IN OUT

Monday, August 26, 2024 7:41 AM

IN	**captures input**
OUT	**sends result**
IN OUT	**same parameter captures input and sends result**

Example on IN OUT:

Define a procedure to find square of a number:

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

Calling from SQL prompt:

```
SQL> VAR a NUMBER
SQL> EXEC :a := 2;
SQL> EXEC square(:a);
SQL> PRINT a
```

pragma autonomous_transaction

Monday, August 26, 2024 8:12 AM

pragma autonomous_transaction:

Transaction:

- is a series of actions [SQL commands]
- Transaction must be successfully finished or cancelled

main program

TRANSACTION t1
update => 7499, 1000
hike(7369, 2000)
commit
t1 ended

PROCEDURE hike(... , ...)

UPDATE => 7369, 2000
ROLLBACK => cancels t1

EMP

EMPNO	ENAME	SAL
7369	SMITH	800+2000 = 2800 rolled back
7499	ALLEN	1600+1000 = 2600 rolled back

main program

TRANSACTION t1
update => 7499, 1000
hike(7369, 2000)
commit => saves t1
t1 ended

PROCEDURE hike(... , ...)

pragma autonomous_transaction

transaction t2
UPDATE => 7369, 2000
ROLLBACK => cancels t2
t2 ended

EMP

EMPNO	ENAME	SAL
7369	SMITH	800+2000 = 2800 rolled back
7499	ALLEN	1600+1000 = 2600 committed

By default, A transaction started in main program will be continued to procedure. To create separate transaction for procedure we use "pragma autonomous_transaction".

It is compiler directive.

It is command to compiler. It instructs that before compiling execute this line.

Example on pragma autonomous_transaction:

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

   ROLLBACK;
END;
/
```

main program:

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

Granting permission on procedure to other user:

c##batch730am:

PROCEDURE addition(x INT, y INT)

GRANT execute

ON addition
TO c##userA;

c##userA:
SQL> EXEC c##batch730am.addition(2,3);
Output:
sum=5

Dropping procedure:

Syntax:
DROP PROCEDURE <procedure_name>;

Example:
DROP PROCEDURE hike;

parameter mapping techniques

user_source
user_procedures

Parameter mapping techniques

Tuesday, August 27, 2024 7:42 AM

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

x, y => formal parameters

```
addition(5,4)
```

5,4 => actual parameters

Parameter mapping techniques /
Parameter association techniques /
Parameter Notations:

There are 3 parameter mapping techniques. They are:

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

Positional mapping:
In this, actual parameters are mapped with formal parameters based on positions.

Example:

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

positions

```
addition(1, 2, 3)
```

Named mapping:
In this, actual parameters are mapped with formal

parameters based on names.

Example:

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

Names

addition(z=>1, x=>2, y=>3)

Mixed mapping:

- **In this, actual parameters are mapped with formal parameters based on positions and names.**

Example:

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

positions and names [mixed]

addition(1,z=>2,y=>3)

addition(z=>1, 2, 3) => ERROR

NOTE:

after named mapping we cannot use positional mapping

Example:

Define a procedure to add 3 numbers:

addition(1,2,3)

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

```
END;
/
```

Calling:

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

Output:

```
sum=60
x=10
y=20
z=30
```

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

Output:

```
sum=60
x=20
y=30
z=10
```

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

Output:

```
sum=60
x=10
y=30
z=20
```

user_procedures
user_source

user_procedures:

- **it is a system table / built-in table.**
- **it maintains all procedures, functions, packages information.**

to see procedures info:

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

user_source:

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

to see procedures info:

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

to see procedure's code:

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

STORED FUNCTIONS

Tuesday, August 27, 2024 8:20 AM

FUNCTION:

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

There are 2 types of functions:

- **Stored functions**
- **Packaged functions**

Stored function:

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

Example:

```
SCHEMA c##batch730am
    FUNCTION check_balance
```

Packaged Function:

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

Example:

```
SCHEMA c##batch730am
    PACKAGE bank
        FUNCTION check_balance
```

NOTE:

- **To perform DML operations define PROCEDURE.**
- **To perform calculations or FETCH [select] operations define FUNCTION.**

Example:

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

experience                 => calculation =>  FUNCTION
```

transaction statement => fetch => FUNCTION
checking balance => fetch => FUNCTION

Syntax to define Stored Function:

```
CREATE OR RELPALCE FUNCTION
<name>(<parameters_list>) RETURN <data_type>
AS
   --declare the variables
BEGIN
    --statements
    return <expressions>;
END;
/
```

