




Administration base de données ORACLE



Le langage PL/SQL



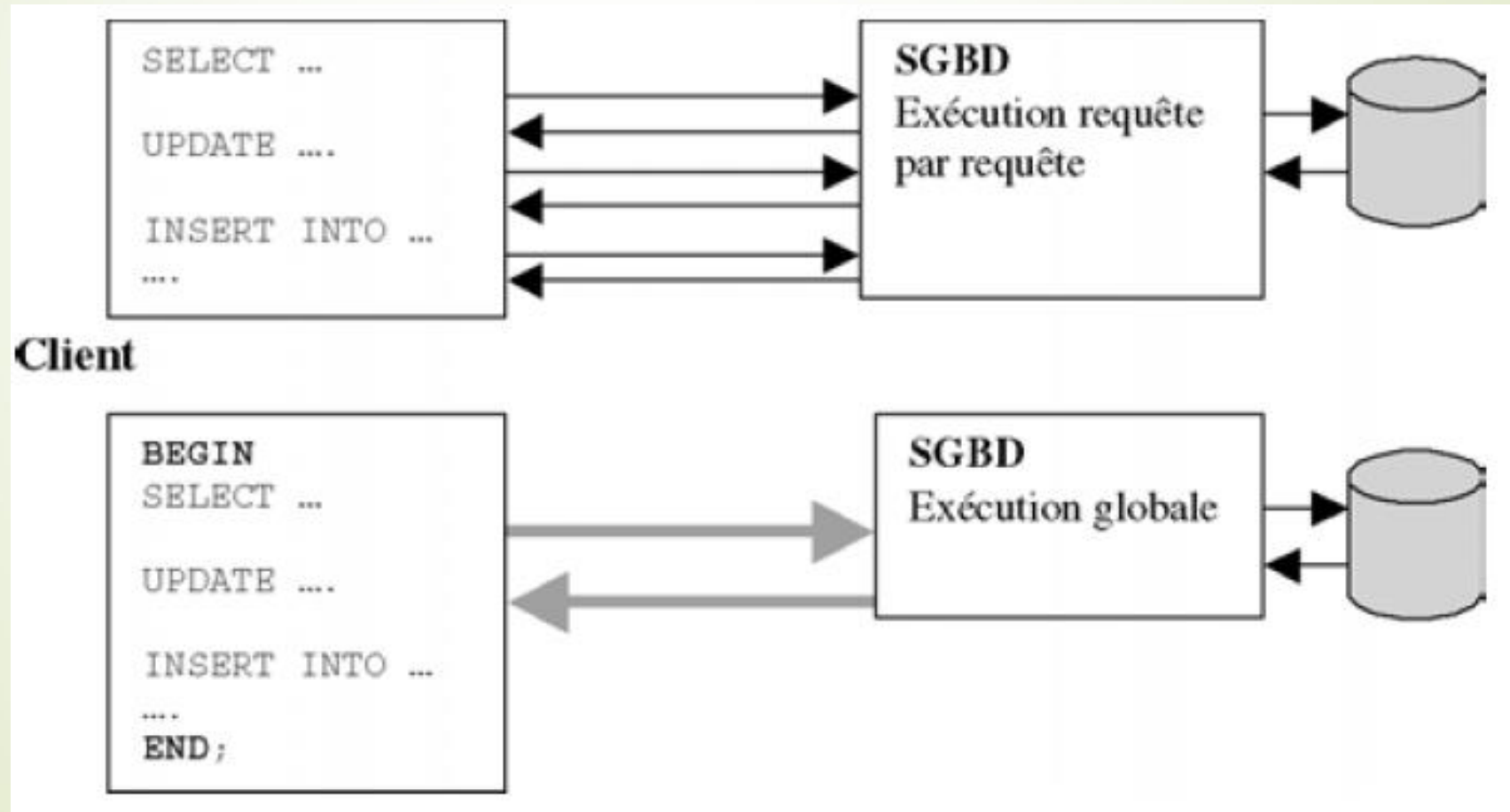
Chapitre 5 – Bases du PL/SQL

Le langage PL/SQL

- PL/SQL (Procedural Language/SQL) est un langage procédural d'Oracle étendant SQL.
- Il permet de combiner les avantages d'un langage de programmation classique (IF, WHILE...) avec les possibilités de manipulation de données offertes par SQL (SELECT, INSERT...)

