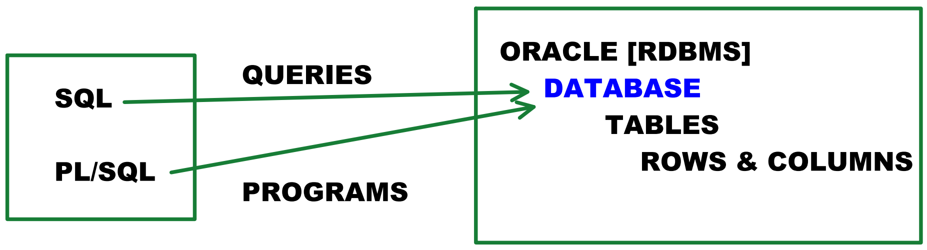
PL/SQL

Saturday, August 3, 2024 8:00 AM



SQL:

- **SQL => Structured Query Language**
- **SQL is a Query Language**
- **SQL is Non-Procedural Language [no programs]**
- **Just we write queries to communicate with ORACLE DB.**

PL/SQL:

- **PL => Procedural Language**
- **PL/SQL is Programming Language**
- **It is Procedural Language. In this, we write a set of statements [program].**
- **In PL/SQL we develop the programs to communicate with ORACLE DB.**

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

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

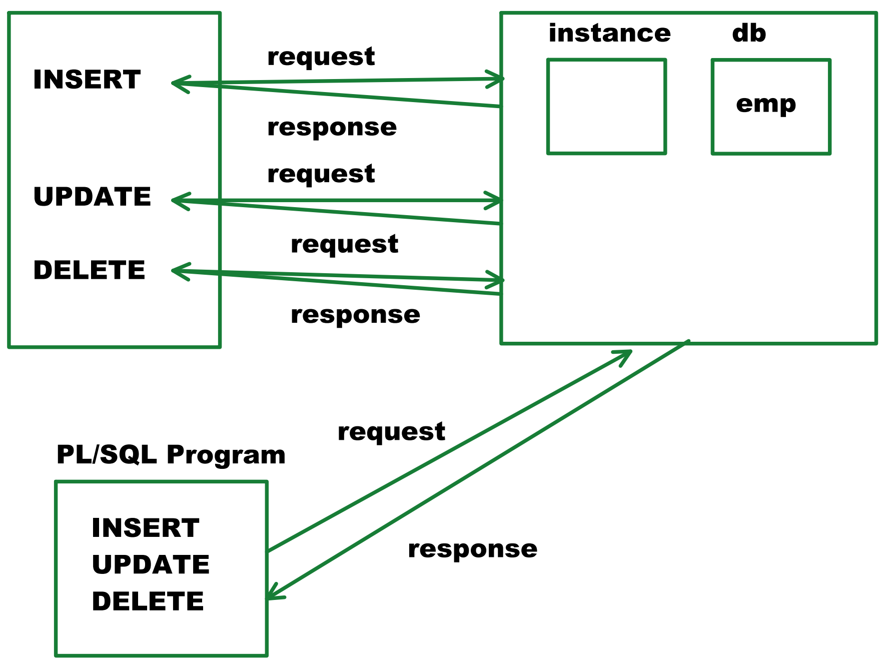
Advantages of PL/SQL:

- **Improves the performance.**
- **Provides Conditional Control Structures.**
- **Provides Looping Control Structures.**
- **Provides Exception Handling.**
- **Provides reusability.**
- **Provides security.**

Improves the performance:

TRANSACTION => is a series of actions

SQL PLUS **ORACLE DB SERVER**



- PL/SQL program reduces no of requests and responses. So, it improves the performance.

Provides Conditional Control Structures:

- PL/SQL provides condition control structures such as IF .. THEN, IF .. THEN .. ELSE ... etc.
- Using Conditional Control Structures we can perform actions based on conditions.

Provides Looping Control Structures:

- PL/SQL provides Looping Control Structures such as Simple Loop, For Loop and While Loop.
- To perform same action repeatedly we use Looping Control Structures.

Provides Exception Handling:

- Exception => Run Time Error.
- When run time error occurs our program will be terminated in middle of execution.
- The way of handling run time errors is called "Exception Handling".

Provides reusability:

- PL/SQL provides procedures, functions, packages and triggers. These concepts provide reusability.

Provides Security:

- Only authorized users can call the procedures and functions.

Types of Blocks:

2 Types:

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

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

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

Example:

Anonymous Block:

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

Named Block:

```
CREATE PROCEDURE p1 AS
BEGIN
  --Statements
END;
```

printing hello

Syntax of Anonymous Block:

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

Declaration part (optional)

Execution part

In C	**printf("HELLO");**
In Java	**System.out.println("HELLO");**
In PL/SQL	**dbms_output.put_line('HELLO');**

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

stored procedure
packaged procedure

c##batch730am
   PROCEDURE p1 => stored procedure

- defined in package "dbms_output".
- It is used to print the data on screen.

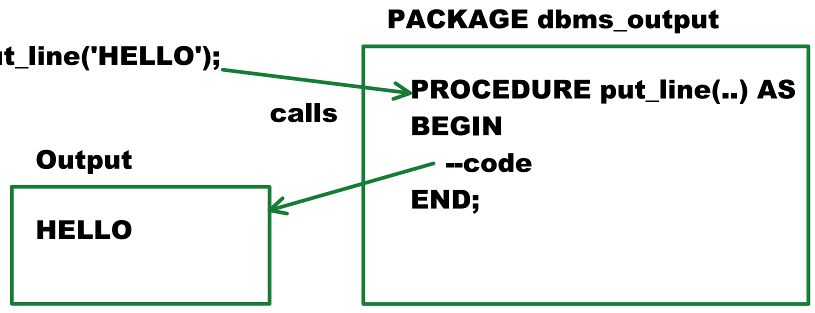
Syntax to call packaged procedure:

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

```
    PROCEDURE p1 => stored procedure

c##batch730am
    PACKAGE demo
        PROCEDURE p1 => packaged procedure
```

Example:

```
dbms_output.put_line('HELLO');
--procedure call
```



Program to print HELLO on screen:

Developing PL/SQL program:

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

- type above program in text editor like:
  notepad
  wordpad
  notepad++
  edit plus
- save it in D: drive, batch730am folder,
  with the name HelloDemo.sql.