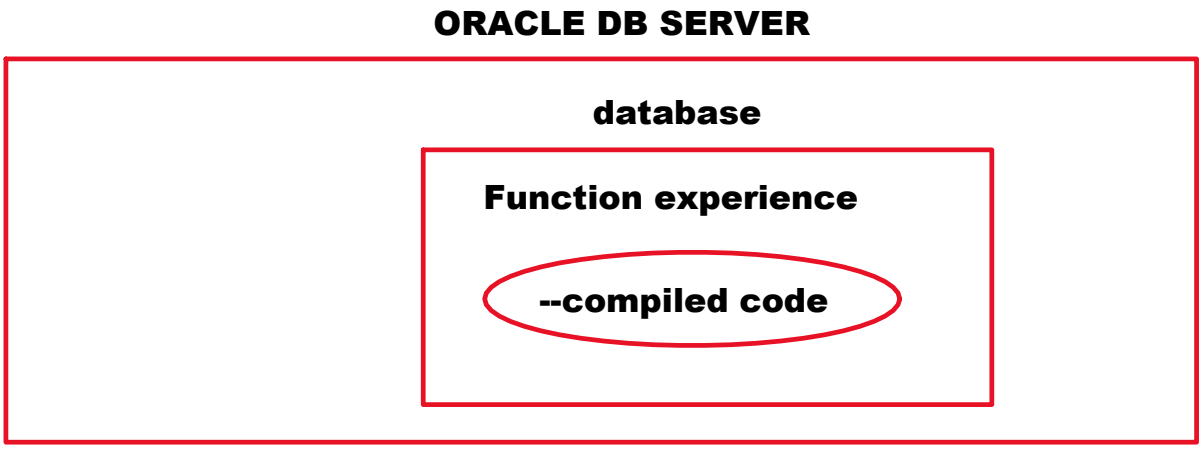
Note:

- **Every function returns the value.**
- **returning value is mandatory.**
- **in Function, take all parameters as IN parameters.**

Examples on defining functions:

Define a function to calculate experience of specific employee:

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

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



Calling a Function:

3 ways:

- **from SQL prompt**
- **from PL/SQL program**
- **from programming languages**

- **from SQL prompt:**

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

Display all emp names, hiredates along with experience.
display emp names in lower case:

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

ENAME	lower(ename)	EMPNO	experience(empno)
SMITH	lower('SMITH')	7369	experience(7369)
ALLEN	lower('ALLEN')	7499	experience(7499)

Calling from PL/SQL program:

DECLARE

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

    v_exp := experience(v_empno);

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

Define a function to check the balance of specific account:

ACCOUNTS

ACNO	NAME	BALANCE
1001	A	50000
1002	B	30000

```
CREATE OR REPLACE FUNCTION
check_balance(p_acno NUMBER) RETURN NUMBER
AS
    v_balance NUMBER;
BEGIN
    SELECT balance INTO v_balance FROM accounts
    WHERE acno=p_acno;

     RETURN v_balance;
END;
/
```

calling:

```
SQL> select check_balance(1001) FROM dual;

CHECK_BALANCE(1001)
-------------------
```

50000

Define a function to get specific dept emp records:

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

       RETURN c1;
END;
/
```

Calling:

```
SQL> SELECT getdept(30) FROM dual;
--displays 30th dept record
```

Define a function to display top-n salaried emp records:

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

    RETURN c1;
END;
/
```

calling:

```
SELECT gettopn(3) FROM dual;
--displays top 3 salaried emp records
```

Differences Procedure and Function:

PROCEDURE	FUNCTION
• PROCEDURE may or may not return the value.	• FUNCTION always returns the value.
• Returning value is optional.	• Returning value is mandatory.
• to return the value we use OUT parameter.	• to return the value we use RETURN keyword.
• PROCEDURE can return any number of values.	• FUNCTION can return 1 value only.
• To perform DML operations define PROCEDURE.	• To perform calculations or fetch operations define FUNCTION.
• Example: withdraw	Example: check_balance
• A procedure cannot be called from SQL command	• A function can be called from SQL command

Can we perform DML operations using FUNCTION?
Yes. It is not recommended.
If we perform DML operation in function, it cannot be called from SELECT command.

Can we take OUT parameters in FUNCTION?
Yes. It is not recommended.
FUNCTION standard is: FUNCTION returns 1 value.
If we take OUT parameter it is against to FUNCTION standard.

Granting permission on function to other user:

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

login as c##userA:

```
SELECT c##batch730am.experience(7369) FROM dual;
```

Dropping Function:

Syntax:

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

Example:

```
DROP FUNCTION getdept;
```

```
user_procedures => all procedures, functions, packages info
user_source     => all procedures, functions, packages and triggers info
                   including code
```

to see list of functions:

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

(or)

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

to see function code:

```
SELECT text
FROM user_source
```

WHERE name='EXPERIENCE';

NOTE:

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

Advantages of Sub Program [Function or procedure]:

- **improves the performance.**
- **provides reusability.**
- **reduces length of code.**
- **improves understandability.**
- **Better maintenance.**
- **provides security.**

PACKAGES

Wednesday, August 28, 2024 8:50 AM

PACKAGE:

- **PACKAGE is one ORACLE DB OBJECT.**
- **It is a collection of procedures, functions, data types, exceptions, cursors and variables.**

Creating Package:

2 steps:

- **Package specification**
- **Package body**

Package specification:

Syntax:

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

Package body:

Syntax:

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

Example on creating package:

PACKAGE math

```
PROCEDURE addition
```

PACKAGE math

PROCEDURE addition
FUNCTION product

PACKAGE SPECIFICATION:

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

PACKAGE BODY:

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

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

Calling:

```
SQL> EXEC math.addition(1,2);
Output:
sum=3

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

Example:

PACKAGE HR

```
PROCEDURE hire      =>  INSERT
PROCEDURE fire     => DELETE
PROCEDURE hike => UPDATE
FUNCTION experience => calculation
```

PACKAGE SPECIFICATION:

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

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

PACKAGE BODY:

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

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

   PROCEDURE hike(p_empno NUMBER, p_amount NUMBER)
   AS
   BEGIN
      UPDATE emp SET sal=sal+p_amount WHERE empno=p_empno;
```

```
        COMMIT;
        dbms_output.put_line('sal increased..');
    END hike;

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

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

Calling:

```
SQL> EXEC HR.HIRE(1001,'A',6000);
Output:
record inserted..

