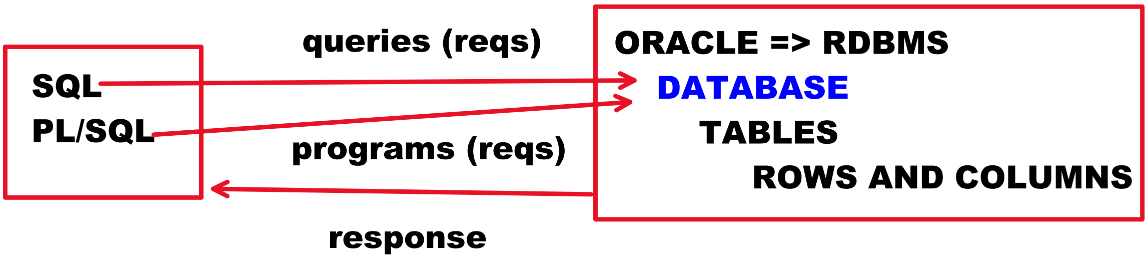
PL/SQL

Tuesday, November 12, 2024 7:46 AM



queries (reqs)

SQL

PL/SQL

programs (reqs)

response

ORACLE => RDBMS

DATABASE

TABLES

ROWS AND COLUMNS

PL/SQL:

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

- It is a programming language
- It is a procedural language
- In this language, we develop the programs to communicate with ORACLE DB.

- PL/SQL = SQL + Programming
  (queries)

- PL/SQL is extension of SQL.

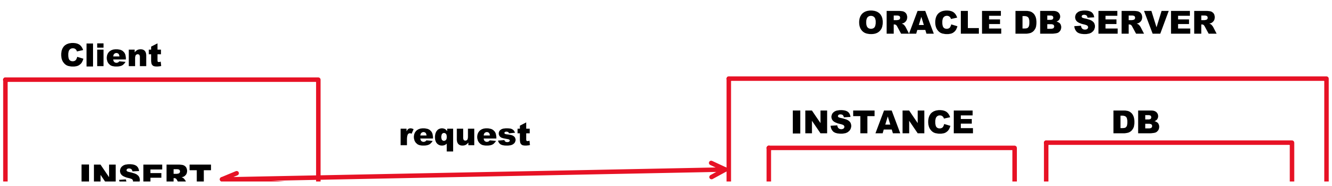
- PL/SQL program = SQL stmts + PL/SQL stmts

- All SQL queries we can write as statements in PL/SQL program.

Advantages:

- It improves performance.
- It provides conditional control structures.
- It provides looping control structures.
- It provides exception handling.
- It provides reusability.
- It provides security.

It improves performance:



ORACLE DB SERVER

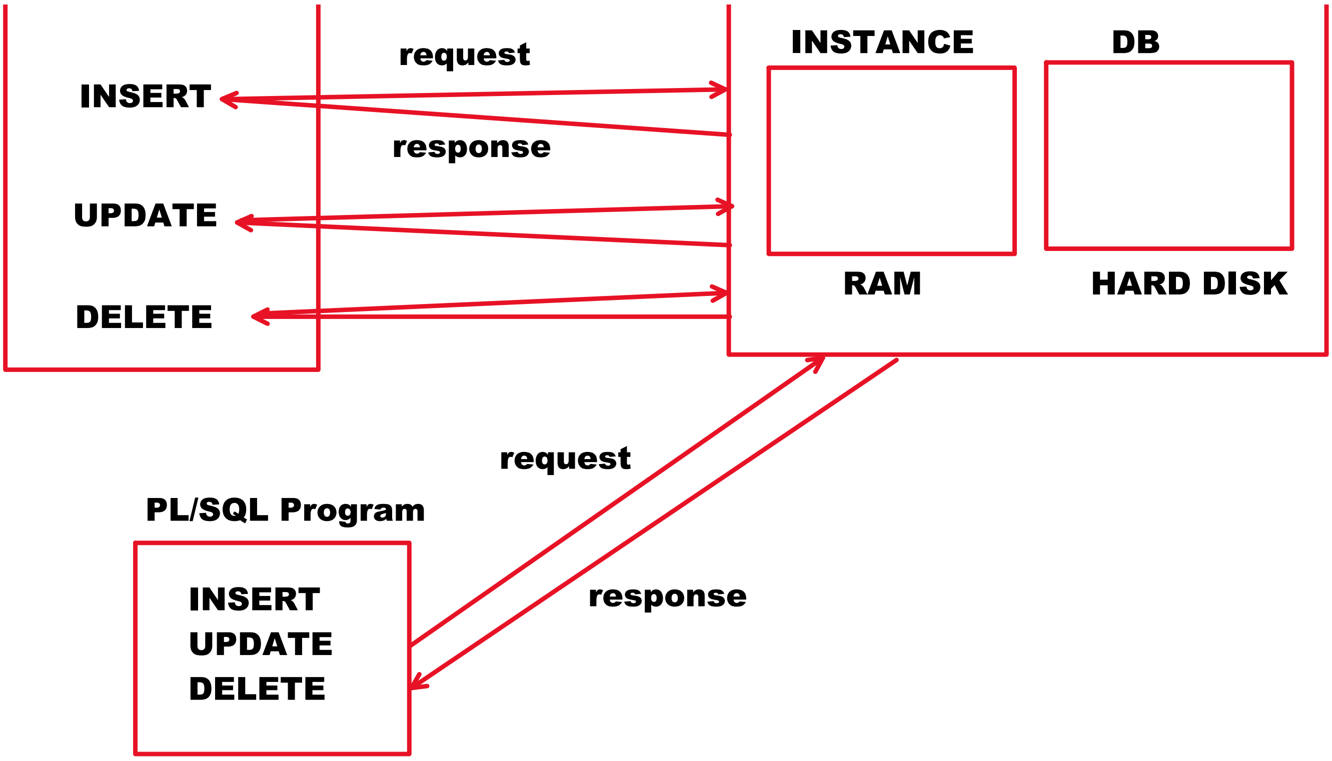
Client

request

INSTANCE

DB

INSERT



In PL/SQL program, we can group SQL queries and we can submit as 1 request. It reduces number of requests and responses. So, it improves performance.