Le langage PL/SQL

Le langage PL/SQL optimise le trafic sur le réseau



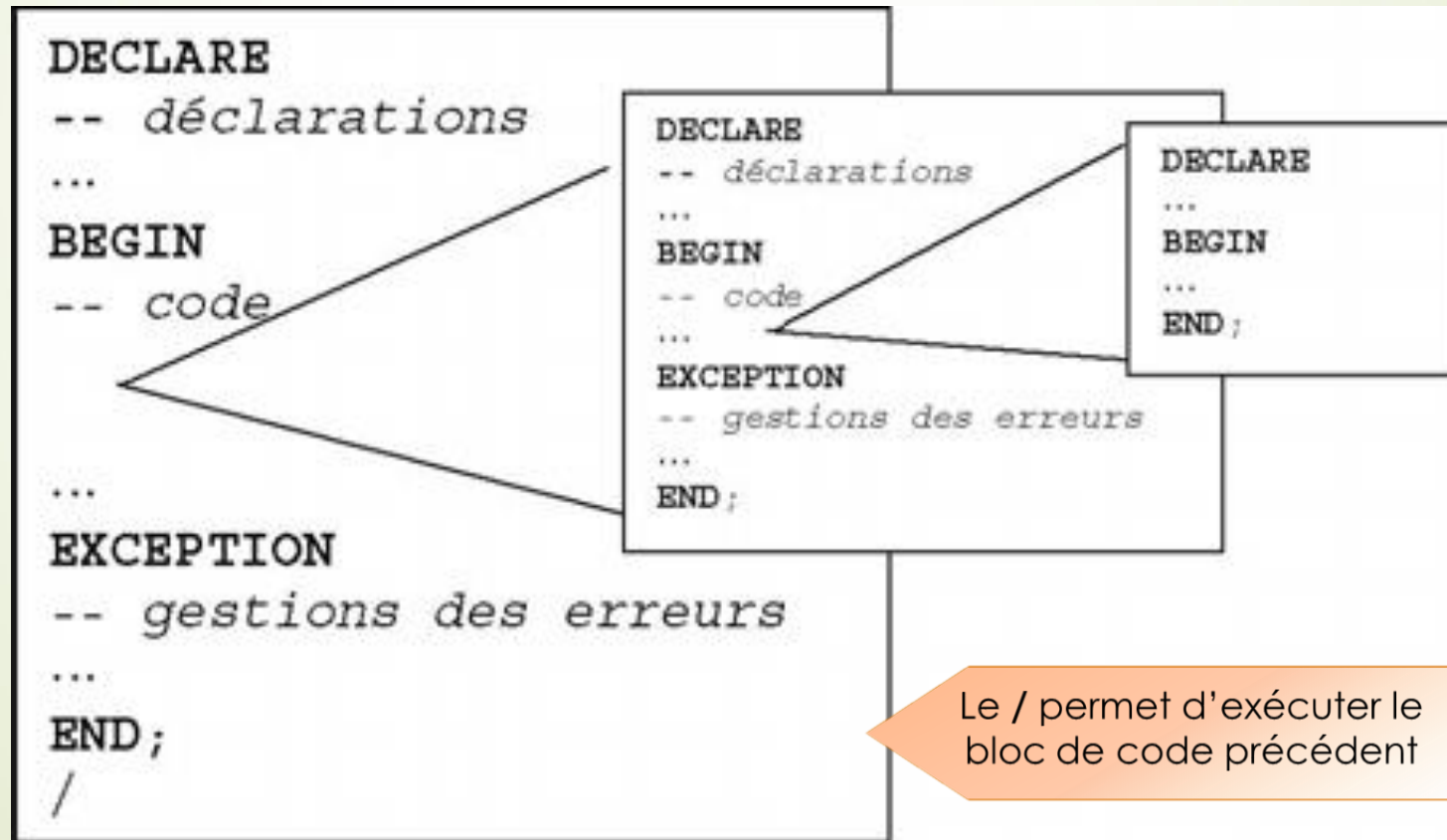
Le langage PL/SQL

Un programme PL/SQL peut être :

- ❑ **Bloc de code** : c'est un sous programme non nommé, et non stocké. Il n'a pas d'arguments et ne retourne aucune valeur.
- ❑ **Fonction ou procédure stockée** : c'est un sous programme nommé qui peut accepter des arguments et qui peut retourner une valeur (cas des fonctions).
- ❑ **Déclencheurs** : représente un sous programme qui invoque des instructions SQL (select,).
- ❑ **Package** : Ensemble de fonctions, procédures, variables, curseurs, déclencheurs, etc. regroupés dans un même fichier.

Structure d'un programme PL/SQL

- Un programme PL/SQL qui n'est pas nommé (**un bloc**) est composé de **3 sections** :





Les instructions PL/SQL

Les instructions PL/SQL se décomposent en :

- Instructions d'affectations;
- Instructions SQL : COMMIT, SELECT, UPDATE...
- Instructions de contrôle, de répétition et d'itération
- Instructions de gestion de curseur
- Instructions de gestion des erreurs

Déclaration des variables

Les variables et les constantes sont déclarées (et **éventuellement initialisées**) dans la section DECLARE.

Syntaxe :

nomVariable Type := valeurInitial ;

Exemple :

```
SQL> DECLARE
```

```
X varchar2(6) ;
```

