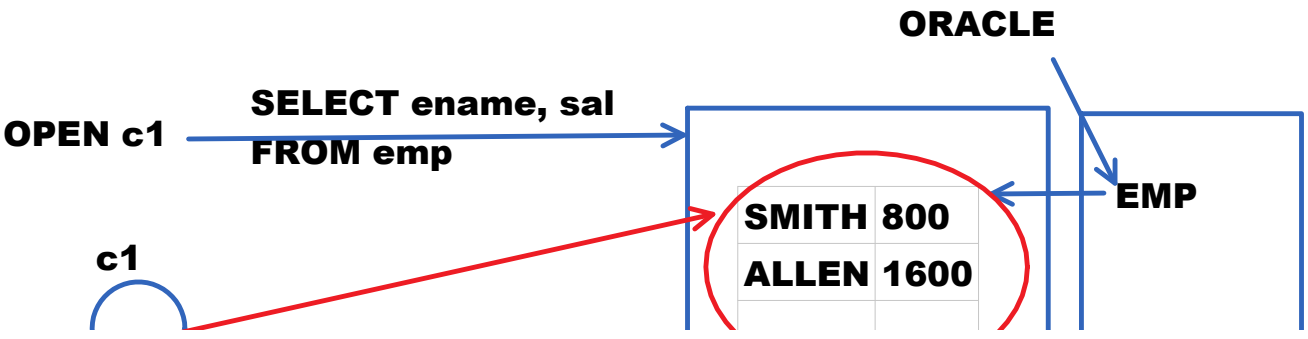
OPENING CURSOR:

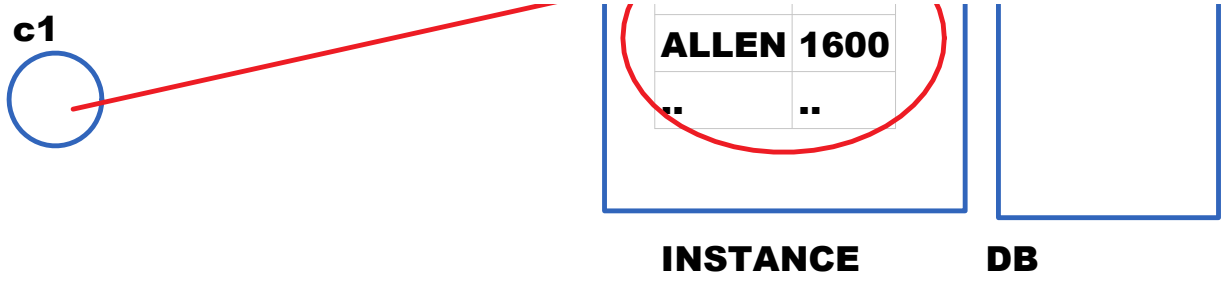
Syntax:

OPEN <cursor_name>;

Example:

OPEN c1;





When CURSOR is opened,

- SELECT query will be submitted to ORACLE
- ORACLE goes to DB
- Selects data
- Copies into some memory location
- This memory location address will be given to cursor variable.

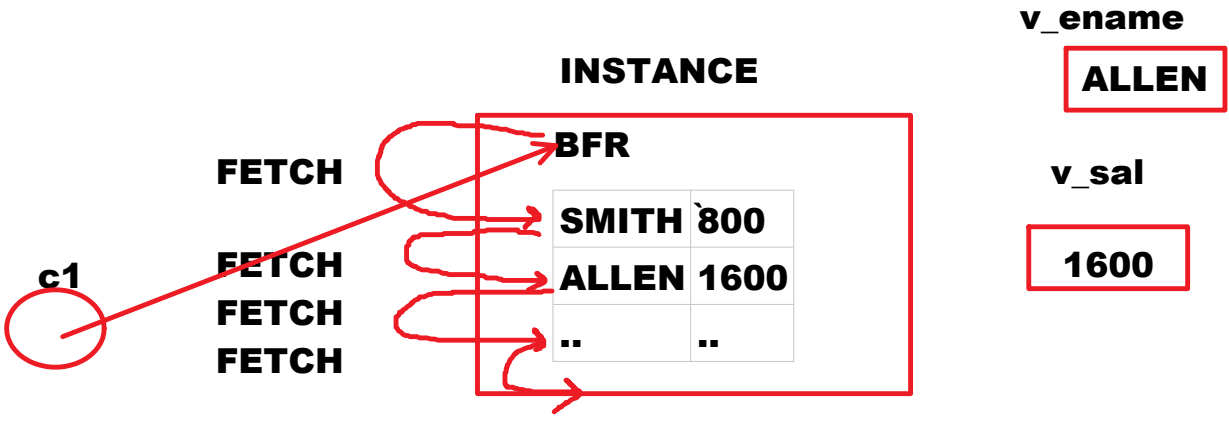
FETCHING RECORD FROM CURSOR:

Syntax:

FETCH <cursor_name> INTO <variables_list>;

Example:

FETCH c1 INTO v_ename, v_sal;



BFR => BEFORE FIRST ROW

when fetch is unsuccessful stop fetching

- When FETCH statement is executed, it fetches next row.
- One FETCH statement can fetch one row only. To fetch

multiple rows we write FETCH statement in LOOP.

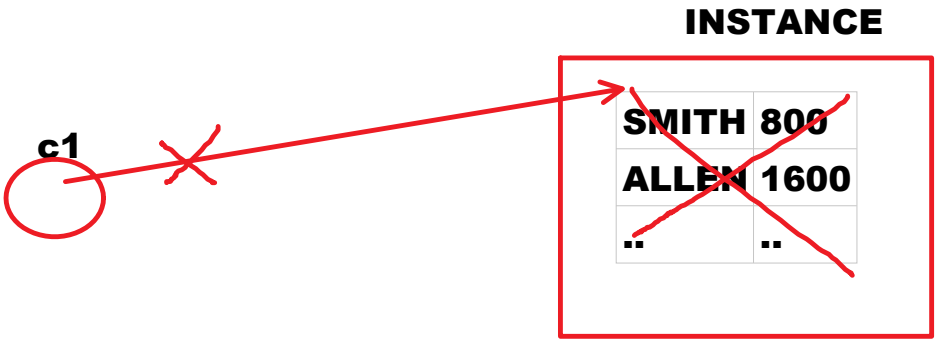
CLOSING CURSOR:

Syntax:

```
CLOSE <cursor_name>;
```

Example:

```
CLOSE c1;
```



When CURSOR is closed,

- **memory will be cleared**
- **reference will be gone.**

DECLARE	**CUROR c1 IS SELECT ename, sal FROM emp**
OPEN	**OPEN c1**
FETCH	**FETCH c1 INTO v_ename, v_Sal**
CLOSE	**CLOSE c1**

Cursor Attributes:

%FOUND
%NOTFOUND
%ROWCOUNT
%ISOPEN

Syntax:

<cursor_name><attribute_name>

Examples:

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

%FOUND:

- It returns boolean value [true or false].
- If record is found, it returns TRUE
- If record is not found, it returns FALSE

%NOTFOUND:

- It returns boolean value [true or false].
- If record is not found, it returns TRUE
- If record is found, it returns FALSE

%ROWCOUNT:

- its default value is 0.
- if record is found, rowcount value incremented by 1

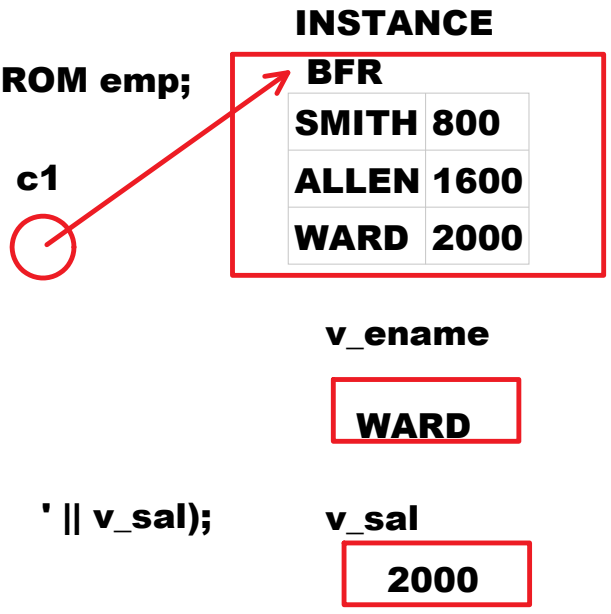
%ISOPEN:

- it returns boolean value.
- If cursor is opened, it returns TRUE.
- If cursor is not opened, it returns FALSE.