Compiling and running PL/SQL program:

- Open SQL PLUS
- login as user

Syntax to compile PL/SQL program:

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

Example:

```
SQL> SET SERVEROUTPUT ON
SQL> @D:\batch730am\HelloDemo.sql
Output:
HELLO
```

SERVEROUTPUT:

- **by default, SERVEROUTPUT value is OFF.**
- **If it is OFF, messages cannot be sent to output.**
- **To send messages to output, we must set SERVEROUTPUT as ON.**

```
SQL> SET SERVEROUTPUT ON
```

it is applicable for entire session.

data types **PL/SQL = SQL + Programming**
declare
assign
print
read

Data Types in PL/SQL:

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

Integer related	**Number(p)** **Integer** **Int** **pls_integer => PL/SQL only** **binary_integer => PL/SQL only**
Floating Point related	**Number(p,s)** **Float** **binary_float** **binary_double**

Date & Time related	**Date** **Timestamp**
Binary related	**BFILE** **BLOB**
Boolean related	**Boolean => PL/SQL only [till oracle 21c]** **In ORACLE 23ai, boolean data type is available 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].**
- **Variable is a name of storage location.**
- **It is used to hold the data.**
- **TO hold the data variable is required**
- **A variable can hold only 1 value at a time.**

Note:

- **to hold the data variable is required.**
- **to allocate the memory for variable we use data type.**

Declaring variable:

Syntax

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

Examples:

```
a NUMBER(4);
b VARCHAr2(10);
c DATE;
```

Assigning Value:

:=	**Assignment Operator**

Syntax:

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

                                          (value)
```

Right side value will be stored in left side variable.

Example:

```
a := 25;

b := 'RAJU';
```

Printing data:

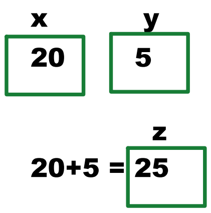
```
dbms_outpurt.put_line(a);
dbms_output.put_line(b);
```

Reading data:

```
a := &a;
Output:
enter value for a: 20
```

declare	x NUMBER(4);
assign	x:=25;
print	dbms_output.put_line(x);
read	x:=&x;

Program to add 2 numbers:



- declare 3 variables as number
- assign 20 to x
- assign 5 to y
- calculate x+y and store it in z
- print z

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

```
         x := 20;
         y := 5;

         z := x+y;

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

Program to add 2 numbers. Read 2 numbers at run time:

```
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;
/
```

```
SQL> SET SERVEROUTPUT ON

SQL> @D:\batch730am\ReadDemo.sql
Output:
Enter value for x: 50
old   6:        x := &x;
new   6:        x := 50;
Enter value for y: 20
old   7:        y := &y;
new   7:        y := 20;
sum=70
```

```
SQL> SET VERIFY OFF
```

if we set VERIFY as OFF, ORACLE does not display old and new parameters.

```
SQL> /
Output:
Enter value for x: 50
Enter value for y: 20
sum=70
```

Using SQL commands in PL/SQL:

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

Using SELECT command in PL/SQL:

Syntax:

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

Example:

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

Display the emp record of given empno:

```
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 value for empno: 7499
ALLEN    1600
```

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

EMP

```
EMP                                   PROGRAM
   empno NUMBER(4)                      v_empno NUMBER(2)
   --------------                         -99 TO 99
   7369
    7499
```

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

```
EMP                                  EMP
   empno NUMBER(4)                      v_empno DATE
```

%TYPE:

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

Syntax:

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

Example:

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

v_empno variable data type will be taken as
EMP table's EMPNO column's data type [NUMBER(4)]

```
v_ename EMP.ENAME%TYPE;

v_sal EMP.SAL%TYPE;
```

Display the emp record of given empno:

```
enter .. empno: 7369
SMITH      800
```

```
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;
/
```

Program to check the balance of given account number:

ACCOUNTS

ACNO	NAME	BALANCE
1234	A	80000
1235	B	40000

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

INSERT INTO accounts VALUES(1234,'A',80000);
INSERT INTO accounts VALUES(1235,'B',40000);
COMMIT;
```

Program:

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

    SELECT balance INTO v_balance FROM accounts
    WHERE acno=v_acno;

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

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:

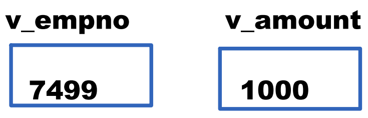
```
ENTER ... EMPNO: 7369
experience=43 years
```

Using UPDATE command in PL/SQL:

Note:
UPDATE, INSERT and DELETE commands syntaxes are same as SQL.

Example:
Program to increase salary of given empno with given amount:

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

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

     COMMIT;

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



```
enter .. empno: 7499
enter .. amount: 1000
sal increased..
```

%ROWTYPE:

- **It is Attribute Related Data Type.**
- **It is used to hold entire row of the table.**
- **It can hold one row at a time.**
- **It reduces no of variables**

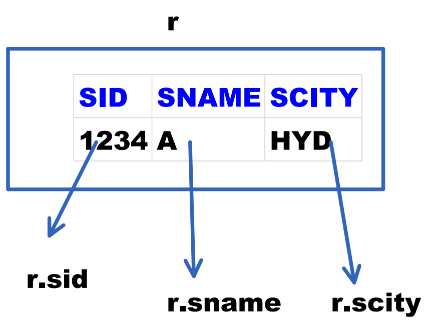
Syntax:
```
    <variable> <table_name>%ROWTYPE;
```

Example:
```
    r STUDENT%ROWTYPE;
```

STUDENT TABLE

SID	SNAME	SCITY
1234	A	HYD
1235	B	BLR
1236	C	PUN

```
SELECT * INTO r FROM student
WHERE sid=1234;
```



Program to insert student record into student table:

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

Program:

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

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

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

r

SID	SNAME	SCITY
1234	A	HYD

Output:
```
enter .. sid: 1234
enter .. sname: A
enter .. scity: HYD
```

