PL/SQL

Wednesday, September 20, 2023 7:34 PM

SQL:

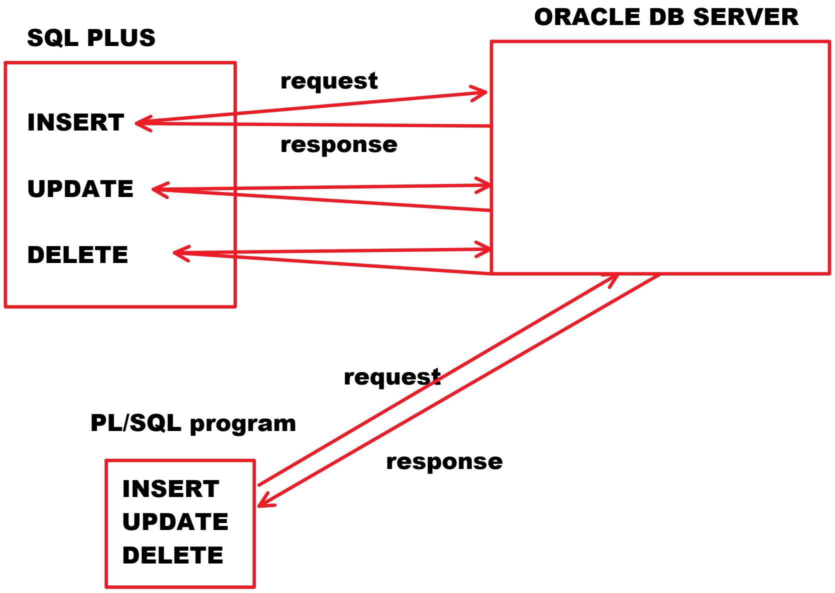
- **Non-Procedural [no set of statements (or) no programs]**

PL/SQL:

- **PL/SQL => Procedural Language/Structured Query Language**
- **PL/SQL is procedural language. It means, we write a set of statements or programs.**
- **PL/SQL = SQL + Programming**
- **PL/SQL is extension of SQL.**
- **In PL/SQL we develop the programs to deal with ORACLE DATABASE.**
- **All SQL queries can be written as statements in PL/SQL program.**

Advantages:

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



- PL/SQL improves performance.
- We can group SQL commands in PL/SQL program and we can submit as one request.
- It reduces no of requests and responses. So, automatically performance will be improved.

provides conditional control structures:

- PL/SQL provides conditional control structures like IF .. THEN,   IF .. THEN .. ELSE, IF .. THEN .. ELSIF

provides looping control structures:

- to perform same action repeatedly, PL/SQL provides looping control structures.
- PL/SQL provides looping control structures such as while loop, simple loop, for loop

Types of Blocks:

2 Types:

- Anonymous Block
- Named Block

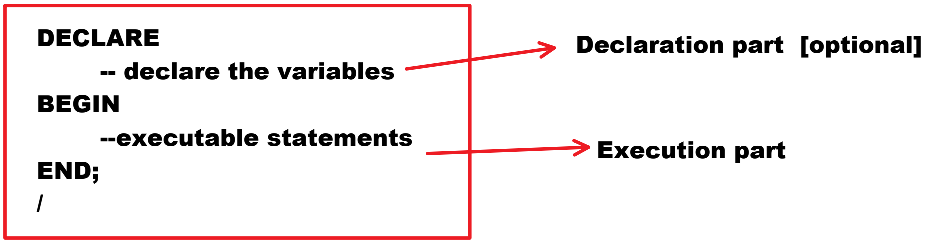
Anonymous Block:

- Anonymous => no name
- A block without name is called "Anonymous Block"

Named Block:

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

Syntax of Anonymous Block:



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

Declaration part [optional]

Execution part

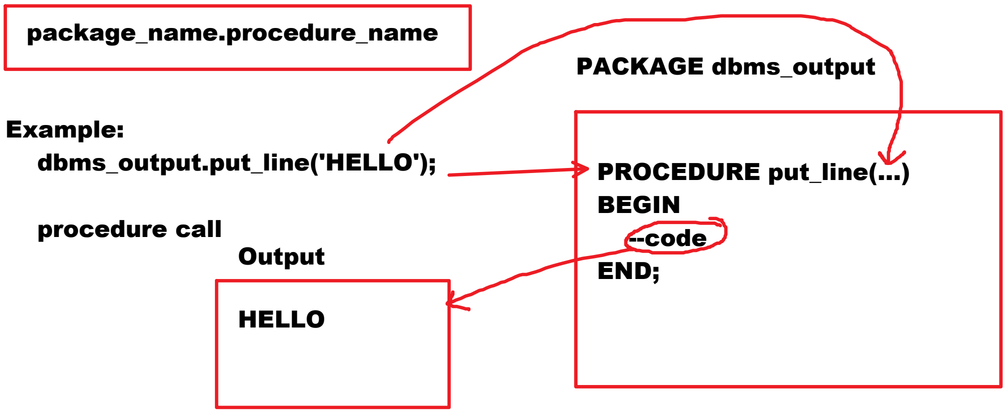
In PL/SQL:

dbms_output	package
put_line()	procedure

put_line():

- **put_line() is a procedure defined in dbms_output package.**
- **It is used to print the data on screen.**

syntax to call packaged procedure:



package_name.procedure_name

PACKAGE dbms_output

Example:
dbms_output.put_line('HELLO');

procedure call

```
PROCEDURE put_line(...)
BEGIN
   --code
END;
```

Output

HELLO

Program to print HELLO on screen:

```
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:

@<path>	compile
/	run

- **Open sqlplus**
- **login as user**

```
SQL> SET SERVEROUTPUT ON

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

SERVEROUTPUT:

- SERVEROUTPUT is a parameter
- Its default value is OFF
- To send messages to output screen we must set serveroutput as ON.

- to set serveroutput as on write following command:

  ```
  SQL> SET SERVEROUTPUT ON
  ```

Data Types
Declare
Assign
Initialize
Print
Read

Data Types in PL/SQL:

Character Related	Char(n)
	varchar2(n)
	LONG
	CLOB
	STRING(n) => PL/SQL only
	
	nChar(n)
	nVarchar2(n)
	nCLOB

Integer Related	Number(p)
	INT
	INTEGER
	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	**%TYPE => PL/SQL only** **%ROWTYPE => PL/SQL only**
Cursor Related	**SYS_REFCURSOR => PL/SQL only**
Boolean	**BOOLEAN** **Till ORACLE 21C, PL/SQL only** **From ORACLE 23C version onwards this data type available for SQL also**

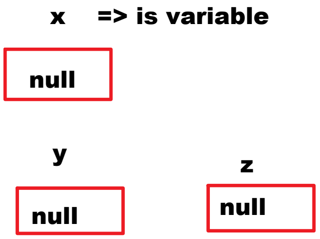
Variable:

- **Variable is an Identifier.**
- **Variable is a name of storage location that contains a value.**
- **A variable can hold 1 value only .**

Declaring Variable:

Syntax:
<variable> <data_type>;

Example:
x NUMBER(4);
y VARCHAR2(10);
z DATE;

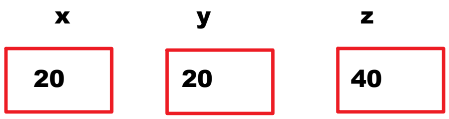


Assigning value:

:=	**Assignment Operator**

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

Examples:



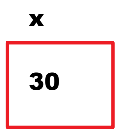
x:=20; --20 is constant

```
y:=x;    --x is variable

z:=x+y;  --x+y is expression


x+y := z;    --ERROR
```

Initializing Variable:

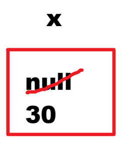
Giving value at the time of declaration is called "initialization".

x NUMBER(4) := 30;



x NUMBER(4);

x:=30;



Printing data:

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

Reading data:

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

Output:
enter value for x: 30

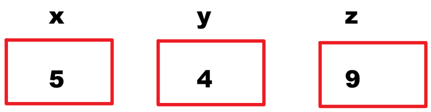
DECLARE	**x NUMBER(4);**
Assign	**x:=30;**
Initialize	**x NUMBER(4) := 30;**
print	**dbms_output.put_line(x);**
read	**x:=&x;**

Program to add 2 numbers:

```
x    y
5    4

5+4 = 9
z:=x+y
```

- **declare x,y,z as number type**
- **assign 5 value to x**
- **assign 4 value to y**
- **add x and y store result in z**
- **print z**

```
DECLARE
   x NUMBER(4);
   y NUMBER(4);
   z NUMBER(4);
BEGIN
   x:=5;
   y:=4;

   z:=x+y;

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



```
Output:
sum=9
```

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;
/

Output:
Enter value for x: 2
old   6:        x:=&x;
new   6:        x:=2;
Enter value for y: 8
old   7:        y:=&y;
new   7:        y:=8;
sum=10
```

To avoid old and new parameters set verify as OFF

```
SQL> SET VERIFY OFF

SQL> @d:\batch6pm\AddDemo.sql
Output:
Enter value for x: 10
Enter value for y: 5
sum=15
```

Using SQL commands in PL/SQL:

- **DML, DRL, TCL commands can be used directly in PL/SQL program.**

- **DDL, DCL commands cannot be used directly in PL/SQL program. To use DDL or DCL commands in PL/SQL we use "DYNAMIC SQL".**

Program to delete an emp record:

- **DECLARE v_empno variable**
- **Read empno**
- **DELETE command**
- **COMMIT**
- **display message => record deleted**

```
DECLARE
    v_empno NUMBER(4);
BEGIN
    v_empno := &empno;

    DELETE FROM emp WHERE empno=v_empno;
    COMMIT;

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

v_empno

7788

```
Output:
enter .. empno: 7788
record deleted..
```

Using SELECT command in PL/SQL:

Syntax of SELECT command in PL/SQL:

```
SELECT <column_list / *> INTO <variable_list>
FROM <table_name>
WHERE <condition>;
```

Example:

```
SELECT ename,sal INTO x,y
FROM emp
WHERE empno=7369;
```

x	y
SMITH	800

Display the emp record of given empno:

```
enter ... empno: 7369                 enter ... empno: 7499
SMITH      800                        ALLEN      1600
```

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

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

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

v_empno	v_ename	v_sal
7499	ALLEN	1600

EMP

EMPNO	ENAME	SAL
7369	SMITH	800
7499	ALLEN	1600
..		

```
Output:
enter ... empno:7499
ALLEN   1600
```

%TYPE:

- It is attribute related data type.
- It is used to declare a variable with table column's data type.
- It avoids mismatch between data type of variable and table column
- It avoids mismatch between field size of variable and table column

Syntax:

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

Examples:

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

Program to demonstrate %TYPE:
Display the emp record of given empno:

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

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

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

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

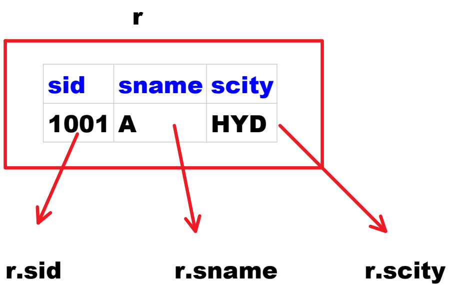
emp table's empno column's data type will be taken as v_empno variable data type.

%ROWTYPE:

STUDENT

sid	sname	scity
1001	A	HYD
1002	B	DLH
..		
1010	..	..

```
r  STUDENT%ROWTYPE;
```



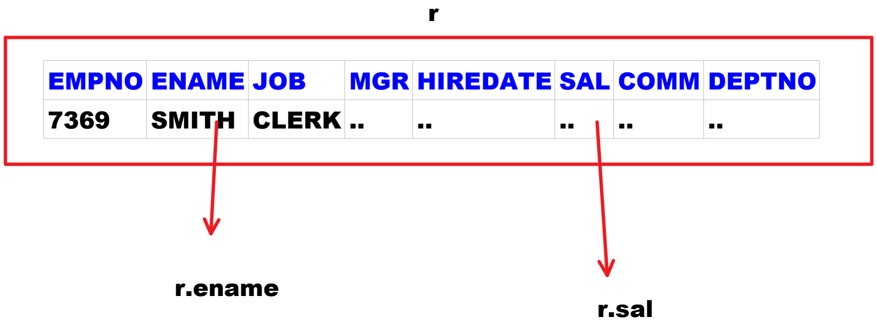
r

sid	sname	scity
1001	A	HYD

EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	..						
7499	..						

```
r EMP%ROWTYPE;
```



r

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	..	..	..	..	..

% ROWTYPE:

- **It is attribute related data type.**
- **it is used to hold entire table row.**
- **It can hold only one row at a time. It cannot hold multiple rows at a time.**
- **It decreases no of variables.**

Syntax:

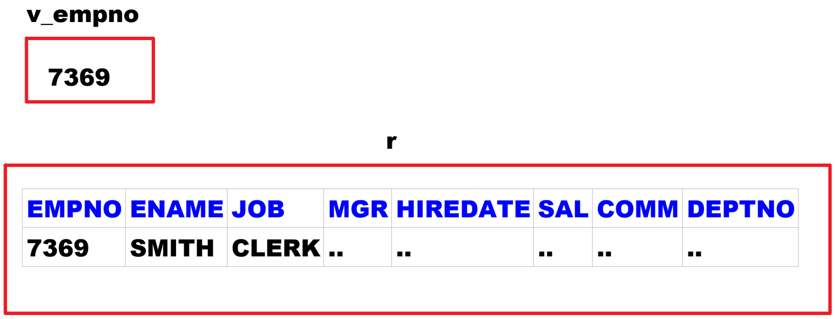
```
<variable> <table_name>%ROWTYPE;
```

Example:

```
r1 EMP%ROWTYPE;
r1 can hold emp table entire row


r2 STUDENT%ROWTYPE;
r2 can hold student table entire row
```

Program to demonstrate %ROWTYPE.
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.sal || '    ' || r.hiredate);
END;
/
```



EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	..	..	..	..	..