SQL> EXEC HR.HIKE(1001,2000);
Output:
sal increased..

SQL> EXEC HR.FIRE(1001);
Output:
record deleted..

SQL> SELECT hr.experience(7934) FROM dual;
Output:
HR.EXPERIENCE(7934)
-------------------
                42
```

Advantages of Package:

- we can group related procedures and functions.
- It improves the performance.
- We can declare global variables.
- We can overload packaged procedures and packaged functions.
- provides security.
- Better maintenance.
- We can make members as public or private.

Note:

- Stored procedure or Stored function cannot be overloaded.

- **packaged procedure and packaged function can be overloaded.**

Overloading:

- **Defining multiple procedures or functions with same name and different signature is called "Overloading".**
- **Different signature means,**
  - **change in number of parameters**
  - **change in data types**
  - **change in order of parameters**

Example:

```
PACKAGE DEMO
    PROCEDURE p1(x INT)
    PROCEDURE p1(x INT, y VARCHAR2)
    PROCEDURE p1(x DATE)
    PROCEDURE p1(x VARCHAR2, y INT)
```

Example on Overloading:

```
PACKAGE OLDEMO

FUNCTION addition => 2 numbers
FUNCTION addition => 3 numbers
Global variable x => initialize with 500
```

PACKAGE SPECIFICATION:

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

PACKAGE BODY:

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

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

END;
/
```

Calling:

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

SQL> SELECT OLDEMO.addition(2,3,4) FROM dual;
Output:
OLDEMO.ADDITION(2,3,4)
----------------------
                   9

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

```
DECLARE
  a INT := 20;
BEGIN
  dbms_output.put_line((OLDEMO.x+a));
END;
/
Output:
520
```

Note:

- **Declaring in package specification means, we are making members as public.**

Example:

PACKAGE SPECIFICATION

```
PACKAGE demo2
   PROCEDURE p2 => public
```

PACKAGE BODY

```
PACKAGE BODY demo2
   PROCEDURE p1  => private
```

```
PACKAGE demo2
   PROCEDURE p2 => public
   PROCEDURE p3 => public
```

```
PACKAGE BODY demo2
   PROCEDURE p1  => private
   PROCEDURE p2
   PROCEDURE p3
```

PACKAGE SPECIFICATION:

```
CREATE OR REPLACE PACKAGE demo2 AS
   PROCEDURE p2;
   PROCEDURE p3;
END;
/

CREATE OR REPLACE PACKAGE BODY demo2 AS
   PROCEDURE p1
   AS
   BEGIN
      dbms_output.put_line('p1 called');
   END p1;

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

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

Calling:

```
SQL> EXEC demo2.p2;
Output:
p1 called
p2 called

SQL> EXEC demo2.p3;
Output:
p1 called
```

p3 called

SQL> EXEC demo2.p1;
Output:
ERROR: p1 is private member.

user_procedures
user_source

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

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

to see package code:

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

Dropping Package:

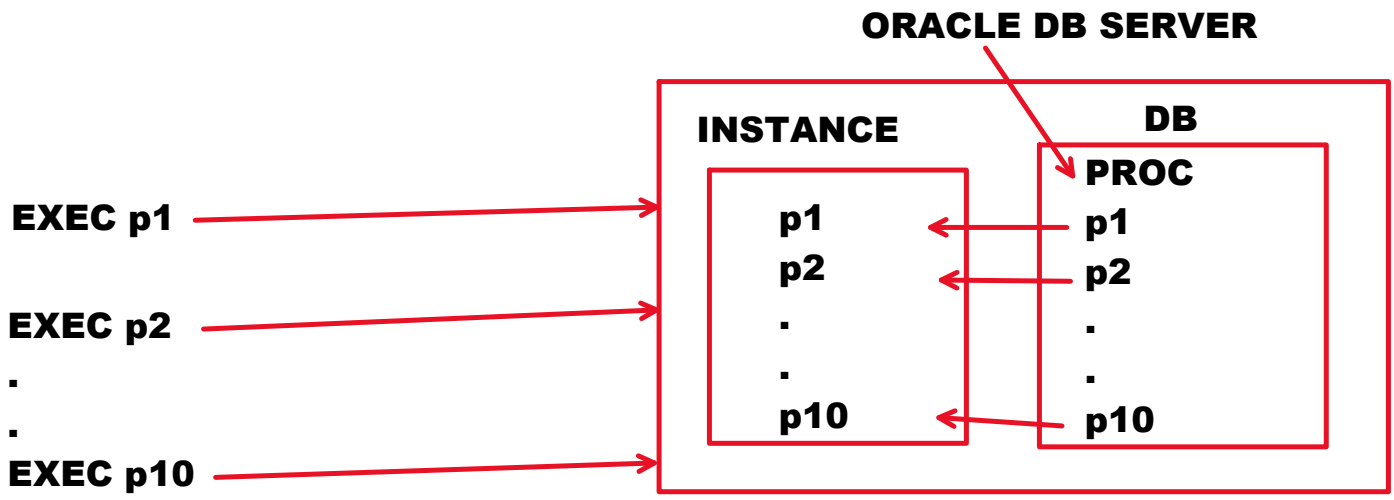
Syntax:
DROP PACKAGE <name>;

Example:
DROP PACKAGE demo2;

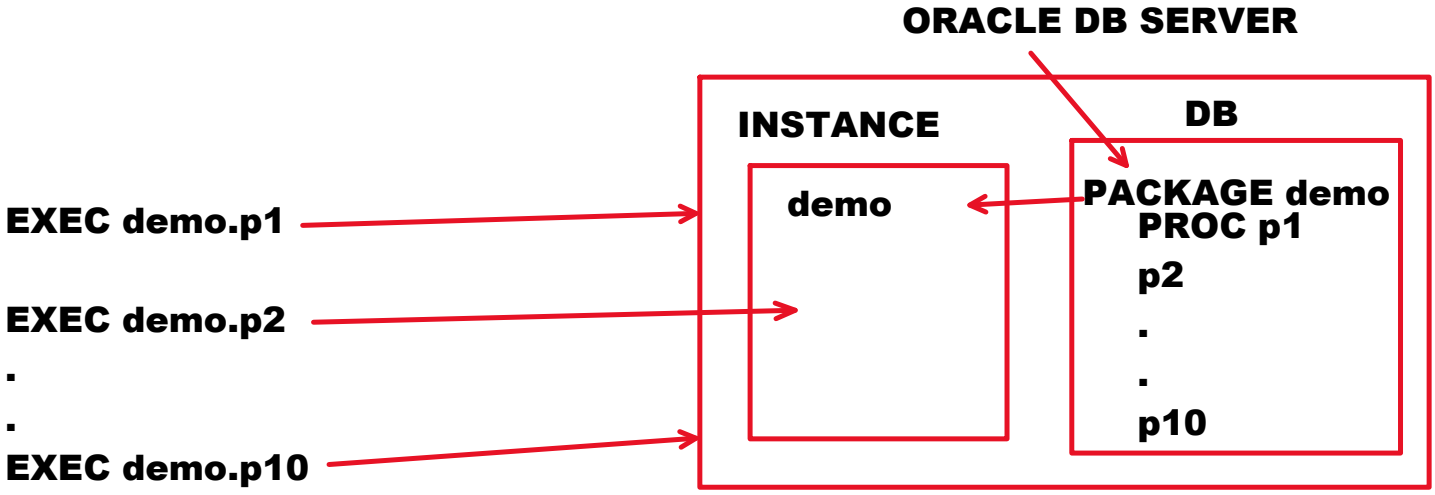
Dropping Package Body:

Syntax:
DROP PACKAGE BODY <name>;

Example:
DROP PACKAGE BODY OLDEMO;



If no of travels to DB are increased, performance will be degraded.

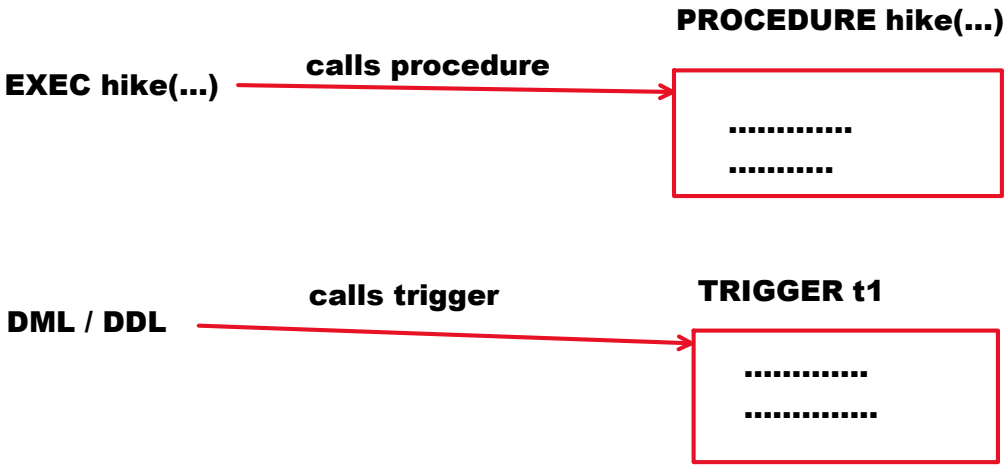


Using PACKAGE, no of travels to DB are reduced. So, it improves the performance.

TRIGGERS

Friday, August 30, 2024 7:44 AM

TRIGGER:



TRIGGER:

- **TRIGGER is one ORACLE DB OBJECT.**

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

- **TRIGGER is same as PROCEDURE.**
  For PROCEDURE execution explicit call is required.
  For TRIGGER execution explicit call is not required.

Note:

- **To perform DML operations, define PROCEDURE.**
- **To Control DML operations, define TRIGGER.**

TRIGGER can be used for following purposes:

- **To control DMLs.**

  Example:
  don't allow user to perform DMLs on SUNDAY
  don't allow user to perform DMLs before and after office timings

- **To audit the tables.**

  Example:
  which user
  on which date
  at which time
  which actions

all above things can be recorded in another table.
This is called "Auditing".

- To implement our own business rules (constraints)

Example:
don't allow the user to decrease the salary

Types of triggers:

3 Types:

- Table Level Trigger / DML Trigger
- Schema Level Trigger / DDL Trigger / System Trigger
- Database Level Trigger / DDL Trigger / System Trigger

Table Level Trigger:
- If trigger is created on table then it is called "Table Level Trigger".

2 Types:
- Statement Level Trigger
- Row Level Trigger

Statement Level Trigger:
This Trigger gets executed one time for 1 DML statement.

Row Level Trigger:
This Trigger gets executed one time for every row affected by DML.

Syntax of Table Level Trigger:

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

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

Example on Statement Level Trigger:

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

Testing:

```
UPDATE emp SET sal=sal+1000;     --calls trigger
```

Output:

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

Example on Row Level Trigger:

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

Testing:

```
UPDATE emp SET sal=sal+1000;     --calls trigger
```

Output:

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

14 times
14 rows updated.

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

mon-sat:	on sun:
UPDATE => allow	UPDATE => don't allow
INSERT => allow	INSERT => don't allow
DELETE => allow	DELETE => don't allow

Note:
Before Trigger:
- First Trigger gets executed.
- Then DML operation will be performed.

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

```
CREATE OR REPLACE TRIGGER t3
BEFORE insert or update or delete
ON emp
BEGIN
   IF to_char(sysdate,'DY')='SUN' THEN
      raise_application_error(-20050, 'you cannot perform DMLs on Sunday..');
   END IF;
END;
/
```

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

On Sunday:
UPDATE emp SET sal=sal+1000;

Define a Trigger to don't allow the user to perform DMLs before or after office timings [office timings: 10AM to 4PM]:

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

      IF h NOT BETWEEN 10 AND 15 THEN
            raise_application_error(-20050, 'you can perform DMLs b/w 10AM to 4PM only');
      END IF;
END;
/
```

:NEW and :OLD:

- These are built-in variables.
- These are bind variables.
- These are %ROWTYPE variables.
- When trigger code is executed, at runtime these variables will be created implicitly.

- :NEW => holds new row
- :OLD => holds old row

- These can be used in row level trigger only.

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

```
INSERT INTO emp
VALUES(1234, 'A', 6000);
```

:NEW

EMPNO	ENAME	SAL
1234	A	6000

:OLD

EMPNO	ENAME	SAL
null	null	null

EMP

EMPNO	ENAME	SAL
1234	A	6000

```
UPDATE emp
```

:NEW

EMPNO	ENAME	SAL

```
UPDATE emp
SET sal=sal+2000
WHERE empno=1234
```

:NEW

EMPNO	ENAME	SAL
1234	A	8000

:OLD

EMPNO	ENAME	SAL
1234	A	6000

EMP

EMPNO	ENAME	SAL
1234	A	8000

```
DELETE FROM emp
WHERE empno=1234;
```

:NEW

EMPNO	ENAME	SAL
null	null	null

:OLD

EMPNO	ENAME	SAL
1234	A	8000

Define a Trigger to maintain deleted records in another table:

EMP_RESIGN

DOR	EMPNO	ENAME	JOB	SAL
7934				

```
delete from emp
where job=7934;
```

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

```
CREATE OR REPLACE TRIGGER t5
AFTER delete
ON emp
FOR EACH ROW
BEGIN
	INSERT INTO emp_resign VALUES(sysdate, :old.empno,
```

```
        :old.ename, :old.job, :old.sal);
END;
/
```

Testing:

```
DELETE FROM emp WHERE empno=7934;
```

Output:

1 row deleted

:old

EMPNO	ENAME	JOB	SAL
7934	MILLER	CLERK	3000

EMP_RESIGN

DOR	EMPNO	ENAME	JOB	SAL
31-AUG-24	7934	MILLER	CLERK	3000

Define a trigger to audit emp table:

emp_audit

uname	op_date_time	op_type	old_empno	old_ename	new_empno	new_ename
user	systimestamp	op	:old.empno	:old.ename	:new.empno	:new.ename

```
create table emp_audit
(
uname VARCHAR2(15),
op_date_time TIMESTAMP,
op_type VARCHAR2(10),
old_empno NUMBER(4),
old_ename VARCHAR2(10),
new_empno NUMBER(4),
new_ename VARCHAR2(10)
);
```

```
CREATE OR REPLACE TRIGGER t6
AFTER insert or delete or update
ON emp
FOR EACH ROW
DECLARE
    op VARCHAR2(10);
BEGIN
    IF inserting THEN
        op := 'INSERT';
    ELSIF deleting THEN
        op := 'DELETE';
    ELSIF updating THEN
```

```
            op := 'UPDATE';
        END IF;

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

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

```
CREATE OR REPLACE TRIGGER t10
BEFORE update
ON emp
FOR EACH ROW
BEGIN
   IF :new.sal < :old.sal THEN
      raise_application_error(-20050, 'you cannot decrease
      sal..');
   END IF;
END;
/
```

Testing:
SQL> UPDATE emp SET sal=sal-1000;
Output:
ERROR:
ORA-20050: you cannot decrease sal..

Define a trigger to don't allow the user to edit empno:

```
CREATE OR REPLACE TRIGGER t11
BEFORE update OF empno
ON emp
BEGIN
   raise_application_error(-20080, 'you cannot update empno..');
END;
/
```

Testing:
update emp set empno=1234
where empno=7369;
Output:
ERROR:
ORA-20080: you cannot update empno..

Schema Level Trigger / DDL Trigger / System Trigger:

- **DBA defines it.**
- **If trigger is created on SCHEMA [user] then it is called "Schema Level Trigger".**
- **It is defined to control 1 user's DDL operations.**

Syntax:

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

Example on Schema Level Trigger:

Define a trigger to don't allow c##batch730am user to drop any DB Object:

login as DBA:
username: system

```
CREATE OR REPLACE TRIGGER st1
BEFORE drop
ON c##batch730am.SCHEMA
BEGIN
      raise_application_error(-20090, 'you cannot drop any db obj..');
END;
/
```

login as c##batch730am:

DROP TABLE emp;
Output:
ERROR:
ORA-20090: you cannot drop any db obj..

DROP PROCEDURE addition;
Output:
ERROR:
ORA-20090: you cannot drop any db obj..

System Variable	Purpose
ora_dict_obj_type	It holds current object type Example: TABLE, PROCEDURE, PACKAGE, VIEW
ora_dict_obj_name	It holds object name Example: EMP, WITHDRAW, HR, V1
ora_sysevent	it holds DDL action Example: CREATE, DROP, ALTER
ora_login_user	it holds current user name Example: C##BATCH730AM

Define a trigger to don't allow c##batch730am user to drop table:

login as DBA:

```
CREATE OR REPLACE TRIGGER st2
BEFORE DROP
ON c##batch730am.SCHEMA
BEGIN
   IF ora_dict_obj_type='TABLE' THEN
      raise_application_error(-20070, 'you cannot drop table..');
   END IF;
END;
/
```

Testing:
SQL> drop procedure square;
Output:
Procedure dropped.

SQL> drop package hr;
Output:
Package dropped.

SQL> drop table emp;
Output:
ERROR:
ORA-04088: error during execution of trigger 'SYSTEM.ST2'
ORA-20070: you cannot drop table..

Database Level Trigger / System Trigger / DDL Trigger:

- **It is defined by DBA.**
- **If a trigger is created on DATABASE then it is called Database level trigger.**
  To control multiple users or all users DDL actions we define it.

  Syntax:

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

Example on Database level trigger:

Define a trigger to don't allow c##batch730am, c##batch4pm to drop any db object:

login as DBA:
username: system

```
CREATE OR REPLACE TRIGGER dt1
BEFORE DROP
ON DATABASE
BEGIN
   IF ora_login_user IN('C##BATCH730AM', 'C##BATCH4PM') THEN
       raise_application_error(-20050, 'you cannot drop any db obj..');
    END IF;
END;
/
```

Testing:

c##batch730am:

SQL> drop table emp;
Output:
ERROR:

ORA-20050: you cannot drop any db obj..

c##batch4pm:
SQL> drop table emp;
Output:
ERROR:
ORA-20050: you cannot drop any db obj..

Disabling and Enabling Trigger:

Syntax:

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

Example:

```
ALTER TRIGGER t3 DISABLE;
--t3 trigger will not work temporarily

ALTER TRIGGER t3 ENABLE;
--t3 trigger works again
```

Dropping Trigger:

Syntax:

```
DROP TRIGEGR <name>;
```

Example:

```
DROP TRIGGER dt1;

DROP TRIGGER st1;

DROP TRIGGER t3;
```

user_triggers:

- **it is a system table / readymade table.**
- **it maintains all triggers info.**

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

user_source:

- **it is a system table / readymade table.**
- **it maintains all triggers info.**

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

To see trigger code:

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

COLLECTIONS

Tuesday, September 3, 2024 7:46 AM

COLLECTION:

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

Example:

x

56
78
41
32

x(1)

NUMBER

y

SMITH
ALLEN
WARD
BLAKE

VARCHAR2

z

10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO

z(1).deptno => 10

DEPT%ROWTYPE

NOTE:

- **CURSOR is used to hold multiple rows.**
- **COLLECTION is used to hold multiple rows.**

- **CURSOR has some drawbacks. To avoid them we use COLLECTION.**

Types of Collections:

3 types:

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

Associative Array:

- **Associative Array is a table of 2 columns. They are: INDEX and ELEMENT.**
- **In this, INDEX can be VARCHAR2 type.**

Examples:

x

INDEX	ELEMENT
1	67
2	44
3	81
4	72

y

INDEX	ELEMENT
HYD	800000
DLH	1000000
BLR	900000

x(1) => 67
x(2) => 44

y('HYD') => 800000
y('DLH') => 1000000

Creating Associative Array:

Follow 2 steps:

- **Define our own data type**
- **Declare variable for that data type**

- **Define our own data type:**

Syntax:

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

Example:

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

Note:
If INDEX is NUMBER type write INDEX TYPE as binary_integer (or) pls_integer.

x

INDEX	ELEMENT
1	67
2	44
3	81
4	72

Declare variable for that data type:

Syntax:
<variable> <data_type>;

Example:
x NUM_ARRAY;

Collection Members:

MEMBER	PURPOSE
first	first index
last	last index
next	next index
prior	previous index

Example on Associative Array:

Create an Associative array as following:

x

INDEX	ELEMENT
1	67
2	44
3	81
4	72

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

    x NUM_ARRAY;
BEGIN
    x := num_array(67,44,81,72);     --oracle 21c only

    /*    x(1) := 67;    x(2) := 44;   ....   */

    dbms_output.put_line('x(1)=' || x(1));    --67
    dbms_output.put_line('first index=' || x.first);
    dbms_output.put_line('last index=' || x.last);
    dbms_output.put_line('next index of 2=' || x.next(2));
    dbms_output.put_line('prev index of 2=' || x.prior(2));

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

END;
/
```