```
record inserted..
```

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;
/
```

v_empno

$\boxed{7499}$

r

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

Output:

```
enter .. empno: 7499
ALLEN     2600     ...
```

Using DELETE command in PL/SQL:

Program to delete emp record of given empno:

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

   DELETE FROM emp WHERE empno=v_empno;
   COMMIT;

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

v_empno

$\boxed{7788}$

Output:

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

declare	x INT;
assign	x:=20;
print	dbms_output.put_line(x);
read	x:=&x;
initialize	x INT := 20;

%TYPE	**used to declare a variable with table column's data type** **v_empno EMP.EMPNO%TYPE;**
%ROWTYPE	**used to hold entire row** **r EMP%ROWTYPE;**

Control Structures

Friday, August 9, 2024 7:52 AM

Control Structures:

- **Control Structure is used to control the flow of execution of program.**
- **Normally, PL/SQL program gets executed sequentially. To change 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	**while**
	for
	simple loop

Jumping	**goto** **exit** **exit when** **continue** **return**

Conditional Control Structures:

Conditional Control Structure execute the statements based on conditions.

PL/SQL provides conditional control structures:

- **IF .. THEN**
- **IF .. THEN .. ELSE**
- **IF .. THEN .. ELSIF**

- **NESTED IF**
- **CASE**

IF .. THEN:

Syntax:

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

condition => T

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

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

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

   SELECT hiredate INTO v_hiredate FROM emp
   WHERE empno=v_empno;

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

   IF v_exp>42 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                          --condition T
ELSE
   --Statements                          --condition F
END IF;
```

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

Example on IF .. THEN .. ELSE:

Program to increase salary of given empno based on job.
if job is MANAGER then increase 20% on sal
for others, increase 10% on salary.

```
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;
/
```

Assignment:
program to increase salary of given empno based on deptno.
if emp is working in
deptno 30 then increase 15.5% on sal
for others, increase 10.2% on sal

program to increase salary of given emp no with given
amount. after increment if sal is more than 10000 cancel it:

```
read empno
read amount
UPDATE the salary

SELECT sal copy into v_sal

IF v_sal>10000 THEN
   ROLLBACK;
ELSE
   COMMIT;
END IF;
```

IF .. THEN .. ELSIF:

Syntax:

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

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

Example on IF .. THEN .. ELSIF:

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

```
if job is MANAGER then increase 20% on sal
          CLERK                         10%
          OTHERS                         5%
```

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_job EMP.JOB%TYPE;
   v_per FLOAT;
BEGIN
```

v_empno

$\boxed{7369}$

v_job

```
    v_per FLOAT;
BEGIN
    v_empno := &empno;

    SELECT job INTO v_job FROM emp
    WHERE empno=v_empno;

    IF v_job='MANAGER' THEN
       v_per := 20;
    ELSIF v_job='CLERK' THEN
       v_per := 10;
    ELSE
       v_per := 5;
    END IF;

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

    COMMIT;

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

$\boxed{7369}$

v_job

$\boxed{\text{CLERK}}$

v_per

$\boxed{10}$

Output:

```
enter .. empno: 7369
job=CLERK
10% on sal increased..
```

Assignment:
Program to increase salary given empno based on joining year as following:

if emp joined in	1980 then increase	20.7% on sal
	1981	18.2%
	1982	15%
	others	12.8%

```
read empno
find given empno's year of joining

SELECT hiredate INTO v_hiredate FROM emp
```

```
WHERE empno=v_empno;

v_year := to_char(v_hiredate, 'YYYY');

IF v_year = 1980 THEN
```

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

Syntax:

```
IF <condition1> THEN
    IF <condition2> THEN
        --Statements
    END IF;
END IF;
```

Program on NESTED IF:

STUDENT

SID	SNAME	M1	M2	M3
1001	A	70	90	80
1002	B	55	30	60

RESULT

SID	TOTAL	AVRG	RESULT
v_sid	r2.total	r2.avrg	r2.result

Program to find total, average and result of given student id:
Max marks: 100
Min marks: 40 for pass in each subject
if pass, check average
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

PROGRAM:

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

   SELECT * INTO r1 FROM student
   WHERE sid=v_sid;

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


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

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

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

v_sid

1001

r1

SID	SNAME	M1	M2	M3
1001	A	70	90	80

r2

SID	TOTAL	AVRG	RESULT
	240	80	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 can 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. When all constants are not matched with expression value, it executes ELSE statements. Writing ELSE is optional.

Example:

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

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

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

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

Syntax of searched CASE:

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

The statements in Searched CASE get executed when corresponding condition is TRUE.
When all conditions are FALSE, it executed ELSE statements.
Writing ELSE is optional.

Example program on Searched CASE:

Program to check whether the given number is +ve or -ve or 0:

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

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

   CASE
   WHEN n>0 THEN
      dbms_output.put_line('+ve');
   WHEN n<0 THEN
      dbms_output.put_line('-ve');
   ELSE
      dbms_output.put_line('0');
   END CASE;
END;
/
```

Assignment:

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

```
if deptno 10 then increase 10% on sal
            20                  20%
          others                 5%
```

```
read empno
find deptno    =>  v_deptno

CASE v_deptno
WHEN 10 THEN
```

```
              v_per := 10;
```

Program to increase salary of given empno as following:
if sal is >=5000   then increase 20% on sal
if sal is between 3000 to 4999 then increase 10% on sal
otherwise, increase 5% on sal

```
read empno
find sal          => v_sal

CASE
WHEN v_sal>=5000 THEN
   v_per := 20;
WHEN v_sal>=3000 THEN
   v_per := 10;
ELSE
   v_per:=5;
END CASE;

UPDATE
```

Looping Control Structures:

1 Lakh statements	1 Lakh hellos
dbms_output.put_line('hello');	hello
dbms_output.put_line('hello');	hello
dbms_output.put_line('hello');	hello
.	
.	
dbms_output.put_line('hello');	

```
LOOP 1 Lakh times
  dbms_output.put_line('hello');
END LOOP;
```

Looping Control structure is used to execute the code repeatedly.

PL/SQL provides 3 Looping Control Structures:

- **while**
- **simple loop**
- **for**

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, it terminates the loop.