Program to Find experience of given empno:

enter .. empno: 7369
experience=42

```
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);

 END;
 /
```

Using Update Command in PL/SQL:

Program to increase salary of given empno with given amount:

```
enter ... empno: 7369
enter ... amount: 1000
salary increased
```

```
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('salary increased');
END;
/
```

v_empno	v_amount
7369	1000

emp

empno	ename	sal
7369	smith	800
7499	allen	1600

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 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;
/
```

r

sid	sname	m1
1	A	70

Control Structures

Saturday, September 23, 2023 6:13 PM

```
DECLARE
   m INT := 70;
BEGIN

   IF m>=40 THEN
      dbms_output.put_line('PASS');
   ELSE
      dbms_output.put_line('FAIL');
   END IF;

END;
/
```

```
PASS
```

Control Structures:

- **Control Structures are used to change the sequential execution.**

- **Normally program gets executed sequentially. To change this sequential execution, to transfer 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	**Simple Loop** **While Loop** **For Loop**
Jumping	**GOTO** **EXIT** **EXIT WHEN** **CONTINUE**

Conditional Control Structures:

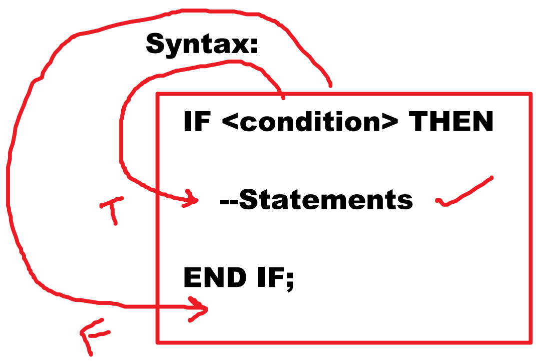
Conditional Control Structures execute the statements

based on conditions.

PL/SQL provides following conditional control structures:

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

IF .. THEN:



Syntax:

```
IF <condition> THEN

    --Statements

END IF;
```

condition=> TRUE

- The statements in "IF .. THEN" get executed when the condition is TRUE.
- If condition is FALSE, it will not execute the statements.

Example on "IF .. THEN":

Program to delete an emp record of given empno.
If experience is more than 41 then only delete the record.

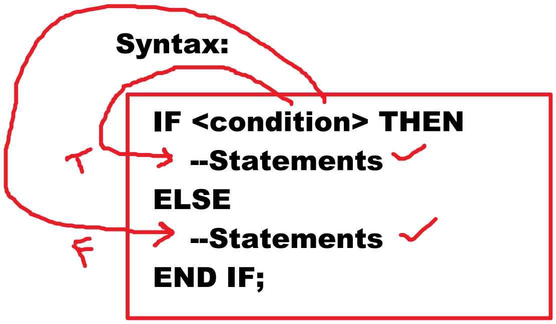
```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_hiredate DATE;
   v_exp INT;
BEGIN
   v_empno := &empno;

   SELECT hiredate INTO v_hiredate FROM emp
   WHERE empno=v_empno;
```

v_empno	v_hiredate	v_exp
7844	----	42

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

        IF v_exp>41 THEN
            DELETE FROM emp WHERE empno=v_empno;
            COMMIT;
            dbms_output.put_line('record deleted');
        END IF;

    END;
    /
```

IF .. THEN .. ELSE:

Syntax:



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

condition=> TRUE

condition => FALSE

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

Example on IF .. THEN .. ELSE:

Program to increase salary of given empno as following:
if job is MANAGER => increase 20% on sal
others => increase 10% on sal

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_job EMP.JOB%TYPE;
```

v_empno	v_job	v_per
7934	CLERK	10

```
DECLARE                                   v_empno    v_job      v_per
   v_empno EMP.EMPNO%TYPE;                 7934      CLERK       10
   v_job EMP.JOB%TYPE;
   v_per INT;
BEGIN
   v_empno := &empno;

   SELECT job INTO v_job FROM emp
   WHERE empno=v_empno;

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

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

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

```
Output:
enter ..empno: 7934
job is CLERK
10% on sal increased
```

IF .. THEN .. ELSIF:

Syntax:

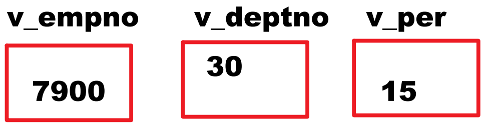
```
IF <condition-1> THEN
   --Statements                    condn1 => TRUE
ELSIF <condition-2> THEN
   --Statements                    cond1 => F  cond2 => T
.
.
[ELSE
   --Statements]                    All condns FALSE
END IF;
```

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

- **To check more than 1 condition we use "IF .. THEN .. ELSIF"**

Example on IF .. THEN .. ELSIF:

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



v_empno v_deptno v_per

7900 30 15

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_deptno EMP.DEPTNO%TYPE;
   v_per INT;
BEGIN
   v_empno := &empno;

   SELECT deptno INTO v_Deptno FROM emp WHERE empno=v_empno;

   IF v_deptno=10 THEN
      v_per := 10;
   ELSIF v_deptno=20 THEN
      v_per := 20;
   ELSIF v_deptno=30 THEN
      v_per := 15;
   ELSE
      v_per := 5;
   END IF;

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

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

NESTED IF:

Syntax:

```
IF <condition-1> THEN
   IF <condition-2> THEN
      --Statements
   END IF;
END IF;
```

--condition1, condition2 => TRUE

The statements in Inner IF get executed when outer condition and inner condition are true.

CASE:

It can be used in 2 ways. They are:

- **Simple CASE   => [same as switch in C/Java]**
- **Searched CASE   => [same as if else if in C/Java]**

Simple CASE:

- **It can check equality condition only**

Searched CASE:

- **It can check any condition like <   >   <=   =**

Syntax of Simple CASE:

```
CASE <expression>
   WHEN <constant-1> THEN
      --Statements
   WHEN <constant-2> THEN
      --Statements
   .
   .
```

```
        Statements
    .
    .
    [ELSE
        --Statements]
END CASE;
```

Example on Simple CASE:

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

EVEN	2,4,6,8,......	divide with 2	remainder 0
ODD	1,3,5,7,......	divide with 2	remainder 1

```
DECLARE
    n INT := &n;
BEGIN
    CASE mod(n,2)
        WHEN 0 THEN
            dbms_output.put_line('EVEN');
        WHEN 1 THEN
            dbms_output.put_line('ODD');
    END CASE;
END;
/
```

Syntax of Searched Case:

```
CASE
    WHEN <condition-1> THEN
        --Statements
    WHEN <condition-2> THEN
        --Statements
    .
    .
    ELSE
        -_Statements
END CASE;
```

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

```
DECLARE
    n INT := &n;
BEGIN
    CASE
     WHEN n>0 THEN
        dbms_output.put_line('POSITIVE');
     WHEN n<0 THEN
        dbms_output.put_line('NEGATIVE');
    WHEN n=0 THEN
        dbms_output.put_line('ZERO');
    END CASE;
END;
/
```

Example Program on NESTED IF:

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

STUDENT

sid	sname	m1	m2	m3
1	A	70	80	60
2	B	50	30	75

RESULT

sid	total	avrg	result
v_sid (or) r1.sid	r2.total	r2.avrg	r2.result

**enter .. sid: 1
result calculated and stored in result table**

STUDENT

sid	sname	m1	m2	m3

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

1	A	70	80	60
2	B	50	30	75

```
INSERT INTO student VALUES(1,'A',70,80,60);
INSERT INTO student VALUES(2,'B',50,30,70);
COMMIT;
```

RESULT

sid	total	avrg	result

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

```
DECLARE
   v_sid STUDENT.SID%TYPE;
   r1 STUDENT%ROWTYPE;
   r2 RESULT%ROWTYPE;
BEGIN
   v_sid := &sid;      --1

   SELECT * INTO r1 FROM student
   WHERE sid=v_sid;  --1

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

v_sid

1

r1

sid	sname	m1	m2	m3
1	A	70	80	60

r2

sid	total	avrg	result
	210	70	FIRST

```
   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';
      ELSIF r2.avrg>=40 THEN
         r2.result := 'THIRD';
      END IF;
   ELSE
      r2.result := 'FAIL';
   END IF;

   INSERT INTO result VALUES(v_sid, r2.total, r2.avrg,
   r2.result);
   COMMIT;
   dbms_output.put_line('result calculated and stored in
   RESULT table');
END;
/
```

Output:

```
Enter value for sid: 1
result calculated and stored in RESULT table
```

Looping Control Structures:

Looping Control Structures are used to execute the statements repeatedly.

PL/SQL provides following Looping Control Structures:

- **While loop**
- **Simple loop**
- **For loop**

While Loop:

Syntax:

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

- **The statements in WHILE LOOP get executed as long as the condition is TRUE.**
- **When the condition is FALSE, LOOP will be terminated.**

Example on WHILE LOOP:

Program to print numbers from 1 to 4:

i=1 2 3 4

Output:

```
i
1
2
3
4
```

i<=4

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

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

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

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

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

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

Program to print 2023 Calendar:

Output:

```
1-JAN-23          d1
2-JAN-23
3-JAN-23
.
.
31-DEC-23         d2
```

```
DECLARE
   d1 DATE;
   d2 DATE;
BEGIN
   d1 := '1-JAN-2023';
   d2 := '31-DEC-2023';

   WHILE d1<=d2
   LOOP
      dbms_output.put_line(d1);
      d1:=d1+1;
   END LOOP;
END;
```

/

Program to print sundays in 2023 CALENDAR:

```
DECLARE
   d1 DATE;
   d2 DATE;
BEGIN
   d1 := '1-JAN-2023';
   d2 := '31-DEC-2023';

   WHILE d1<=d2
   LOOP
      IF to_char(d1,'DY')='SUN' THEN
         dbms_output.put_line(d1 || '   ' || to_char(d1,'DAY'));
      END IF;
      d1:=d1+1;
   END LOOP;
END;
/
```

Simple Loop:

Syntax:

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

Example on Simple Loop:

Program to print numbers from 1 to 4 using simple loop:

Output:

```
DECLARE
   i INT;
```

Output:

```
i
1
2
3
4
```

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

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

END;
/
```

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

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

EXIT WHEN:

- It is used to terminate the loop in the middle of execution.
- We can use it in LOOP only.

Syntax:

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

EXIT:

- It is used to terminate the loop in the middle of execution.
- We can use it in LOOP only.

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

Output:

ERROR:

illegal EXIT/CONTINUE statement; it must appear inside a loop

For Loop:

Syntax:

```
FOR <variable> IN [REVRSE] <lower> .. <upper>
LOOP
   --statements
END LOOP;
```

Example on FOR:

Program to print numbers from 1 to 4:

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

```
Output:
1
2
3
4
```

- **We have no need to declare loop variable.**

- **For LOOP variable is read-only variable. We cannot write data into loop variable.**

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

```
Output:
ERROR:
i cannot be used as an assignment target
```

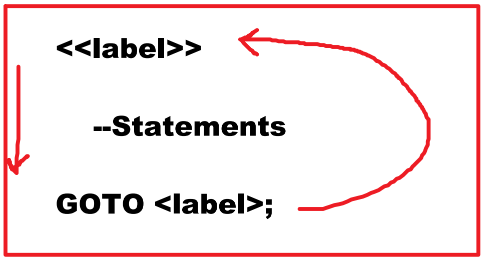
- **Loop variable scope is limited to LOOP only**

```
BEGIN
FOR i IN 1 .. 10
LOOP
   dbms_output.put_line(i);
END LOOP;
   dbms_output.put_line(i);  -- ERROR: i must be declared
END;
/
```

GOTO:

- **used to transfer the control to specified label**

Syntax:



```
<<label>>

   --Statements

GOTO <label>;
```

Example on GOTO:

Program to print numbers from 1 to 4 using GOTO:

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

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

```
Output:
i

1
2
3
4
```

CONTINUE:

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

Example:

Program to print numbers from 1to 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;
/
```

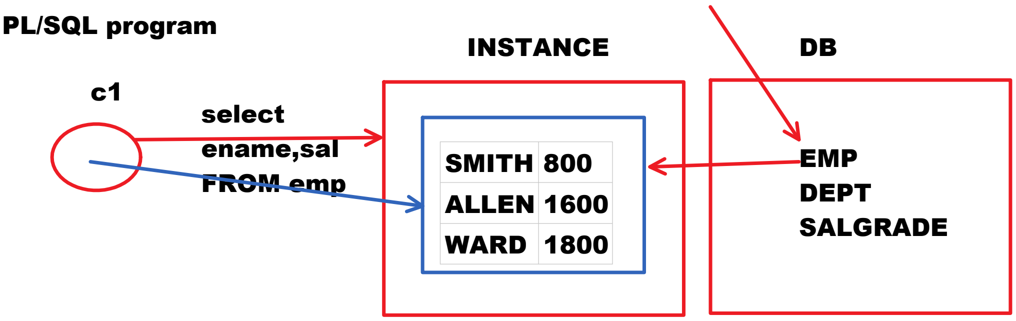
CURSORS

Tuesday, September 26, 2023 6:43 PM

CURSORS:

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

TO HOLD 1 COLUMN VALUE	**USE %TYPE** **EXAMPLE:** **V_ENAME EMP.ENAME%TYPE;** **V_ENAME** **SMITH**
TO HOLD 1 ROW	**USE %ROWTYPE** **EXAMPLE:** **R EMP%ROWTYPE;** **r** **EMPNO | ENAME | JOB | SAL**
TO HOLD MULTIPLE ROWS	**USE CURSOR**



SMITH	800
ALLEN	1600
WARD	1800

Cursor:

- Cursor is a pointer to memory location in ORACLE INSTANCE.

- This memory location holds multiple rows.

- To fetch multiple rows one by one we use CURSOR.

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

Steps to use CURSOR:

To use CURSOR, follow 4 steps. They are:

- DECLARE
- OPEN
- FETCH
- CLOSE

DECLARING CURSOR:

Syntax:

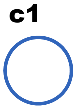
```
CURSOR <cursor_name> IS <select_query>;
```

Example:

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

When we declare the cursor,

- cursor variable will be created
- SEELCT query will be identified



Opening Cursor:

Syntax:

```
OPEN <cursor_name>;
```

Example:

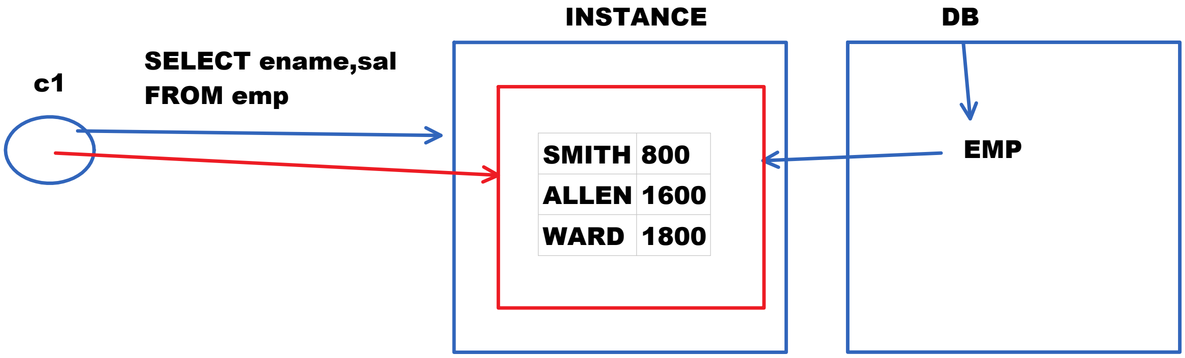
```
OPEN c1;
```

When we open the cursor,

- SELECT query will be submitted to ORACLE.
- ORACLE goes to database, SELECTs the data and copies

into some memory location in ORACLE INSTANCE

- This memory location address will be given to c1



SMITH	800
ALLEN	1600
WARD	1800

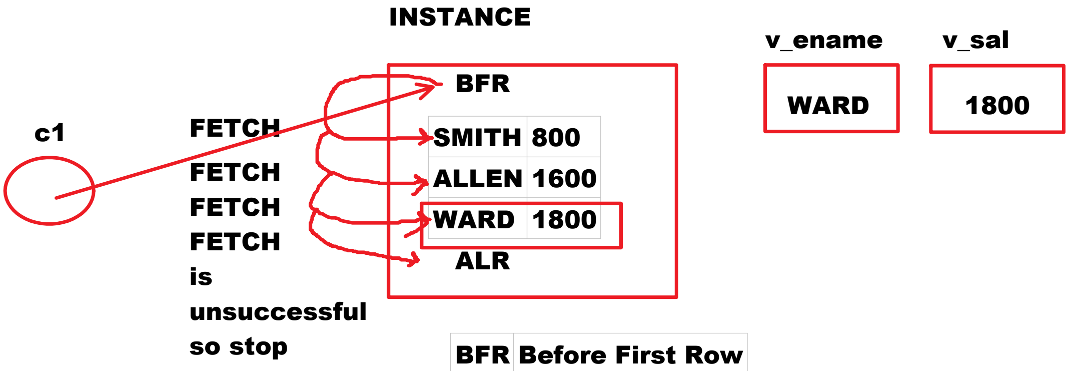
FETCHING RECORDS FROM CURSOR:

Syntax:

```
FETCH <cursor_name> INTO <variable_list>;
```

Example:

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



SMITH	800
ALLEN	1600
WARD	1800

BFR	Before First Row
ALR	After Last Row

- One FETCH statement can fetch one row only. To fetch multiple rows write FETCH statement inside of LOOP.

CLOSING CURSOR:

Syntax:

```
CLOSE <cursor_name>;
```

Example:

```
CLOSE c1;
```

When CURSOR is closed,

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

CURSOR ATTRIBUTES:

There are 4 cursor attributes. they are:

- **%FOUND**
- **%NOTFOUND**
- **%ROWCOUNT**
- **%ISOPEN**

Syntax:

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

Example:

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

%FOUND:

- **it returns boolean value like TRUE OR FALSE.**
- **If record is found, it returns TRUE.**
- **If record is not found, it returns FALSE.**

%NOTFOUND:

- **it returns boolean value like TRUE OR FALSE.**
- **If record is found, it returns FALSE.**
- **If record is not found, it returns TRUE.**

%ROWCOUNT:

- **Its default value is 0.**
- **If FETCH is successful, ROWCOUNT value will be incremented by 1.**

%ISOPEN:

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

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;

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

   CLOSE c1;
END;
/
```

Increase salary to all emps based on hike table data:

EMPLOYEE

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

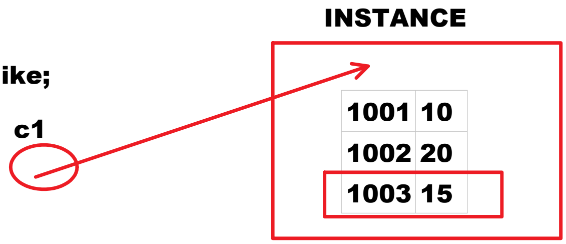
HIKE

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;

   LOOP
      FETCH c1 INTO r;
      EXIT WHEN c1%NOTFOUND;
      UPDATE employee SET sal=sal+sal*r.per/100
      WHERE empno=r.empno;
   END LOOP;

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

   CLOSE c1;
END;
/
```



r

empno	per
1003	15

Program to find result of all students and insert total, avrg and result values in result table:

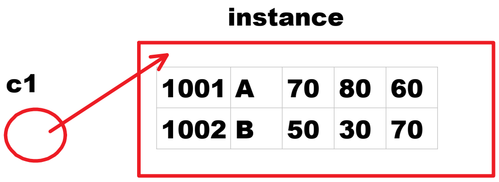
STUDENT

sid	sname	m1	m2	m3
1001	A	70	80	60
1002	B	50	30	70

RESULT

sid	total	avrg	result

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

   LOOP
      FETCH c1 INTO r1;

      EXIT WHEN c1%NOTFOUND;

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

      IF r1.m1>=40 AND r1.m2>=40 AND r1.m3>=40 THEN
```



r1

sid	sname	m1	m2	m3
1001	A	70	80	60

r2

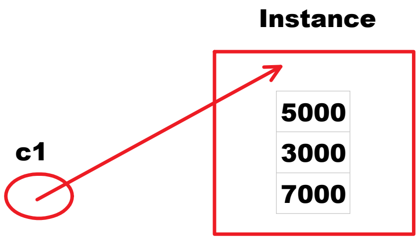
sid	total	avrg	result
	210	70	PASS

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

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

    COMMIT;

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

    CLOSE c1;
END;
/
```

Write a program to find sum of salaries of all emps:

EMP

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

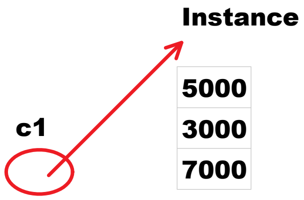


```
            0       +  5000 + 3000 + 7000
v_sum :=    v_sum + v_sal
```

v_sum

0 5000 8000
15000

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

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

    CLOSE c1;
```



v_sal

~~5000 3000~~
7000

v_sum

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

Output:
sum of salaries=15000

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

    CLOSE c1;
END;
/
```

Output:
sum of salaries=15000

5000
3000
7000

Assignment:
Find max salary in all emps

```
v_max                      v_sal>v_max
                              v_max := v_sal;
0  5000  7000
```

Display all emp records using CURSOR.
Do it using WHILE LOOP:

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

    FETCH c1 INTO r;

    WHILE c1%FOUND
    LOOP
        dbms_output.put_line(RPAD(r.ename,10) || '    ' || r.sal);
        FETCH c1 INTO r;
    END LOOP;

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

    CLOSE c1;
END;
/
```

CURSOR FOR LOOP:

Syntax:

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

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

- **If we use CURSOR FOR loop, we have no need to open the cursor, fetch the record from cursor and close the cursor. All these 3 actions will be done implicitly.**

- **No need to declare cursor for loop variable. implicitly it will be declared as %rowtype.**

Display all emp records using cursor for loop:

```
   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 [Inline For Loop]:

If SELECT query is specified in CURSOR FOR loop then it is called "Inline Cursor".

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 parameter is called "Parameterized Cursor".**

- **We pass this parameter value at the time of opening cursor.**

- **When we don't know exact value at the time of declaration then we use parameterized cursor.**

Syntax to declare parameterized cursor:

```
CURSOR <name>(<parameter_list>) IS <SELECT query>;
```

Syntax to Open Parameterized Cursor:

```
OPEN <cursor_name>(<value_list>);
```

Example on Parameterized Cursor:

Display the emp records based on deptno:

```
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;
/
```

Note:
n is paramter
for parameter declaration don't use sizes

n NUMBER(4)	Invalid
n NUMBER	valid

REF CURSOR:

Simple Cursor

```
SELECT * FROM emp  => CURSOR c1
SELECT * FROM dept  => CURSOR c2
SELECT * FROM salgrade => CURSOR c3
```

REF CURSOR

```
SELECT * FROM Emp => Cursor c1
SELECT * FROM Dept => Cursor c1
SELECT * FROM Salgrade => Cursor c1
```

- **In Simple Cursor, one cursor can be used for one SELECT query only.**
- **In REF CURSOR, same cursor can be used for multiple SELECT queries. It reduces no of cursors.**

- **in Ref cursor, we specify SELECT query at the time of opening cursor.**

- **For ref cursor we use data type "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;
--fetch and process
--close

OPEN c1 FOR SELECT * FROM dept;
--fetch and process
--close

OPEN c1 FOR SELECT * FROM salgrade;
--fetch and process
--close
```

Example on REF CURSOR:

**Display all emp table records using cursor c1.
using same cursor display 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 || '    ' || r2.loc);
   END LOOP;

   CLOSE c1;
END;
/
```

Differences between Simple Cursor and Ref Cursor:

Simple Cursor	**Ref Cursor**
One cursor can be used for one select query only	**One cursor can be used for multiple select queries**
Here, select query is fixed. it is static.	**Here, select query can be changed. it is dynamic.**
SELECT query will be specified at the time of declaration.	**SELECT query will be specified at the time of opening cursor.**
It has no data type.	**It has data type. that is: sys_refcursor**
It cannot be used as procedure parameter. It has no data type.	**It can be used as procedure parameter. Because, it has data type.**

Note:
To process any DRL or DML command a cursor is required.

Types of Cursors:

2 Types:

- **Implicit Cursor**
- **Explicit Cursor**
    - **Simple Cursor**
    - **Ref Cursor**

Implicit Cursor:

- **To process any DRL or DML command a cursor is required.**
- **To execute any DRL or DML command implicitly ORACLE uses a cursor. This cursor is called "Implicit Cursor".**
- **This Implicit Cursor name is: SQL**

  Example:

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

Examples on Implicit Cursor:

Program to increase 1000 rupees salary to all emps:

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

    dbms_output.put_line(SQL%ROWCOUNT || ' rows updated..');
    COMMIT;
END;
/
```

Program to increase salary of given empno with given amount.
if emp record is not found then display messge as emp not existed.
otherwise, display sal increased:

```
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('no emp existed with this empno');
   ELSE
      dbms_output.put_line('salary increased..');
   END IF;

   COMMIT;
END;
/
```

CURSOR:

- **is a pointer to memory location in instance [RAM]**

- **GOAL: to hold multiple rows and process them one by one**

- **4 steps:**
- **DECLARE**
- **OPEN**
- **FETCH**
- **CLOSE**

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

- **parameterized cursor =>**

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


OPEN c1(20);
```

Inline Cursor:
We specify select query in cursor for loop

Ref cursor:
same cursor can be used for multiple select queries

Exception Handling

Thursday, September 28, 2023 7:01 PM

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.**

  Example:
  missing
  missing)
  missing '
  missing END IF
  missing END LOOP

Run Time Errors:

- **Run Time Errors occur at run time.**

- **When run time error occurs, our application will be closed in that line only. it will not execute remaining code. It is called "Abnormal Termination". With abnormal termination we may loss the data. That's why Run time error must be handled. The mechanism of handling run-time error is called "Exception Handling".**

Exception	**Run Time Error**	**Problem:** **abnormal termination**
Exception Handling	**The way of handing run-time error**	**Solution:** **converts abnormal termination to normal termination**

Examples:
record is not found
size is exceeded
inserting duplicate value in PK

Logical Errors:

- **Logical Errors occur due to mistake in logic.**
- **ORACLE will not check any logical error.**
- **we are responsible to define correct logic.**
- **With Logical Errors, we get wrong results.**

Example:

withdrawing amount => we need to subtract withdrawl amount from balance
if we add then it is logical error.

Exception:

- **Exception means run-time error.**
- **When run-time error occurs, our application will be closed abnormally.**
- **With abnormal termination we may loss the data.**

Exception Handling:

- **The way of handling run-time errors is called "Exception Handling".**
- **It converts abnormal termination to normal termination.**

- **For Exception Handling add a block. that is: 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
        .
        .
END;
/
```

Program to demonstrate exception handling:

Program to divide 2 numbers:

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

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

    EXCEPTION
        WHEN zero_divide THEN
            dbms_output.put_line('cannot divide with 0');
        WHEN value_error THEN
            dbms_output.put_line('size is exceeded or wrong input');
        WHEN others THEN
            dbms_output.put_line('some RTE occurred');
END;
/
```

```
runtime -1:
enter value for x: 10
enter value for y: 2
z=5


runtime -2:
enter value for x: 10
enter value for y: 0
cannot divide with 0


runtime -3:
enter value for x: 123456
enter value for y: 2
size is exceeded or wrong input

runtime -3:
enter value for x: 'RAJU'
enter value for y: 2
size is exceeded or wrong input
```

Types of Exceptions:

2 Types:

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

Built-In Exception:

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

Examples:

zero_divide
value_error
no_data_found
too_many_rows
invalid_cursor
cursor_already_open
dup_val_on_index

User-Defined exception:

- **We can define our own exception. this is called "User-Defined Exception".**
- **We raise it explicitly.**

  Examples:
    xyz
    one_divide
    sunday_not_allow

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

value_error:
when we wrong input or size is exceeded, value_error exception will be raised.

No_Data_Found:
When we retrieve the data if record is not found then no_data_found exception will be raised.

Example on No_Data_Found:

Display the emp record of given empno.
If emp is not existed, it raises exception. Handle it:

```
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('record not found in emp table');
END;
/
```

runtime-1:

```
enter value for empno: 7369
SMITH      800


runtime-1:
enter value for empno: 1234
record not found in emp table
```

too_many_rows:

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

Example on too_many_rows:

Display the emp record of 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);

   EXCEPTION
      WHEN too_many_rows THEN
         dbms_output.put_line('multiple rows selected..');
END;
/

runtime-1:
enter value for job: PRESIDENT
KING    PRESIDENT

runtime-1:
enter value for job: MANAGER
multiple rows selected..
```

Invalid_cursor:
when we try to fetch the record without opening cursor then invalid_cursor exception will be raised.

