Stored Procedures

Friday, November 1, 2024 6:20 PM

Procedure:

- **Procedure is a named block of statements that gets executed on calling.**
- **Procedure can be also called as "Sub Program".**
- **PROCEDURE is one ORACLE DB OBJECT.**

Types of Procedures:

2 Types:

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

Stored Procedure:

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

  Example:

  ```
  SCHEMA c##batch6pm
      PROCEDURE deposit    => Stored Procedure
  ```

Packaged Procedure:

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

  Example:

  ```
  SCHEMA c##batch6pm
      PACKAGE bank
          PROCEDURE deposit    => Packaged Procedure
  ```

Syntax to define stored procedure:

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

Procedure header / Procedure Specification

Procedure Body

Procedure = Procedure header + Procedure body

Example on Stored procedure:

Define a procedure to add 2 numbers:

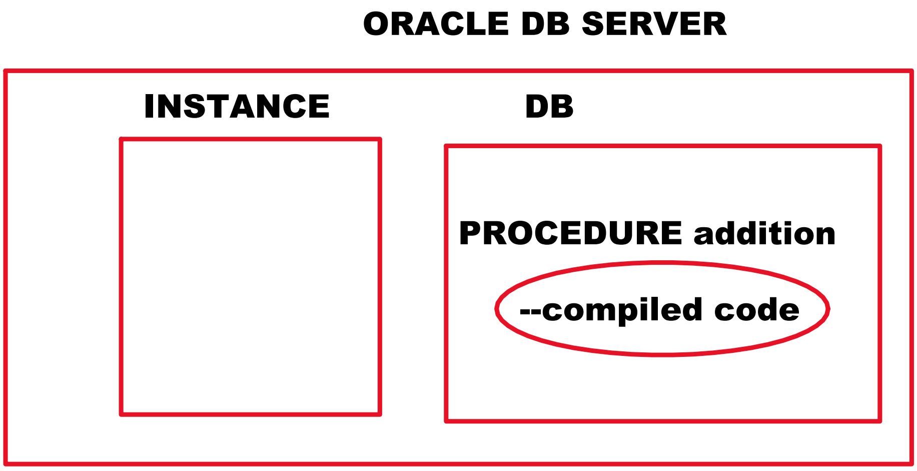
- **Open text editor => notepad**
- **Type following code**

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

- **save it in D: drive, batch6pm Folder, with the name ProcedureDemo.sql**

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

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



Calling a stored procedure:

3 ways:

- **Calling from SQL prompt**
- **Calling from PL/SQL program [main program]**
- **Calling from Programming languages [Java, Python, C#]**

Calling from SQL prompt:

Syntax:

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

Example:

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

Calling from PL/SQL program [main program]:

```
DECLARE
   a NUMBER(4);
   b NUMBER(4);

BEGIN
   a := &a;
   b := &b;

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

Note:
to see errors type following command:
SQL> SHOW ERRORS

Parameter:

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

Syntax:

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

Parameter modes:

3 modes:

- **IN**
- **OUT**
- **IN OUT**

IN:

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

Example:

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

Output:
PROCEDURE CREATED WITH COMPILATION ERRORS

```
SQL> SHOW ERRORS
```

Output:
ERROR: x cannot be used as assignment target

OUT:

- **It sends output.**
- **It is used to send the result out of the procedure.**
- **It is read-write parameter.**

- **In Procedure call, it must be variable only.**

IN OUT:

- **It takes input and sends output.**
- **It is read-write parameter.**
- **In Procedure call, it must be variable only.**

Example on OUT parameter:

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

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

calling from sql prompt:

```
SQL> VAR s NUMBER
SQL> EXEC addition(2,3,:s);
SQL> PRINT s
```

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:

procedure call:

```
update_salary(7369, 2000)
```

```
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 from sql prompt:

```
SQL> EXEC update_salary(7369, 2000);
```

Output:

```
sal increased..
```

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

```
EXEC update_salary(7369, 1000, :s)
```

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

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

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

Calling from SQL prompt:

```
SQL> VAR s NUMBER
SQL> EXEC update_salary(7369, 1000, :s);
SQL> PRINT s
```