Example program on WHILE:

Program to print numbers from 1 to 4:

i

```
1
```

Output:
i

```
i:=1;
```

```
1
```

Output:
i

```
1
2
3
4
```

```
i:=1;

d_o.p_l(i);   --1
i:=i+1;     --i=2

d_o.p_l(i);   --2
i:=i+1;     --i=3

d_o.p_l(i);   --3
i:=i+1;     --i=4

d_o.p_l(i);   --4
```

```
WHILE 4 times
LOOP
   d_o.p_l(i);
   i:=i+1;
END LOOP;
```

```
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;
```

Program to print numbers from 1 to 4:

```
i

1
2
3
4
```

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

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

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

$\longrightarrow$ equivalent to

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

EXIT:

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

EXIT WHEN:

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

```
BEGIN
   dbms_output.put_line('hi');
```

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

Output:
ERROR:
EXIT can be used in loop only

For Loop:

Syntax:

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

Example on FOR loop:

Program to print numbers from 1 to 4:

```
i
1
2
3
4
```

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

Note:

- We have no need to declare loop variable. Implicitly it will be declared as NUMBER type.

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

  Example:
```
BEGIN
    FOR i IN 1 .. 10
    LOOP
        i := 5;
        dbms_output.put_line(i);
    END LOOP;
END;
/
```
  Output:
```
ERROR:
'I' cannot be used as an assignment target
```

- **For Loop variable scope is limited to LOOP only.**
  Scope => availability

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

        dbms_output.put_line(i);
    END;
    /
```
  Output:
```
    ERROR: I must be declared
```

Program to print numbers from 4 to 1:

```
BEGIN
```

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

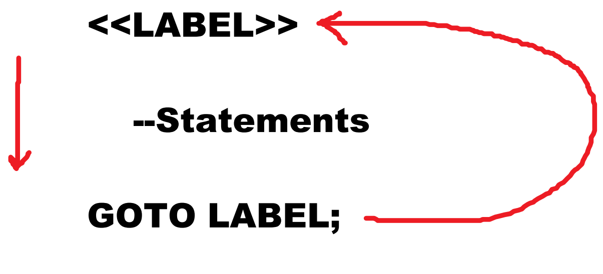
Program to print even numbers b/w 1 to 20:

```
2
4
6
8
.
.
20
```

```
BEGIN
    FOR i IN 2 .. 20 BY 2          --from oracle 21c
    LOOP
      dbms_output.put_line(i);
     END LOOP;
END;
/
```

GOTO:
When GOTO statement is executed execution jumps to specified label.

Syntax:



```
<<LABEL>>

    --Statements

GOTO LABEL;
```

Example on GOTO:

Program to print numbers from 1 to 4:

```
i

1
2
3
4
```

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

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

CONTINUE:

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

Example on CONTINUE:

Program to print numbers from 1 to 10 except 7:

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

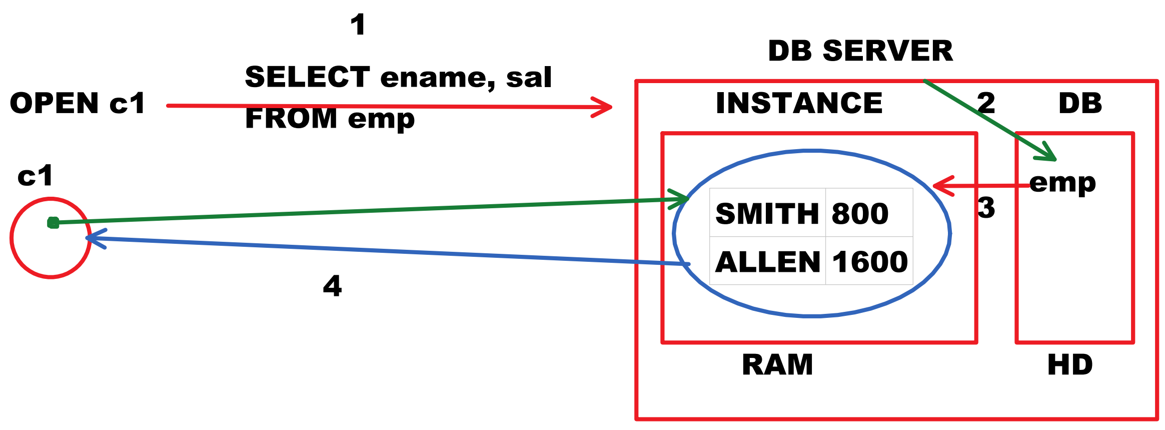
CURSORS

Wednesday, August 14, 2024 8:46 AM

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

To hold 1 column value we use %TYPE
To hold 1 row we use %ROWTYPE
To hold multiple rows we use CURSOR

Note:
Every CURSOR is associated with SELECT QUERY.



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

Steps to use CURSOR:

4 steps:

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

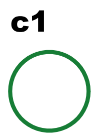
Declaring Cursor:

Syntax:

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

Example:

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



When cursor is declared,

- **cursor variable will be created.**
- **select query will be identified.**

Opening Cursor:

Syntax:

```
OPEN <cursor_name>;
```

Example:

```
OPEN c1;
```

When cursor is opened,

1. **SELECT query will be submitted to ORACLE.**
2. **ORACLE goes to DB.**
3. **Selects the data from table and copies into ORACLE INSTANCE.**
4. **This memory location address will be given to CURSOR.**

Now cursor has multiple rows.

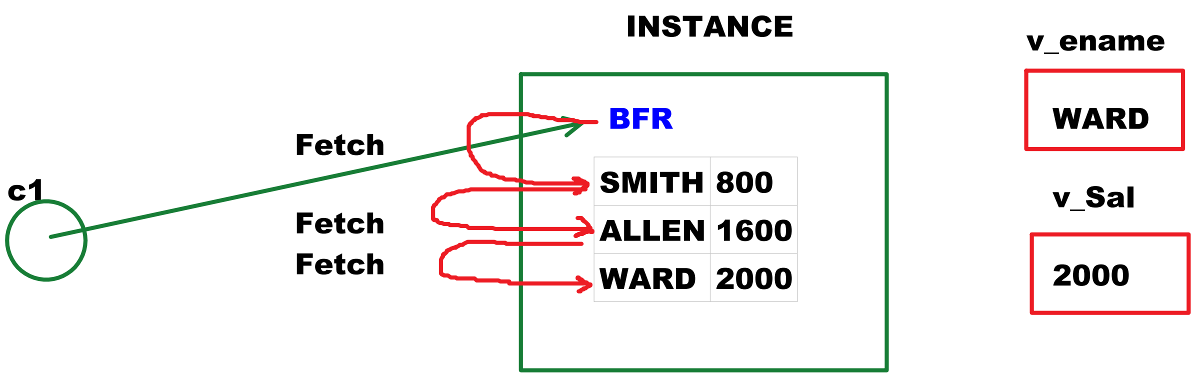
Fetching record from CURSOR:

Syntax:

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

Example:

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



BFR => Before First Row

When FETCH statement is executed,

- **it goes to next row and copies into corresponding variables.**

One FETCH statement can fetch one row only.
To FETCH multiple rows we write FETCH statement in LOOP.

Closing Cursor:

Syntax:

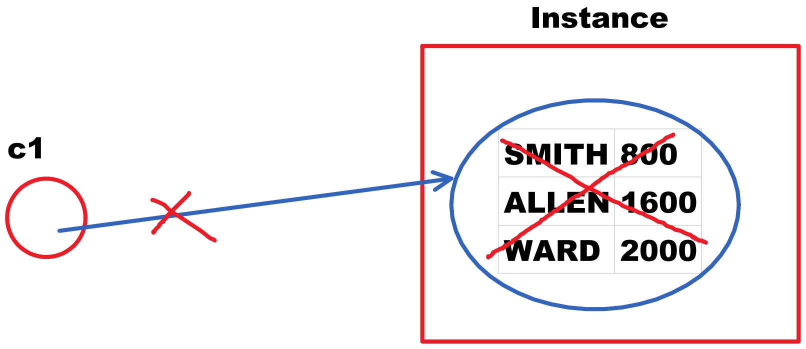
```
CLOSE <cursor_name>;
```

Example:

```
CLOSE c1;
```

When CURSOR is closed,

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



Cursor Attributes:

4 Cursor Attributes:

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

Syntax to use cursor attribute:

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

Examples:

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

%FOUND:
- **It returns boolean value [true / false].**
- **If record is found, it returns TRUE.**
- **If record is not found, it returns FALSE.**

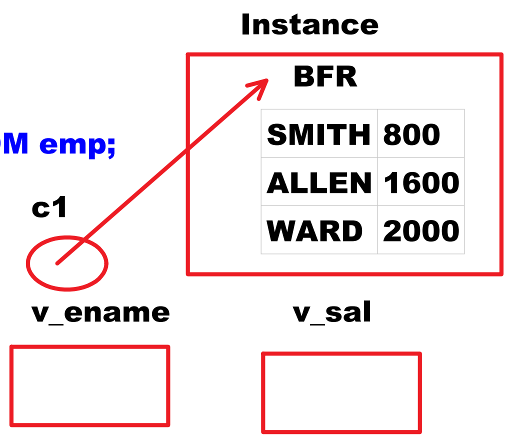
%NOTFOUND:
- **It returns boolean value [true / false].**
- **If record is not found, it returns TRUE.**
- **If record is found, it returns FALSE.**

%ROWCOUNT:
- **by default, it's value is 0.**
- **If fetch is successful, ROWCOUNT value will be incremented by 1.**

%ISOPEN:
- **it returns boolean value.**
- **If cursor is opened, it returns TRUE.**
- **If cursor is not opened, it returns FALSE.**

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

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 fetched..');

   CLOSE c1;
END;
/
```

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

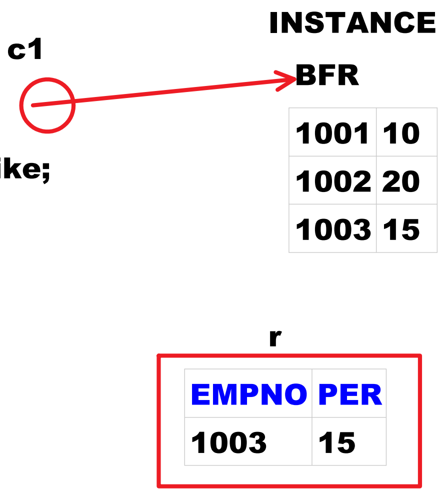
EMPLOYEE

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

HIKE

EMPNO	PER
1001	10
1002	20
1003	15

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

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

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

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



1001	10
1002	20
1003	15

EMPNO	PER
1003	15

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

   LOOP
      FETCH c1 INTO r;

      EXIT WHEN c1%notfound;

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

   COMMIT;

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

```
    CLOSE c1;
END;
/
```

Program to find total, average and result of all students and insert them in RESULT table:

STUDENT

SID	SNAME	M1	M2	M3
1001	A	70	90	80
1002	B	55	30	60

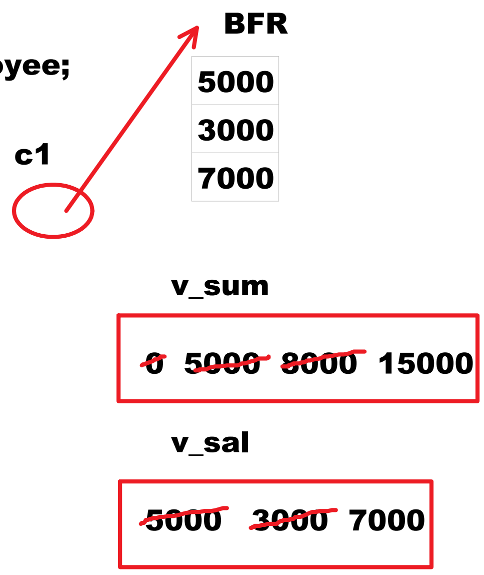
RESULT

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 calculated and stored in result table');

    CLOSE c1;
END;
/
```

Program to find sum of salaries of all emps:

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

   LOOP
      FETCH c1 INTO v_sal;

      EXIT WHEN c1%notfound;

      v_sum := v_sum+v_sal;
   END LOOP;

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

   CLOSE c1;
END;
/
```



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.

