

# Chapitre I

## PL/SQL

### Partie 2

# LES CURSEURS

- Ce sont des zones mémoires SQL privé permettant d'accéder aux informations qui y sont stockées lors de l'exécution d'une instruction SQL.
- C'est une sorte de pointeur servant à parcourir une série ordonnées de lignes pointant successivement vers chacune d'entre elle pour en fournir les adresses individuelles.

# LES CURSEURS

- Il existe deux types de pointeurs :

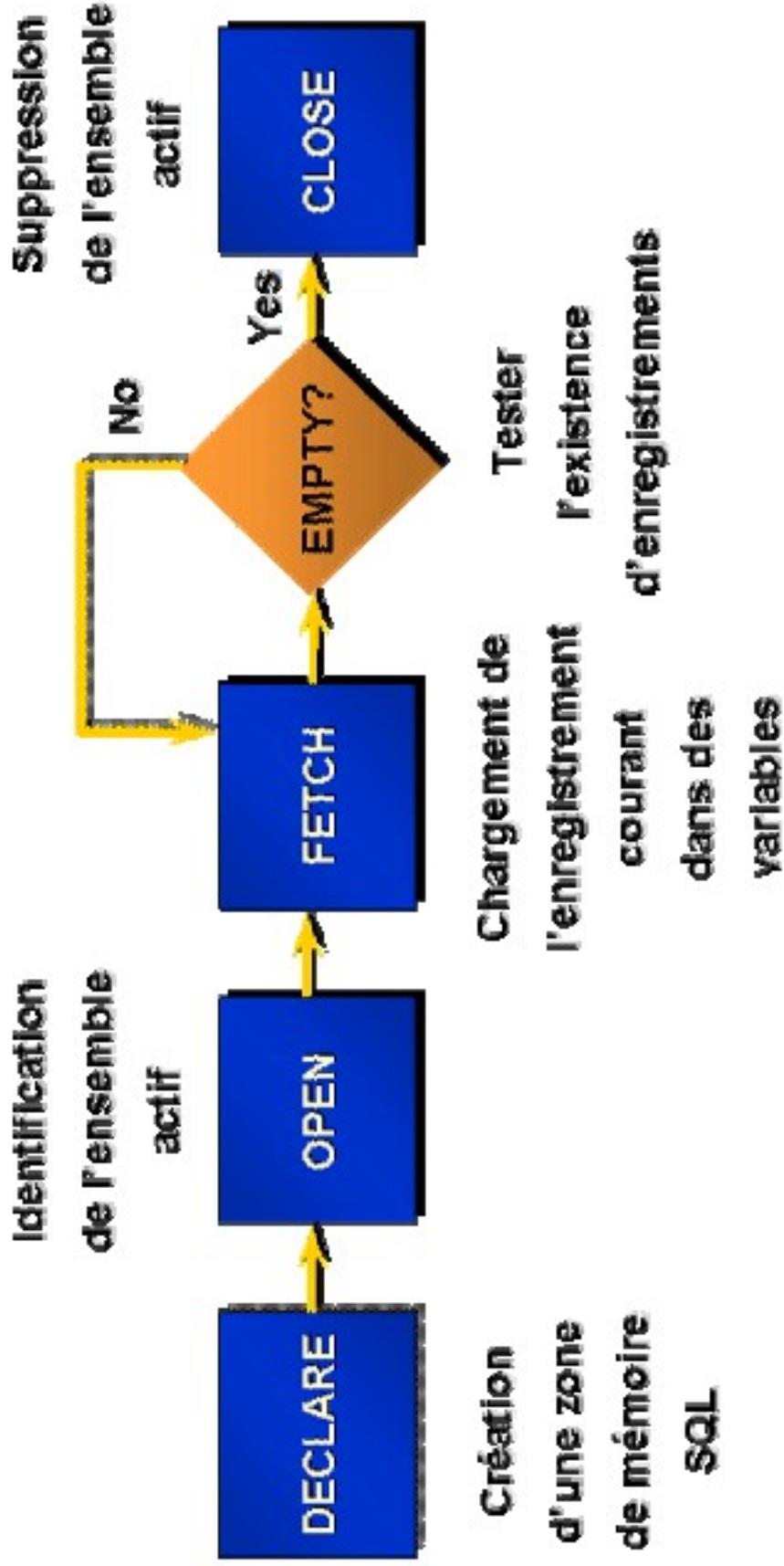
## 1. Curseur implicite :

Il est généré automatiquement par le noyau pour toute requête SQL non déclarée dans un curseur explicite.

## 2. Curseur explicite :

Il est créée et géré par l'utilisateur pour traité un ordre SELECT qui ramène plusieurs lignes. Le traitement du résultat pourra ce faire alors ligne par ligne.

# Étape d'utilisation d'un curseur explicite



# Les curseurs explicite

1. Déclaration du curseur  
La déclaration / définition du curseur permet juste d'affecter le nom du curseur à l'ordre SELECT, la requête n'est pas exécuter immédiatement.