Example on invalid_cursor:

Display all emp records. if cursor is not opened run time error will occur. handle it:

```
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 the opened cursor then cursor_already_open exception will be raised.

Example on cursor_already_open:

Program to display all emp record. if we try to open opened cursor run time error will occur. handle it:

```
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
```

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 on dup_val_on_index:

Program to insert student record in student table:

STUDENT
PK

sid	sname	scity

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

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

   dbms_output.put_line('record inserted..');


   EXCEPTION
      WHEN dup_val_on_index THEN
         dbms_output.put_line('PK does not accept duplicate values');
END;
/
```

```
runtime-1:
Enter value for sid: 1
Enter value for sname: A
Enter value for scity: HYD
record inserted..
```

```
runtime-2:
Enter value for sid: 1
Enter value for sname: B
Enter value for scity: DELHI
PK does not accept duplicate values
```

User-Defined Exception:

- **We can define our own exception. This is called "User-Defined exception".**
- **It will be raised explicitly.**

For user-defined exception, follow 3 steps:

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

built-in exception	**user-defined exception**
1 step	**3 steps**
handle	**declare**
	raise
	handle

Declaring Exception:

Syntax:

```
<exception_name> EXCEPTION;
```

Examples:

```
one_divide EXCEPTION;
xyz EXCEPTION;
Sunday_not_allow EXCEPTION;
```

- **to declare the exception name use EXCEPTION data type.**
- **declare it in declaration part.**

Raise the exception:

Syntax:

```
RAISE <exception_name>;
```

Example:

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

- **RAISE is a keyword. It is used to raise the exception explicitly**

Handling Excerption:

Syntax:

```
EXCEPTION
    WHEN <exception_name> THEN
        --Handling code
```

Example:

```
EXCEPTION
    WHEN one_divide THEN
        dbms_output.put_line('cannot divide with 1');
```

Example on user-defined exception:

program to divide 2 numbers. if denominator is 1 raise our own exception one_dvide and handle it:

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

   IF y=1 THEN
      RAISE one_divide;      --raising exception
   END IF;

   z := x/y;

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

   EXCEPTION
      WHEN zero_divide THEN
         dbms_output.put_line('cannot divide with 0');
      WHEN one_divide THEN
         dbms_output.put_line('cannot divide with 1');
END;
/
```

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

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

runtime-3:
Enter value for x: 10
Enter value for y: 1
cannot divide with 1
```

Program to increase salary of given empno with given amount. If user tries to update on Sunday raise the exception and handle it:

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

    IF to_char(sysdate,'dy')='sat' THEN
        RAISE sunday_not_allow;            --raise
    END IF;

    UPDATE emp SET sal=sal+v_amount WHERE empno=v_empno;
    COMMIT;
    dbms_output.put_line('salary increased..');

    EXCEPTION
    WHEN sunday_not_allow THEN          --handle
        dbms_output.put_line('you cannot update on sunday..');
END;
/
```

```
on Monday:
enter value for empno: 7499
enter value for amount: 2000
salary increased..

on Sunday:
enter value for empno: 7499
enter value for amount: 2000
you cannot update on sunday
```

We can raise the exception using 2 ways. They are:

- **using RAISE keyword**
- **using RAISE_APLLICATION_ERROR() procedure**

RAISE_APLLICATION_ERROR():

- **It is a procedure.**
- **It is used to raise the exception using our own error code and our own error message.**
- **user-defined error code valid range is: -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 program on RAISE_APPLICATION_ERROR():

program to divide 2 numbers. if denominator 1 raise the exception using raise_application_error() procedure:

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

   IF y=1 THEN
      RAISE_APPLICATION_ERROR(-20050, 'you cannot divide with 1');
   END IF;

   z := x/y;

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

END;
/
```

```
Output:
enter value for x:20
enter value for y:1
ERROR at line 1:
ORA-20050: you cannot divide with 1
```

Differences b/w RAISE and RAISE_APPLICATION_ERROR():

RAISE	RAISE_APPLICATION_ERROR()
• RAISE is keyword	• RAISE_APPLICATION_ERROR() is procedure
• Exception will be raised using name here.	• Exception will be raised using error code here.
• Example: RAISE one_divide;	• Example: RAISE_APPLICATION_ERROR(-20050, 'you cannot divide with 1');

PRAGMA EXCEPTION_INIT():

STUDENT

SID	SNAME	M1
PK		CHECK

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

INSERT INTO student VALUES(1,'A',60);

INSERT INTO student VALUES(1,'B',50);
Output:
ERROR:
ORA-00001: unique constraint (C##BATCH6PM.SYS_C008442) violated
```

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

```
INSERT INTO student VALUES(null, 'B', 80);
Output:
ERROR:
ORA-01400: cannot insert NULL into "C##BATCH6PM"."STUDENT"."SID")
```

-1400	Error Code
cannot insert NULL	Message
NO ERROR NAME defined	Error Name

```
INSERT INTO student VALUES(2,'B',789);
Output:
ERROR:
ORA-02290: check constraint (C##BATCH6PM.SYS_C008441) violated
```

-2290	Error Code
check constraint violated	Error Message
NO ERROR NAME defined	Error Name

Note:

- Any instruction started with PRAGMA is called Compiler Directive.
- It is instruction to the compiler.

- **It instructs that, before compiling PL/SQL program first execute this line.**

PRAGMA EXCEPTION_INIT():

- **It is a compiler directive.**
- **directive => command**
- **It is a command to the compiler.**

- **Some errors have names.**
- **Some errors does not have names.**
- **To defined name for unnamed exception, we use PRAGMA EXCEPTION_INIT().**
- **We write this instruction in DECLARATION PART.**

Syntax:

```
PRAGMA EXCEPTION_INIT(<user_Defined_Exception_name>, <built-in_error_code);
```

Example:

```
    check_violate EXCEPTION;

    PRAGMA EXCEPTION_INIT(check_violate, -2290);
```

- **To handle the exception in EXCEPTION block a name is required.**
- **That's why to define name for unnamed exception, we use PRAGMA EXCEPTION_INIT().**

Example program on PRAGMA EXCEPTION_INIT():

Program to insert student record into STUDENT table:

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

DECLARE
    check_violate EXCEPTION;
    PRAGMA EXCEPTION_INIT(check_violate,-2290);
BEGIN
    INSERT INTO student VALUES(&sid, '&sname', &m1);
    COMMIT;
    dbms_output.put_line('record inserted..');

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

```
/


 Output:
 Enter value for sid: 1
 Enter value for sname: A
 Enter value for m1: 80
 record inserted..

 Output:
 Enter value for sid: 1
 Enter value for sname: A
 Enter value for m1: 80
 PK does not accept duplicates..

 Output:
 Enter value for sid: 2
 Enter value for sname: B
 Enter value for m1: 457
 marks must be b/w 0 to 100
```

Exception Handling

Exception	**runtime error**
Exception handling	**the way of handling run time error** **add EXCEPTIO block**

2 types:
built-in

- **zero_divide**
- **value_error**
- **no_data_found**
- **too_many_rows**
- **invalid_Cursor**
- **cursor_already_open**
- **dup_val_on_index**
- **others**

user-defined

3 steps:
declare
raise

handle

we can raise exception in 2 ways:
raise
raise_Application_error()

pragma_exception_init():

- **compiler directive**
- **used to define a name to unnamed exception**

Stored Procedures

Monday, October 2, 2023 6:10 PM

Procedures:

- **Procedure is a DB Object. It is stored physically in ORACLE DB.**
- **Procedure is a named block of statements that gets executed on calling.**
- **A procedure can be also called as "Sub Program".**

In C:
Function:
a set of statements => calling

In Java:
Method:
a set of statements => calling

Types of procedures:

2 Types:

- **Stored procedure**
- **Packaged procedure**

Stored procedure:
If a procedure is defined in SCHEMA then it is called "Stored procedure".

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

Packaged procedure:
If a procedure is defined in PACKAGE then it is called "Stored procedure".

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

Syntax to define Stored Procedure:

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

HEADER

+

BODY

Define a procedure to add 2 numbers:

Note:

Granting permission to create procedure:

login as DBA:

Define a procedure to add 2 numbers:

Note:
don't give sizes for parameters

x NUMBER	**valid**
x NUMBER(4)	**invalid**

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

- **Open text editor**
- **Write above code**
- **Save it in "D:" Drive, "batch6pm" Folder, with the name "ProcedureDemo.sql".**

- **Open sqlplus**
- **login as user and write following command:**

```
SQL> @d:\batch6pm\ProcedureDemo.sql
Output:
Procedure created.
```

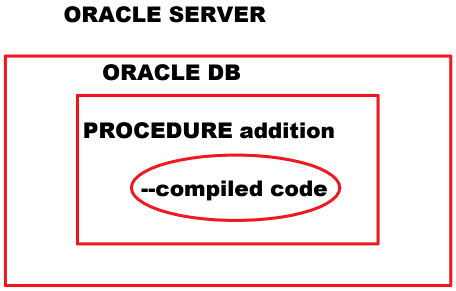
Granting permission to create procedure:

login as DBA:
username: system
password: naresh

```
SQL> GRANT create procedure TO c##batch6pm;
Output:
Grant succeeded.
```

When procedure is created, compiled code will be stored in it.

When we call the procedure, it runs compiled code. Every time code will not be compiled. Just, it runs compiled code. So, it improves the performance.



Calling Stored procedure:

We can call a Stored procedure using 3 ways:
- **From SQL prompt**
- **From PL/SQL program**
- **From Front-End Application [Java, Python, C#]**

- **From SQL prompt:**

- **"EXEC[UTE]" command is used to call a stored procedure from SQL prompt.**

Syntax:

```
SQL> EXEC[UTE] <procedure_name>(<argument_list>);
```

Example:

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

Calling from PL/SQL program:

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

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

```
Output:
enter value for a: 5
enter value for b: 4
sum=9
```

Parameters:

```
CREATE OR REPLACE 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
Header

Body

```
SQL> EXEC addition(5,4);   --procedure call          5,4 Actual parameters
```

- **Parameter / Argument:**
- **A Parameter can be also called as Argument.**

- **A local variable which is declared in procedure header is called "Parameter".**

  Parameters are 2 types:
  - **Formal parameter**
  - **Actual parameter**

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

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

Syntax to define formal parameter:

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

Example:

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

Parameter Modes:

- There are 3 parameter modes in PL/SQL. They are:
  - IN   [default]
  - OUT
  - IN OUT

IN:

- It is default one.
- It captures input.
- It is used to bring value into procedure from out of procedure.
- In procedure call, it can be constant or variable.
- It is read-only parameter.

OUT:

- It is used to send the result [output] out of procedure.
- In procedure call, it must be variable only.
- It is read-write parameter.

IN OUT:

- It is used to bring value into procedure from out of procedure and same parameter can be used to send the result out of procedure.
- In procedure call, it must be variable only.
- It is read-write parameter.

Example on OUT parameter:

Define a procedure to add 2 numbers. And send the result out of procedure.

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

Calling from SQL prompt:

```
SQL> VARIABLE a NUMBER
SQL> EXEC addition(2,3,:a);
SQL> PRINT a
Output:
a
--
5
```

In above example a is bind variable

Bind Variable:

- **A variable which is declared at SQL prompt is called "Bind variable".**
- **To declare the variable at SQL prompt, we use the command "VAR[IABLE]".**
- **To assign value to bind variable, we use bind operator : [colon].**
- **PRINT command is used to print bind variable value.**

Calling from PL/SQL program:

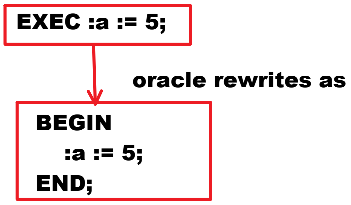
```
DECLARE
    a NUMBER(4);
    b NUMBER(4);
    c NUMBER(4);
BEGIN
    a := &a;
    b := &b;
    addition(a,b,c);
    dbms_output.put_line('sum=' || c);
END;
/
```

Example on IN OUT parameter:

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

Calling from SQL prompt:

```
SQL> VARIABLE a NUMBER
SQL> EXEC :a := 5;
SQL> EXEC square(:a);
SQL> PRINT a
Output: 25
```



```
EXEC :a := 5;
```

oracle rewrites as

```
BEGIN
    :a := 5;
END;
```

IN	**used to bring value into procedure from out of procedure**
OUT	**used to send result out of procedure**
IN OUT	**used to bring value into procedure from out of procedure and same parameter can be used to send result out of procedure**

Parameter mapping techniques /

Parameter association techniques /
Parameter Notations:

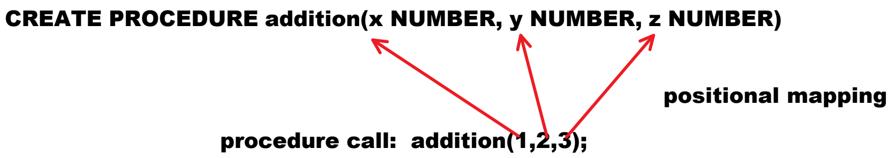
There are 3 parameter mapping techniques. they are:

- **Positional Mapping**
- **Named Mapping**
- **Mixed Mapping**

Positional Mapping:

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

Example:



CREATE PROCEDURE addition(x NUMBER, y NUMBER, z NUMBER)

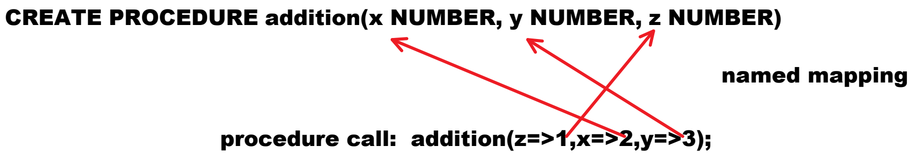
positional mapping

procedure call: addition(1,2,3);

Named Mapping:

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

Example:



CREATE PROCEDURE addition(x NUMBER, y NUMBER, z NUMBER)

named mapping

procedure call: addition(z=>1,x=>2,y=>3);

Mixed Mapping:

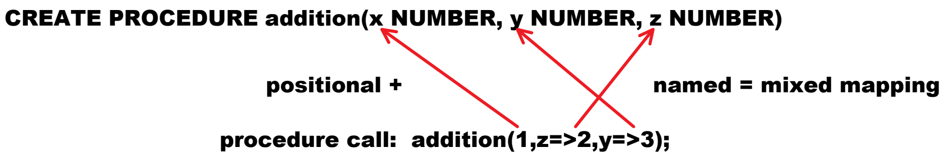
In mixed mapping, actual parameters will be mapped with formal parameters based on positions and names.