It provides conditional control structures:

Using conditional control structure,
we can perform actions based on conditions.
PL/SQL provides conditional control structure like: IF .. THEN, IF .. THEN .. ELSE, IF .. THEN .. ELSIF

It provides looping control structures:

- Using looping control structure, we can perform same task repeatedly.
- PL/SQL provides looping control structures like:
  FOR, WHILE, SIMPLE LOOP

It provides exception handling:

Exception => problem => Run Time Error
Exception Handling => solution => we can handle run time

errors

If RTE occurs, program will be terminated in middle of execution.

It provides reusability:

PL/SQL provides functions, procedures and packages.
With this, we get reusability.
We define code only once. But, we can use it for any number of times by calling.

It provides security:
Only authorized users can call our procedures and functions.

Types of Blocks

Tuesday, November 12, 2024 8:38 AM

Types of Blocks:

There are 2 types of blocks. They are:

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

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

Named Block:
A block with name is called "Named Block".

Examples:
procedures, functions, packages, triggers

Anonymous Block:

Block

```
BEGIN
  --statements
END;
```

Named block:

```
CREATE PROCEDURE demo
BEGIN
    --statements
END;
```

Syntax of Anonymous block:

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

declaration part (optional)

execution part

Printing data:

In C	printf("hello");	printf() => function

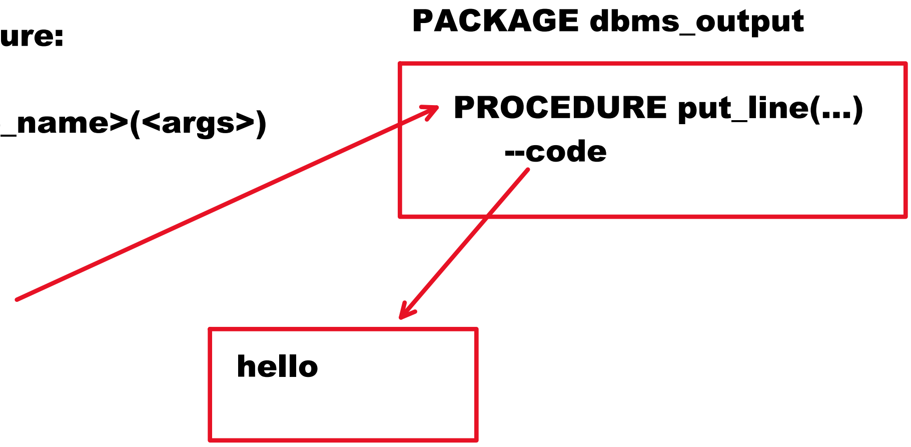
In Java	**System.out.println("hello");**	**println() => method**
In PL/SQL	**dbms_output.put_line('hello');**	**put_line() => procedure**

Syntax to call packaged procedure:

<package_name>.<procedure_name>(<args>)

Example:

dbms_output.put_line('hello');
--procedure call



PACKAGE dbms_output

PROCEDURE put_line(...)
--code

hello

Developing PL/SQL program:

Program to print hello:

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

- **Type above code in any text editor like: notepad, edit plus, notepad++**
- **save it in D: drive, batch730am Folder, with the name "HelloDemo.sql".**

Compiling and Running PL/SQL program:

Syntax:

```
SQL>   @<path of program file>
```

- **open sql plus**
- **login as user**

```
SQL> SET SERVEROUTPUT ON

SQL> @d:\batch730am\HelloDemo.sql
```

SERVEROUTPUT	**OFF**
PAGESIZE	**14**
LINESIZE	**80**

NOTE:
By default SERVEROUTPUT value is OFF.
If is OFF, messages cannot be sent to output.

To send messages to output, we must set SERVEROUTPUT as ON.

Data Types in PL/SQL

Wednesday, November 13, 2024 7:51 AM

Data Types in PL/SQL:

PL/SQL = SQL + Programming

PL/SQL provides following data types:

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

Integer Related	**Number(p)** **Integer** **Int** **Binary_Integer PL/SQL only** **Pls_Integer PL/SQL only**
Floating point Related	**Number(p,s)** **Float** **Binary_Float** **Binary_Double**
Date and Time Related	**Date** **Timestamp**
Binary Related Data Types	**BFILE** **BLOB**
Boolean related	**Boolean** **till oracle 21c => Boolean data type available in PL/SQL only.** **In oracle 23ai, boolean data type added in SQL also.**

Attribute Related	**%TYPE PL/SQL only** **%ROWTYPE PL/SQL only**
Cursor Related	**SYS_REFCURSOR PL/SQL only**
Exception Related	**EXCEPTION PL/SQL only**

Variable:

- **Variable is an Identifier [name].**

- **To identify every memory location uniquely we give a name to memory location. This memory location name is called "Variable".**

- **Variable means, storage location name.**
- **Variable is used to hold the data.**
- **Variable is temporary.**
- **It can hold only 1 value at a time.**

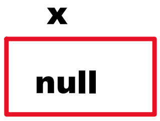
Declaring Variable:

Syntax:

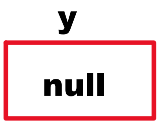
```
<variable> <data_type>;
```

Examples:

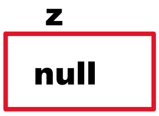
```
x NUMBER(4);
```



```
y VARCHAR2(10);
```



```
z DATE;
```



Assigning value:

:=	**Assignment Operator**

Syntax:

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

Examples:

```
x := 1234;
y := 'RAJU';
z := to_date('25-DEC-2023');
```

Printing data:

```
dbms_output.put_line(x);          --prints 1234
dbms_output.put_line(y);          --prints RAJU
dbms_output.put_line(z);          --prints 25-DEC-23
```

Reading data:

Syntax:

```
<variable> := &<text>;
```

Examples:

```
x := &x;
Output:
Enter value for x: 20

y := '&y';
Output:
Enter value for y: RAJU
```

Declare	x NUMBER(4);
Assign	x := 50;
Print	dbms_output.put_line(x);
Read	x := &x;
Initialize	x NUMBER(4) := 50;