Syntaxe:

**CURSOR nom\_curseur  
IS commande SELECT;**

# Exemple

Étant donnée la base de donnée suivante :

EMP (EMPNO, ENAME, JOB, #MGR, HIREDATE, SAL, COMM, #DEPTNO)  
DEPT (DEPTNO, DNAME, LOC)

```
DECLARE
    --premier curseur
    CURSOR emp_cursor
    IS SELECT empno, sal * 10 FROM emp;

    CURSOR dept_cursor
    IS SELECT * FROM dept WHERE deptno = 10 ;

BEGIN
    ...
END ;
```

# Les curseurs explicites

## 2. Ouverture du curseur :

- Allocation mémoire du curseur
- Analyse syntaxique et sémantique de la requête.
- positionne le pointeur juste avant la première ligne du curseur.

Remarque:

Un curseur ne peut être ouvert que s'il ne l'a jamais été ouvert ou ouvert puis fermé.

# Les curseurs explicites

## Syntaxe :

```
OPEN nom_cursor;
```

## Exemple:

```
OPEN emp_cursor;  
OPEN dept_cursor;
```

# Les curseurs explicites

3. Traitement des lignes ramenées
  - les lignes ramenées par le curseur doivent être traitées séquentiellement.
  - chaque ligne doit renseigner un ensemble de variables réceptrices définies dans la partie DECLARE du bloc.
  - Le nombre de ces variables doit être identique aux nombres de champ dans la ligne ramenée et avoir le même type.

# Les curseurs explicites

L'ordre **FETCH** permet d'extraire la ligne suivante dans le curseur et déplace le pointeur séquentiellement d'une seule position vers l'avant.

**FETCH nom\_curseur INTO {nom\_variable [,  
nom\_variable]...|nom-record}**

# Exemple

```
DECLARE
  CURSOR dept10 IS
    SELECT ename, sal FROM emp WHERE dept=10;
    nom emp.ename%TYPE,
    salaire emp.sal%TYPE;

BEGIN
  OPEN dept10;
  LOOP
    FETCH dept10 INTO nom, salaire;
    /*traitement des lignes*/
  END LOOP;
  ...
END;
```

# Les curseurs explicites

## 4. Fermeture du curseur :

- Désactivation du curseur.
- libération des ressources.

Syntaxe :  
**CLOSE nom curseur;**

Exemple :  
**CLOSE dept10;**

# Exemple

```
DECLARE
  CURSOR dept10 IS
    SELECT ename, sal FROM emp WHERE dept=10;
    nom emp.ename%TYPE,
    salaire emp.sal%TYPE;

BEGIN
  OPEN dept10;
  LOOP
    FETCH dept10 INTO nom, salaire;
    /*traitement des lignes*/
  END LOOP;
  ...
  CLOSE dept10;
END;
```

# Attributs des curseurs explicites

1. L'attribut %ISOPEN :
  - Attribut booléen : TRUE si le curseur est ouvert, FALSE sinon.

Exemple :  
**BEGIN**

...  
**IF NOT dept10%ISOPEN THEN**  
  **OPEN dept10;**  
**END IF;**  
...  
**END;**

# Attributs des curseurs explicites

2. L'attribut %FOUND :
  - Attribut booléen : TRUE si l'ordre FETCH ramène au moins une ligne.

Exemple :  
**BEGIN**

```
...  
OPEN dept10;  
WHILE dept10%FOUND LOOP  
  FETCH dept10 INTO nom, salaire;  
END LOOP;  
...  
END;
```

# Attributs des curseurs explicites

3. L'attribut %NOTFOUND :
  - Attribut booléen : TRUE si le curseur ne ramène plus de ligne.

Exemple :

```
BEGIN
```

```
...  
OPEN dept10;  
LOOP  
  FETCH dept10 INTO nom, salaire;  
  EXIT WHEN dept10%NOTFOUND OR dept10%NOTFOUND IS NULL;  
END LOOP;  
...  
END;
```

# Attributs des curseurs explicites

4. L'attribut %ROWCOUNT :
  - Attribut numérique : cette attribut est incrémenté à chaque ordre FETCH. Il traduit la nième ligne traitée.

Exemple :

```
BEGIN
```

```
...  
OPEN dept10;  
LOOP  
  FETCH dept10 INTO nom, salaire;  
  EXIT WHEN dept10%ROWCOUNT > 5;  
END LOOP;  
...  
END;
```

# Utilisation avancée des curseurs

## L'attribut %ROWTYPE :

- Permet la déclaration implicite d'une structure dont les éléments sont d'un type identique aux colonnes ramenées par le curseur.

### Exemple :

```
DECLARE
CURSOR C1 IS SELECT sal, ename, empno FROM emp;
var C1%ROWTYPE;
BEGIN
OPEN C1;
LOOP
FECTH C1 INTO var;
EXIT WHEN C1%NOTFOUND;
INSERT INTO temp (A,B,C) VALUES (var.empno, var.ename, var.sal);
END LOOP;
END;
```

# Utilisation avancée des curseurs

## *Les boucles et les curseurs*

L'objectif est de fournir au programmeur une structure simple et efficace pour utiliser les structures de boucle et les curseurs.