Example:

CREATE PROCEDURE addition(x NUMBER, y NUMBER, z NUMBER)

procedure call: addition(z=>1,2,3); --ERROR

Note:
After named mapping we cannot use positional mapping



CREATE PROCEDURE addition(x NUMBER, y NUMBER, z NUMBER)

positional +   **named = mixed mapping**

procedure call: addition(1,z=>2,y=>3);

Define a procedure to add 3 numbers:

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

END;
/
```

```
SQL> EXEC addition(1,2,3);    --positional mapping
sum=6
x=1
y=2
z=3
```

```
SQL> EXEC addition(z=>1,x=>2,y=>3);    --named mapping
sum=6
x=2
y=3
z=1
```

```
SQL> EXEC addition(z=>1,2,3);
Output:
ERROR: a positional parameter association may not follow a
named association
```

```
SQL> EXEC addition(1,z=>2,y=>3);     --mixed mapping
sum=6
x=1
y=3
z=2
```

Note:
To perform DML operations define PROCEDURE.

Define a procedure to increase salary of an employee with specific amount:

```
CREATE OR REPLACE PROCEDURE
update_salary(p_empno EMP.EMPNO%TYPE, p_amount FLOAT)
AS
BEGIN
   UPDATE emp SET sal=sal+p_amount WHERE empno=p_empno;
   COMMIT;
   dbms_output.put_line('salary increased..');
END;
/
```

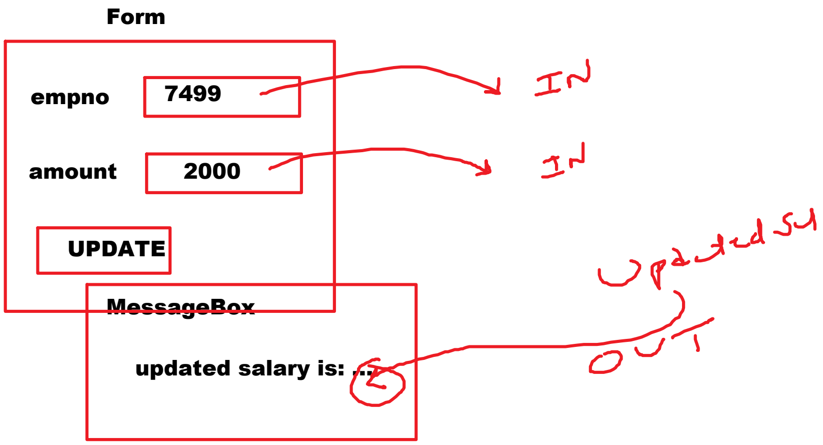
Calling Procedure:

```
SQL> EXEC update_salary(7499,2000);
```

Output:

```
salary increased..
```

Define a procedure to increase salary of an employee with specific amount. Send updated salary out of procedure:



procedure call:

```
update_salary(7499,2000,:s);
```

```
CREATE OR REPLACE PROCEDURE
update_salary(p_empno IN NUMBER, p_amount IN FLOAT, p_sal OUT NUMBER)
AS
BEGIN
   UPDATE emp SET sal=sal+p_amount WHERE empno=p_empno;
   COMMIT;
   dbms_output.put_line('salary increased..');

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

Calling procedure:

```
   SQL> VARIABLE s NUMBER
   SQL> EXEC update_salary(7499,2000,:s);
   SQL> PRINT s
```

emp

empno	ename	sal
7369	SMITH	800
7499	ALLEN	5000

ACCOUNT

ACNO	NAME	BALANCE
1001	A	500000
1002	B	700000

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

```
 INSERT INTO account VALUES(1001,'A',500000);
 INSERT INTO account VALUES(1002,'B',700000);
 COMMIT;
```

Define a procedure to perform withdraw operation:

p_acno	p_amount	v_balance
1001	100000	500000

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

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

   UPDATE account SET balance=balance-p_amount WHERE acno=p_acno;
   COMMIT;

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

ACNO	NAME	BALANCE
1001	A	500000
1002	B	700000

Calling Procedure:

```
SQL> EXEC withdraw(1001,800000);
Output:
ERROR at line 1:
ORA-20050: Insufficient balance


SQL> EXEC withdraw(1001,100000);
successful withdrawl
```

Define a procedure to perform deposit operation:

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

      COMMIT;

      dbms_output.put_line('deposit is successful');
   END;
   /
```

calling procedure:

```
SQL> EXEC deposit(1001,400000);
deposit is successful
```

Define a procedure to perform deposit operation.

After performing deposit operation updated balance send out of the procedure:

```
CREATE OR REPLACE PROCEDURE
deposit(p_acno IN NUMBER, p_amount IN NUMBER,
p_balance OUT NUMBER)
AS
BEGIN
   UPDATE account SET balance=balance+p_amount
   WHERE acno=p_acno;
   COMMIT;
   dbms_output.put_line('amount credited');

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

p_acno	p_amount	p_balance
1001	200000	1000000

ACNO	NAME	BALANCE
1001	A	~~800000~~ 1000000
1002	B	700000

b
1000000

calling procedure:

```
SQL> VAR b NUMBER
SQL> EXEC deposit(1001,200000,:b);
Output:
amount credited

SQL> PRINT b
Output:
b
--
1000000
```

Define a procedure to perform fund transfer operation:

ACCOUNT

ACNO	NAME	BALANCE
1001	A	1000000
1002	B	700000

1001 account holder is transferring 200000 to 1002
after transferring

ACCOUNT

ACNO	NAME	BALANCE
1001	A	800000
1002	B	900000

procedure call:

```
   fund_transfer(1001, 1002, 200000);
```

p_from	p_to	p_amount

Update
bal=bal-p_amount

Update
bal=bal+p_amount

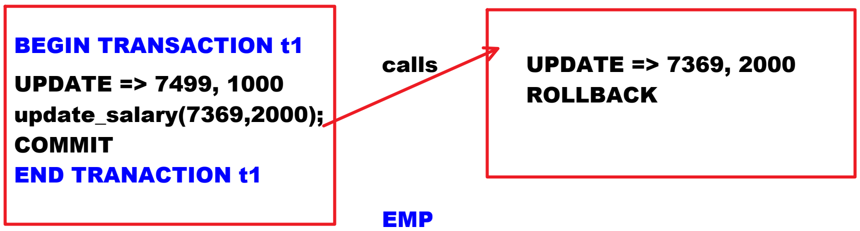
PRAGMA AUTONOMOUS_TRANSACTION:

PL/SQL program [main]

PROCEDURE update_salary

PL/SQL program [main]

PROCEDURE update_salary



BEGIN TRANSACTION t1

UPDATE => 7499, 1000
update_salary(7369,2000);
COMMIT

END TRANACTION t1

calls

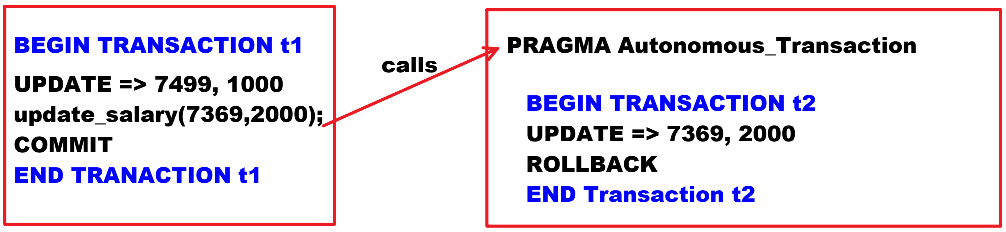
UPDATE => 7369, 2000
ROLLBACK

EMP

EMPNO	ENAME	SAL
7369	SMITH	5000+2000 = 7000 rolled back => 5000
7499	ALLEN	8000+1000 = 9000 rolled back => 8000

PL/SQL program [main]

PROCEDURE update_salary



BEGIN TRANSACTION t1

UPDATE => 7499, 1000
update_salary(7369,2000);
COMMIT

END TRANACTION t1

calls

PRAGMA Autonomous_Transaction

BEGIN TRANSACTION t2
UPDATE => 7369, 2000
ROLLBACK
END Transaction t2

EMP

EMPNO	ENAME	SAL
7369	SMITH	5000+2000 = 7000 => rolled back => 5000
7499	ALLEN	8000+1000 = 9000 => committed => 9000

- By default, a separate transaction will not be created for procedure.
- A transaction started in main program will be continued for PROCEDURE also.
- COMMIT or ROLLBACK in the procedure will be applied for main program also. This is the problem.

- To solve this problem, we need to create a separate transaction for procedure.

- To create a separate transaction for procedure we use PRAGMA AUTONOMOUS_TRANSACTION.

- PRAGMA AUTONOMOUS_TRANSACTION is a compiler directive [command]. It is a command to the compiler.

- We declare it in declaration part.

Define a procedure to increase salary of an employee with specific amount. Create a separate transaction for the procedure:

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

```
/
```

main program:

```
BEGIN
    UPDATE emp SET sal=sal+2000 WHERE empno=7499;
    update_salary(7369,1000);
    COMMIT;
END;
/
```

user_procedures
user_source

user_procedures:

- it is a built-in table / system table

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

  DESC user_procedures;

  To see all procedures information:

  ```
  SELECT object_name, object_type
  FROM user_procedures;
  ```

user_source:

- it is a built-in table / system table

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

  DESC user_source;

  To see all procedures information:

  ```
  SELECT DISTINCT name, type
  FROM user_source;
  ```

  To see procedure's code:

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

Dropping Procedure:

Syntax:

```
DROP PROCEDURE <procedure_name>;
```

Example:

```
DROP PROCEDURE square;
```

c##batch6pm	c##userA
PROCEDURE addition	
	EXEC c##batch6pm.addition(1,2,3); Output: ERROR: addition is invalid identifier
to give permission on procedure to other user: GRANT execute ON addition TO c##userA;	

	EXEC c##batch6pm.addition(1,2,3);
	Output:
	sum=6

Note:

- A procedure can be also called as sub program
- A function can be also called as sub program

Advantages of sub programs [procedures and functions]:

- It improves the performance.
- It provides reusability.
- It reduces length of code.
- It provides security. Only authorized can call the procedure.
- Better maintenance.
- It improves the understandability.

Fund Transfer Procedure Assignment

Monday, October 9, 2023 7:51 PM

Code:

```
CREATE OR REPLACE PROCEDURE
fund_transfer(p_from NUMBER, p_to NUMBER, p_amount NUMBER)
AS
BEGIN
    UPDATE account SET balance=balance-p_amount WHERE
    acno=p_from;
    UPDATE account SET balance=balance+p_amount WHERE acno=p_to;
    COMMIT;
    dbms_output.put_line('transaction successful');
END;
/
```

Stored Functions

Thursday, October 5, 2023 6:54 PM

Functions:

- Function is a DB Object.

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

- To perform DML operations, define PROCEDURE.
  To perform FETCH operations [SELECT] or calculations, define FUNCTION.

Example:

Opening_Account => INSERT => PROCEDURE
Withdraw => UPDATE => PROCEDURE
Deposit => UPDATE => PROCEDURE

Check_Balance => SELECT => FUNCTION
experience => Calculation => FUNCTION
Trans_statement => SELECT => FUNCTION

Types of Functions:

2 Types:

- Stored Functions
- Packaged Functions

Stored Function:

- A Function which is defined in SCHEMA is called "Stored Function".

Example:

SCHEMA c##batch6pm
FUNCTION check_balance => Stored Function

Packaged Function:

- A Function which is defined in PACKAGE is called "Packaged Function".

Example:

```
SCHEMA c##batch6pm
    PACKAGE bank
            FUNCTION check_balance    => Packaged Function
```

Syntax to define Stored Function:

```
CREATE [OR REPLACE] FUNCTION
<function_name>[(<parameter_list>)] RETURN <return_type>
IS / AS
    --declare the variables
BEGIN
    --executable statements
    RETURN <expression>;
END;
/
```

Note:

- A Function returns the value.
- Here, returning value is mandatory.
- If we don't want to return the value then define PROCEDURE.
- For procedure returning value is optional.

Example on Stored Function:

Define a function to multiply 2 numbers:

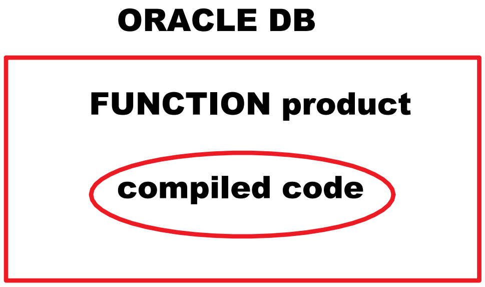
```
CREATE OR REPLACE FUNCTION
product(x NUMBER, y NUMBER) RETURN NUMBER
AS
    z NUMBER(4);
BEGIN
    z:=x*y;

    RETURN z;
END;
```

```
/
```

- **Write above code in a new file. save it in "D:" Drive, "batch6pm" folder, with the name "FunctionDemo.sql".**

- **open sqlplus. login as user. write following code:**

```
SQL>@d:\batch6pm\FunctionDemo.sql
Output:
Function created.
```



ORACLE DB

FUNCTION product

compiled code

Calling the function:

3 ways:

- **From SQL prompt**
- **From PL/SQL program [main program]**
- **From Front-End Application [Java, Python, C#]**

From SQL prompt:

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

From PL/SQL program [main program]:

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

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

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

Define a function to find experience of an employee:

```
CREATE OR REPLACE 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 Function:

```
SQL> SELECT experience(7934) FROM dual;
Output:
EXPERIENCE(7934)
--------------------------------
             41
```

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

Output:
 EMPNO ENAME      HIREDATE        EXP
------------------------------------------------------------------
      7369 SMITH      17-DEC-80        42
      7499 ALLEN      20-FEB-81        42
      7521 WARD       22-FEB-81        42
      7566 JONES      02-APR-81        42
```

Define a Function to display the emp records of specific dept:

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

   RETURN c1;
END;
/
```

Calling Function:

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

Define a function to display top n salaried emp records:

```
CREATE OR REPLACE FUNCTION
gettopn(n NUMBER) RETURN SYS_REFCURSOR
AS
   c1 SYS_REFCURSOR;
BEGIN
   OPEN c1 FOR SELECT * FROM (SELECT ename,sal,
   dense_rank() over(order by sal desc) as rnk
   FROM emp)
   WHERE rnk<=n;

   RETURN c1;
END;
/
```

Calling Function:

```
   SELECT gettopn(3) FROM dual;
```

Differences between 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
• We use OUT parameter to return the value.	• We use RETURN keyword to return the value.
• A procedure can return multiple values.	• A function can return 1 value only.
• To perform DML operations, define PROCEDURE.	• To perform FETCH operations (or) calculations, define FUNCTION.
• Example: WITHDRAW => update	• Example: CHECK_BALANCE => select
• PROCEDURE cannot be called from SQL command like select, update.	• FUNCTION can be called from SQL command like select, update.
• To call a procedure from SQL prompt use EXEC command	• To call a function from SQL prompt use SQL command like select, update.

Can we perform DML operations using FUNCTION?
YES. But, it is not recommended.
If we perform DML operation through FUNCTION, this function

cannot be called from SQL command like SELECT, UPDATE.