```
Y constant number :=10 ;
```

```
Z integer :=3 ;
```

```
v BOOLEAN NOT NULL DEFAULT FALSE;
```

Remarque :

Les déclarations multiples comme suit : *i, j, k NUMBER;* ne sont **pas permises**

Déclaration des variables

On trouve 3 types de variables en PL/SQL.

- Un des types utilisés en SQL (number, varchar...).
- Un type particulier au PL/SQL :
 - ✓ Boolean : True, False.
 - ✓ Natural : 0 à 2147438647
 - ✓ Positive : (1 à 2147438647)
- Un type faisant référence à celui d'une colonne d'une table ou une suite de colonnes.
 - ✓ **%type** : le type d'une variable équivaut au type d'un champ.
 - ✓ **%RowType** : le type d'une variable équivaut au type d'un enregistrement.

Exemple

DECLARE

unclient **Client%ROWTYPE**;

--unclient est composée de toutes les colonnes de la table Client.

nomclient **Client.nom%TYPE**;

-- nomclient a le même type que la colonne nom de la table Client

BEGIN

SELECT * INTO **unclient** FROM Client WHERE nom='ALAOUI';

monClient := unclient .nom;

--Accès à des valeurs de l'enregistrement par la notation pointée.

END;

/

Affectation

Affectation procédural

L'affectation est réalisée avec l'opérateur **:=**
nom_variable := valeur;

Affectation SQL

La clause **INTO** associée à SELECT permet d'affecter le résultat de la requête aux variables listées derrière INTO.