Program to add 2 numbers:

20   15

20+15 = 35

Declare 3 variables as number type => x,y,z
Assign 20 to x
Assign 15 to y
Calculate x+y and store it in z
Print z

Program:

```
DECLARE
   x NUMBER(4);
   y NUMBER(4);
   z NUMBER(4);
BEGIN
   x := 20;
   y := 15;

   z := x+y;

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

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

Program:

```
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-1:

```
Enter value for x: 90
old   6:        x := &x;
new   6:        x := 90;
Enter value for y: 10
old   7:        y := &y;
new   7:        y := 10;
sum=100
```

TO avoid old and new parameters we have to set VERIFY as OFF.

```
SQL> SET VERIFY OFF

SQL> /
Enter value for x: 90
Enter value for y: 10
sum=100
```

Using SQL commands in PL/SQL:

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

Using SELECT command in PL/SQL:

Syntax:

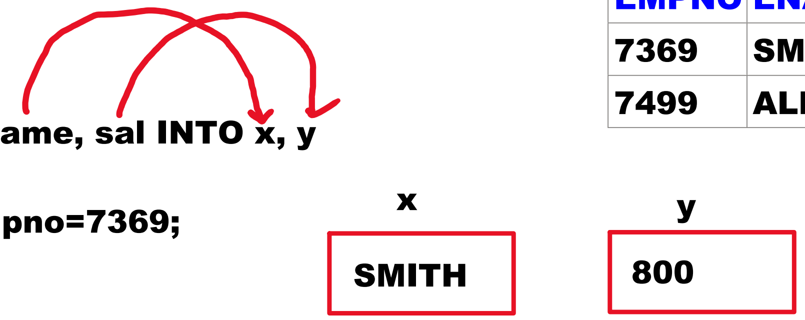
```
SELECT <column_list> INTO <variable_list>
FROM <table_name>
WHERE <condition>;
```

EMP

EMPNO	ENAME	SAL	..
7369	SMITH	800	..
7499	ALLEN	1600	..

Example:

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



NOTE:

column names can be used in
SQL statements only.
column names cannot be used in
PL/SQL statements. To use column
in PL/SQL statement copy column data into
variable. Using variable we can access column data.

To copy column data into variable we
use INTO clause.

Examples on SELECT command:

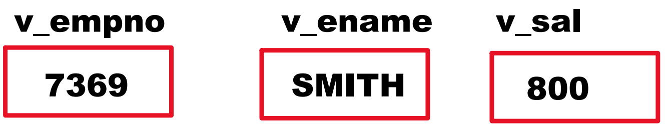
Program to display emp record of given empno:

steps:
read empno => v_empno

select data from table copy into variables
v_ename, v_sal

print emp record

Program:



```
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;
/
```

Output:
```
Enter .. empno: 7369
SMITH    800
```

Output:
```
Enter value for empno: 7934
MILLER     3630
```

Program to check the balance of given account number:

ACCOUNTS

ACNO	BALANCE
1234	80000
1235	50000

```
DECLARE
   v_acno NUMBER(4);
   v_balance NUMBER(9,2);
BEGIN
   v_acno := &acno;

   SELECT balance INTO v_balance FROM accounts
   WHERE acno=v_acno;

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

Output:

```
Enter .... acno: 1234
balance=80000
```

%TYPE:

Problem-1:
variable field size and column field size are mismatching

v_empno NUMBER(2)

EMP
EMPNO NUMBER(4)
7369
7499

Problem-2:
variable data type and column data type are mismatching

v_empno DATE

EMP
EMPNO NUMBER(4)

To solve above problems, PL/SQL provides %TYPE data type.

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

Syntax:

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

Example:

```
v_empno EMP.EMPNO%TYPE;
v_acno    ACCOUNTS.ACNO%TYPE;
```

Example on %TYPE:

program to display emp record of given empno:

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

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

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

Example:

Program to display balance of given acno:

```
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;
/
```

NOTE:
v_empno EMP.EMPNO%TYPE

above statement instructs that,
take v_empno variable data type as EMP table's EMPNO column's data type.

%ROWTYPE:

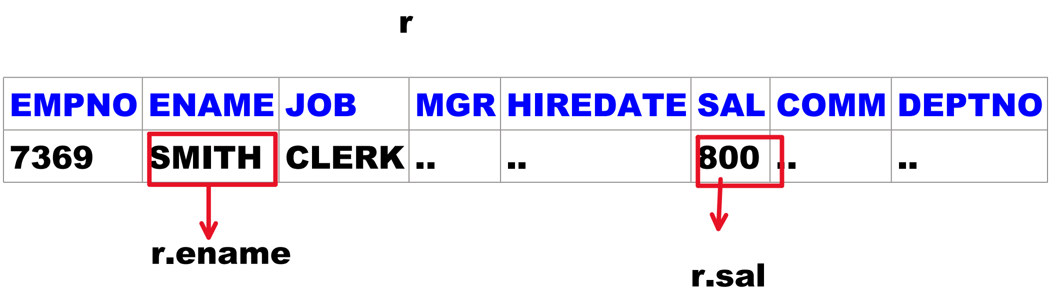
- **It is attribute related data type.**
- **It is used to hold entire row of a table.**
- **It reduces number of variables.**
- **It can hold only 1 row at a time.**

Syntax:

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

Example:

```
r EMP%ROWTYPE;
```



r

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	..	..	800	..	..

r.ename

r.sal

SELECT * INTO r FROM emp WHERE empno=7369;

Example on %ROWTYPE:

Display emp record of given empno:

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

   SELECT * INTO r FROM emp WHERE empno=v_empno;

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

```
Output:
Enter value for empno: 7900
JAMES    2950
```

Program to find experience of given empno:

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

   SELECT hiredate INTO v_hiredate FROM emp
   WHERE empno=v_empno;

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

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

```
Output:
Enter value for empno: 7934
experience=42 years
```

Program to find today's weekday:

```
DECLARE
   wd VARCHAR2(10);
BEGIN
   wd := to_char(sysdate, 'DAY');

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

Using UPDATE in PL/SQL:

Example:

Program to increase salary of given empno with given amount:

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

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

   COMMIT;

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

   SELECT sal INTO v_sal FROM emp
   WHERE empno=v_empno;

   dbms_output.put_line('after incr sal=' || v_sal);
END;
/
```

Output:

```
Enter value for empno: 7934
Enter value for amount: 1000
sal increased..
after incr sal=7630
```

Using DELETE in PL/SQL:

Program to delete emp record of given empno:

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

   DELETE FROM emp WHERE empno=v_empno;

   COMMIT;

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

Output:

```
Enter .. empno: 7900
record deleted..
```

Using INSERT in PL/SQL:

STUDENT

SID	SNAME	M1

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

Program to insert student record into STUDENT table:

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

Output:

```
Enter value for sid: 1004
Enter value for sname: D
Enter value for m1: 55
record inserted..
```

data types

declare	x NUMBER(4);
assign	x := 50;
read	x := &x;
print	d_o.p_l(x);
initialize	x NUMBER(4) := 50;

%TYPE	is used to hold 1 column value
%ROWTYPE	is used to hold 1 row

SELECT
UPDATE
INSERT
DELETE

CONTROL STRUCTURES

Friday, November 15, 2024 8:35 AM

max marks: 100
min marks: 40 for pass

```
DECLARE
      m INT := 70;
BEGIN
   IF m>=40 THEN
      dbms_output.put_line('PASS');
   ELSE
      dbms_output.put_line('FAIL');
   END IF;
END;
/
```

Control Structures:

- Control Structure is used to control the flow of execution of program.

- Normally, program gets executed sequentially. To change sequential execution, to transfer to our desired location we use Control Structures.

PL/SQL provides following Control Structures:

Conditional	IF .. THEN
	IF .. THEN .. ELSE
	IF .. THEN .. ELSIF
	NESTED IF
	CASE

Looping	WHILE FOR SIMPLE LOOP
Jumping	GOTO

	EXIT EXIT WHEN

Conditional Control Structures:

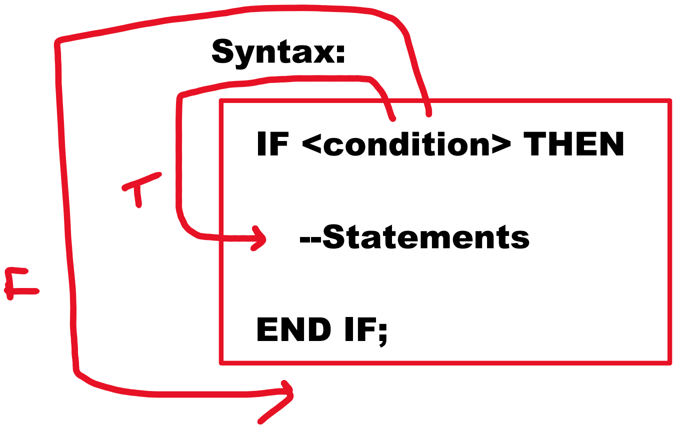
Conditional Control Structure executes the statements based on conditions.

PL/SQL provides following Conditional Control Structures:

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

IF .. THEN:

Syntax:



```
IF <condition> THEN

   --Statements

END IF;
```

condn => TRUE

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

Example on IF ..THEN:

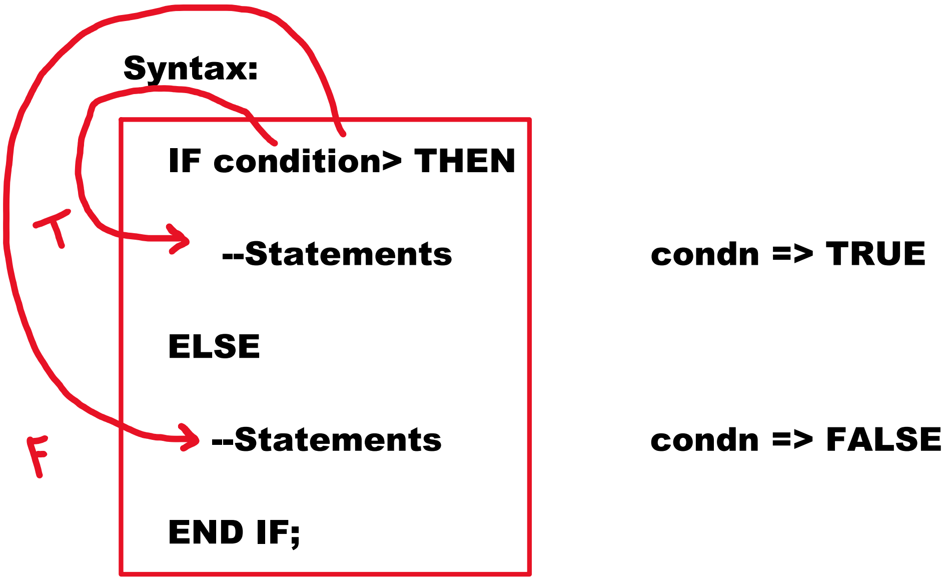
Program to delete emp record of given empno.
If experience is more than 43 then only delete the record:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_hiredate DATE;
   v_exp INT;
```

```
BEGIN
   v_empno := &empno;

   SELECT hiredate INTO v_hiredate FROM emp
   WHERE empno=v_empno;

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

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

```
Output-1:
Enter value for empno: 7900
experience=42 years


Output-2:
Enter value for empno: 7499
experience=43 years
record deleted
```

IF .. THEN .. ELSE:



Syntax:

```
IF condition> THEN

   --Statements                      condn => TRUE

ELSE

   --Statements                      condn => FALSE

END IF;
```

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

Example:

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

if job is MANAGER then increase	20% on sal	
OTHERS	10%	

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

   SELECT job INTO v_job FROM emp
   WHERE empno=v_empno;

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

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

   COMMIT;

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

IF .. THEN .. ELSIF:

Syntax:

Syntax:

```
IF <condition1> THEN
    --statements                          condn1 => T
ELSIF <condition2> THEN
    --statements                          condn1 => F, condn2 => T
.
.
[ELSE
    --statements]                         All condns => F
```

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

Example on IF .. THEN .. ELSIF:

Program to increase salary of given empno based on job.
if job is MANAGER then increase 20% on sal
CLERK 15%
others 5%

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

    SELECT job INTO v_job FROM emp
    WHERE empno=v_empno;

    IF v_job='MANAGER' THEN
        v_per := 20;
    ELSIF v_job='CLERK' THEN
        v_per := 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('job=' || v_job);
    dbms_output.put_line(v_per || '% on sal increased..');
END;
/
```

NESTED IF:

Writing IF in another IF is called "Nested If".