Output:

```
x(1)=67
first index=1
last index=4
next index of 2=3
prev index of 2=1
67
44
81
72
```

Example:

Create an Associative Array and hold dept names in it and print them:

d

INDEX ELEMENT

```
-----------------------------------------------------
1                    ACCOUNTING
2                    RESEARCH
3                    SALES
4                    OPERATIONS
```

Program:

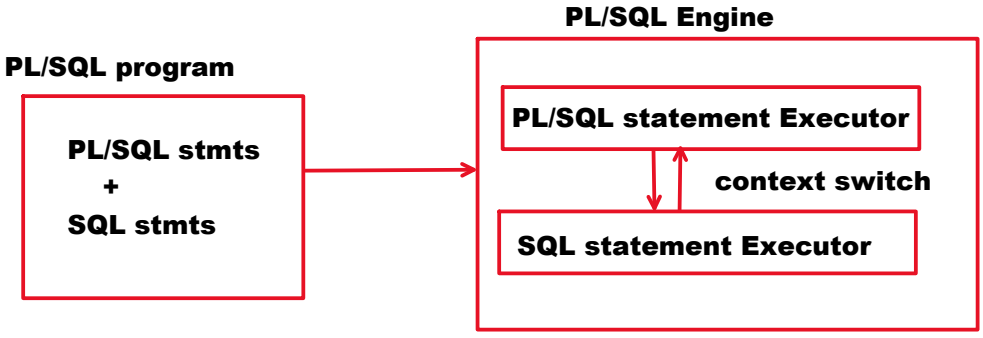
```
DECLARE
      TYPE dept_array IS TABLE OF varchar2(10)
      INDEX BY binary_integer;

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

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

Output:

```
---------------
ACCOUNTING
RESEARCH
SALES
OPERATITONS
```

Note:
Above program degrades the performance.

If no of context switches are increased then performance will be degraded.
To improve the performance we use BULK COLLECT.



BULK COLLECT:
=> It is used to collect entire data at a time with single context switch.

=> it reduces no of context switches.
=> it improves the performance.

Program to demonstrate BULK COLLECT:

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

      d DEPT_ARRAY;
BEGIN

       SELECT dname BULK COLLECT INTO d FROM dept;

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

Output:

ACCOUNTING
RESEARCH
SALES
OPERATITONS

Example:

Create an Associative Array, hold emp table records in it and print them:

e

INDEX	ELEMENT		
1	empno	ename	sal
	7369	SMITH	800
2	empno	ename	sal
	7499	ALLEN	1600

Program:

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

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

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

Assignment:
All programs of CURSOR => try with COLLECTION

Example:

EMPLOYEE

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

HIKE

EMPID	PER
1001	20
1002	10
1003	15

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

h

INDEX	ELEMENT		
1	EMPID: 1001, PER: 20	h(1)	h(1).per => 20 h(1).empid => 1001
2	EMPID: 1002, PER: 10	h(2)	
3	EMPID: 1003, PER: 15	h(3)	

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

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

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

```
      WHERE empid=h(i).empid;
   END LOOP;

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

Note:
Above program degrades the performance.
FOR LOOP increases no of context switches.
If no of context switches are increased performance will be degraded

To improve performance of above program we use
BULK BIND.

BULK BIND:

- **It is used to submit BULK INSERT / BULK UPDATE / BULK DELETE commands.**
- **For BULK BIND, we define FORALL loop.**
- **Using this, BULK INSERT commands / BULK UPDATE commands / BULK DELETE commands can be executed with single context switch.**
- **It reduces no of context switches. So, performance will be increased.**

Syntax of FOR ALL:

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

Example:

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

NOTE:

- **BULK COLLECT is related to DRL.**
- **BULK BIND is related to DML.**

Nested Table:

- **It is same as single dimensional array in C/Java.**
- **Nested table is a table of 1 column. i.e: ELEMENT.**
- **No need to maintain INDEX.**
- **Always INDEX is NUMBER type.**

Example:

x

ELEMENT
50

45
70
20

Creating Nested table:

2 steps:

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

define our own data type:

Syntax:

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

Example:

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

declaring variable for our own data type"

Syntax:

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

Example:

```
x NUM_ARRAY;
```

Example on Nested Table:

Create a nested table as following:

x

ELEMENT
50
45
70
20

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

   x NUM_ARRAY;
BEGIN
   x := num_array(50,45,70,20);

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

```
            dbms_output.put_line(x(i));
        END LOOP;
    END;
    /
```

Create a nested table, hold emp table records in it and print them:

e

EMPNO	ENAME	SAL
7369	SMITH	800

e(1)