Example on CURSOR FOR LOOP:

Display all emp records:

```
DECLARE
   CURSOR c1 IS SELECT * FROM emp;
BEGIN

   FOR r IN c1
   LOOP
      dbms_output.put_line(r.ename || '   ' || r.sal);
   END LOOP;
END;
/
```

Assignment:
Find sum of salaries of all emps using CURSOR FOR LOOP.

Inline Cursor:

- **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;
/
```

Parameterized Cursor:

- **Cursor with parameter is called "Parameterized Cursor".**
- **When we don't know exact value at the time of declaration we take that as the parameter and this value will be passed at the time opening cursor.**

Syntax:

```
CURSOR <cursor_name>(<parameters_list>) IS <select query>;
```

Example:

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

```
OPEN c1(10);
```

n value is 10
c1 holds 10th dept records

Example on parameterized cursor:

```
DECLARE
   CURSOR c1(n NUMBER) IS SELECT * FROM emp WHERE deptno=n;
   r EMP%ROWTYPE;
BEGIN
   OPEN c1(10);

   LOOP
      FETCH c1 INTO r;

      EXIT WHEN c1%notfound;

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

   CLOSE c1;
END;
/
```

Types of Cursors:

2 types:
- ○ **Implicit Cursor**
- ○ **Explicit Cursor**
  - ▪ **Simple Cursor**
  - ▪ **Ref Cursor**

Simple Cursor:
- **In Simple Cursor,**
  One Cursor can be used for 1 select query only.
- **It is fixed.**
- **it has no data type.**

- it cannot be used as procedure parameter.

Ref Cursor:

- In Ref Cursor, Same cursor can be used for multiple select query.
- In this, select query can be changed.
- It reduces no of cursors.
- it has data type. i.e: SYS_REFCURSOR
- It can be used as procedure parameter.

Simple Cursor:	Ref Cursor:
CURSOR c1 => SELECT * FROM emp	CURSOR c1 => SELECT * FROM emp
CURSOR c2 => SELECT * FROM dept	=> SELECT * FROM dept
CURSOR c3 => SELECT * FROM salgrade	=> SELECT * FROM salgrade

Declaring Ref Cursor and Opening Ref Cursor:

Declaring Ref Cursor:

Syntax:

```
<cursor_name> SYS_REFCURSOR;
```

Example:

```
c1 SYS_REFCURSOR;
```

Opening Ref Cursor:

Syntax:

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

Example:

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

Example on Ref Cursor:

Program to display all emp records Using cursor.
Using same cursor also 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 b/w Simple Cursor and Ref Cursor:

Simple Cursor	Ref Cursor
• In simple cursor, one cursor can be used for 1 select query only.	• In ref cursor, same cursor can be used for multiple select queries.
• in this, select query is fixed.	• in this, select query 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. Because, it has data type.
• in this, select query will be specified at the time of declaring cursor.	• in this, select query will be specified at the time of opening cursor.

Types of Cursors:

2 types:

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

Note:
To execute any DRL or DML command 1 cursor is required.

Implicit Cursor:

- **To execute any DRL or DML command implicitly ORACLE uses a CURSOR. it is called "Implicit Cursor".**
- **Implicit Cursor name is: SQL.**

  SQL%FOUND
  SQL%NOTFOUND
  SQL%ROWCOUNT

SQL%ISOPEN

Program to increase 1000 rupees salary to all emps:

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

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

   COMMIT;
END;
/
```

Program to increase salary of given empno with given amount:

```
DECLARE
   v_empno EMP.EMPNO%TYPE;
   v_amount 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('emp not existed with this empno');
   ELSE
      COMMIT;
      dbms_output.put_line('sal increased..');
   END IF;
END;
/
```

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

4 steps:

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

Cursor For Loop:
no need to open, fetch and close

Inline cursor:
no need to declare also.
we specify select query in cursor for loop.

parameterized cursor:
a cursor with parameter
CURSOR c1(n NUMBER)

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

Types of cursor:

2 types:

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

Exception Handling

Monday, August 19, 2024 8:49 AM

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

Types of Errors:

3 Types:

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

Compile Time Errors:

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

  Examples:
  missing ;
  missing '
  missing)
  mising END IF
  missing END LOOP

Run Time Errors:

- **These errors occur at run time.**
- **These errors occur due to following reasons:**
  - **when record is not found**
  - **when size is exceeded**
  - **when wrong input is given**
  - **when we try to divide with 0**
  - **when we insert duplicate value in PK**
  - **when check constraint violated**

When run time error occurs program will be terminated abnormally.

Problem:
Abnormal termination
it leads to wrong results. we may loss the data.

Logical Errors:

- **These errors occur due to mistake in logic.**
- **it leads to wrong results.**
- **Programmer is responsible to develop correct logic.**

  Example:
  Withdraw => curr_balance+withdrawl_amount

Exception:

- **Exception means Run Time Error.**
- **It is a problem.**
- **When run time error occurs our program will be terminated in the middle of execution.**

Problem:
Abnormal Termination.

It leads to wrong results.
we may loss the data.

Exception Handling:

- **It is the solution for Exception [Run Time Error].**
- **The way of handling run time errors is called Exception handling.**
- **To handle run time errors we define EXCEPTION block.**

  Syntax:

```
DECLARE
```

```
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('you cannot divide with
            0');
        WHEN value_error THEN
            dbms_output.put_line('value is out of range or
            wrong input is given');
        WHEN others THEN
            dbms_output.put_line('something went
            wrong..');
END;
/
```

```
Output-1:
Enter value for x: 20
Enter value for y: 5
z=4

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

Output-3:
Enter value for x: 123456
Enter value for y: 2
value is out of range or wrong input is given

Output-4:
Enter value for x: 'RAJU'
Enter value for y: 2
value is out of range or wrong input is given
```

Types of Exceptions:

2 Types:

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

Built-In Exception:

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

Examples:

```
    zero_divide
    value_error
    no_data_found
    dup_val_on_index
    too_many_rows
    invalid_cursor
    cursor_already_open
    others
```

User-Defined Exception:

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

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

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

no_data_found:
when we retrieve the data if record is not found, no_data_found exception will be raised.