Example on IN OUT parameter:

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

Calling from SQL prompt:

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

NOTE:

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

EXEC addition(2,3); **2,3 => Actual parameters**

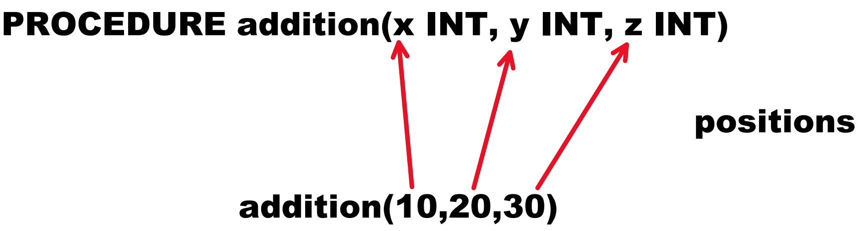
Parameter mapping techniques /
Parameter association techniques /
Parameter notations:

3 parameter mapping techniques:

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

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

Example:



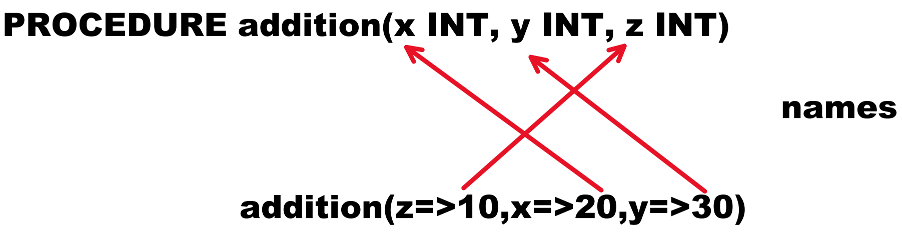
PROCEDURE addition(x INT, y INT, z INT)

positions

addition(10,20,30)

Named mapping:
In named mapping,
actual parameters are mapped with formal parameters
based on names.

Example:



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

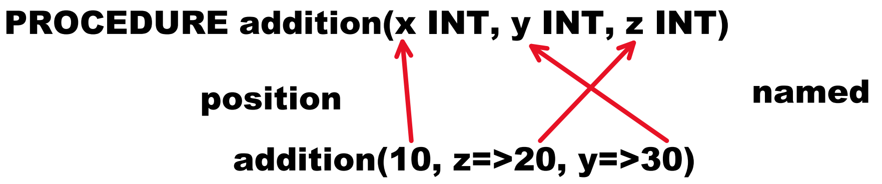
names

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

Mixed mapping:

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

Example:


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

position

named

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

```
addition(z=>10, 20, 30)
--error: after named mapping we cannot
         use positional mapping
```

Example:

Define a procedure for adding 3 numbers:

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

```
DESC user_procedures
```

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 information.
- it maintains code also.

```
DESC user_source
```

to see procedures info:

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

to see procedure code:

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

Granting permission on ADDITION procedure to c##userA:

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

Login as c##userA:

```
SQL> SET SERVEROUTPUT ON

SQL> EXEC c##batch6pm.addition(10,20,30);
```

Output:

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

Dropping a Procedure:

```
Syntax:
   DROP PROCEDURE <name>;

Example:
   DROP PROCEDURE addition;
```

Stored Functions

Monday, November 25, 2024 5:48 PM

FUNCTION:

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

Types of Functions:

2 Types:

- **Stored Function**
- **Packaged Function**

Stored Function:

- **A function which is defined SCHEMA is called "Stored Function".**

  Example:

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

Packaged Function:

- **A function which is defined in PACKAGE 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 operations, define FUNCTION.

Example:

```
insert_emp      => INSERT  => PROCEDURE
update_salary  => UPDATE => PROCEDURE

experience      => calculation   => FUNCTION
```

getdept => fetch => FUNCTION

Syntax to define a Stored Function:

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

Note:

- **In PL/SQL,**
  Function always returns the value.
  Returning value is mandatory.