Examples on CURSOR:

Display all emp names and salaries:

```
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;
/
```




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

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

   CLOSE c1;
END;
/
```

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

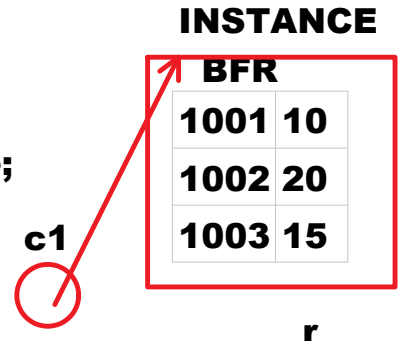
EMPLOYEE

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;
```




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

```
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;

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

   CLOSE c1;
END;
/
```

r

EMPNO	PER
1003	15

Program to find sum of salaries of all emps:

```
SAL
---------                    0 + 5000 + 10000 + 4000
5000
10000       v_sum:=v_sum+v_sal
4000
```

INSTANCE

BFR
5000
10000
4000

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

   LOOP
      FETCH c1 INTO v_sal;
```

c1

v_Sal	v_sum
5000	0
10000	5000

```
   LOOP
      FETCH c1 INTO v_sal;
      EXIT WHEN c1%NOTFOUND;
      v_sum := v_sum+v_sal;
   END LOOP;

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

   CLOSE c1;
END;
/
```

```
5000
10000
4000
```

```
0
5000
15000
19000
```

Output:
sum=19000

Cursor For 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 loop variable. it will be declared implicitly as %ROWTYPE.**

Syntax:

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

Example:

```
FOR r IN c1
LOOP
   dbms_output.put_line(r.ename);
END LOOP;
```

Example on Cursor For Loop:

Display all emp names and salaries:

```
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 SEELCT 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:

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

Parameterized Cursor:

- **A cursor which is declared using parameters is called "Parameterized Cursor".**
- **When we open the cursor we pass parameter value.**

Syntax:

```
CURSOR <cursor_name>(<parameters_list>) IS <SELECT QUERY>;
```

Example on parameterized cursor:

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

   r EMP%ROWTYPE;
BEGIN
   OPEN c1(20);

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

END;
/
```

Ref Cursor:

Simple Cursor:

```
c1 => SELECT * FROM emp
c2 => SELECT * FROM dept
c3 => 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. WHERE AS in Ref Cursor, Same cursor can be used for multiple SELECT QUERIES.**

- **REF CURSOR has a data type. i.e: SYS_REFCURSOR.**

Declaring Ref Cursor:

Syntax:

```
<cursor_name> SYS_REFCURSOR;
```

Example:

```
c1 SYS_REFCURSOR;
```

Opening Ref Cursor:

Syntax:

```
OPEN <cursor_name> FOR <SELECT QUERY>;
```

Example:

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

Example on Ref Cursor:

Program to display emp table records and 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%notfound;
        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 b/w Simple Cursor and Ref Cursor:

Simple Cursor	Ref Cursor
• in this, one cursor can be used for one select query only	• in this, same cursor can be used for multiple select queries
• it is static. select query is fixed.	• it is dynamic. select query can be changed.
• no data type	• it has data type. i.e: SYS_REFCURSOR
• in this, select query will be specified at the time of declaration	• in this, select query will be specified at the time of opening cursor.
• it cannot be used as procedure parameter. Because, it has no data type.	• it can be used as procedure parameter. because, it has data type.

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. it is called "Implicit Cursor".**
- **This Implicit Cursor name is: SQL.**

SQL%FOUND
SQL%NOTFOUND
SQL%ROWCOUNT
SQL%ISOPEN

Example on Implicit Cursor:

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 to 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;

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

END;
/
```

CURSOR GOAL:
to hold multiple rows and process them one by one

4 steps:

DECLARE	**CURSOR c1 IS SELECT * FROM emp**
OPEN	**OPEN c1**
FETCH	**FETCH c1 INTO r**
CLOSE	**CLOSE c1**

Cursor For loop:
- **no need to open, fetch and close**

Inline Cursor:
- **we specify select query in cursor for loop**

Parameterized cursor:
- **a cursor with parameter**

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

OPEN c1(20);
```

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

Saturday, June 15, 2024 7:23 PM

Exception Handling:

Exception [problem]	**Run Time Error**
Exception Handling [solution]	**The way of handling run time errors**

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 at run time. it means, these errors occur during program execution.**
- **These errors occur due to several reasons as following:**
  - **divide with 0**
  - **when record is not found**
  - **when size is exceeded**
  - **when we give wrong input**
  - **if we insert duplicate value in PRIMARY KEY**

Problem:

- **abnormal termination. it leads to wrong results.**

we may loss the data.

Logical Errors:

- These errors occur due to mistake logic.
- it leads to wrong results.
- programmer must develop correct logic.

Example:

withdraw:
v_balance := v_balance + v_amount

Exception:

- Exception means Run Time Error.
- it is a problem
- When run time error occurs our program will be terminated in the middle of execution. So, abnormal termination will occur.
- Abnormal termination leads to wrong results or invalid data.
- That is why we must handle the runtime errors.

Exception Handling:

- The way of handling runtime errors is called "Exception Handling".
- For Exception handling we add EXCEPTION block.

Syntax of Exception Handling:

```
DECLARE
   --declare the variables
BEGIN
   --executable statements
   EXCEPTION
      WHEN <Exception_name> THEN
         --Handling code
      WHEN <Exception_name> THEN
         --Handling code
      .
```

```
        --Handling code
    .
    .
END;
/
```

Example on Exception Handling:

Write a 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('you cannot divide with 0');
      WHEN value_error THEN
         dbms_output.put_line('size is exceeded or wrong input given');
END;
/
```

Output-1:

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

Output-2:

```
Enter value for x: 20
```

Enter value for y: 0
you cannot divide with 0

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

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

Types of Exceptions:

2 Types:
- **Built-In Exception**
- **User-Defined Exception**

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

Examples:
zero_divide
value_error
no_data_found
dup_val_on_index
too_many_rows
invalid_cursor
cursor_already_open

zero_divide:
- **This exception will be raised when we try to divide with 0.**

value_error:

- **when we give wrong input or when size is exceeded then value_error will occur.**

no_data_found:

- **when we retrieve the data if record is not found 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 emp exited with
         this empno');
END;
/
```

Output-1:

```
Enter value for empno: 7698
BLAKE   8104
```

Output-2:

```
Enter value for empno: 9123
no emp exited with this empno
```

dup_val_on_index:

when we try to insert duplicate value in Primary Key column then dup_val_on_index exception will be raised.

Example:

```
CREATE TABLE student
(
sid NUMBER(4) CONSTRAINT con80 PRIMARY KEY,
sname VARCHAR2(10)
);
```

Program to insert student record into table:

```
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('this sid already assigned..');
END;
/
```

too_many_rows:

- **when select query selects multiple rows then too_many_rows exception will be raised.**

Example on too_man_rows:

Display the emp record 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 rows selected..');
END;
/
```

Output-1:

```
Enter value for job: PRESIDENT
KING    PRESIDENT    12000
```

Output-2:

```
Enter value for job: CLERK
many rows selected..
```

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 is not opened..');
END;
/
```

Output:

```
cursor is 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:

Display all emp records:

```
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..

Others:

- **It can handle any exception.**

Example on Others:

```
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 others THEN
         dbms_output.put_line('RTE occurred..');
END;
/
```

Output-1:

```
Enter value for x: 20
Enter value for y: 0
RTE occurred..
```

Output-2:

```
Enter value for x: 123456
Enter value for y: 2
RTE occurred..
```

Output-3:

```
Enter value for x: 'RAJU'
Enter value for y: 2
RTE occurred..
```

SQLERRM	• is a built-in function • it returns error message
SQLCODE	• is a built-in function • it returns error code

Example on SQLERRM and SQLCODE:

```
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 others THEN
         dbms_output.put_line(SQLERRM);
         --dbms_output.put_line(SQLCODE);
         --dbms_output.put_line('RTE occurred..');
END;
/
```

User-Defined Exception:

- An exception which is defined by user is called "User-Defined Exception".
- We define it explicitly and we raise it explicitly.