```
CREATE OR REPLACE FUNCTION demo123 RETURN NUMBER
AS
BEGIN
    UPDATE emp SET sal=sal+1000;
    COMMIT;
    RETURN 5;
END;
/
Output:
Function Created.

SQL> SELECT demo123 FROM dual;
Output:
ERROR: cannot perform a DML operation inside a query
```

Can we define OUT parameters in FUNCTION?

YES. But, it is not recommended.

- **Don't use OUT parameters in FUNCTION.**
- **If we define OUT parameter in FUNCTION, it cannot be called from SQL command.**
- **It changes meaning of FUNCTION. FUNCTION standard is: function must return 1 value.**

Dropping a Function:

Syntax:

```
DROP FUNCTION <function_name>;
```

Example:

```
DROP FUNCTION product;
```

user_procedures:

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

to see functions list created by user:

```
column object_name format a20

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 all functions list:

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

To see Function code:

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

To give permission to other user on Function:

c##batch6pm:

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

Login as userA:

```
SELECT c##batch6pm.experience(7369) as exp FROM dual;
Output:
exp
-----
42
```

PACKAGES

Friday, October 6, 2023 7:22 PM

PACKAGES:

- PACKAGE is a DB Object.
- PACKAGE is a collection of procedures, functions, global variables, cursors and exceptions.
- All related procedures and functions can be placed at one place.

Example:

PACKAGE bank

```
PROCEDURE OPENING_ACCOUNT => INSERT
PROCEDURE CLOSING_ACCOUNT => DELETE
PROCEDURE WITHDRAW => UPDATE

FUNCTION CHECK_BALANCE => SELECT
FUNCTION TRANS_STATEMENT => SELECT
FUNCTION EXPERIENCE => CALCULATION
```

packaged procedures

packaged functions

Creating a Package:

- To create a package follow 2 steps. They are:
  - Define Package Specification
  - Define Package Body

Defining Package Specification:

Syntax:

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

Defining Package Body:

Syntax:

Defining Package Body:

Syntax:

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

Create following Package:

PACKAGE math

PROCEDURE addition
FUNCTION product

granting permission to user to create the package:

login as DBA:
username: system
password: naresh

```
GRANT resource
TO c##batch6pm;
```

Defining Package Specification:
loin as user: c##batch6pm

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

Defining 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;
```

```
/
```

Calling Packaged Procedure:

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

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

Create following package:

PACKAGE HR

```
PROCEDURE HIRE => INSERT
PROCEDURE HIKE => UPDATE
PROCEDURE FIRE => DELETE

FUNCTION EXPERIENCE => CALCULATION
```

Defining Package Specification:

```
CREATE OR REPLACE PACKAGE HR
AS
   PROCEDURE HIRE(p_empno NUMBER, p_ename VARCHAR2);
   PROCEDURE HIKE(p_empno NUMBER, p_amount NUMBER);
   PROCEDURE FIRE(p_empno NUMBER);

   FUNCTION EXPERIENCE(p_empno NUMBER) RETURN NUMBER;
END;
/
```

Defining 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 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('salary increased..');
    END HIKE;


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


    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 Packaged procedure:

```
    SQL> EXEC HR.HIRE(5001,'AA');
    Output:
    record inserted..

    SQL> EXEC HR.HIKE(7499,1000);
    Output:
    salary increased..

    SQL> EXEC HR.FIRE(5001);
    Output:
```

```
record deleted..

SQL> SELECT HR.EXPERIENCE(7369) FROM dual;
Output:
HR.EXPERIENCE(7369)
-------------------
             42

SQL> select empno,ename,hiredate,
       HR.experience(empno) AS exp
       FROM emp;
Output:
    EMPNO ENAME      HIREDATE         EXP
---------------------------------------------------------------------
     7369 SMITH      17-DEC-80         42
     7499 ALLEN      20-FEB-81         42
     7521 WARD       22-FEB-81         42
```

CREATE Following Package:

ACCOUNT

ACNO	NAME	BALANCE
1001	A	800000
1002	B	500000

PACKAGE bank

PROCEDURE opening_account => INSERT
PROCEDURE withdraw => UPDATE
PROCEDURE deposit => UPDATE
PROCEDURE fund_transfer => UPDATE

FUNCTION check_balance => SELECT

Advantages of Package:

- **We can group related procedures and functions.**
- **It improves the performance.**
- **Better maintenance.**
- **Packaged Procedures and Packaged Functions can be overloaded.**
  Note: Stored Procedure and Stored Function cannot be overloaded.
- **We can declare global variable.**
- **We can make members as public or private.**

NOTE:
Packaged Procedures and Packaged Functions can be overloaded.
Stored Procedure and Stored Function cannot be overloaded.

OVERLOADING:

- **Defining multiple sub programs [procedures/functions] with same name and different signature is called "OVERLOADING".**

- **Different signature means,**
  - **change in no of parameters**
  - **change in data types**
  - **change in order of parameters**

Example:
PACKAGE p1

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

change in
no of parameters

PACKAGE p2

```
FUNCTION demo(x NUMBER, y VARCHAR2)
FUNCTION demo(x DATE, y TIMESTAMP)
```

change in
data types

PACKAGE p3

```
FUNCTION demo(x NUMBER, y VARCHAR2)
FUNCTION demo(x VARCHAR2, y NUMBER)
```

change in
order of parameters

Example on OVERLOADING:

PACKAGE OLdemo

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

Defining Package Specification:

```
CREATE OR REPLACE PACKAGE Oldemo
AS
   x NUMBER := 500;   --global variable

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

Defining Package Body:

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

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

Calling:

```
   SQL> SELECT Oldemo.addition(2,3) FROM dual;
   Output:
   5

   SQL> SELECT Oldemo.addition(2,3,4) FROM dual;
   Output:
   9
```

```
SQL> EXEC dbms_output.put_line(OLDemo.x);
Output:
500


DECLARE
   a NUMBER := 100;
   b NUMBER(4);
BEGIN
   b := OLDemo.x+a;

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

Note:
Using PACKAGE, We can make members as public or private.

Defining Package Specification means, we are making members as public.

Package Specification

```
PROCEDURE p2
PROCEDURE p3
```

p2 => public
p3 => public

Within SCHEMA,
from anywhere we can call

Package Body demo

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

p1 => private
within package only it can be used

- **If a procedure is defined in PACKAGE BODY, but not declared in PACKAGE SPECIFICATION then it is called "private member".**
- **Private member can be used within PACKAGE only**

- **If a procedure is defined in PACKAGE BODY also it is declared in**

PACKAGE SPECIFICATION then it is called "Public Member".

- **Public member can be used within SCHEMA .**

Dropping Package:

Syntax:

```
DROP PACKAGE <package_name>;
```

Example:

```
DROP PACKAGE OLdemo;
```

user_source:

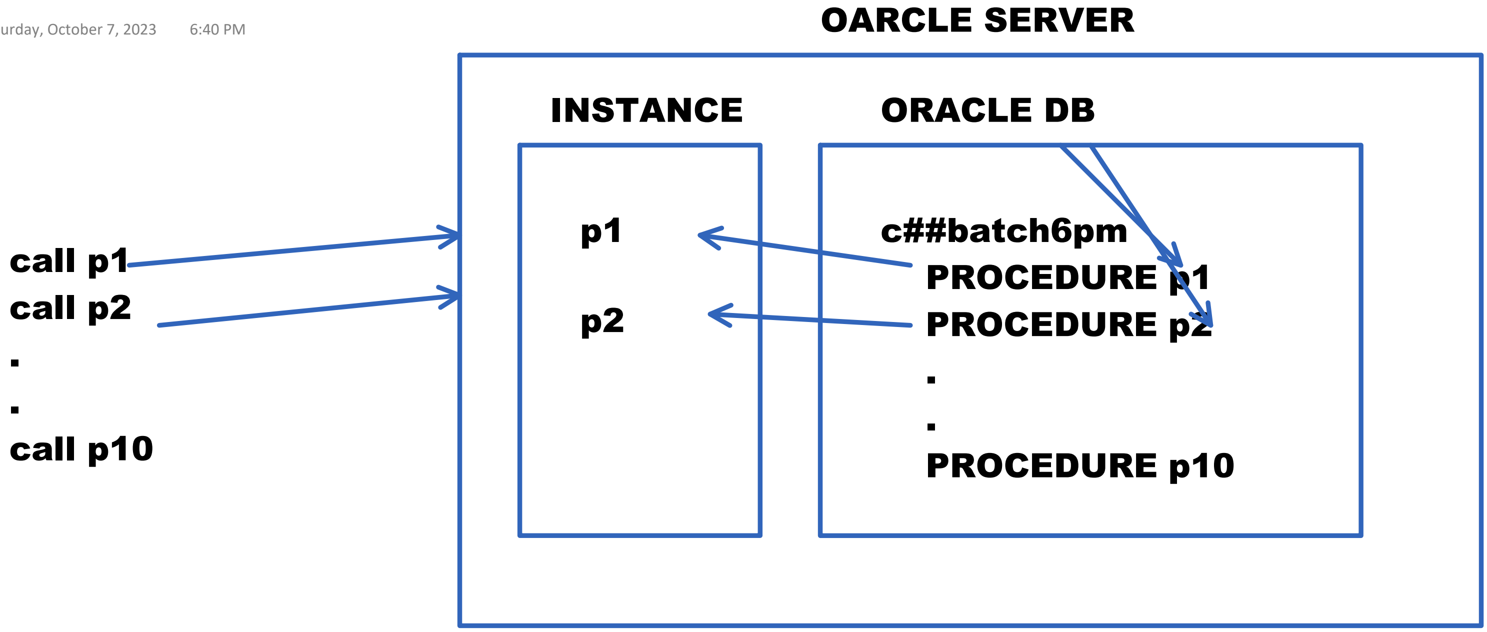
- **it is a system table.**
- **It maintains all procedures, functions, packages and triggers information including code**

to see packages list:

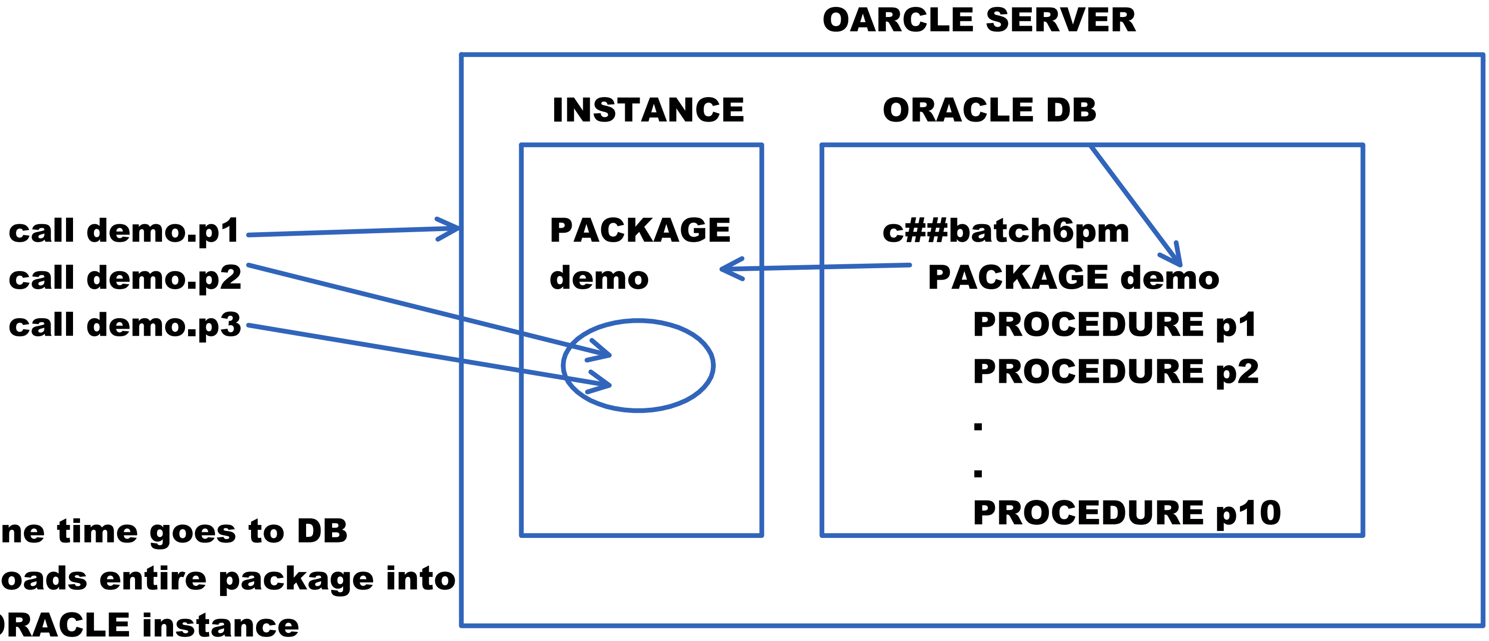
```
SELECT DISTINCT name, type
FROM user_source
WHERE type='PACKAGE';
```

to see package code:

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



10 times goes to DB

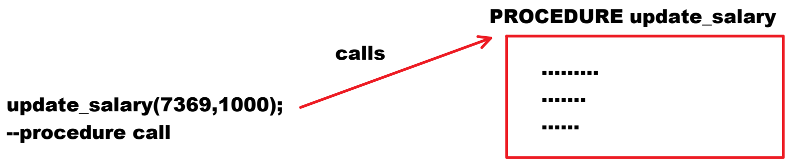


one time goes to DB
Loads entire package into
ORACLE instance

It improves the performance

TRIGGERS

Monday, October 9, 2023 6:10 PM

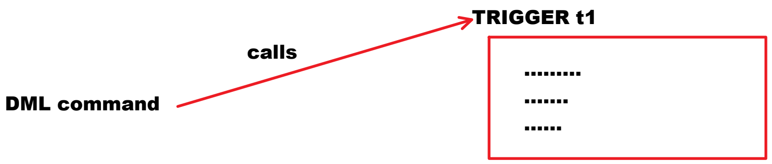


update_salary(7369,1000);
--procedure call

calls

PROCEDURE update_salary

....
....
......



DML command

calls

TRIGGER t1

....
....
......

TRIGGER:

- TRIGGER is a DB Object.

- TRIGGER is a named block of statements that gets executed automatically when we submit DML command.

- TRIGGER is same as PROCEDURE. But,
  For PROCEDURE EXECUTION explicit call is required.
  For TRIGGER EXECUTION explicit call is not required.
  TRIGGER will be called implicitly when we submit the DML command.

- PROCEDURE is used to perform DML operations.
- TRIGGER is used to control the DML operations.

updating salary
PROCEDURE

you can update b/w 10AM to 4PM
Before 10Am and After 4PM don't allow DMLs
TRIGGER

TRIGGER can be used for 3 purposes:

- To control the DMLs.
  Example:
  Don't allow DMLs on Sunday
  Before 10AM and After 4PM don't allow DMLs

- For Auditing the tables or databases.
  Example:
  - which user
  - on which date & at which time
  - which data he accessed
  - what was old data
  - what is new data

    Above details can be recorded in another table

- To implement our own business rules [constraints]
  Example:
  don't allow user to decrease the salary

Types of Triggers:

3 Types:

- **Table Level Trigger / DML Trigger   [it is defined by PL/SQL Developer]**
- **Schema Level Trigger / DDL Trigger / System Trigger [it is defined by DBA]**
- **Database Level Trigger / DDL Trigger / System Trigger [it is defined by DBA]**

Table Level Trigger / DML Trigger:

- **A Trigger which is created on table is called "Table Level Trigger".**
- **PL/SQL Developer defines it.**

Table level Triggers are 2 types. They are:

- **Statement Level Trigger**
- **Row Level Trigger**

Statement Level Trigger:
This Trigger gets executed once for one SQL statement.

Example:

```
UPDATE emp SET sal=sal+1000;
--calls trigger. trigger code gets executed once
```

Row Level Trigger:
This Trigger gets executed once for one row affected by DML command.

Example:

```
UPDATE emp SET sal=sal+1000;
--14 rows updated.
--calls trigger. trigger code gets executed 14 times

UPDATE emp SET sal=sal+1000 WHERE job='SALESMAN';
--6 rows updated.
--calls trigger. trigger code gets executed 6 times
```

Syntax of defining Table Level Trigger:

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

Example on Statement Level Trigger:

```
CREATE OR REPLACE TRIGGER t1
AFTER INSERT OR UPDATE OR DELETE
ON emp
BEGIN
   dbms_output.put_line('Statement level trigger executed');
END;
/
```

Testing:

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

Output:

```
14 rows updated.
Statement level trigger executed
```

Example on Row Level Trigger:

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

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

Examples on Controlling DMLs:

Define a trigger not to allow the user to perform DMLs on Sunday:

```
CREATE OR REPLACE TRIGGER t3
BEFORE INSERT OR UPDATE OR DELETE
ON emp
BEGIN
   IF to_char(sysdate,'dy')='sun' THEN
      RAISE_APPLICATION_ERROR(-20050, 'You cannot perform
      DMLS on sunday');
   END IF;
END;
/
```

```
Testing:
From Monday to Saturday:
UPDATE emp SET sal=sal+1000;
Output:
12 rows updated.

On Sunday:
UPDATE emp SET sal=sal+1000;
Output:
ERROR at line 1:
ORA-20050: You cannot perform DMLS on sunday
ORA-06512: at "C##BATCH6PM.T3"
```

Define a trigger not to allow the user to perform DMLs as following:
From Monday To Friday:
Before 10 AM and After 4 PM don't allow DMLs
On Saturday:
Before 10 AM and After 2 PM don't allow DMLs
On Sunday:
Don't allow DMLs

capture weekday number => d := to_char(sysdate,'d')
capture hours part in 24 hrs format => h := to_char(sysdate,'HH24');

Program:

```
CREATE OR REPLACE TRIGGER t4
BEFORE INSERT OR UPDATE OR DELETE
ON emp
DECLARE
   d INT;
   h INT;
BEGIN
   d := to_char(sysdate,'d');
   h := to_char(sysdate,'HH24');

   IF d BETWEEN 2 AND 6 AND h NOT BETWEEN 10 AND 15 THEN
      raise_application_error(-20050,'from mon to fri, before 10am
      and after 4pm dmls not allowed');
   ELSIF d=7 AND h NOT BETWEEN 10 AND 13 THEN
      raise_application_error(-20060,'on sat, before 10am and after
      2pm dmls not allowed');
   ELSIF d=1 THEN
      raise_application_error(-20070,'on sun, dmls are not allowed');
   END IF;
END;
/
```

Before Trigger:

- **First Trigger gets executed**
- **then DML will be performed**

After Trigger:

- **First DML will be performed**
- **then Trigger gets executed**

Disabling and Enabling Trigger:

```
ALTER TRIGGER t4 disable;
--t4 trigger disable. temporarily it will not work


ALTER TRIGGER t4 enable;
--again t4 trigger will work
```

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

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

Testing:

```
update emp set empno=1234
where ename='SMITH';
```

Output:

```
ERROR at line 1:
ORA-20050: you cannot update empno
```

ORA-06512: at "C##BATCH6PM.T5", line 2

OF empno	**user cannot update empno**
OF empno,ename	**user cannot update empno, ename**

:NEW and :OLD:

- **:NEW and :OLD are built in variables.**
- **These are bind variables.**
- **When Trigger code is executed, at runtime 2 variables will be created. They are: :NEW and :OLD**
- **These are ROWTYPE variables. They can hold entire row.**
- **:NEW and :OLD are called pseudo records.**
- **:NEW variable holds NEW ROW.**
- **:OLD variable holds OLD ROW.**
- **:NEW and :OLD can be used in Row Level Trigger only. These cannot be used in Statement Level Trigger.**

DML	:NEW	:OLD
INSERT	NEW ROW	NULL
UPDATE	NEW ROW	OLD ROW
DELETE	NULL	OLD ROW

EMP

EMPNO	ENAME	SAL
7369	SMITH	6000
7499	ALLEN	5000
7521	WARD	3000

In case of INSERT:

INSERT INTO emp VALUES(1001,'A',2000);

:NEW

EMPNO	ENAME	SAL
1001	A	2000

:NEW.SAL

:OLD

EMPNO	ENAME	SAL
null	null	null

:OLD.SAL

In case of UPDATE:

UPDATE emp SET sal=sal+2000 WHERE empno=7369;

:NEW

EMPNO	ENAME	SAL
7369	SMITH	8000

:OLD

EMPNO	ENAME	SAL
7369	SMITH	6000

EMP

EMPNO	ENAME	SAL
7369	SMITH	~~6000~~ 8000
7499	ALLEN	5000

7521	WARD	3000

In case of DELETE:

```
DELETE FROM emp WHERE empno=7369;
```

:NEW

EMPNO	ENAME	SAL
null	null	null

:OLD

EMPNO	ENAME	SAL
7369	SMITH	8000

Define a trigger to record deleted emp records in "Emp_Resign" Table:

EMP

EMPNO	ENAME	SAL
7369	SMITH	8000
7499	ALLEN	5000
7521	WARD	3000

EMP_RESIGN

EMPNO	ENAME	SAL
:OLD.EMPNO	:OLD.ENAME	:OLD.SAL

```
DELETE FROM emp WHERE empno=7521;
```

:NEW

EMPNO	ENAME	SAL
null	null	null

:OLD

EMPNO	ENAME	SAL
7521	WARD	3000

```
CREATE TABLE emp_resign
(
empno NUMBER(4),
ename 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.SAL, sysdate);
END;
/
```

Testing:

```
DELETE FROM emp WHERE empno=7934;
--7934 record will be recorded in emp_resign

DELETE FROM emp WHERE job='SALESMAN';
--all SALESMEN  records will be recorded in emp_resign table
```

Auditing the Table:
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
c##batch6pm	10-OCT-23 7:18.0.0 PM	INSERT	null	null	null	1001	A	4000
USER	SYSTIMESTAMP	op	:old.empno	:old.ename	:old.sal	:new.empno	:new.ename	:new.sal

```
CREATE TABLE emp_audit
(
uname VARCHAR2(15),
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 UPDATING THEN
      op := 'UPDATE;
   ELSIF DELETING THEN
      op := 'DELETE';
   END IF;

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

Testing:

```
INSERT  ...
UPDATE  ..
DELETE  ..

SELECT * FROM emp_audit;
```

Define a Trigger not to allow the user to decrease the salary:

```
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 the
      sal');
```

```
    END IF;
END;
/
```

Testing:

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

ERROR at line 1:
ORA-20050: you cannot decrease the sal
ORA-06512: at "C##BATCH6PM.T8", line 3
```

Schema Level Trigger / DDL Trigger / System trigger:

- **Schema => User**
- **Schema Level Trigger is used to control one user.**
- **DBA defines it.**

Syntax of Schema Level Trigger:

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
BEFORE / AFTER <DDL_LIST>
ON <user_name>.SCHEMA
DECLARE
    --declare the variables
BEGIN
    --executable statements
END;
/
```

Example on Schema Level Trigger:

Define a trigger not to 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 dtop any DB object..');
END;
/
```

Testing:
login as c##batch6pm user:

```
DROP TABLE emp;
Output:
ERROR
```

System Variable	Purpose
ORA_DICT_OBJ_TYPE	It holds object type Example: TABLE, VIEW, PROCEDURE, TRIGGER
ORA_DICT_OBJ_NAME	It holds Object Name Example: EMP, WITHDRAW, T5

ORA_LOGIN_USER	**It holds user name** **Example:** **C##BATCH6PM**
ORA_SYSEVENT	**It holds DDL action performed by user** **Example:** **DROP, TRUNCATE, CREATE, ALTER**

Define a trigger no to allow the user to drop the table.
Allow him to drop remaining DB Objects:
Login as DBA:

```
CREATE OR REPLACE TRIGGER st2
BEFORE DROP
ON c##batch6pm.SCHEMA
BEGIN
    IF ora_dict_obj_type='TABLE' THEN
        raise_application_error(-20050,'you cannot drop');
    END IF;
END;
/
```

Testing:
Login as c##batch6pm:

```
DROP PROCEDURE withdraw;
Output:
Procedure dropped

DROP TABLE emp;
Output:
Error
```

Database Level Trigger / DDL Trigger / System Trigger:

- **DBA creates it.**
- **To control more than one user we use Database Level Trigger.**

Syntax:

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
BEFORE/AFTER <DDL_LIST>
ON DATABASE
DECLARE
    --declare the variables
BEGIN
    --executable statements
END;
/
```

Define a trigger not to allow c##batch6pm, c##batch2pm
users to drop the DB Objects:

Login as DBA:

```
CREATE OR REPLACE TRIGGER dt1
BEFORE DROP
ON DATABASE
BEGIN
    IF ora_login_user IN('C##BATCH2PM', 'C##BATCH6PM')
    THEN
        raise_application_error(-20050,'you cannot drop any
        DB OBject');
    END IF;
END;
/
```

Testing:

```
Login as c##batch6pm:
    DROP TABLE emp;
    Output:
        ERROR

Login as c##batch2pm:
    DROP TABLE emp;
    Output:
        ERROR
```

Dropping Trigger:

Syntax:

```
DROP TRIGGER <trigger_name>;
```

Example:

```
DROP TRIGGER t5;
```

How can you see on which table, on which column triggers are created?

using "user_triggers" system table we can see it

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

to see trigger_code:

```
SELECT text
FROM user_source
WHERE name='T5';
```

Collections

Thursday, October 12, 2023 7:22 PM

Note:
CURSOR is used to hold multiple records.
COLLECTION is used to hold multiple records.

CURSOR has some drawbacks. To avoid drawbacks of CURSOR, COLLECTION concept introduced.

All programs which we can do using CURSOR, can be done using COLLECTION

Collection:

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

Example:

50	x(1)
90	x(2)
30	x(3)
80	x(4)
40	x(5)

number type elements

'RAJU'	a(1)
'KIRAN'	a(2)
'SAI'	a(3)
'VIJAY'	a(4)

string type elements

EMPNO	ENAME	SAL
1001	A	6000

e(1) **e(1).ename** **e(1).sal**

EMPNO	ENAME	SAL
1002	B	4000

e(2) **e(2).ename** **e(2).sal**

EMPNO	ENAME	SAL
1003	C	8000

e(3)

EMPNO	ENAME	SAL
1004	D	3000

e(4)

emp%rowtype elements

Types of Collections:

3 Types:

- **Index By table / Associative Array / PL SQL TABLE**
- **Nested Table**
- **V-Array [Variable size Array]**

x

50

1. declare
x number(2);

data type is ready. so declare it.[1 step]

50
90
30
80
40

1. define data type
2. declare variable for that data type

data type is not ready.
so define data type and declare variable for it.
[2 steps]

Note:

- **For Collection data type is not ready.**
- **We need to define our own collection data type.**
- **Then declare variable for it**

Index By Table / Associative Array / PL SQL Table:

- **Associative array is a table of 2 columns. Those 2 columns are:**
  - **Index**
  - **Element**

  Example:

INDEX	ELEMENT
1	78
2	54
3	91
4	32

Creating Associative Array:

To create associative array follow 2 steps:

- **define our own data type**
- **declare variable for that data type**

defining our own data type:

Syntax:

```
TYPE <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, we can use BINARY_INTEGER (or) PLS_INTEGER.

Declare variable for our own data type:

Syntax:

```
<variable> <type>;
```

Example:

```
x NUM_ARRAY;
```

Collection members:

first	x.first	gives first index
last	x.last	gives last index
next	x.next(2)	gives next index of 2 . i.e. 3
prior	x.prior(2)	gives previuos index of 2. i.e. 1

Create an associative array to hold numbers:

INDEX	ELEMENT
1	40
2	90
3	50
4	38

```
DECLARE
    TYPE num_array IS TABLE OF NUMBER(4) INDEX BY binary_integer;
    x NUM_ARRAY;
BEGIN
    x := num_array(40,90,50,38);

    dbms_output.put_line('x(2)=' || x(2));  --90

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

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

    dbms_output.put_line('x elements are:');
    for i IN x.first .. x.last
    loop
        dbms_output.put_line(x(i));
    end loop;
END;
/
```

Output:

```
x(2)=90
first index=1
last index=4
next index of 2 is:3
prev index of 2 is:1
x elements are:
40
90
50
38
```

In above example:

```
    x := num_array(40,90,50,38):
```

- num_array() is called "Collection Constructor"
- Collection Constructor is a special function. When we define a data type implicitly collection constructor will be defined by ORACLE.
- It is used to initialize the collection.

Create an Associative Array to store dept name:

d

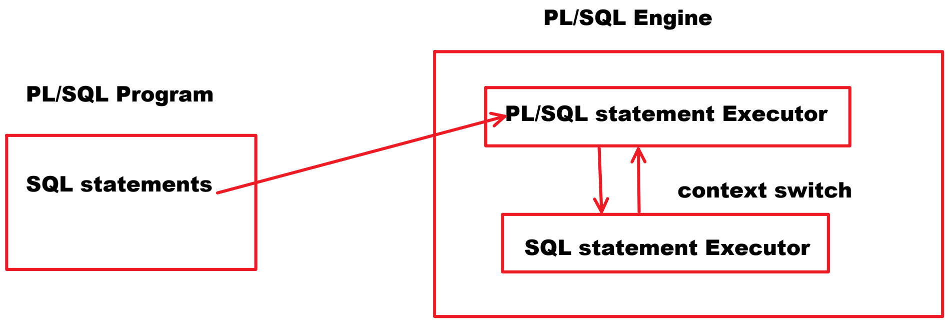
INDEX	ELEMENT
1	ACCOUNTING
2	RESEARCH
3	SALES
4	OPERATIONS