Syntax:

```
IF <condn1> THEN
   IF <condn2> THEN

      --statements

   END IF;
END IF;
```

condn1, codn2 => T

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

Example on NESTED IF:

STUDENT

SID	SNAME	M1	M2	M3
1001	A	60	70	50
1002	B	80	30	45

RESULT

SID	TOTAL	AVRG	RESULT

Program to find total, avrg, result of given student id and insert those values into RESULT table:

max marks: 100
min marks: 40 in each subject for pass
if pass, check avrg.
if avrg is 60 or more => FIRST DIV
if avrg is b/w 50 to 59 => SECOND DIV
if avrg is b/w 40 to 49 => THIRD DIV

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

    SELECT * INTO r1 FROM student
    WHERE sid=v_sid;

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


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

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

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

v_sid

1001

r1

SID	SNAME	M1	M2	M3
1001	A	60	70	50

r2

SID	TOTAL	AVRG	RESULT
	180	60	FIRST

CASE:

- **It can be used in 2 ways. They are:**
    - **Simple CASE [same as switch in JAVA]**
    - **Searched CASE [same as if else if in JAVA]**

Simple CASE:
It can check equality condition only

Searched CASE:
It ca check any condition

Syntax of Simple CASE:

```
CASE <expression>
   WHEN <constant1> THEN
        --statements
   WHEN <constant2> THEN
        --statements
   .
   .
   [ELSE
       --statements]
END CASE;
```

The statements in Simple CASE get executed when constant value is equals to expression value.
If constants are not equal then it executes ELSE statements.

Example:

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

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

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

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

END;
/
```

Searched CASE:

Syntax:

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

Example:

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

+ve	>0
-ve	<0

```
DECLARE
    n INT;
BEGIN
```

```
        n := &n;

        CASE
            WHEN n>0 THEN
                dbms_output.put_line('+ve');
            WHEN n<0 THEN
                dbms_output.put_line('-ve');
            WHEN n=0 THEN
                dbms_output.put_line('zero');
        END CASE;

    END;
    /
```

Looping Control Structures:

Looping Control Structure is used to execute the statements repeatedly.

PL/SQL provides following Looping Control Structured:

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

WHILE:

```
WHILE <condn>
LOOP

    --statements

END LOOP;
```

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

Example on WHILE:

Program to print numbers from 1 to 4:

Output:

```
i

1
2
3
4
```

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

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

Simple Loop:

Syntax:

```
LOOP

   --statements
   EXIT WHEN <condition>;   / EXIT;

END LOOP;
```

Example on Simple Loop:

Program to print numbers from 1 to 4:

Output:

```
DECLARE
   i INT;
```

Output:

```
i

1
2
3
4
```

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

    LOOP
        dbms_output.put_line(i);
        i := i+1;
        EXIT WHEN i=5;
    END LOOP;

END;
/
```

```
EXIT WHEN i=5;

(or)

IF i=5 THEN
  EXIT;
END IF;
```

EXIT WHEN:

- it is a jumping control structure.
- it is used to terminate the loop.
- it can be used in loop only.

Syntax:

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

EXIT:

- it is a jumping control structure.
- it is used to terminate the loop.
- it can be used in loop only.

Syntax:

```
EXIT;
```

What is the output?

```
BEGIN
    dbms_output.put_line('hi');
    EXIT;
    dbms_output.put_line('bye');
END;
/
```

Output:

ERROR: EXIT can be used inside of loop only

FOR:

Syntax:

```
FOR <variable> IN <lower> .. <upper>
LOOP

    --statements

END LOOP;
```

Example on FOR:

Program to print numbers from 1 to 4:

```
i

1
2
3
4
```

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

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

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

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

Output:

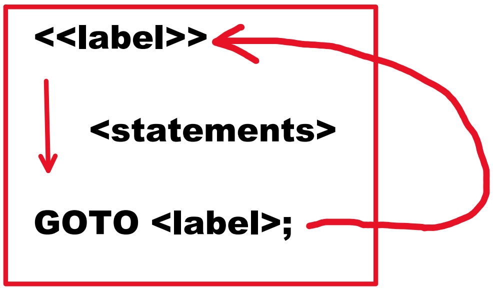
ERROR: i cannot be used as assignment target
i => read-only variable

- **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
END;
/
```

GOTO:

Syntax:



```
<<label>>

    <statements>

GOTO <label>;
```

When GOTO statement is executed, it goes to specified label.

Example on GOTO:

Program to print numbers from 1 to 4:

Output:

```
DECLARE
   i INT;
BEGIN
```

```
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
```

CURSORS

Tuesday, November 19, 2024 7:47 AM

CURSORS:

GOAL:

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

to hold 1 column value	use %TYPE
to hold 1 row	use %ROWTYPE
to hold multiple rows	use CURSOR or COLLECTION

CURSOR:

- **CURSOR is a pointer to a memory location which is in INSTANCE. This memory location contains multiple rows.**

- **To hold multiple rows and process them one by one we are using CURSOR.**

Steps to use a CURSOR:

4 steps:

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

NOTE:

- **CURSOR is associated with SELECT query.**
- **This SELECT query result will be stored in CURSOR.**

Declaring Cursor:

Syntax:

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

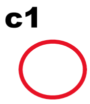
Example:

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

When we declare cursor, **c1**

When we declare cursor,

- CURSOR variable will be created.
- SELECT query will be identified.



Opening Cursor:

Syntax:

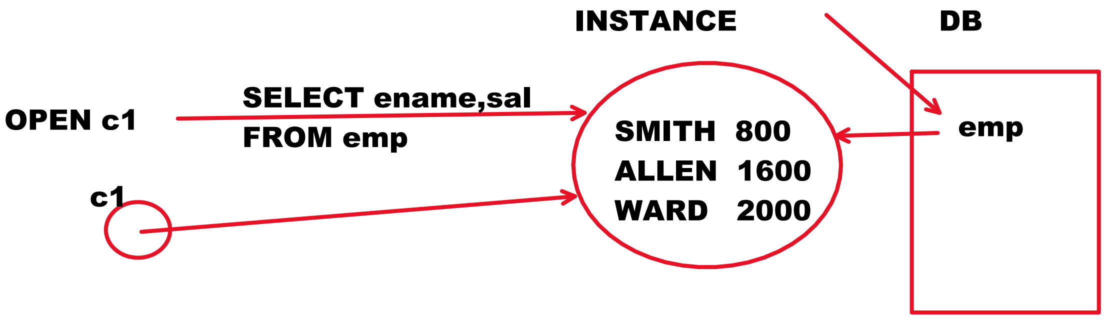
```
OPEN <cursor_name>;
```

Example:

```
OPEN c1;
```

When CURSOR is opened,

- SELECT query will be submitted to ORACLE.
- ORACLE goes to DB, selects the data and loads the result into some memory location which is in INSTANCE.
- This memory location address will be given to CURSOR variable.


INSTANCE

DB

OPEN c1

SELECT ename,sal
FROM emp

c1

SMITH 800
ALLEN 1600
WARD 2000

emp

when cursor is opened,
select query result will be loaded in INSTANCE.

Fetching Record from CURSOR:

Syntax:

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

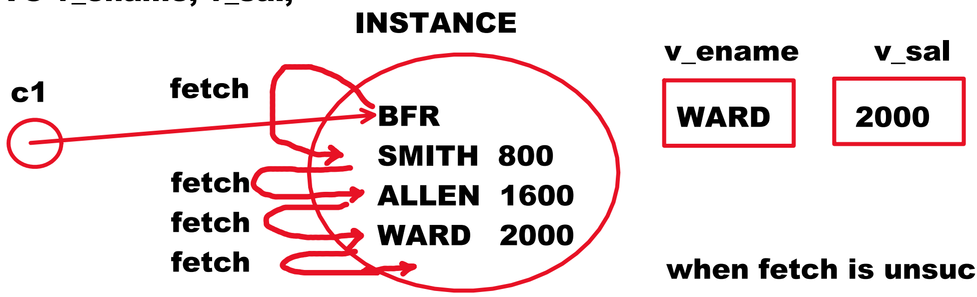
Example:

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

When FETCH statement is executed it fetches next row and copies into variables.

1 fetch statement can fetch 1 row.
to fetch multiple rows and process them write
fetch statement in loop.

FETCH c1 INTO v_ename, v_sal;



INSTANCE

c1

fetch

BFR
SMITH 800
ALLEN 1600
WARD 2000

fetch
fetch
fetch

v_ename

WARD

v_sal

2000

when fetch is unsuccessful
stop fetching.

BFR	Before First Row

Closing CURSOR:

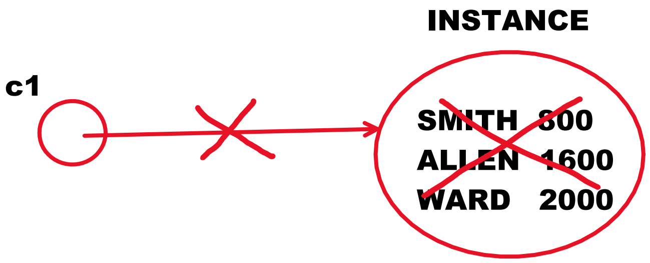
Syntax:

CLOSE <cursor_name>;

Example:
CLOSE c1;

When CURSOR is closed,

- The data in the INSTANCE will be cleared.
- Reference will be gone.



INSTANCE

c1

SMITH 800
ALLEN 1600
WARD 2000

DECLARE	CURSOR c1 IS SELECT ename,sal FROM emp
OPEN	OPEN c1
FETCH	FETCH c1 INTO v_ename, v_sal
CLOSE	CLOSE c1

Cursor Attributes:

Syntax:

<cursor_name><attribute_name>

Cursor provides following attributes:

- **%found** row
- **%notfound** row
- **%rowcount** row
- **%isopen**

Examples:

c1%found
c1%notfound
c1%rowcount
c1%isopen

Example Program on CURSOR:

Program to display all emp names and salaries:

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

   LOOP
      FETCH c1 INTO r;

      EXIT WHEN c1%NOTFOUND;

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

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

   CLOSE c1;
END;
/
```

Example:

EMPLOYEE

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

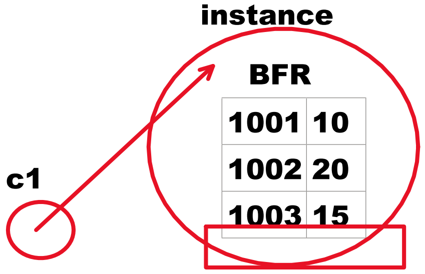
HIKE

EMPNO	PER
1001	10
1002	20
1003	15

1001	A	5000
1002	B	3000
1003	C	7000

1001	10
1002	20
1003	15

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



1001	10
1002	20
1003	15

r

EMPNO	PER
1003	15

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

   LOOP
      FETCH c1 INTO r;

      EXIT WHEN c1%notfound;

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

   COMMIT;

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

   CLOSE c1;
END;
/
```

Example:

Program to calculate total, avrg and result of all students.

STUDENT

SID	SNAME	M1	M2	M3
1001	A	60	70	50
1002	B	80	30	45

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
           r2.result := 'PASS';
        ELSE
           r2.result := 'FAIL';
        END IF;

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

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

     CLOSE c1;
  END;
  /
```

CURSOR FOR LOOP:

Syntax:

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

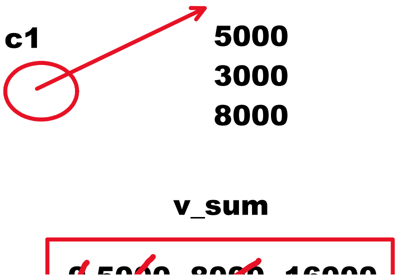
- If we use CURSOR FOR LOOP,
  we have no need to open, fetch and close the cursor.
  All these 3 actions will be done implicitly.