Built-In Exception:

1 step:
HANDLE

User-Defined exception:

3 steps:

- **DECLARE**
- **RAISE**
- **HANDLE**

For user-defined exception follow 3 steps.

They are:

- **DECLARE**
- **RAISE**
- **HANDLE**

Declaring user-defined exception:

- **to declare the exception we use EXCEPTION data type.**

Syntax:

```
<exception_name> EXCEPTION;
```

Examples:

```
one_divide EXCEPTION;
xyz EXCEPTION;
sunday_not_allow EXCEPTION;
comm_is_null EXCEPTION;
```

RAISING user-defined exception:

- **RAISE keyword can be used to raise the exception**

Syntax:

```
RAISE <exception_name>;
```

Examples:

```
RAISE one_divide;
RAISE xyz;
```

Handling user-defined exception:

Syntax:

```
EXCEPTION
   WHEN <Exception_name> THEN
      --handling code
```

Example:

```
EXCEPTION
   WHEN one_divide THEN
      dbms_output.put_line('denominator cannot be 1');
```

Example on user-defined exception:

program to divide 2 numbers. if denominator is 0 run time error will occur handle it. if denominator is 1 then 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('denominator cannot be 0');
      WHEN one_divide THEN
         dbms_output.put_line('denominator cannot be 1');

END;
/
```

Output-1:

```
Enter value for x: 10
Enter value for y: 0
denominator cannot be 0
```

Output-2:

```
Enter value for x: 10
Enter value for y: 1
denominator cannot be 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 FLOAT;
   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('sal cannot be increased on SUNDAY..');
END;
/

Output-1 [mon-sat]:
Enter value for empno: 7934
Enter value for amount: 1000
sal increased..
```

Output-2 [on Sunday]:

```
Enter value for empno: 7934
Enter value for amount: 1000
sal cannot be increased on SUNDAY..
```

Note:

- we can raise the exception in 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 exception with our own error code and error message.
- our own code must be ranges from -20000 to -20999

Syntax:

```
RAISE_APPLICATION_ERROR(<user_defined_error_code>,
<user_Defined_error_message>);
```

Example:

```
RAISE_APPLICATION_ERROR(-20050,'you cannot divide with 1')
```

Output:

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

Example on raise_application_error():

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_amount FLOAT;
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-1 [on Sunday]:
ORA-20050: you cannot update on sunday..

What are the differences b/w RAISE and RAISE_APPLICATION_ERROR():

RAISE	RAISE_APLLICATION_ERROR()
• it is a keyword • it raises exception using name	• it is a procedure • it raises exception using code

pragma exception_init():

-1476	Error Code
divisor is equal to zero	Error Message
zero_divide	Error name

-1	Error Code
unique constraint violated	Error Message
dup_val_on_index	Error name

-2290	Error Code
check constraint violated	Error Message
NO ERROR NAME AVAILABLE	..

- some errors have names. some errors does not have names. To give name for unnamed exceptions we use pragma exception_init().

  Syntax:

  pragma exception_init(<user_Defined_exception_name>, <built-in_error_code>)

  Example:

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

- pragma exception_init() is called compiler directive. It is an instruction to the compiler. It instructs that before compiling program , first execute this line. directive = command / instruction

Example on pragma exception_init():

Program insert a record into student table:

```
CREATE TABLE student
(
sid NUMBER(4) CONSTRAINT c90 PRIMARY KEY,
sname VARCHAR2(10),
m1 NUMBER(3) CONSTRAINT c91 CHECK(m1 BETWEEN 0 AND
100)
);
```

Program:

```
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 duplicates..');
      WHEN check_violate THEN
         dbms_output.put_line('marks must be b/w 0 to 100');
END;
/
```

Output-1:

```
Enter value for sid: 1005
Enter value for sname: E
Enter value for m1: 78
record inserted..
```

Output-2:

```
Enter value for sid: 1005
Enter value for sname: F
Enter value for m1: 90
PK does not accept duplicates..
```

Output-3:

```
Enter value for sid: 1006
Enter value for sname: G
Enter value for m1: 567
marks must be b/w 0 to 100
```

STORED PROCEDURES

Wednesday, June 19, 2024 6:49 PM

PROCEDURE:

- **PROCEDURE is one ORACLE DB Object.**
- **PROCEDURE is a named block of statements that gets executed on calling.**

Types of Procedures:

2 Types:

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

Stored Procedure:

- **A procedure which is defined in SCHEMA [user] is called "Stored Procedure".**

Example:
SCHEMA c##batch6pm
PROCEDURE withdraw => Stored Procedure

Packaged Procedure:

- **A procedure which is defined in PACKAGE is called "Packaged Procedure".**

Example:
SCHEMA c##batch6pm
PACKAGE bank
PROCEDURE withdraw => Packaged Procedure

Syntax to define Sored Procedure:

```
CREATE [OR REPLACE] PROCEDURE
<name>[(<parameters_list>)]
```

→ **procedure header / procedure specification**

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

procedure header / procedure specification

procedure body

procedure = header + body

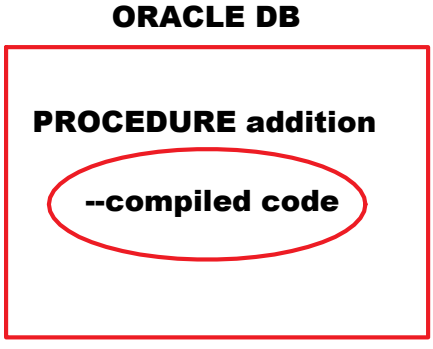
Define a procedure to add 2 numbers:

```
CREATE 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 any text editor like notepad.
- save it in d: drive, batch6pm folder, with the name ProcedureDemo.sql.

Open SQL PLUS
Login as user

```
SQL> @d:\batch6pm\ProcedureDemo.sql
```



Calling Stored Procedure:

- **from SQL prompt**
- **from PL/SQL program**
- **from programming language like JAVA, C#, PYTHON.**

- **from SQL prompt:**

Syntax:

```
EXEC[UTE] <procedure_name>(<parameters>);
```

Example:

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

from PL/SQL program [main program]:

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

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

Output:
Enter .. a: 2
Enter .. b: 3
sum=5
```

Note:
TO modify existing procedure's code we use REPLACE.

```
CREATE OR REPLACE PROCEDURE                HEADER
addition(x NUMBER, y NUMBER)          -- x,y => parameters
AS
```

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

Parameter:

- **Parameter is a local variable which 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 mods. they are:**
  - **IN**
  - **OUT**
    IN OUT

IN:

- **It is default one.**
- **It 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.**

Example:

```
CREATE OR REPLACE PROCEDURE
addition(x IN NUMBER, y IN NUMBER)
AS
   z NUMBER(4);
BEGIN
   x := 500;
   z := x+y;
   dbms_output.put_line('sum=' || z);
END;
/
```

Output:
ERROR: Procedure created with compilation errors

to see errors:
SQL> SHOW ERRORS

OUT:
- **it 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:
- **it captures input and sends output.**
- **same parameter takes input and sends 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 out of the 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 s NUMBER
SQL> EXEC addition(10,20,:s);
SQL> PRINT s
Output:
30

Note:
Bind variable:
- **A variable which is declared at SQL command prompt is called**

"Bind Variable".

- to write data into bind variable we use bind operator : [colon].
- PRINT command is used to print bind variable value

Syntax to declare bind variable:

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

Example:

```
VAR a NUMBER
```

Calling from PL/SQL 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;
/
```

Example:

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(7499, 2000);
Output:
sal increased..
```

Example:

Define a procedure to increase salary of specific employee.
After increment, increased salary send 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(7934,1000,:s);
Output:
sal increased..

SQL> PRINT s
```

Example on IN OUT parameter:

Define a procedure to find square of a number:

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

SQL> VAR a NUMBER
SQL> EXEC :a := 5;
SQL> PRINT a
Output:
5
SQL> EXEC SQUARE(:a);
SQL> PRINT a
Output:
25
```

Example:

Define a procedure to perform withdraw operation:

ACCOUNTS

ACNO	NAME	BALANCE
1001	A	70000
1002	B	50000

```
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',50000);

COMMIT;
```

```
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 funds..');
   END IF;

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

   COMMIT;

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

Output-1:

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

Output-2:

```
SQL> EXEC withdraw(1001,10000);
Output:
transaction successful..
```

Define a procedure to perform deposit operation:

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

```
   COMMIT;

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

Calling:

```
SQL> EXEC deposit(1001,30000);
```

Output:

```
transaction successful..
```

Define a procedure to perform deposit operation. After depositing, send the 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(1001,10000,:b);
transaction successful..

SQL> print b

         B
----------
   100000
```

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

x,y => Formal Parameters

Calling:

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

10,20 => Actual Parameters

Formal Parameter:

- A parameter which is declared in procedure header

Actual Parameter:

- A parameter in procedure call

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 this, actual parameter will be mapped with formal parameter based on position.