- **In function don't define OUT parameters.**
- **Define all parameters as IN parameters only.**

Define a function to multiply 2 numbers:

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

   RETURN z;
END;
/
```

A function can be called in 3 ways. They are:

- **From SQL prompt**
- **From PL/SQL program**
- **From Programming Languages**

Calling From SQL prompt:

- **We can call a function from SQL commands.**

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

Calling from PL/SQL program:

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

      c := product(a,b);

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

Define a Function to calculate experience of an employee:

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

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

Calling:

```
SQL> SELECT experience(7369) FROM dual;
```

Display all emp names and 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)
SMITH	lower('SMITH') => smith
ALLEN	lower('ALLEN') => allen
WARD	lower('WARD') => ward

EMPNO	experience(empno)
7369	experience(7369) => 43
7499	experience(7499) => ..
7521	experience(7521) => ..

Define a function to display specific dept 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(20) FROM dual;
```

Define a function to display top n salaried emp records:

```
CREATE OR REPLACE FUNCTION
gettopn(n NUMBER) RETURN SYS_REFCURSOR
```

```
AS
   c1 SYS_REFCURSOR;
BEGIN
   OPEN c1 FOR SELECT * FROM (SELECT empno, ename,
   sal, dense_rank() over(order by sal desc) AS rank
   FROM emp) WHERE rank<=n;

   RETURN c1;
END;
/
```

Calling:

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

Differences b/w Procedure and Function:

PROCEDURE	FUNCTION
• PROCEDURE may or may not return the value.	• FUNCTION always returns 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.
• A PROCEDURE can return multiple values.	• A FUNCTION can return 1 value only.
• A PROCEDURE cannot be called from SQL command.	• A FUNCTION can be called from SQL command.

• To perform DML operations define PROCEDURE.
Examples:
insert_emp => procedure
update_sal => procedure
delete_emp => procedure
	• To perform calculations or fetch operations define FUNCTION.
	Examples:
	getdept => select
	experience => calc

• We cannot write RETURN statement in PROCEDURE.	• We can write RETURN statement in FUNCTION.

Can we perform DML operation through FUNCTION?
YES. It is not recommended.
If we perform DML operation through FUNCTION,
it cannot be called from SQL commands.

Can we define OUT parameters in FUNCTION?
YES. It is not recommended.
It is against to function standard.
Function standard is: A function returns 1 value only.

user_procedures
user_source

user_procedures:

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

```
DESC user_procedures
```

to see functions info:

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

user_source:

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

```
DESC user_source
```

to see functions info:

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

to see function's code:

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

Granting permission on PROUCT function to c##userA:

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

Login as c##userA:

```
SQL> SET SERVEROUTPUT ON

SQL> SELECT c##batch730am.product(2,3) FORM dual;
Output:
6
```

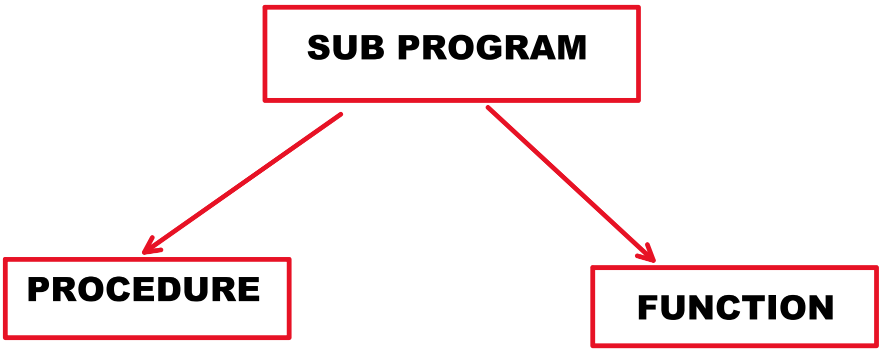
Dropping a Function:

Syntax:

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

Example:

```
DROP FUNCTION product;
```



Advantages of Sub Program:

- It improves performance. [it holds compiled code]
- It provides reusability.
- It reduces length of code.
- It improves understandability.
- Better maintenance.
- It provides security.