Exemple :

```
DECLARE
unclient Client%ROWTYPE;
nomclient Client.nom%TYPE;
BEGIN
SELECT * INTO unclient FROM Client WHERE nom='ALAOUI';
monClient := unclient .nom;
END;
```

Rq : l'instruction DBMS_OUTPUT.PUT_LINE
n'affiche rien sauf si on exécute auparavant :
SET SERVEROUTPUT ON;



Exercice

- Affiche 'DBA' à l'écran
- Regroupez les deux chaînes « ORACLE » et « DBA »



Exercise

```
DECLARE
W_TEMP VARCHAR2(10)
BEGIN
W_TEMP:='DBA';
DBMS_OUTPUT.PUTLINE('W_TEMP');
END;
```

```
DECLARE
W_TEMP VARCHAR2(10);
W_TEMP2 VARCHAR2(10);
BEGIN
W_TEMP:='ORACLE';
W_TEMP2:='DBA';
W_TEMP=W_TEMP||W_TEMP2;
DBMS_OUTPUT.PUTLINE(W_TEMP);
END ;
```

Opérateurs

PL/SQL supporte les opérateurs suivants :

- Arithmétique : + - * /
- Concaténation : ||
- Parenthèses (priorités entre opérations): ()
- Affectation : :=
- Comparison: =, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN
- Logique : AND, OR, NOT

Structures conditionnelles

Syntaxe :

IF condition1 THEN

instructions;

ELSIF condition2 THEN

instructions;

ELSE

instructions;

END IF;

Structures conditionnelles

Exemple :

DECLARE

v_tel CHAR(14) NOT NULL := '06-76-85-14-89';

BEGIN

IF SUBSTR(v_tel,1,2)='06' **THEN**

DBMS_OUTPUT.PUT_LINE ('C'est un portable!');

ELSE

DBMS_OUTPUT.PUT_LINE ('C'est un fixe...');

END IF;

END;

Structure CASE

DECLARE

v_mention CHAR(2);

v_note NUMBER(4,2) := 9.8;

BEGIN

CASE

WHEN v_note >= 16 **THEN** v_mention := 'TB';

WHEN v_note >= 14 **THEN** v_mention := 'B';

WHEN v_note >= 12 **THEN** v_mention := 'AB';

WHEN v_note >= 10 **THEN** v_mention := 'P';

ELSE v_mention := 'R' ; --ELSE est optionnelle

END CASE;

END;

/

Boucles - Structure tant que

```
WHILE condition LOOP  
instructions;  
END LOOP;
```

Exemple :

```
DECLARE  
v_somme NUMBER(4) := 0;  
v_entier NUMBER(3) := 1;  
  
BEGIN  
WHILE (v_entier <= 100) LOOP  
v_somme := v_somme+v_entier;  
v_entier := v_entier + 1;  
END LOOP;  
  
DBMS_OUTPUT.PUT_LINE ('Somme = ' || v_somme);  
END;  
/
```

Boucles - Structure faire-tantque

```
LOOP  
instructions;  
EXIT [WHEN condition;]  
END LOOP;
```

Exemple :

```
DECLARE  
v_somme NUMBER(4) := 0;  
v_entier NUMBER(3) := 1;  
  
BEGIN  
LOOP  
v_somme := v_somme+v_entier;  
v_entier := v_entier + 1;  
  
EXIT WHEN v_entier > 100;  
END LOOP;  
  
DBMS_OUTPUT.PUT_LINE('Somme = ' || v_somme);  
END;
```

Boucles - Structure for

```
FOR compteur IN valeurInf..valeurSup LOOP  
instructions;  
END LOOP;
```

Exemple :

```
DECLARE  
v_somme NUMBER(4) := 0;  
  
BEGIN  
FOR v_entier IN 1..100 LOOP  
v_somme := v_somme+v_entier;  
END LOOP;
```

```
DBMS_OUTPUT.PUT_LINE ('Somme = ' || v_somme);  
END;
```



Exercices

- Affiche les nombres de 1 à 15 avec la boucle while, loop et for

Exercices

- Affiche les nombres de 1 à 15 avec la boucle while, loop et for

```
DECLARE
W_TEMP VARCHAR2(20) ;
BEGIN
FOR I IN 1..15 LOOP
W_TEMP:=i ;
DBMS_OUTPUT.PUT_LINE(W_TEMP);
END LOOP;
END;
```



Programmation avancée



Curseurs

- Les requêtes renvoient très souvent un **nombre important** et non prévisible **de lignes**.
- **Un curseur est une variable dynamique** qui prend pour valeur le résultat d'une **requête multi-lignes**.
- **Un curseur est une zone mémoire** qui permet de traiter **individuellement chaque ligne renvoyée** par un SELECT.