- We have no need to declare CURSOR FOR LOOP variable.
  Implicitly it will be declared as %ROWTYPE.

Example on CURSOR FOR LOOP:

Program to find sum of salaries of all emps:

```
DECLARE
   CURSOR c1 IS SELECT sal FROM emp;
   v_sum NUMBER := 0;
BEGIN

   FOR r IN c1
   LOOP
```



```
    FOR r IN c1
    LOOP
       v_sum := v_sum + r.sal;
    END LOOP;

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

v_sum

~~0~~ ~~5000~~ ~~8000~~ 16000

v_sum := v_sum + r.sal;

0 + 5000
5000+3000 = 8000
8000+8000 = 16000

Assignment:

Display all emp records using cursor for loop

increase salary of all emps in employee table according to hike table percentages using cursor for loop

find total, avrg and result of all students and insert them into result table using cursor for loop

Inline Cursor:

Syntax:

```
FOR <variable> IN (<select query>)
LOOP
   -statements
END LOOP;
```

- If select query is specified in CURSOR FOR LOOP then it is called "Inline Cursor".

Example on Inline Cursor:

Display all emp records using INLINE CURSOR:

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

Ref Cursor:

In Simple Cursor

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

In Ref Cursor

c1 => SELECT * FROM emp
c1 => SELECT * FROM dept
c1 => SELECT * FROM salgrade

- **In Simple Cursor,**
  One cursor can be used for 1 select query only.
  It is fixed.

- **In Ref Cursor,**
- **Same Cursor can be used for multiple select queries.**
  Select query can be changed at run time.

- **It has data type. i.e.: SYS_REFCURSOR.**
- **It can be used as procedure parameter.**

Declaring Ref Cursor:

Syntax:
```
<cursor_name> SYS_REFCURSOR;
```

Example:
```
c1 SYS_REFCURSOR;
```

Opening Ref Cursor:

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

Example:
```
OPEN c1 FOR SELECT * FROM emp;
```

Example on Ref Cursor:

Program to display emp table records and dept table records using ref cursor:

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

   LOOP
```

```
        FETCH c1 INTO r1;
        EXIT WHEN c1%notfound;
        dbms_output.put_line(r1.ename || '     ' || r1.sal);
    END LOOP;

    CLOSE c1;

    OPEN c1 FOR SELECT * FROM dept;

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

    CLOSE c1;
END;
/
```

Differences b/w Simple Cursor and Ref Cursor:

Simple Cursor	Ref Cursor
In Simple Cursor, 1 cursor can be used for 1 select query only	In Ref Cursor, same cursor can be used for multiple select queries
It is fixed.	It can be changed.
It is static.	It is dynamic.
It has no data type.	It has data type. i.e: SYS_REFCURSOR
It cannot be used as procedure parameter. Because, it has no data type.	It can be used as procedure parameter.
In this, we specify SELECT QUERY at the time of declaration.	In this, we specify SELECT QUERY at the time of opening cursor.

Parameterized Cursor:

- **Cursor with parameters is called "Parameterized Cursor".**
- **This parameter value will be passed at the time of opening cursor.**

Syntax:

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

Example:

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


OPEN c1(10);
```

Example on Parameterized Cursor:

Program to hold specific dept records in cursor and process them using parameterized cursor:

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

   r EMP%ROWTYPE;
BEGIN
   OPEN c1(30);

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

   CLOSE c1;
END;
/
```

Types of Cursors:

2 types:

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

Implicit Cursor:

- **To execute any DRL or DML command implicitly ORACLE uses a cursor. It is called "Implicit Cursor".**
- **Implicit Cursor name is: SQL.**
- **We will not declare, open, fetch or close implicit cursor. All these actions will be done implicitly.**
- **We can use cursor attributes in coding using cursor name SQL.**

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

Example on Implicit Cursor:

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

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

   IF sql%notfound THEN
      dbms_output.put_line('no emp existed with this empno');
   ELSE
      dbms_output.put_line('sal increased..');
   END IF;
END;
/
```

Example:

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;
/
```

CURSOR	**is a pointer to a memory location in instance**

purpose	**to hold multiple rows and process them one by one**
4 steps	**DECLARE OPEN FETCH CLOSE**
Ref Cursor	**Same cursor can be used for multiple select queries**
Cursor for loop	**no need to open, fetch, close**
Inline cursor	**we specify select query in cursor for loop** **no need to declare also**
Parameterized cursor	**cursor with parameters c1(n NUMBER)**
Types of cursors	**Implicit cursor => SQL** **Explicit Cursor** **simple cursor** **ref cursor**

STORED PROCEDURES

Friday, November 22, 2024 7:46 AM

PROCEDURE:

- **PROCEDURE named block of statements that gets executed on calling.**
- **PROCEDURE can be also called as SUB PROGRAM.**

Types of Procedures:

2 Types:

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

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

Example:
SCHAMA c##batch730am
PROCEDURE withdraw **Stored procedure**

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

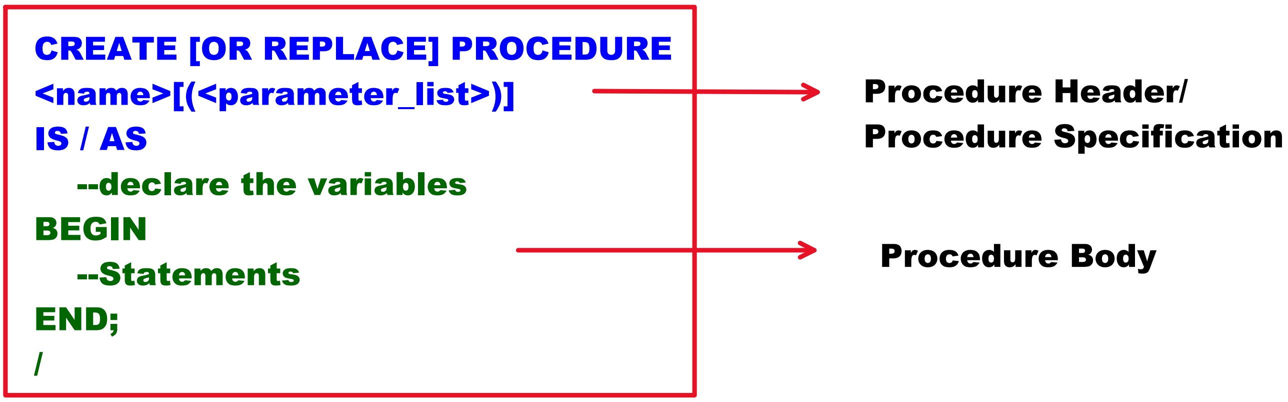
Example:
SCHAMA c##batch730am
PACKAGE bank
PROCEDURE withdraw **Packaged procedure**

Syntax to define Stored Procedure:

In C:
Function:
is a set of statements
calling

In Java:
Method:
is a set of statements
calling

IN PL/SQL:
Procedure
Function
is a set of statements
calling



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

Procedure Header/
Procedure Specification

Procedure Body

Example on Stored Procedure:

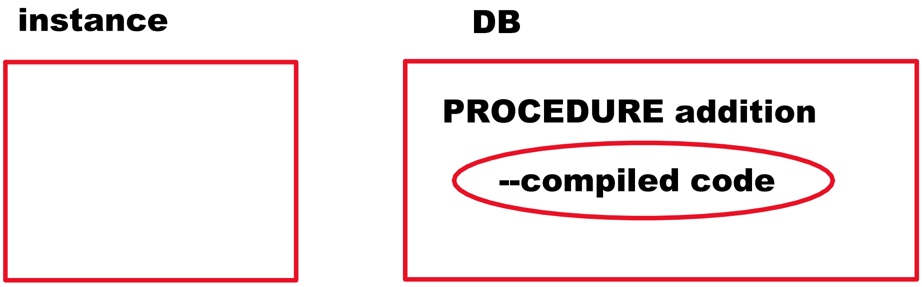
Define a procedure to add 2 numbers:

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

- **Type above code in text editor.**
- **Save it in D: Drive, batch730 am Folder, with the name ProcedureDemo.sql.**

- **open sql plus.**
- **login as user.**

```
SQL> @d:\batch730am\ProcedureDemo.sql
Output:
Procedure created.
```



Calling Stored Procedure:

3 ways:

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

Calling from SQL prompt:

Syntax:

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

Example:

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

Calling from PL/SQL Program:

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

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

Parameter:

- Parameter is a local variable which is declared in Header.

Syntax:

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

Example:

```
x IN NUMBER
```

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

Parameter modes:

3 modes:

- **IN   [default]**
- **OUT**
- **IN OUT**

IN:

- **It is default one.**
- **It takes input.**
- **It is used to bring value into procedure from out of procedure**
- **In procedure call, IN parameter can be variable or constant.**

OUT:

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

IN OUT:

- **Same parameter takes input and sends output.**
- **In procedure call, it must be variable only.**

Example on OUT parameter:

Define a procedure to add 2 numbers.
Send the result out of the procedure:

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

Calling from SQL prompt:

```
SQL> VAR s NUMBER

SQL> EXEC addition(2,3,:s);

SQL> PRINT s
Output:
5
```

NOTE:

Bind Variable:

- **A variable which is declared at SQL prompt is called "Bind Variable".**
- **In above example s is bind variable.**
- **To declare bind variable we use VAR[IABLE] command**
- **To write data into bind variable use bind operator : [colon]**
- **To print bind variable value use PRINT command.**

Calling from PL/SQL program:

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

   addition(a,b,c);

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

Example:

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

```
CREATE OR REPLACE PROCEDURE
update_salary(p_empno IN NUMBER, p_amount IN NUMBER)
```

```
AS
BEGIN
   UPDATE emp SET sal=sal+p_amount
   WHERE empno=p_empno;

   COMMIT;

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

Calling:

```
SQL> EXEC update_salary(7934,2000);
```

Output:

```
sal increased..
```

Define a procedure to increase salary of specific employee with specific amount. Updated salary send out of the procedure:

```
--procedure call:
--  EXEC update_salary(7934, 1000, :s)

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

   dbms_output.put_line('sal increased..');
   SELECT sal INTO p_sal FROM emp
   WHERE empno=p_empno;
END;
/
```

Calling:

```
SQL> VAR s NUMBER
SQL> EXEC update_salary(7934,1000,:s);
```

Output:

```
sal increased..
SQL> PRINT s
```

Output:

```
12000
```

NOTE:

to see errors of procedure write following command:

```
SQL> SHOW ERRORS
```

Example:

Define a procedure to perform withdraw transaction:

Accounts

ACNO	BALANCE
1234	80000
1235	30000

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

   IF p_amount>v_balance THEN
      dbms_output.put_line('insufficient funds..');
   ELSE
      UPDATE accounts SET balance=balance-p_amount
      WHERE acno=p_acno;

      COMMIT;

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

```
SQL> EXEC withdraw(1234, 90000);
Output:
insufficient funds..
```

```
SQL> EXEC withdraw(1234, 10000);
Output:
```

```
transaction successful..
```

Example:
Define a procedure to perform deposit transaction:

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

   COMMIT;

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

Calling:

```
SQL> EXEC deposit(1234,20000);
```

Output:

```
transaction successful..
```

Assignment:

Accounts

ACNO	BALANCE
1234	80000
1235	30000

Define a procedure to perform fund transfer operation:

procedure call:

```
fund_transfer(1234, 1235, 10000);
```

```
create procedure
fund_transfer(p_from NUMBER, p_to NUMBER, p_amount NUMBER)
AS
BEGIN
   check Sufficient funds available or not
   if available
   UPDATE from account balance
```

```
      UPDATE to account balance
      save transaction
      display message: transaction successful
  END;
  /
```

parameter mapping techniques:

positional
named
mixed