PACKAGES

Wednesday, November 27, 2024 7:49 AM

PACKAGE:

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

Creating a Package:

2 steps:

- **Package Specification [declarations]**
- **Package Body [body]**

Package Specification:

- **in this, we declare procedures, functions, exceptions, cursors, global variables ..etc.**

Syntax:

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

Package Body:

- **in this, we define body of procedures and functions.**

Syntax:

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

Example on creating package:

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 from SQL prompt:

```
SQL> EXEC math.addition(10,20);
Output:
sum=30

SQL> SELECT math.product(10,20) FROM dual;
Output:
200
```

Calling from PL/SQL program:

```
DECLARE
    a INT;
    b INT;
    c INT;
BEGIN
    a := &a;
    b := &b;

    math.addition(a,b);

    c := math.product(a,b);
    dbms_output.put_line('product=' || c);
END;
/
```

Example:

PACKAGE HR

PROCEDURE hire => INSERT
PROCEDURE fire => DELETE

```
PROCEDURE hire => INSERT
PROCEDURE fire => DELETE
PROCEDURE hike => UPDATE

FUNCTION experience => calc
```

```
--PACKAGE SPECIFICATION

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

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



--PACKAGE BODY

CREATE OR REPLACE PACKAGE BODY HR
AS
   PROCEDURE hire(p_empno NUMBER, p_ename VARCHAR2)
   AS
   BEGIN
      INSERT INTO emp(empno, ename)
      VALUES(p_empno, p_ename);

      COMMIT;

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

   PROCEDURE fire(p_empno NUMBER)
   AS
   BEGIN
      DELETE FROM emp WHERE empno=p_empno;
```

```
        COMMIT;
        dbms_output.put_line('record deleted..');
    END fire;

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

        COMMIT;

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

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

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

Calling:

```
SQL> EXEC hr.hire(1234, 'A');
```

Output:

```
record inserted..
```

```
SQL> EXEC hr.fire(1234);
```

Output:

```
record deleted..
```

```
SQL> EXEC hr.hike(7369, 1000);
```

Output:

```
sal increased..
```

```
SQL> SELECT hr.experience(7369) FROM dual;
Output:
HR.EXPERIENCE(7369)
------------------
                43

SQL> select ename, hr.experience(empno) as exp
        FROM emp;
Output:
ENAME          EXP
---------- ---------------------------
SMITH          43
JONES           43
MARTIN           43
```

Advantages of Package:

- **We can group related procedures and functions.**
- **It improves performance.**
- **It provides reusability.**
- **It reduces length of code.**
- **It improves understandability.**
- **Better maintenance.**
- **It provides security.**

- **We can declare global variables.**
- **Packaged procedures or Packaged functions can be overloaded.**
- **We can make members as public or private.**

Overloading:

- **Defining multiple procedures or functions with same name and different signatures is called "Overloading".**

- **Different signature means**
    - ○ **change in number of parameters**
    - ○ **change in data types**
    - ○ **change in order of parameters**

- **Stored procedures or stored functions cannot be overloaded WHERE AS packaged procedure or packaged function can be overloaded.**

  Example:

  ```
  PACKAGE demo
      PROCEDURE p1
      PROCEDURE p1(x INT)

      PROCEDURE p1(x DATE)
      PROCEDURE p1(x INT, y DATE)
      PROCEDURE p1(x DATE, y INT)
  ```

Example on Overloading:

```
PACKAGE OLDEMO
```

```
x INT     --global variable
FUNCTION addition(x INT, y INT)
FUNCTION addition(x INT, y INT, z INT)
```

```
--PACKAGE SPECIFICATION

CREATE OR REPLACE PACKAGE OLDEMO
AS
    x INT := 500;          --global variable
    FUNCTION addition(x INT, y INT) RETURN INT;
    FUNCTION addition(x INT, y INT, z INT) RETURN INT;
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(1,2), oldemo.addition(1,2,3)
         FROM dual;