Example:

addition(x NUMBER, y NUMBER, z NUMBER)

positional mapping

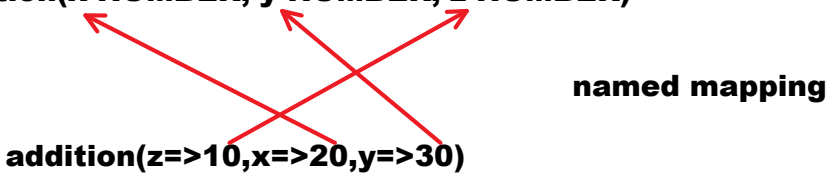
addition(10,20,30)

Named mapping:

- **in this, actual parameter will be mapped with formal parameter based on name.**

Example:

addition(x NUMBER, y NUMBER, z NUMBER)



named mapping

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

Mixed mapping:

- **in this, actual parameters will be mapped with formal parameters based on positions and names.**

Example:

addition(x NUMBER, y NUMBER, z NUMBER)

mixed mapping

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

addition(z=>10,20,30)
ERROR: a positional mapping may not follow named mapping

Example on parameter mapping techniques:

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;
```

/

Calling:

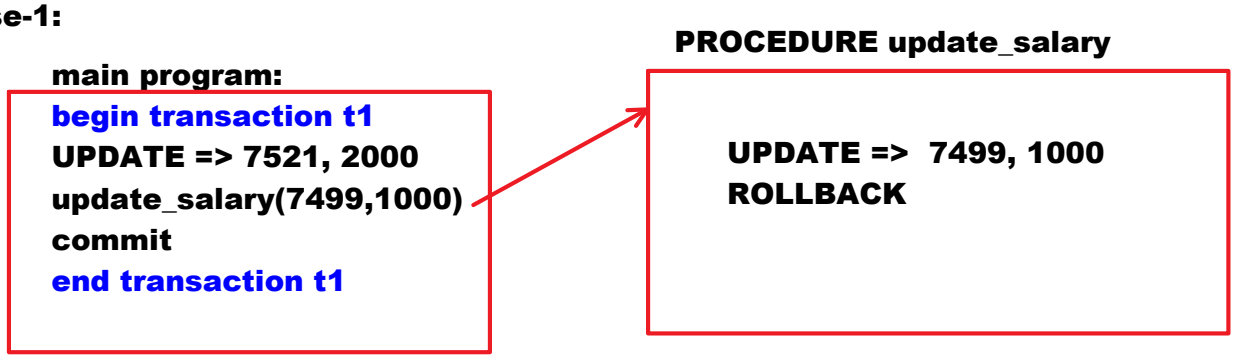
```
SQL> EXEC addition(10,20,30);
Output:
sum=60
x=10
y=20
z=30
```

```
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
```

Pragma Autonomous_Transaction:

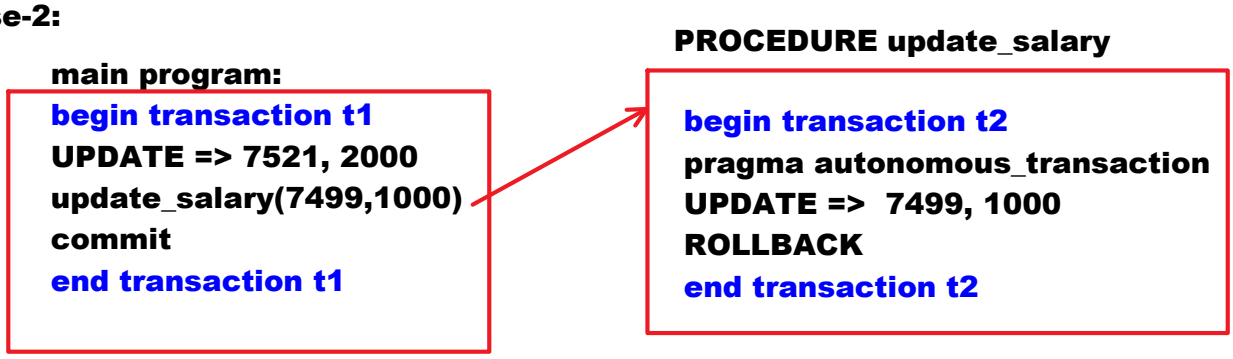
case-1:



EMP

EMPNO	ENAME	SAL
7499	ALLEN	1600+1000 = 2600 rolled back 1600
7521	WARD	2000+2000 = 4000 rolled back 2000

case-2:



EMP

EMPNO	ENAME	SAL
7499	ALLEN	1600+1000 = 2600 rolled back 1600
7521	WARD	2000+2000 = 4000 committed 4000

```
CREATE OR REPLACE PROCEDURE
update_salary(p_empno NUMBER, p_amount NUMBER)
AS
   pragma autonomous_transaction;
BEGIN
   UPDATE emp SET sal=sal+1000
   WHERE empno=p_empno;

   ROLLBACK;
END;
/

BEGIN
   UPDATE emp SET sal=sal+2000 WHERE empno=7521;

   update_salary(7499,1000);

   COMMIT;
END;
/
```

Note:
procedure actin cancelled
main action committed

- **by default a separate transaction will not be created for a procedure.**
- **a transaction started in main program will be**

continued to procedure.

- to create a separate transaction for procedure we use "pragma autonomous_transaction".

Granting permission on procedure to other users:

login as c##batch6pm:

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

login as c##userA:

```
EXEC c##batch6pm.addition(2,3,4);
```

Dropping procedure:

Syntax:

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

Example:

```
DROP PROCEDURE deposit;
```

user_procedures
user_source

user_procedures:

- it is a system table.
- it maintains all procedures, functions and packages information.

to see procedures list:

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

user_source:

- **it is a system table.**
- **it maintains all procedures, functions, packages and triggers information including code.**

to see list of procedures:

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

to see procedure's code:

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

STORED FUNCTIONS

Saturday, June 22, 2024 6:36 PM

FUNCTION:

- **FUNCTION is a named block of statements that gets executed on calling.**
- **It can be also called as "Sub Program".**

Types of Functions:

2 types:

- **Stored Functions**
- **Packaged Functions**

Stored Function:

A function which is defined in SCHEMA

Example:

SCHEMA c##batch6pm

FUNCTION check_balance => stored function

Packaged Function:

A function which is defined in PACKAGE

Example:

SCHEMA c##batch6pm

PACKAGE bank

FUNCTION check_balance => packaged function

Note:

- **To perform DML operations define PROCEDURE.**
- **To perform calculations or fetch operations define FUNCTION.**

Example:

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

find experience  => calculation => FUNCTION
account balance => fetch        => FUNCTION
transactions statement => fetch => FUNCTION
```

Syntax to define stored function:

```
CREATE OR REPLACE FUNCTION
<name>(<parameters_list>) RETURN <return_type>
AS
BEGIN
   --Statements
   RETURN <expression>;
END;
/
```

Note:

- **In FUNCTION, always take IN parameters only.**
- **Don't take OUT parameters.**

Example on defining stored function:

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

Calling from SQL prompt:

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

Calling PL/SQL program [main program]:

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

   c := product(a,b);

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

Example:

Define a function to find experience of an employee:

```
CREATE 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;
/
```

Calling:

```
SQL> SELECT experience(7788) FROM dual;
```

Output:

```
EXPERIENCE(7788)
----------------
            41
```

```
select lower(ename) as ename,
hiredate, experience(empno) as experience
FROM emp;
```

Define a function to check account balance:

ACCOUNTS

ACNO	NAME	BALANCE
1001	A	100000
1002	B	50000

```
CREATE FUNCTION
check_balance(p_acno NUMBER) 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:
SQL> select check_balance(1002) FROM dual;
Output:
CHECK_BALANCE(1002)
-------------------
            50000
```

define a function to display emp records of specific dept:

```
CREATE 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:
SELECT getdept(10) FROM dual;
```

Differences procedures and functions:

PROCEDURE	FUNCTION
procedure may or may not return the value.	function returns the value
returning value is optional	returning value is mandatory
to return the value we use OUT parameter.	to return the value we use RETURN keyword.
it can return multiple values.	it can return 1 value only.
it cannot be called from SQL command.	it can be called from SQL command
to perform DML operations define procedure. Example: PROCEDURE withdraw	to perform fetch operations or calculations define function. Example: FUNCTION check_balance

Can we perform DML operations through FUNCTION?
YES. it is not recommended
If we perform DML operation in FUNCTION, it cannot be

called from SELECT command.