Déclaration d'un curseur

Syntaxe :

`CURSOR monCurseur IS requête ;`

Exemple :

`CURSOR clientCur IS SELECT * FROM Client;`

- Le curseur de nom clientCur est chargé dans cet exemple de récupérer le résultat de la requête qui suit.
- Il peut alors être ouvert lorsqu'on souhaite l'utiliser (dans le corps d'un bloc).

Utilisation d'un curseur

❑ **OPEN nomCurseur;**

Ouverture du curseur (chargement des lignes). Aucune exception n'est levée si la requête ne ramène **aucune** ligne.

Exemple : OPEN cur;

❑ **FETCH nomCurseur INTO listeVariables ;**

Chargement de l'enregistrement courant dans une ou plusieurs variables. Et positionnement sur la ligne suivante.

Exemple : FETCH cur **INTO** var1,var2,var3;

❑ **CLOSE nomCurseur;**

Ferme le curseur. L'exception **INVALID_CURSOR** se déclenche si des accès au curseur sont opérés après sa fermeture.

Exemple : CLOSE cur;

Utilisation d'un curseur

❑ *nomCurseur*%**ISOPEN**

Retourne TRUE si le curseur est ouvert, FALSE sinon.

Exemple : IF zone1%**ISOPEN** THEN ...

❑ *nomCurseur*%**NOTFOUND**

Retourne TRUE si le dernier FETCH n'a pas renvoyé de ligne (fin de curseur).

Exemple : EXIT WHEN zone1%**NOTFOUND**;

❑ *nomCurseur*%**FOUND**

Retourne TRUE si le dernier FETCH a renvoyé une ligne.

Exemple : WHILE (zone1%**FOUND**) LOOP

❑ *nomCurseur*%**ROWCOUNT**

Retourne le nombre total de lignes traitées jusqu'à présent.

Principe d'un curseur

```
DECLARE  
CURSOR clientCur IS SELECT * FROM Client;  
varclient Client%ROWTYPE;
```

```
BEGIN  
OPEN clientCur ;  
FETCH clientCur INTO varclient;  
CLOSE clientCur ;  
END;  
/
```

clientCur *curseur*

C1	Smith	31/10/03
C2	Jones	30/11/03
C3	Blake	30/11/03
C4	Clark	31/12/03
C5	Adams	31/12/03

C1	Smith	31/10/03
----	-------	----------

Parcours d'un curseur

- La commande FETCH ne ramène qu'une **seule ligne**, il faut donc la mettre dans **une boucle** pour pouvoir **parcourir** toutes les lignes.
- La sortie d'une boucle de parcours de curseur se fait à l'aide de la condition composée :

EXIT WHEN **nomCurseur%NOTFOUND**
OR nomCurseur%NOTFOUND IS NULL.

Parcours d'un curseur : LOOP

sélectionner l'ensemble des employés dont le salaire ne dépasse pas 6000 et les augmenter de 500:

DECLARE

CURSOR SalCur IS

SELECT * FROM EMP WHERE SAL<6000.00;

employe EMP%ROWTYPE;

BEGIN

OPEN SalCur;

LOOP

FETCH SalCur INTO employe;

EXIT WHEN SalCur%NOTFOUND;

UPDATE EMP

SET SAL=SAL+500.00 WHERE EMPNO=employe.empno;

END LOOP;

CLOSE SalCur ;

END;

/

Parcours simplifié : FOR

DECLARE

CURSOR SalCur IS

SELECT * FROM EMP WHERE SAL<6000.00;

employe EMP%ROWTYPE;