SQL> exec dbms_output.put_line(oldemo.x+200);
```

NOTE:

- **Using PACKAGE, we can make members as public or private.**
- **Declaring in PACKAGE SPECIFICATION means, we are making members as public.**

Example:

PACKAGE SPECIFICATION

PACKAGE demo1

PROCEDURE p2;
PROCEDURE p3;

PACKAGE BODY

PACKAGE demo1

PROCEDURE p1
PROCEDURE p2
PROCEDURE p3

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

private member can be accessed with in PACKAGE only.

public members can be accessed from anywhere within SCHEMA.

user_procedures
user_source

user_procedures:

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

to see packages list created by a user:

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

user_source:

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

to see packages info:

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

to see package code:

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

Dropping Package:

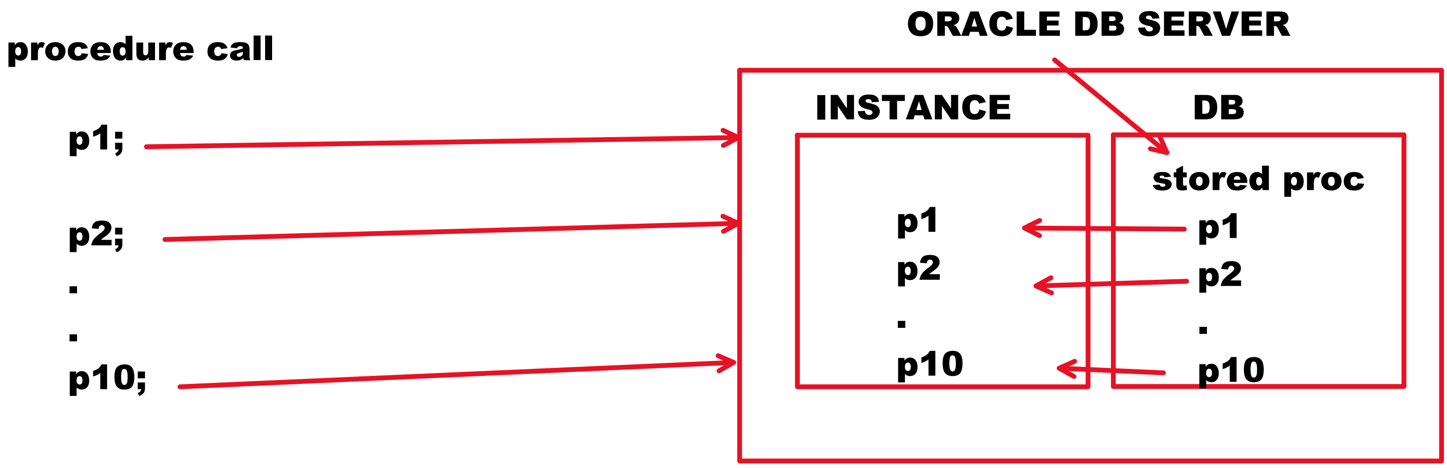
Syntax:

```
DROP PACKAGE <name>;
```

Example:

```
DROP PACKAGE demo1;
```

Wednesday, November 27, 2024 9:01 AM



procedure call

p1;

p2;

.

.

p10;

ORACLE DB SERVER

INSTANCE

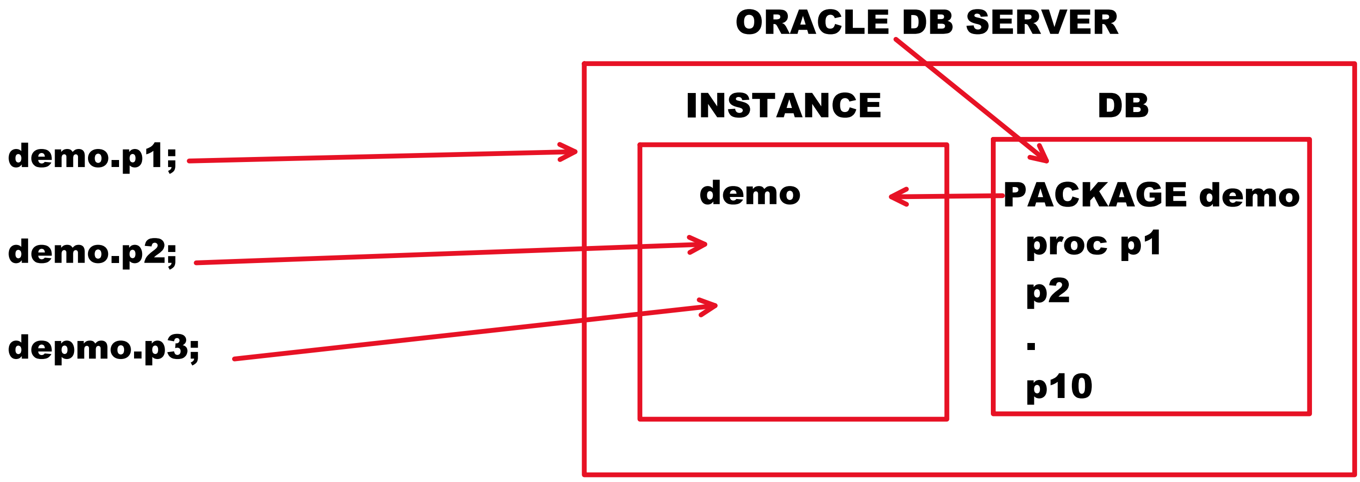
DB

p1
p2
.
p10

stored proc
p1
p2
.
p10

For every procedure call,
ORACLE goes to DB,
searches for procedure,
loads compiled code into INSTANCE
and executes it.

It degrades performance.
If no of travels to DB are increased then performance will be degraded.



demo.p1;

demo.p2;

depmo.p3;

ORACLE DB SERVER

INSTANCE

DB

demo

PACKAGE demo
proc p1
p2
.
p10

PACKAGE reduces no of travels to DB.
So, it improves performance.

Exception Handling

Thursday, November 28, 2024 9:02 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 or semantic mistakes.**

  Example:

  missing ;
  missing END IF
  missing '
  missing)

Run Time Errors:

- **These errors occur at run time [during program execution].**
- **These errors occur due to several reasons like:**
  - **divide with 0**
  - **when we retrieve data if record is not found**
  - **if we insert duplicate value in PK**
  - **wrong input is given**
  - **if check constraint violated**

- **When Run Time Error occurs, program will be terminated in the middle of execution.**

  Problem: Abnormal Termination
  With Abnormal Termination we may loss the data.

That's why we must handle run time errors.

Logical Errors:

- **These errors occur due to mistake in logic.**
- **It leads to wrong results due to mistake in logic.**
- **As a developer, we are responsible to develop correct logic.**

Example:

```
Withdraw =>   balance := balance+amount

                         50000+10000  = 60000
```

Exception Handling:

- **Exception => Run Time Error**
- **The way of handling run time errors is called "Exception Handling".**

- **To handle the run time error we define EXCEPTION block.**

Syntax of Exception Handling:

```
DECLARE
    --declare the variables
BEGIN
    --statements

    EXCEPTION
        WHEN <exception_name> THEN
            --handling code
        WHEN <exception_name> THEN
            --handling code
        .
        .
END;
/
```

Example on 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 / wrong input');
      WHEN others THEN
         dbms_output.put_line('something went wrong');
END;
/
```

NOTE:

"others" can handle any run time error

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 / wrong input
```

Output-4:

```
Enter value for x: 'raju'
Enter value for y: 2
value is out of range / wrong input
```

Types of Exceptions:

2 types:

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

Built-In Exception:
Built-in exceptions are already defined by ORACLE SOFTWARE DEVELOPERS and these will be raised implicitly.

Examples:

zero_divide
value_error
no_data_found
dup_val_on_index
too_many_rows
invalid_cursor
cursor_already_open

User-Defined Exception:
We can define our own exceptions. These are called "User-Defined Exception".

Example:

one_divide
Sunday_not_allow
xyz
raju

zero_divide:
When we try to divide with 0
then zero_divide exception will be raised

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

no_data_found:
when we retrieve data from table if record is not found
then no_data_found exception will be raised.

Example on no_data_found:

Program to display emp record of given empno:

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

   SELECT * INTO r FROM emp WHERE empno=v_empno;

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

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

Output-1:

```
Enter value for empno: 7369
SMITH    13601.35
```

Output-2:

```
Enter value for empno: 1234
no emp existed with this empno
```

dup_val_on_index:

When we insert duplicate value in PRIMARY KEY column, dup_val_on_index exception will be raised.

Example on dup_val_on_index:

Program to insert customer record into customer table:

CUSTOMER

CID	CNAME	CCITY

PK

```
CREATE TABLE customer
(
cid NUMBER(4) CONSTRAINT c300 PRIMARY KEY,
cname VARCHAR2(10),
ccity CHAR(3)
);
```

Program:

```
BEGIN
   INSERT INTO customer VALUES(&cid, '&cname',
   '&ccity');
   COMMIT;
   dbms_output.put_line('record saved..');

   EXCEPTION
      WHEN dup_val_on_index THEN
         dbms_output.put_line('custid already existed..');
END;
/
```

too_many_rows:

When we retrieve the data if select query selects multiple rows, too_many_rows exception will be raised.

Example on too_many_rows:

Program to display the emp records based on given job:

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

    SELECT * INTO r FROM emp WHERE job=v_job;

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

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

```
Output-1:
Enter value for job: PRESIDENT
KING    PRESIDENT   9050

Output-2:
Enter value for job: MANAGER
many rows selected..
```

Invalid_Cursor:

When we try fetch 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 not opened..');
END;
/
```

Output:
cursor not opened..

cursor_already_open:
when we try to open opened cursor,
cursor_already_open exception will be raised

Example on cursor_already_open:

Program to display all emp records:

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

    OPEN c1;

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

    CLOSE c1;

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

Output:

```
cursor already opened..
```

Built-In Exception:	User-Defined Exception:
name is ready	we define a name
it will be raised implicitly	raise it explicitly
just handle it	handle it
1 step:	3 steps:
Handle the Exception	• declare • raise • handle

User-Defined Exception:

We can define our own exceptions. These are called "user-defined exceptions".

For user-defined exception follow 3 steps. They are:

- Declare
- Raise
- Handle

Declaring Exception:

Syntax:

```
<exception_name> EXCEPTION;
```

Examples:

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

using EXCEPTION data type we can declare exception name.

Raising Exception:

Syntax:

```
RAISE <Exception_name>;
```

Examples:

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

Using RAISE keyword we can raise exception.

Handling the Exception:

Syntax:

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

Example:

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

To handle the exception define EXCEPTION block.

declare	one_divide EXCEPTION;
raise	RAISE one_divide;
handle	EXCEPTION WHEN one_divide THEN d_o.p_l('you cannot divide with 1');

Example on user-defined exception:

Program to divide 2 numbers.
if denominator is 0 run time error occurs. handle it.
if denominator is 1 raise run time error 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                                  --handle
        WHEN zero_divide THEN
            dbms_output.put_line('denominator cannot be 0');
        WHEN one_divide THEN
            dbms_output.put_line('denominator cannot be 1');
END;
/
```

NOTE:
we can raise the error using 2 ways. They are:

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

Raise_Application_Error():

- **Raise_Application_Error() procedure is used to raise the error explicitly with our own code and message.**

- **our own error code must be b/w -20000 to -20999.**

Syntax:

```
Raise_Application_Error(<error_code>, <error_message>)
```

Example:

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

Example on raise_application_error():

program to divide 2 numbers.
if denominator 1 then raise the exception using 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:

```
Enter value for x: 20
Enter value for y: 1
ORA-20050: you cannot divide with 1
```

Differences b/w RAISE and RAISE_APPLICATION_ERROR():

RAISE	• **it is a keyword** • **it raises error using name**
RAISE_APPLICATION_ERROR()	• **it is a procedure** • **it raises error using code**

pragma exception_init()

TRIGGERS
COLLECTIONS
working with lobs
dynamic sql

views
indexes
sequences
m.views
synonyms