```
DECLARE
   TYPE dept_array IS TABLE OF varchar2(10) INDEX BY binary_integer;
   d dept_array;
BEGIN
   SELECT dname INTO d(1) FROM dept WHERE deptno=10;
   SELECT dname INTO d(2) FROM dept WHERE deptno=20;
   SELECT dname INTO d(3) FROM dept WHERE deptno=30;
   SELECT dname INTO d(4) FROM dept WHERE deptno=40;

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

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

- **Above Program degrades the performance.**
- **IN above program, 4 context switches will occur to get 4 dept names.**
- **If no of context switches are increased then performance will be degraded.**
- **To avoid this problem, we use "BULK COLLECT".**

context switch:
travelling from PL/SQL statement executor to SQL statement executor and from SQL statement executor to PL/SQL statement executor is called one context switch



```
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;
```

BULK COLLECT clause:

- **BULK COLLECT clause is used to get entire data and store it in collection with one context switch.**

- **It reduces no of context switches.**
- **It improves the performance.**

Create an Associative Array and store emp table records in it:

INDEX	ELEMENT	
1	EMPNO ENAME SAL 1001 A 6000	e(1)
2	EMPNO ENAME SAL 1002 B 4000	e(2)
3	EMPNO ENAME SAL 1003 C 8000	e(3)
4	EMPNO ENAME SAL 1004 D 3000	e(4)

```
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;

   dbms_output.put_line('4th rec:' || e(4).ename || '   ' || e(4).sal);
   dbms_output.put_line('3rd rec:' || e(e.prior(4)).ename || '   ' || e(e.prior(4)).sal);

END;
/
```

Create an Associative Array and hold "HIKE" table records in it. According to HIKE table data increase salary to all emps in EMPLOYEE Table:

HIKE

EMPNO	PER
1001	10
1002	20
1003	15

EMPLOYEE

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

h

INDEX	ELEMENT

n

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

```
UPDATE employee
SET sal=sal+sal*h(1).per/100
WHERE empno=h(1).empno;
```

```
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
   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;

   dbms_output.put_line('salaries are updated..');
   COMMIT;
END;
/
```

Above program degrades the performance.

For LOOP will be executed by PL/SQL statement executor.
UPDATE command will be executed by SQL statement executor.
No of context switches will be increased here. So, performance will be degraded.

To improve performance of above program use BULK BIND.

BULK BIND:

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

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

- **FORALL loop is faster than for loop.**

Syntax of FORALL loop:

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

Create an Associative Array and hold "HIKE" table records in it.
According to HIKE table data increase salary to all emps in EMPLOYEE Table:

```
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;

   dbms_output.put_line('salaries are updated..');
   COMMIT;
END;
/
```

- **Associative Array is a table of 2 columns. They are:**
  - **Index**
  - **Element**

- **In Associative Array index can be NUMBER type or CHAR Type.**

INDEX	ELEMENT
1	RAJU
2	KIRAN
3	SAI

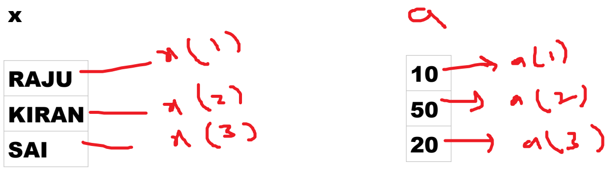
INDEX	ELEMENT
A	RAJU
B	KIRAN
C	SAI

Nested Table:

- **Nested Table is a table of column.**
- **That one column is element. No need to maintain INDEX here. Because always INDEX is number type.**

- **Always indexing starts from 1.**
- **It is same as single dimensional array in C.**

Example:



Creating Nested Table:

follow 2 steps:

- **define our own data type.**
- **declare variable for our own data type.**

define our own data type:

Syntax:

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

Example:

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

declaring variable for our own data type:

Syntax:

```
<variable_name> <type>;
```

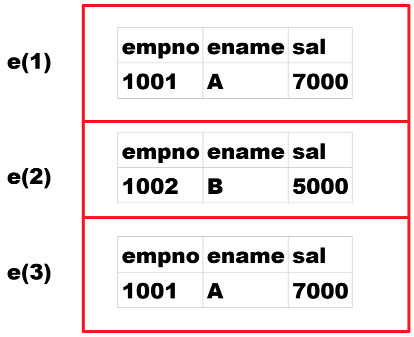
Example:

```
x NUM_ARRAY;
```

Example on Nested Table:

```
DECLARE
    TYPE num_array IS TABLE OF number(4);
    x NUM_ARRAY;
BEGIN
    x := num_array(10,50,20);

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

Create a nested table and hold emp table records in it:



e(1)

empno	ename	sal
1001	A	7000

e(2)

empno	ename	sal
1002	B	5000

e(3)

empno	ename	sal
1001	A	7000

```
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;

   dbms_output.put_line(e.count || ' rows selected..');
END;
/
```

- In Associative Array, unlimited no of elements can be stored.
- memory will not be wasted.

- In Nested Array, unlimited no of elements can be stored.
- memory will not be wasted.

V-Array [Variable Size Array]:

- For V-Array size is fixed.
- We must specify size for V-ARRAY.
- When we know exact number elements then use V-ARRAY.

- In V-Array we can store limited number of elements.
- It is same as nested table. but size is fixed.
- It maintains element only.
- Index is always number type.
- memory wastage may be there.

V-ARRAY => size is 10

10	40	50							

3 elements

10	40	50	78	90					

5 elements

Creating V-Array:

follow 2 steps:

- define our own data type
- declare variable for our own data type

defining data type:

Syntax:

```
TYPE <type_name> IS VARRAY(<size>) OF <element_type>;
```

Example:

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

declaring variable:

Syntax:

```
<variable> <type>;
```

Example:

```
x NUM_ARRAY;
```

Example program on V-Array:

```
DECLARE
   TYPE num_array IS VARRAY(10) OF number(4);
   x NUM_ARRAY;
BEGIN
   x := num_array(10,50,20,56,78);

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

Output:

```
10
50
20
56
78
```

Create VARRAY and hold emp table records in it:

```
DECLARE
   TYPE emp_array IS VARRAY(15) 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 between Cursor and Collection:

CURSOR	COLLECTION
• fetches row by row	• it collects entire data at a time and stores in collection.
• It can move forward only	• it can move in any direction.
• Sequential accessing only. Random accessing is not possible.	• Random Accessing is possible.
• It is slower.	• It is faster.

Collection Type	No of elements	Index type	dense or sparse
Associative Array	unlimited	BINARY_INTEGER (or) VARCHAR2	sparse (or) dense we can start index from anywhere
Nested Table	unlimited	BINARY_INTEGER	starts as dense it can become sparse index starts from 1
V-Array	limited	BINARY_INTEGER	dense index starts from 1

dense	no gaps can be there
sparse	gaps can be there

```
x num_array;

x(100) := 70;
x(200) := 56;   sparse

x(1)
x(2)
x(3)                  dense
```

```
DECLARE
    TYPE num_array IS TABLE OF number(4) INDEX BY VARCHAR2(1);
    x NUM_ARRAY;
BEGIN
    x('A') := 20;

    dbms_output.put_line(x('A'));
END;
/
```

Working with LOBs

Monday, October 16, 2023 6:09 PM

Working with LOBs:

Binary Related Data Types:

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

BFILE & BLOB data types are used to maintain multimedia objects [Large Objects - LOBs] like images, audios, videos, animations, documents, ... etc.

BFILE:

- **BFILE => Binary File Large Object.**
- **It is a pointer to multimedia object. It means, it maintains path of multimedia object.**
- **multimedia object will be stored out of the database. That's why BFILE can be also called "External Large Object".**
- **It is not secured one.**

Example:

D1 => Directory Object

D1 => D:\photos

Database

EMP1

EMPNO	**ENAME**	**EPHOTO [BFILE]**
1	**raju**	**bfilename(D1,raju.jpg)**

D:\
Photos Folder

raju.jpg

Directory Object:

- **Directory Object is ORACLE DB Object.**

- **It is a pointer to specific folder.**
- **DBA creates directory object.**

Syntax to create directory object:

```
CREATE DIRECTORY <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:

login as c##batch6pm user:

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

INSERT INTO emp1
VALUES(1,'A',bfilename('D1','ellison.jpg'));

SELECT * FROM emp1;
```

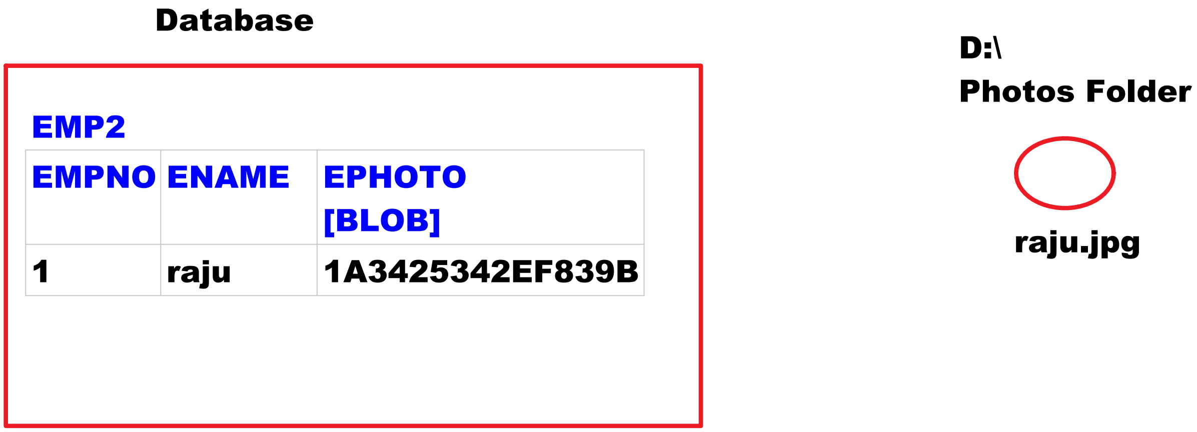
Output:

```
empno       ename        ephoto
--------    ----------   --------------------
1            A          bfilename('D1','ellison.jpg')
```

BLOB:

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

Example:



Database

EMP2

EMPNO	ENAME	EPHOTO [BLOB]
1	raju	1A3425342EF839B

D:\
Photos Folder

Example on BLOB:

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

INSERT INTO emp2
```

NUMBER	null
CHAR	''
BLOB	empty_blob()

VALUES(1, 'A', empty_blob());

empty_blob():

- **it is a function**
- **it is used to insert null in blob type column.**

DBMS_LOB PACKAGE

OPEN procedure	**used to open the file**
CLOSE procedure	**used to close the file**
GETLENGTH function	**used to find size of file**
LOADFROMFILE procedure	**used to read binary data and store into variable**

steps to store image in table:

- **s BFILE := bfilename('D1','ellison.jpg');**
  t BLOB;

- **Identify LOB LOCATOR and LOCK the record.**

  Example:

  SELECT ephoto INTO t FROM emp2 WHERE empno=1 FOR UPDATE;

FOR UPDATE	**is used to lock the record**

- **Open the file in read mode.**
  Example:
  dbms_lob.open(s, dbms_lob.lob_readonly);

- **find size of the file.**
  Example:
  len INT;
  len := dbms_lob.getlength(s);   --6638

- **read len no of bytes data from s and store it in t. t has image now.**
  Example:
  dbms_lob.loadfromfile(t,s,len);

- **Update the t image in table.**

Example:

```
UPDATE emp2 SET ephoto=t WHERE empno=1;
```

- Commit the data

- Close the opened file.

  Example:

  ```
  dbms_output.close(s);
  ```

- display the message: image saved

Procedure to update image:

```
CREATE OR REPLACE PROCEDURE
update_photo(p_empno NUMBER, p_name VARCHAR2)
AS
   s BFILE;
   t BLOB;
   len INT;
BEGIN
   s := bfilename('D1',p_name);

   SELECT ephoto INTO t FROM emp2
   WHERE empno=p_empno FOR UPDATE;

   dbms_lob.open(s, dbms_lob.lob_readonly);

   len := dbms_lob.getlength(s);

   dbms_lob.loadfromfile(t,s,len);

   UPDATE emp2 SET ephoto=t WHERE empno=p_empno;
   COMMIT;

   dbms_lob.close(s);

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

Calling procedure:

```
EXEC update_photo(1,'ellison.jpg');
```

Output:
image saved in table

Dynamic SQL

Tuesday, October 17, 2023 6:09 PM

- DRL, TCL, DML commands can be used directly in PL/SQL.

- DDL, DCL commands cannot be used directly in PL/SQL.
  TO use DDL, DCL commands in PL/SQL we use "Dynamic SQL".

Dynamic SQL:

- Building SQL query at runtime is called "Dynamic SQL".

- "EXECUTE IMMEDIATE" command executes the dynamic query.

- Submit Dynamic query as string to "EXECUTE IMMEDIATE" command

  Example:

  ```
  DROP TABLE emp;     --static query

  EXECUTE IMMEDIATE 'DROP TABLE ' || n;     --Dynamic query
  ```

Define a procedure to drop the table:

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

Calling procedure:

```
EXEC drop_table('student');
```

Output:

```
student table dropped
```

define a procedure to drop any db object:

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

Calling Procedure:

```
   EXEC drop_object('TABLE','HIKE');
   EXEC drop_object('TRIGGER','T6');
```

Define a procedure to drop all triggers:

```
CREATE OR REPLACE PROCEDURE
drop_all_triggers
AS
   CURSOR c1 IS select trigger_name from user_triggers;
   v_name VARCHAR2(10);
BEGIN
   OPEN c1;

   LOOP
      FETCH c1 INTO v_name;
```

```
        EXIT WHEN c1%notfound;
        EXECUTE IMMEDIATE 'DROP TRIGGER ' || v_name;
    END LOOP;

    dbms_output.put_line(c1%rowcount || ' triggers dropped');

    CLOSE c1;
END;
/
```