BEGIN

FOR employe IN SalCur LOOP

UPDATE EMP

SET SAL=6500.00 WHERE EMPNO=employe.empno;

END LOOP;

END;

/



Exercice : Curseur

1. Lecture table emp et affichage du premier enregistrement trouvé

Exercice : Curseur

1. Lecture table emp et affichage du premier enregistrement trouvé

```
DECLARE
CURSOR CTP IS SELECT ename FROM emp;
W_LIBELLE VARCHAR(30);
BEGIN
OPEN CTP;
FETCH CTP INTO W_LIBELLE;
CLOSE CTP;
DBMS_OUTPUT.PUT_LINE(W_LIBELLE);
END;
```

Les déclencheurs "triggers"

- Les **déclencheurs** ou "**triggers**" sont des séquences d'actions définies par le programmeur qui se **déclenchent**, non pas sur un appel, mais directement **quand un événement** particulier (spécifié lors de la définition du trigger) sur une ou plusieurs tables **se produit**.
- Un trigger sera un objet stocké (comme une table ou une procédure)

Les déclencheurs “triggers”

La syntaxe :

```
CREATE [OR REPLACE] TRIGGER montrigger  
{BEFORE | AFTER}  
{INSERT | DELETE | UPDATE}  
ON ma table  
[FOR EACH ROW [WHEN (<condition>)]]  
<corps du trigger>
```

Les déclencheurs “triggers”

CREATE OR REPLACE

Recréer le déclencheur s'il existe déjà.

BEFORE | AFTER

Avant ou après l'événement déclenchant le trigger, qui peut être une insertion, destruction ou mise à jour (**INSERT | DELETE | UPDATE**) sur une table.

L'option **FOR EACH ROW [WHEN (condition)]** fait s'exécuter le trigger à chaque modification d'une ligne de la table spécifiée (on dit que le trigger est de "niveau ligne").

En l'absence de cette option, le trigger est exécuté une seule fois ("niveau table").

<corps du trigger> représente un bloc de code à exécuter au déclenchement d'un trigger.

Exemple

```
CREATE OR REPLACE TRIGGER declench  
AFTER DELETE  
ON emprunt  
FOR EACH ROW  
BEGIN  
  dbms_output.put_line('ligne supprimé');  
END;  
/  
DELETE FROM emprunt;
```

Remarque : Le message « ligne supprimé » s'affichera autant de fois qu'il ya de ligne dans la table. En absence de FOR EACH ROW le message sera affiché **une seule fois**.



Sous-programmes

- ❑ Dans le vocabulaire des bases de données, on appelle les sous-programmes fonctions ou procédures **stockées**, car ils sont compilés et **résident dans la base** de données.
- ❑ Il est possible de retrouver leur code au niveau du **dictionnaire de données**. Le sous-programme peut être ainsi partagé dans un contexte multi-utilisateurs.



Sous-programmes

Les sous-programmes sont des blocs PL/SQL **nommés** et capables d'inclure des **paramètres** en entrée et en sortie.

Il existe deux types de sous-programmes PL/SQL :

- **les procédures** qui réalisent des actions.
- **les fonctions** qui retournent un unique résultat.

Procédure stockée : Syntaxe

```
CREATE [OR REPLACE] PROCEDURE nom_procedure  
  [ (param1 IN typeSQL [,parametre2 OUT typeSQL ...]) ]  
IS ou AS  
  [section_declaration]  
BEGIN  
  section_executable  
END [nom_procedure];
```

IN désigne un paramètre d'entrée (lecture seule),

OUT un paramètre de sortie (écriture seule),

IN OUT un paramètre d'entrée et de sortie (lecture et écriture) .

Il est possible d'initialiser chaque paramètre par une valeur DEFAULT.