Can we define OUT parameters in FUNCTION?
Yes. it is not recommended.
It is against to function standard

Dropping Function:

Syntax:

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

Example:

```
DROP FUNCTION experience;
```

user_procedures
user_source

user_procedures:

- **it is a system table.**
- **it maintains all procedures, functions and packages information.**

to see functions list:

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

user_source:

- **it is a system table.**

- **it maintains all procedures, functions, packages and triggers information including code.**

to see list of procedures:

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

to see function's code:

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

Procedure or Function can be also called as Sub program.

Advantages of Sub Program:

- **improves the performance**
- **provides reusability**
- **decreases length of code**
- **provides security**
- **improves understandability**
- **better maintenance**

PACKAGES

Monday, June 24, 2024 6:21 PM

PACKAGE:

- **PACKAGE is one ORACLE DB Object.**
- **PACKAGE is a collection of procedures, functions, data types, exceptions, global variables and cursors.**
- **it is used to group related procedures and functions.**

Creating Package:

2 steps:

- **Package Specification**
- **Package Body**

Package Specification:

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

In PACKAGE specification we declare procedures and functions.

Package Body:

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

```
    --define body of the functions
END;
/
```

In package bode, we define body of procedures and functions.

Example on creating 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 clal packaged procedure or packaged function:

```
   <package_name>.<procedure/function_name>(<parameters>)
```

Calling:

Calling Packaged procedure:

```
EXEC math.addition(5,10);
Output:
sum=15
```

Calling Packaged function:

```
SELECT math.product(5,10) FROM dual;
Output:
50
```

Example:

PACKAGE HR

```
PROCEDURE hire => insert
PROCEDURE fire  => delete
PROCEDURE hike => update

FUNCTION experience => calc
```

--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:

```
EXEC hr.hire(1001,'A');

EXEC hr.fire(1001);

EXEC hr.hike(7499,1000);

SELECT hr.experience(7499) FROM dual;
```

Assignment:

ACCOUNTS

ACNO	NAME	BALANCE
1234	A	70000
1235	B	50000

PACKAGE BANK

```
PROCEDURE opening_account => insert
PROCEDURE closing_account  => delete
PROCEDURE withdraw         => update
PROCEDURE deposit          => update

FUNCTION check_balance     => select
```

Advantages of Packages:

- we can group related procedures and functions.
- improves the performance.
- we can declare global variables.
- Stored procedure or stored function cannot be overloaded WHERE AS packaged procedure or

packaged function can be overloaded.

- We can make members as public or private.

- provides reusability
- decreases length of code
- provides security
- improves understandability
- better maintenance

Overloading:
Defining multiple functions / procedures with same name and different signatures is called "Overloading".

Example:
PACKAGE OLDEMO

```
global variable x=500
FUNCTION addition(x NUMBER, y NUMBER)
FUNCTION addition(x NUMBER, y NUMBER, z NUMBER)
```

```
--Package Specification

CREATE OR REPLACE PACKAGE OLDEMO
AS
   x NUMBER(3) := 500;
   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:
SQL> SELECT oldemo.addition(1,2) from dual;

OLDEMO.ADDITION(1,2)
--------------------
                   3

SQL> SELECT oldemo.addition(1,2,3) from dual;

OLDEMO.ADDITION(1,2,3)
----------------------
                     6

SQL> EXEC dbms_output.put_line(oldemo.x);
500
```

Example:

PACKAGE SPECIFICATION

PACKAGE demo

```
PROCEDURE p2
PROCEDURE p3
```

PACKAGE BODY demo

```
PROCEDURE p1
PROCEDURE p2
PROCEDURE p3
```

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

We are defining package specification means, we are making members as public.

```
--PACKAGE SPECIFICATION

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

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;
/
```

Dropping package:

Syntax:
DROP PACKAGE <name>;

Example:
DROP PACKAGE hr;

user_prcoedures:
it maintains all procedures, functions and packages info.

to see packages list:

```
column object_name format a15  --reduces column width

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

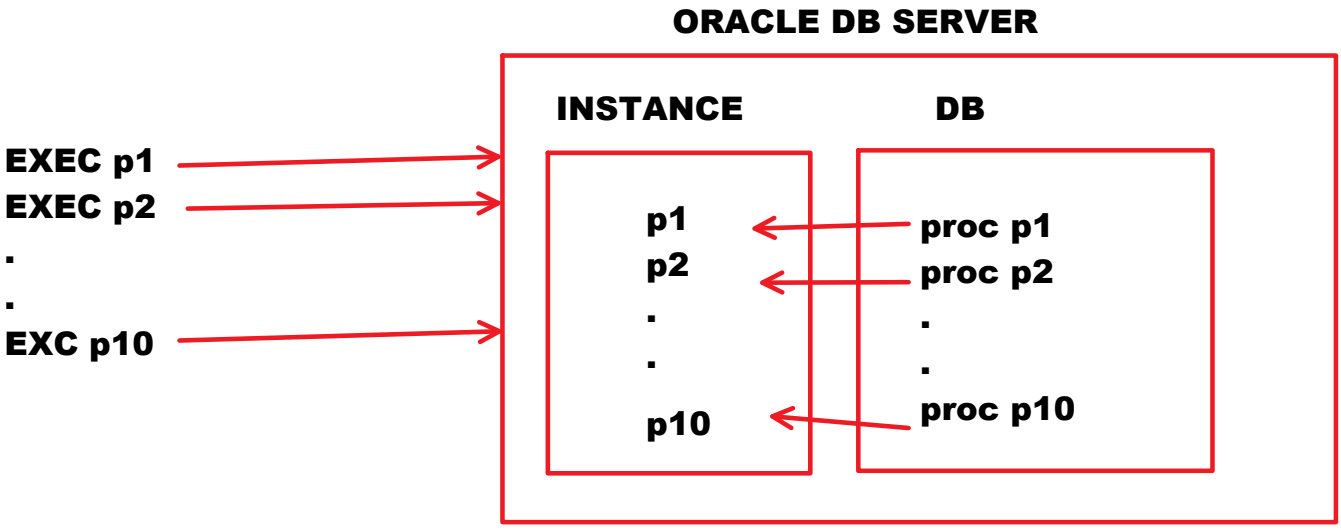
user_source:
it maintains procedures, functions,
packages and triggers info including code.

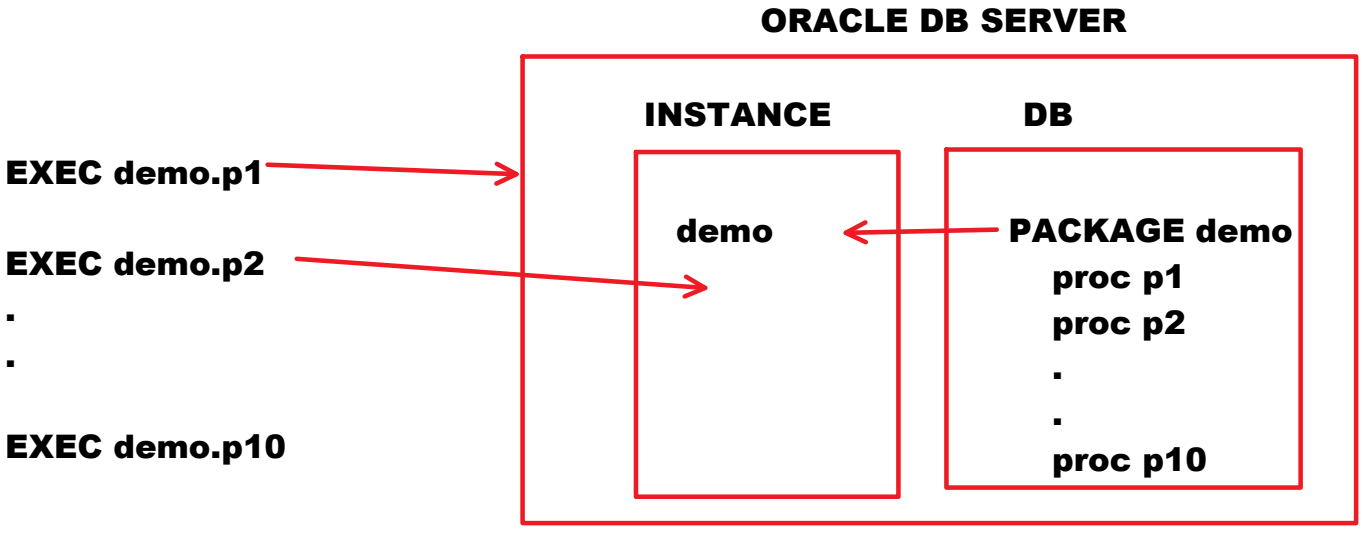
to see packages list:

```
SELECT DISTINCT name
FROM user_source
WHERE type='PACKAGE';
```

to see package code:

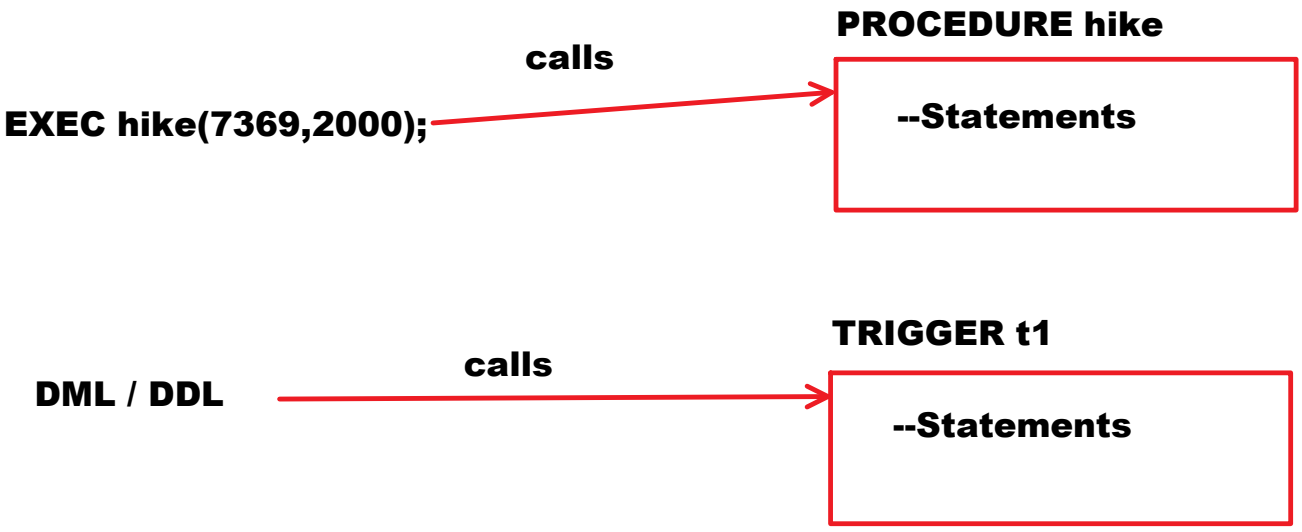
```
SELECT text
FROM user_source
WHERE name='HR';
```


ORACLE DB SERVER
INSTANCE
DB
EXEC p1
EXEC p2
.
.
EXC p10
p1
p2
.
.
p10
proc p1
proc p2
.
.
proc p10


ORACLE DB SERVER
INSTANCE
DB
EXEC demo.p1
EXEC demo.p2
.
.
EXEC demo.p10
demo
PACKAGE demo
proc p1
proc p2
.
.
proc p10

TRIGGERS

Tuesday, June 25, 2024 6:28 PM



TRIGGER:

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

Note:

- **To perform DMLs, we define PROCEDURE.**
- **To control DMLs, we define TRIGGER.**

TRIGGER can be mainly used for 3 purposes. They are:

- **To control the DMLs.**

Examples:

don't allow DMLs on SUNDAY
don't allow DMLs before or after office timings

- **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 the background.
  It is called AUDITING.

- **To define our own constraints (business rules).**

  Example:
  don’t decrease emp 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 trigger is created on table then it is called "Table Level Trigger".**

- **SQL DEVELOPER defines Table Level Trigger.**

Syntax of Table Level Trigger:

```
CREATE [OR REPLACE] TRIGGER <name>
BEFORE/AFTER <dmls_list>
ON <table_name>                          --> Trigger Header / Trigger Specification
[FOR EACH ROW]
DECLARE
   --declare the variables
BEGIN                                    --> Trigger Body
   --statements
END;
/
```

Statement Level Trigger:

- **This trigger gets executed once for one SQL statement.**

Example:

```
UPDATE emp SET sal=sal+1000;    --1 time
```

Row Level Trigger:

- **This trigger gets once for every row affected by DML.**

Example:

```
UPDATE emp SET sal=sal+1000;     -- 14 times
Output:
14 rows updated
```

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

Example on statement level trigger:

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

Testing:
UPDATE emp SET sal=sal+1000;
Output:
statement level trigger executed..
14 rows updated.
```

Example on row level trigger:

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

Testing:
UPDATE emp SET sal=sal+1000;
Output:
row level trigger executed..
```

row level trigger executed..

.

. 14 times

14 rows updated.

Disable and Enable Trigger:

Syntax:

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

Example:

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

ALTER TRIGGER t1 ENABLE;  --t1 trigger works again
```

Dropping trigger:

Syntax:

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

Example:

```
DROP TRIGGER t1;
```

Before trigger:

- first trigger code gets executed.
- then DML operation will be performed.

After trigger:

- first DML operation will be performed.
- then trigger gets executed.

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')='TUE' THEN
      raise_application_error(-20050, 'you cannot perform DMLs
      on SUNDAY..');
   END IF;
END;
/
```

Testing:
[on Sunday]:
UPDATE emp SET sal=sal+1000;
Output:
ERROR:
ORA-20050: you cannot perform DMLs on SUNDAY..

[from mon to sat]:
UPDATE emp SET sal=sal+1000;
Output:
14 rows updated

Define a trigger not to allow the user to update empno:

```
CREATE TRIGGER t4
BEFORE update OF empno
ON emp
BEGIN
   raise_application_error(-20050, 'you cannot update empno..');
END;
/
```

Testing:
update emp set empno=1234
where empno=7499;
Output:
ERROR:
ORA-20050: you cannot update empno..

:NEW and :OLD:

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

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

- **These variables can be used in row level trigger only.**

DML	:NEW	:OLD
INSERT	new row	null
UPDATE	new row	old row
DELETE	null	old row

EMP

EMPNO	ENAME	SAL
1001	A	5000
1002	B	6000

```
INSERT INTO emp
VALUES(5001,'AB',10000);
```

:new

EMPNO	ENAME	SAL
5001	AB	10000

:old

EMPNO	ENAME	SAL
null	null	null

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

:new

EMPNO	ENAME	SAL
null	null	null

:old

EMPNO	ENAME	SAL
1001	A	5000

```
UPDATE emp
SET sal=sal+2000
WHERE empno=1002;
```

:new

EMPNO	ENAME	SAL
1002	B	8000

:old

EMPNO	ENAME	SAL
1002	B	6000

Examples on row level trigger:

Define a trigger to record deleted records in emp_resign table:

EMP

EMPNO	ENAME	JOB	SAL
1001	A	MANAGER	5000
1002	B	CLERK	6000
1003	C	CLERK	8000
..			

EMP_RESIGN

EMPNO	ENAME	JOB	SAL	DOR

:old

EMPNO	ENAME	JOB	SAL
1003	C	CLERK	8000