Example on no_data_found:

Program to display emp record of given empno:

```
DECLARE
    v_empno EMP.EMPNO%TYPE;
    r EMP%ROWTYPE;
BEGIN
```

```
      v_empno := &empno;

      SELECT * INTO r FROM emp WHERE
      empno=v_empno;

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

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

```
   Output-1:
   Enter .. empno: 7499
   ALLEN     1600

   Output-2:
   Enter value for empno: 6123
   no employee existed with this empno..
```

dup_val_on_index:
when we try to insert duplicate value in PRIMARY KEY, dup_val_on_index exception will be raised.

Example on dup_val_on_index:

Program to insert student record in student table:

STUDENT

SID	SNAME	M1

```
CREATE TABLE student
(
sid NUMBER(4) CONSTRAINT c100 PRIMARY KEY,
sname VARCHAR2(10),
m1 NUMBER(4)
);


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..');

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

r

sid	sname	m1
1001	A	70

Output-1:

```
Enter value for sid: 1003
Enter value for sname: C
Enter value for m1: 45
record inserted..
```

Output-2:

```
Enter value for sid: 1003
Enter value for sname: D
Enter value for m1: 77
PK does not accept dups..
```

too_many_rows:

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

Example on too_many_rows:

Program to display emp records based on job:

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

   SELECT * INTO r FROM emp WHERE job=v_job;

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

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

```
output-1:
Enter value for job: PRESIDENT
KING   PRESIDENT   8000

output-2:
Enter value for job: CLERK
selected multiple rows..
```

Invalid_Cursor:

- when we try to ftech for the record without opening cursor, invalid_cursor exception will be raised.

Example on Invalid_Cursor:

Program to display all emp records:

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

      EXIT WHEN c1%notfound;

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

   CLOSE c1;

   EXCEPTION
      WHEN invalid_Cursor THEN
         dbms_output.put_line('cursor is not opened..');
END;
/
```
Output:
```
cursor is not opened..
```

Cursor_Already_Open:
When we try to open opened cursor,
Curosr_Already_Open exception will be raised.

Example on Cursor_Already_Open:

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

   OPEN c1;

   LOOP
      FETCH c1 INTO r;
```

```
        EXIT WHEN c1%notfound;

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

    CLOSE c1;

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

Output:

```
cursor already opened..
```

others:

- **it can handle any exception**

Example on others:

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

    z := x/y;

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

    EXCEPTION
        WHEN others THEN
```

```
            dbms_output.put_line('something went wrong..');
END;
/
```

```
output-1:
Enter value for x: 10
Enter value for y: 0
something went wrong..

output-2:
Enter value for x: 'raju'
Enter value for y: 2
something went wrong..
```

User-Defined Exception:

- **We can define our own exceptions. These are called "User-Defined Exceptions".**
- **If our requirement is not fulfilled with Built-In Exception then we define user-defined exception.**

Examples:

```
one_divide
sunday_not_allow
xyz
raju
```

Note:

For Built-In Exception, follow 1 step:

- **HANDLE**

[name is ready, it will be raised implicitly]

For User-Defined Exception follow 3 steps:

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

Declaring Exception:

Syntax:

```
<exception_name> EXCEPTION;
```

Examples:

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

Raising Exception:

Syntax:

```
RAISE <exception_name>;
```

Examples:

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

HANDLING EXCEPTION:

Syntax:

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

Example:

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

Example on user-defined exception:

Program to divide 2 numbers.

if denominator is 0 then run time error occurs. handle it.
if denominator is 1 then raise the exception and handle it:

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

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

   z := x/y;

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

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

Output-1:

```
Enter value for x: 10
Enter value for y: 5
z=2
```

Output-2:

```
Enter value for x: 10
Enter value for y: 0
you cannot divide with 0
```

Output-3:

```
Enter value for x: 10
Enter value for y: 1
you cannot divide with 1
```

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

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

   IF to_char(sysdate,'DY')='SUN' THEN
      RAISE sunday_not_allow;
   END IF;

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

   COMMIT;

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

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

Output-1 [mon-sat]:

```
Enter value for empno: 7499
Enter value for amount: 1000
```

```
sal increased..
```

Output-2 [on Sunday]:

```
Enter value for empno: 7499
Enter value for amount: 1000
you cannot update on sunday..
```

We can RAISE the EXCEPTION using 2 ways:

- **Using RAISE keyword**
- **Using RAISE_APPLICATION_ERROR() procedure**

RAISE_APPLICATION_ERROR():

- **it is a procedur.**
- **It is used to raise the exception explicitly with our own error code and error message.**
- **User-defined error code must be b/w -20000 TO -20999.**

  Syntax:
  ```
  RAISE_APPLICATION_ERROR(<user-defined error code>,
  <user-defined error message>)
  ```

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

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

Example on Raise_Application_Error():

```
DECLARE
   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-1:
Enter value for x: 20
Enter value for y: 0
ERROR at line 1:
ORA-01476: divisor is equal to zero

Output-2:
Enter value for x: 20
Enter value for y: 1
ERROR at line 1:
ORA-20050: you cannot divide with 1
```

Differences RAISE and RAISE_APPLICATION_ERROR():

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

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

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

   z := x/y;

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

   EXCEPTION
      WHEN others THEN
         dbms_output.put_line(SQLERRM);
         dbms_output.put_line(SQLCODE);
END;
/
```

Output:

```
Enter value for x: 20
Enter value for y: 0
ORA-01476: divisor is equal to zero
-1476
```

Pragma Exception_Init():

-1476	**Error Code**

divisor is equal to zero	Error Message
zero_divide	Exception name

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

-2290	Error Code
Check constraint violated	Error Message
NO EXCEPTION NAME defined	Exception name

-1400	Error Code
cannot insert NULL	Error Message
NO EXCEPTION NAME defined	Exception name

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

- **To handle run time error name is required in exception block.**
- **To define name for unnamed exception we use Pragma Exception_Init().**

  Syntax:

  ```
  Pragma Exception_init(<user_defined_exception_name>,
  <built-in_error_code>)
  ```

  Example:

  ```
  Pragma Exception_init(check_violate, -2290);
  Pragma Exception_init(null_not_allow, -1400);
  ```

  - **Pragma Exception_init() is a compiler directive.**
  - **directive => command / instruction.**
  - **It instructs that before compiling execute this line.**

Example on pragma exception init():

STUDENT

SID	SNAME	M1

```
PK                CHECK
c200              c201
```

```
CREATE TABLE student
(
sid NUMBER(4) CONSTRAINT c200 PRIMARY KEY,
sname VARCHAR2(10),
m1 NUMBER(3) CONSTRAINT c201 CHECK(m1 BETWEEN 0 AND 100)
);
```

Program:

```
DECLARE
   r STUDENT%ROWTYPE;
   check_violate EXCEPTION;
   pragma Exception_Init(check_violate, -2290);
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..');

   EXCEPTION
      WHEN dup_val_on_index THEN
         dbms_output.put_line('student existed with this sid..');
      WHEN check_violate THEN
         dbms_output.put_line('marks must be b/w 0 to 100..');
END;
```

/

Exception [problem]	**Run time error** **problem:** **abnormal termination** **we may loss the data** **we may get wrong result**
Exception handling [solution]	**is a mechanism of handling run time errors** **for exception handling we add EXCEPTION block**

Types of Exceptions:

2 types:

built-in => 1step   => handle
user-defined => 3 steps => declare, raise, handle

we can raise error in 2 ways:
RAISE keyword
RAISE_APPLICATION_ERROR() procedure

PRAGMA EXCEPTION_INIT():
to define name for unnamed unnamed exception we use PRAGMA EXCEPTION_INIT().

Stored Procedures

Thursday, August 22, 2024 8:50 AM

Procedure:

- **is a named block of statements that gets executed on calling.**
- **It can be also called as sub program.**

Types of procedures:

2 types:

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

Stored Procedure:
If procedure is defined in SCHEMA [user] then it is called "Stored Procedure".

Example:

```
SCHEMA c##batch730am
      PROCEDURE withdraw
```

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

Example:

```
SCHEMA c##batch730am
   PACKAGE bank
      PROCEDURE withdraw
```

Syntax of Stored procedure:

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

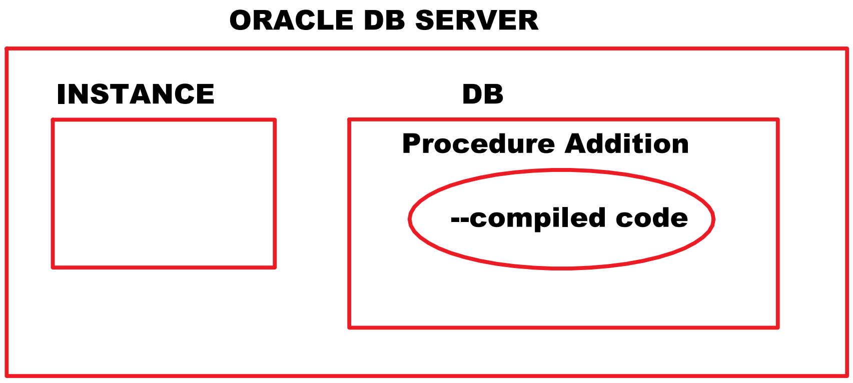
Define a procedure to add 2 numbers:

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

- write above code in text editor like notepad.
- save it in D: drive, batch730am folder, with the name ProcedureDemo.sql.

- Open SQL PLUS
- log in as user

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



Calling a procedure:

3 ways:

- From SQL command prompt
- From PL/SQL program [main program]
- From Programming Language [java, python, .net]

Calling from SQL command prompt:

Syntax:

```
   EXEC[UTE] <procedure_name>(<arguments_list>)
```

Example:

```
   SQL> SET SERVEROUTPUT ON
```

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

Calling from PL/SQL program [main program]:

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

   addition(a,b);
END;
/
```

Note:
to modify existing procedure's code we use "OR REPLACE".

Advantages:
- **it provides reusability.**
- **it improves the performance.**
- **it decreases length of code.**

Note:
PROCEDURE = HEADER + BODY

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

PROCEDURE HEADER /
PROCEDURE SPECIFICATION

PROCEDURE BODY

Parameters:
- **a local variable which is declared in procedure header is called "Parameter".**

Syntax:

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

Examples:

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

Parameter modes:

There are 3 parameter modes. They are:

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

IN:

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

Example:

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

Output:

```
procedure created with compilation errors

SQL> SHOW ERRORS
```

OUT:

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

IN OUT:

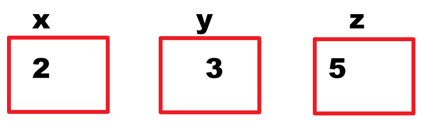
- **It is read-write parameter.**

- **Same parameter can be used to capture input and send output.**
- **In procedure call , it must be variable only.**

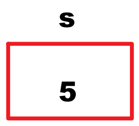
Example on OUT parameter:

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

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;
/
```

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(7369, 2000);
```

Output:

```
sal increased..
```

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

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

   COMMIT;

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

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

Calling from SQL prompt:

```
SQL> VAR s NUMBER
SQL>EXEC update_salary(7369, 2000, :s);
SQL> PRINT s
```

ACCOUNTS

ACNO	NAME	BALANCE
1234	A	80000
1235	B	40000

Define a procedure to perform withdraw operation:

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

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

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

   COMMIT;

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

Calling:

```
SQL> EXEC withdraw(1234, 100000);
```

Output:

```
ORA-20050: Insufficient funds..
```

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

Output:

```
transaction successful..
```

Define a procedure to perform deposit operation.
After depositing, 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 accounts SET balance=balance+p_amount
   WHERE acno=p_acno;

   COMMIT;

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

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

ACCOUNTS

ACNO	NAME	BALANCE
1234	A	80000
1235	B	40000

Calling:

```
SQL> VAR b NUMBER;

SQL> EXEC deposit(1235, 20000, :b);
```

Output:

```
transaction successful..

SQL> PRINT b
```