EMPNO	ENAME	SAL
7499	ALLEN	1600

e(2)

```
DECLARE
   TYPE emp_array IS TABLE OF emp%rowtype;
   e EMP_ARRAY;
BEGIN
   SELECT * BULK COLLECT INTO e FROM emp;

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

V-ARRAY [variable size array]:

- **we must specify the size.**
- **it is same as nested table. it means, it maintains element only.**
- **always INDEX is NUMBER type.**

Example:

x

ELEMENT
50
45
70
20

Creating V-Array:

2 steps:

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

- define our own data type:

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

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

- declare variable for that data type:

  Syntax:
  <variable> <data_type>;

  Example:
  x num_array;

Example on V-Array:

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

   x NUM_ARRAY;
BEGIN
   x := num_array(50,45,70,20);

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

Example:

```
DECLARE
   TYPE emp_array IS VARRAY(20) OF emp%rowtype;
   e EMP_ARRAY;
BEGIN
   SELECT * BULK COLLECT INTO e FROM emp;

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

Differences b/w CURSOR and COLLECTION:

CURSOR	COLLECTION
• it can move forward only.	• it can move in any direction.
• It supports to sequential accessing only.	• supports to random accessing.
• CURSOR is slower.	• COLLECTION is faster.

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

COLLECTION	INDEX	NO OF ELEMENTS	DENSE or SPARSE
Associative Array	NUMBER / VARCHAR2	unlimited	Dense or Sparse
Nested Table	NUMBER	unlimited	starts as Dense it can be come Sparse
V-Array	NUMBER	limited	Dense

DENSE => no gaps

x(1)
x(2)
x(3)

SPARSE => gaps can be there

x(10)
x(20)
x(45)

Working with LOBs

Friday, September 6, 2024 7:46 AM

Working with LOBs:

- **LOB => Large Object => image, audio, video, large text**

Binary Related Data Types:

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

BFILE [binary file large object]:

- **It is used to maintain multimedia objects like images, audios, videos, ...etc.**
- **It is a pointer to multimedia object. It means, It maintains multimedia object's path.**
- **It can be also called as "External large object".**
- **It is not secured.**

Example

directory object
d1 => d:\photos

DATABASE

EMP1

EMPNO	ENAME	EPHOTO [BFILE]
1234	RAVI	bfilename('D1', 'ravi.jpg')

D: drive
photos Folder

ravi.jpg

Directory Object:

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

Syntax:

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

Example:
login as DBA:
username: system

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

GRANT read, write ON DIRECTORY d1
TO c##batch730am;
```

Example on BFILE:

login as c##batch730am:

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

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

COMMIT;

SELECT * FROM emp1;
```

BLOB:

- **BLOB => Binary Large Object**
- **It is used to maintain multimedia objects like images, audios, videos, ...etc.**
- **it maintains multimedia object inside of database.**
- **It can be also called "Internal Large Object".**
- **It is secured.**

Example:

DATABASE

EMP2

EMPNO	ENAME	EPHOTO [BLOB]

D: drive

photos folder

EMP2

EMPNO	ENAME	EPHOTO [BLOB]
1234	ravi	45AB67567E557F44A565BC

ravi.jpg

files are 2 types:

text file => contains text => .txt
binary file => binary data [0s, 1s] => images, audios, videos, .gif

Example on BLOB:

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

INSERT INTO emp2 VALUES(1234, 'ellison', empty_blob());
```

EMP2

EMPNO	ENAME	EPHOTO [BLOB]
1234	ellison	

Define a procedure to update emp photo:

procedure call:
update_photo(1234, 'ellison.jpg');

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

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

    dbms_lob.open(s, dbms_lob.lob_readonly);

    length := dbms_lob.getlength(s);   --3643

    dbms_lob.LoadFromFile(t,s,length);

    UPDATE emp2 SET ephoto=t WHERE empno=p_empno;

    COMMIT;

    dbms_output.put_line('image saved..');

    dbms_lob.close(s);

END;
/
```

DYNAMIC SQL

Monday, September 9, 2024 7:46 AM

DYNAMIC SQL:

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

- DYNAMIC SQL is used to execute dynamic queries.

- The query which is built at runtime is called "Dynamic Query".

static query:

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

Dynamic query:

```
'DROP TABLE ' || v_tname;
```

```
DROP TABLE emp
--this query built at run time
```

- EXECUTE IMMEDIATE command is used to execute dynamic query.

  Example:

  ```
  EXECUTE IMMEDIATE 'DROP TABLE ' || v_tname
  ```

Examples on Dynamic SQL:

Define a procedure to drop a table:

```
CREATE OR REPLACE PROCEDURE
drop_table(p_tname VARCHAR2)
AS
BEGIN
   EXECUTE IMMEDIATE 'DROP TABLE ' || p_tname;
```

```
   dbms_output.put_line(p_tname || ' table dropped');
END;
/
```

Calling:

```
SQL> EXEC drop_table('emp_audit');
Output:
emp_audit table dropped
```

Define a procedure to drop any DB Object:

```
CREATE OR REPLACE PROCEDURE
drop_object(p_obj_type VARCHAR2, p_obj_name VARCHAR2)
AS
BEGIN
   EXECUTE IMMEDIATE 'DROP ' || p_obj_type || ' ' || p_obj_name;

   dbms_output.put_line(p_obj_name || ' ' || p_obj_type || ' dropped');
END;
/
```

Calling:

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

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