```
.. CREATE TABLE emp_resign
   (
   empno number(4),
   ename varchar2(10),
   job varchar2(10),
   sal number(7,2),
   dor date
   );


   CREATE OR REPLACE TRIGGER t6
   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=7788;
   COMMIT;

   SELECT * FROM emp_resign;
   Output:
   7788
```

Example:

define a trigger to audit emp table:

EMP_AUDIT

uname	op_date_time	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 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)
);
```

```
CREATE OR REPLACE TRIGGER t7
AFTER insert or delete or update
ON emp
FOR EACH ROW
DECLARE
   op VARCHAR2(10);
BEGIN
   IF inserting THEN
      op := 'INSERT';
   ELSIF deleting THEN
      op := 'DELETE';
   ELSIF updating THEN
      op :='UPDATE';
   END IF;

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

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

EMPNO	ENAME	SAL
1001	A	5000
1002	B	8000
1003	C	7000

:old

EMPNO	ENAME	SAL
1003	C	7000

```
UPDATE emp
SET sal=sal-2000
WHERE empno=1003;
```

:new

EMPNO	ENAME	SAL
1003	C	5000

```
CREATE OR REPLACE TRIGGER t8
BEFORE update
ON emp
FOR EACH ROW
BEGIN
   IF :new.sal<:old.sal THEN
      raise_application_error(-20050,'you cannot decrease salary');
   END IF;
END;
/
```

Schema Level Trigger:

- **SCHEMA => user**
- **If trigger is created on schema [1 user] then it is called "Schema Level Trigger".**
- **It can be also called as DDL Trigger / System Trigger.**
- **it is developed by DBA.**

Syntax:

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

```
DECLARE
   --declare the variables
BEGIN
   -- statements
END;
/
```

Define a trigger to don't allow c##batch6pm user to drop any db object:

login as DBA:
username: system
password: naresh

```
CREATE OR REPLACE TRIGGER st1
BEFORE drop
ON c##batch6pm.SCHEMA
BEGIN
   raise_Application_error(-20050, 'you cannot drop any db object..');
END;
/
```

Testing:
DROP TABLE salgrade;
Output:
ERROR:
ORA-20050: you cannot drop any db object..

DROP PROCEDURE withdraw;
Output:
ERROR:
ORA-20050: you cannot drop any db object..

Define a trigger to don't allow c##batch6pm user to drop the table:

```
CREATE OR REPLACE TRIGGER st1
BEFORE drop
ON c##batch6pm.SCHEMA
BEGIN
   IF ora_dict_obj_type='TABLE' THEN
      raise_Application_error(-20050, 'you cannot
```

```
        drop table..');
    END IF;
END;
/
```

Testing:
DROP TABLE salgrade;
Output:
ERROR:
ORA-20050: you cannot drop table..

DROP PROCEDURE withdraw;
Output:
procedure dropped

Database level trigger:

- **If trigger is created on database then it is called "database level trigger".**
- **it can be also called as system trigger / ddl trigger.**
- **it is developed by DBA.**

Note:
to control 1 user define SCHEMA LEVEL TRIGGER
to control multiple users or all users define DATABASE LEVEL TRIGGER

Syntax:

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

Define a trigger to don't allow c##batch6pm, c##batch730am, c##batch2pm to drop any db object:

```
CREATE OR REPLACE TRIGGER dt1
BEFORE drop
ON database
BEGIN
   IF user IN('C##BATCH6PM', 'C##BATCH730AM',
   'C##BATCH2PM') THEN
      raise_application_error(-20050, 'you cannot drop any db
      object..');
   END IF;
END;
/
```

Testing:
login as c##batch730am:

```
DROP TABLE emp;
```

Output:
ERROR

login as c##batch2pm:

```
DROP TABLE emp;
```

Output:
ERROR

user_triggers:
it maintains all triggers info

```
   DESC user_triggers

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

to see trigger code:

```
   SELECT text
   FROM user_Source
   WHERE name='T6';
```

COLLECTIONS

Thursday, June 27, 2024 6:11 PM

COLLECTION:

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

Examples:

NUMBER
x

45
56
32
89

x(1) => 45
x(2) => 56

VARCHAR2
a

SMITH
ALLEN
MILLER
SCOTT

a(1) => SMITH
a(2) => ALLEN

STUDENT%ROWTYPE
s

SID	SNAME	SCITY
1001	A	HYD

s(1)

SID	SNAME	SCITY
1002	B	DLH

s(2)

SID	SNAME	SCITY
1003	C	BLR

s(3)

s(1).sid => 1001
s(1).sname => A
s(1).scity => HYD

Types of Collections:

3 Types:

- **Associative Array / PL SQL Table / Index By Table**
- **Nested table**
- **V-Array**

Associative Array:

- **ASSOCIATIVE ARRAY is a table of 2 columns. They are:**
  INDEX
  ELEMENT

x

INDEX	ELEMENT
1	10
2	50

a

INDEX	ELEMENT
HYD	900000
BLR	1200000

1	10
2	50
3	30
4	90

HYD	900000
BLR	1200000
DLH	1100000
CHN	800000

Creating Associative Array:

2 steps:
- **define the data type**
- **declare variable for that data type**

define the data type:

Syntax:

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

Example:

```
TYPE num_array IS TABLE OF number(4)
INDEX BY binary_integer;
```

Note:
If INDEX is NUMBER type then we can use binary_integer or pls_integer

x

INDEX	ELEMENT
1	10
2	50
3	30
4	90

- **declare variable for that data type:**

Syntax:

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

Example:

```
x num_array;
```

Example on Associative Array:

Create an associative array as following:

x

INDEX	ELEMENT
1	10
2	50
3	30
4	90

```
DECLARE
   TYPE num_array IS TABLE OF number(4)
   INDEX BY binary_integer;

   x num_array;
BEGIN
   x := num_array(10,50,30,90);       --21C

   /* x(1):=10;
      x(2):=20;
      x(3):=30;
      x(4):=90;    */                      --19c

   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));


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

x

INDEX	ELEMENT
1	10
2	50
3	30
4	90

```
Output:
first index=1
last index=4
next index of 2=3
prev index of 2=1
10
50
30
90
```

In above program, num_array() is a collection constructor.

collection constructor:

- **is a special function.**
- **is used to initialize the collection.**
- **initializing collection means bringing values into collection.**

Collection members:

first	**first index**	**x.first => 1**
last	**last index**	**x.last => 4**
next	**next index**	**x.next(2) => 3** **--next index of 2 = 3**
prior	**previous index**	**x.prior(2) => 1** **--prev index of 2 = 1**

Create an associative array and hold all dept names in it:

d

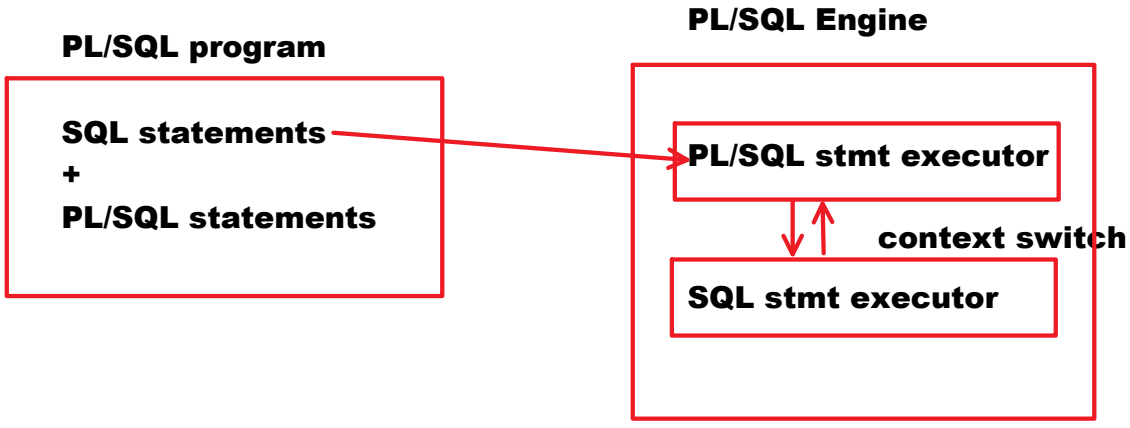
INDEX	ELEMENT
1	ACCOUNTING
2	RESEARCH
3	SALES
4	OPERATIONS

```
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 code degrades the performance.
To collect 4 dept names 4 context switches will occur.
If no of context switches are increased then performance will be degraded.