Procédure stockée : Exemple

```
CREATE OR REPLACE PROCEDURE client_details
IS
    CURSOR cli_cur IS
    SELECT * FROM client;
    un_cli cli_cur%rowtype;
BEGIN
    FOR un_cli in cli_cur
    LOOP
        dbms_output.put_line(un_cli.nom || ' ' || un_cli.date_fin_ab);
    END LOOP;
END;
/
```

Exécution

sous SQL*Plus : **EXECUTE** [ou EXEC] **client_details** ;
Appel dans un programme PL/SQL : **client_details** ;

Procédure stockée : Avec paramètres

```
CREATE PROCEDURE essai
(x IN INTEGER,
y OUT INTEGER,
z IN OUT INTEGER)
IS
BEGIN
y:=x*z;
z:=z*z;
dbms_output.put_line(x || ' ' || y || ' ' || z);
END;
```

```
DECLARE
a INTEGER;
b INTEGER;
BEGIN
a:=5;
b:=10;
dbms_output.put_line( a || ' ' ||b);
essai(2,a,b);
dbms_output.put_line(a || ' ' ||b);
END;
```



Exercice : Procédure

1. Recherchez le nom et le job d'un employée passé en paramètre
- 

Exercice : Procédure

1. Recherchez le nom et le job d'un employée passé en paramètre

```
CREATE OR REPLACE PROCEDURE P_INFOS  
(P_Num in number, P_Nom out varchar2, P_job out varchar2) is  
CURSOR C IS  
SELECT ename, job FROM emp WHERE EMPNO =P_NUM;  
BEGIN  
OPEN C;  
FETCH C INTO P_Nom, P_job;  
CLOSE C;  
END P_INFOS;
```

Fonction stockée : Syntaxe

```
CREATE [OR REPLACE] FUNCTION nom_fonction  
    [ (parametre1 IN | OUT typeSQL [,parametre2...]) ]
```

```
RETURN typeSQL
```

```
IS ou AS
```

```
    [section_declaration]
```

```
BEGIN
```

```
    section_executable
```

```
        RETURN variable;
```

```
END [nom_fonction];
```

Fonction stockée : Exemple

```
CREATE OR REPLACE FUNCTION client_details_func  
RETURN client.nom%type AS  
nom_client client.nom%type;  
BEGIN  
SELECT nom INTO nom_client  
FROM client WHERE id_client = 'C1';  
RETURN nom_client;  
END;  
/
```

Création de la fonction

Appel de la fonction

```
DECLARE  
rst client.nom%type ;  
BEGIN  
rst:=client_details_func;  
dbms_output.put_line(  
'resultat : ' || rst );  
END;  
/
```



Exercice : Fonctions

1. Fonction ajout de deux nombres



Exercice : Fonctions

1. Fonction ajout de deux nombres

```
CREATE OR REPLACE FUNCTION F_AJOUT (P1 NUMBER, P2 NUMBER) RETURN  
NUMBER IS  
CURSOR C_SOMME IS SELECT P1+P2 FROM DUAL;  
W_SOMME NUMBER;  
BEGIN  
OPEN C_SOMME;  
FETCH C_SOMME INTO W_SOMME;  
DBMS_OUTPUT.PUT_LINE(W_SOMME);  
CLOSE C_SOMME;  
RETURN W_SOMME;  
END;
```

```
SELECT F_AJOUT(3,4) FROM DUAL;  
  
Ou  
  
DECLARE  
W_SOMME NUMBER:=0;  
BEGIN  
W_SOMME:=F_AJOUT(3,5);  
DBMS_OUTPUT.PUT_LINE(W_SOMME) ;  
END;
```


Notes

- ❑ Si la procédure ou la fonction est exécutée avec erreurs, utiliser **SHOW ERRORS** pour les afficher.
- ❑ Eliminer les types de paramètres qui ont des parenthèses, Exemple :
VARCHAR(40) à remplacer par STRING
NUMBER à remplacer par INTEGER ou REAL
- ❑ Dans une procédure, comme dans une fonction, il n'existe **pas de mot DECLARE.**