To improve the performance we use BULK COLLECT clause.

BULK COLLECT:

- **BULK COLLECT clause is used to collect entire data at a time with single context switch.**
- **It improves the performance.**
- **It reduces no of context switches and improves 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;
/
```

Output:

```
ACCOUNTING
RESEARCH
SALES
OPERATIONS
```

Create an Associative Array. Hold all emp records in it. and print them:

EMP

EMPNO	ENAME	SAL
1001	A	6000
1002	B	8000
1003	C	7000

e

INDEX	ELEMENT
1	EMPNO: 1001, ENAME: A, SAL: 6000
2	EMPNO: 1002, ENAME: B, SAL: 8000
3	EMPNO: 1003, ENAME: C, SAL: 7000

```
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 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;
```

HIKE

EMPNO	PER
1001	10
1002	20
1003	15

h

INDEX	ELEMENT
1	EMPNO: 1001, PER: 10
2	EMPNO: 1002, PER: 20
3	EMPNO: 1003, PER: 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.
To improve performance we use BULK BIND.

BULK BIND:

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

- **With Single context switch BULK INSERT / BULK UPDATE / BULK DELETE commands get executed. So, improves the performance.**

- **For BULK BIND we define FORALL loop.**

Note:
We will not use BULK BIND keyword to use BULK BIND.
Just we define FORALL loop.

Syntax:

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

Example:

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

```
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:

- NETSED TABLE is a table of 1 column. i.e: ELEMENT.

Example:

ELEMENT
10
50
30
90

Creating nested table:

2 steps:

- create our own data type
- declare variable for that data type

create our own data type:

Syntax:

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

Example:

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

declare variable for that data type:

Syntax:

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

Example:

```
x NUM_ARRAY;
```

Create a nested table as following:

x

ELEMENT
10
50

30
90

```
DECLARE
   TYPE num_array IS TABLE OF number(4);
   x NUM_ARRAY;
BEGIN
   x := num_array(10,50,30,90);

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

Display 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;
/
```

V-ARRAY:

- **V-ARRAY => variable size array**
- **It is same as nested table. But, it can hold limited number of elements.**

Creating V-array:

2 steps:

- create our own data type
- declare variable for that data type

- create our own data type:

  Syntax:
  ```
  TYPE <name> IS VARRAY(<size>) OF <element_type>;
  ```

  Example:
  ```
  TYPE num_array IS VARRAY(10) OF number(4);
  ```

- declare variable for that data type:

  Syntax:
  ```
  <variable> <data_type>
  ```

  Example:
  ```
  x num_array;
  ```

Example on v-array:

```
DECLARE
   TYPE num_array IS VARRAY(10) OF number(4);

   x num_array;
BEGIN
   x := num_array(10,50,30,90);

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

Hold all emp table records in varray:

```
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 CURSOR and COLLECTION:

CURSOR	COLLECTION
• it fetches row by row	• it fetches all rows at a time and stores in collection
• Random accessing is not possible. It supports to sequential accessing only.	• Random accessing is possible.
• CURSOR is slower	• COLLECTION is Faster
• CURSOR can move forward only	• COLLECTION can move in any direction.

Differences among Associative Array, Nested Table and V-Array:

COLLECTION	INDEX	NO OF ELEMENTS	DENSE OR SPARSE
Associative Array	NUMBER or VARCHAR2	unlimited	dense (or) sparse
Nested Table	NUMBER	unlimited	starts as dense it can become sparse
V-Array	NUMBER	limited	dense

DENSE => no gaps

x(1)
x(2)
x(3)
x(4)
x(5)

SPARSE => gaps can be there

x(10)
x(20)
x(50)
x(90)

WORKING WITH LOBS

Monday, July 1, 2024 6:38 PM

Working with LOBs:

Binary Related data types:
Binary Related data types are used to maintain multimedia objects like images, audios, videos, animations, documents.

ORACLE provides following Binary Related data types:

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

BFILE:

- **BFILE => Binary File Large Object**
- **It is a pointer to multimedia object.**
- **It is used to maintain multimedia object's path.**
- **It can be also called as "External Large Object".**
- **It is not secured one.**

Example:

Directory Object:

- **Directory object is a pointer to specific folder.**
- **DBA creates it.**

Syntax:

CREATE DIRECTORY <dir_obj_name> AS <folder_path>;

Example:

login as DBA:

username: system
password: naresh

```
CREATE DIRECTORY d1 AS 'D:\photos';

GRANT read, write ON DIRECTORY d1
TO c##batch6pm;
```

Example on BFILE:

EMP1

EMPNO	ENAME	EPHOTO [BFILE]

Login as c##batch6pm user:

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

INSERT INTO emp1
VALUES(1234, 'Ellison', bfilename('D1','Ellison.jpg'));

COMMIT;

SELECT * FROM emp1;
```

BLOB:

- **BLOB => Binary Large Object**
- **It is used to maintain multimedia object inside of table.**
- **It can be also called as Internal Large Object.**
- **It is secured one.**

Example: **DB**

Example on BLOB:

EMP2

EMPNo	ENAME	EPHOTO [BLOB]

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

INSERT INTO emp2 VALUES(1234, 'Ellison', empty_blob());

COMMIT;
```

Define a procedure to update emp photo:

```
CREATE OR REPLACE PROCEDURE
update_photo(p_empno NUMBER, p_fname VARCHAR2)
AS
   s BFILE;
   t BLOB;
```

```
   length NUMBER;
BEGIN
   s := bfilename('D1', p_fname);      --s holds ellison image path

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

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

   length := dbms_lob.getlength(s);               --finds size of s file

   dbms_lob.LoadFromFile(t,s,length);         -- s file data reads and writes into t
   --now t has image

   UPDATE emp2 SET ephoto=t WHERE empno=p_empno;  --t image stores in table

   COMMIT;

   dbms_output.put_line('image updated...');
END;
/

SQL> EXEC update_photo(1234, 'Ellison.jpg');
Output:
image updated...

SQL> select * from emp2;

    EMPNO ENAME      EPHOTO
---------- ---------- --------------------------------------------------------------------------
1234       Ellison      FFD8FFE000104A4649460001010000010010000

SQL> select length(ephoto) from emp2;

LENGTH(EPHOTO)
--------------
          6638
```

DYNAMIC SQL

Tuesday, July 2, 2024 6:13 PM

DYNAMIC SQL:

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

Static Query:

```
DROP TABLE emp;
```

Dynamic Query:

```
'DROP TABLE ' || tname;

DROP TABLE emp
DROP TABLE dept
DROP TABLE salgrade
```

- **The query which is built at runtime is called "Dynamic Query".**
- **To execute dynamic queries we use the concept DYNAMIC SQL.**
- **Submit Dynamic query as string to EXECUTE IMMEDIATE command.**

Examples on Dynamic SQL:

Define a procedure to drop the table:

```
CREATE OR REPLACE PROCEDURE
drop_table(p_tname VARCHAR2)
AS
```

```
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE ' || p_tname;
   dbms_output.put_line(p_tname || ' table dropped..');
END;
/
```

Calling:

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

Output:

```
salgrade table dropped..
```

Define a procedure to drop any db object:

```
CREATE OR REPLACE PROCEDURE
drop_object(p_obj_type VARCHAR2, p_obj_name VARCHAR2)
AS
BEGIN
   EXECUTE IMMEDIATE 'DROP ' || p_obj_type || ' ' || p_obj_name;
   dbms_output.put_line(p_obj_name || ' ' || p_obj_type || ' dropped..');
END;
/
```

Calling:

```
SQL> EXEC drop_object('procedure','addition');
```

Output:

```
addition procedure dropped..

SQL> EXEC drop_object('table','hike');
Output:
hike table dropped..
```