Syntaxe :

```
FOR nom_record IN nom curseur LOOP  
    --les instructions  
END LOOP;
```

La boucle ouvre et ferme automatiquement le curseur.

Pas besoin de déclaré nom\_record.

# Utilisation avancée des curseurs

Exemple :

```
DECLARE  
CURSOR emp_cursor IS  
SELECT ename, deptno FROM emp;  
nom emp.ename%TYPE;  
dept emp.deptno%TYPE;  
BEGIN  
OPEN emp_cursor;  
WHILE (emp_cursor%FOUND) LOOP  
FETCH emp_cursor INTO nom,dept;  
END LOOP;  
CLOSE emp_cursor;  
END;
```

# Utilisation avancée des curseurs

## *Curseur Paramétré :*

Il permet d'utiliser des variables dans le curseur.  
Principalement dans la clause WHERE.

### Syntaxe:

```
CURSOR nom_curseur  
[(nom_parametre type [, nom_parametre type,...])]  
IS commande SELECT;
```

**But :** utilisation du même curseur dans différentes situations pour extraire différentes informations.

# Utilisation avancée des curseurs

*Exemple :*

```
DECLARE CURSOR C ( d NUMBER) IS  
SELECT salaire FROM emp WHERE deptno= d;  
BEGIN OPEN C(20);  
--Traitement  
END ;
```

*Ou*

```
DECLARE CURSOR C ( d NUMBER) IS  
SELECT salaire FROM emp WHERE deptno= d;  
BEGIN FOR x IN C(20) LOOP  
--Traitement  
END LOOP;  
END .
```

# Exercices

## Utiliser à chaque fois les curseurs

### Exercice 1

Ecrire un programme PL/SQL qui calcule la moyenne des salaires des employés dont l'âge est entre 30 et 40 ans.

### Exercice 2

Ecrire un bloc PL/SQL qui recherche tous les employés dont les revenus mensuels (salaire + commission) sont > 20000 et les insérer dans une table TEMP que vous créerez auparavant.

# Les procédures et les fonctions

# Les procédures

- Une procédure est un bloc PL/SQL nommé qui peut prendre des paramètres / arguments et être invoqué.
- La procédure facilite la réutilisation et la manipulation de code car une fois enregistrée, une procédure peut être utilisée par plusieurs autres applications.

# Syntaxe de création d'une procédure

```
CREATE [OR REPLACE] PROCEDURE nom_procédure
(paramètre1 [mode1] datatype1 /DEFAULT valeur1,
 paramètre2 [mode2] datatype2 /DEFAULT valeur2,...)
```

**IS/AS**

*Bloc PL/SQL*

- Le bloc PL/SQL commence par un BEGIN ou la déclaration de variables locales et ce termine par END ou END *nom\_procédure*.
- L'option REPLACE indique que si la procédure existe déjà, elle sera supprimée et remplacée par la nouvelle.

## *Les modes de paramètre de la procédure*

- Les paramètres de la procédure permettent de transférer des valeurs depuis et vers l'environnement appelant.
- On trouve 3 modes :

Type de paramètre	Description
IN (par défaut)	Une valeur constante est passée de l'environnement appelant vers la procédure
OUT	Une valeur est passée de la procédure vers l'environnement appelant
IN OUT	Une valeur constante est passée de l'environnement appelant vers la procédure et une valeur différente peut être renvoyée à l'environnement en utilisant le même paramètre.

# Exemple avec paramètre IN

```
CREATE OR REPLACE PROCEDURE augmentation_salaire (id  
IN emp.empno%TYPE, coef IN NUMBER)  
IS  
BEGIN  
    UPDATE emp SET sal = sal * coef  
    WHERE empno = id;  
END augmentation_salaire;  
➔ Procedure created
```

Pour exécuter cette procédure il suffit de faire :

```
EXECUTE augmentation_salaire (7369, 1.5);  
➔ PL/SQL procedure successfully completed.
```

# *Exemple avec paramètre OUT*

```
CREATE OR REPLACE PROCEDURE
    out_procedure
    (id      IN emp.empno%TYPE,
     nom      OUT emp.ename%TYPE,
     salaire  OUT emp.sal%TYPE)
IS
BEGIN
    SELECT ename, sal INTO nom, salaire FROM
        emp
    WHERE empno = id;
END out_procedure;
```

# Exemple avec paramètre OUT

- Pour qu'une procédure avec des paramètres OUT fonctionne il faut déclarer autant de variables hôtes que de valeurs renvoyées.
- Ensuite après la déclaration on fait passer ces paramètres à la procédure précédés par « : »
- Exécution de la procédure
  - **EXECUTE out\_procedure(7369, :v\_nom, :v\_sal)**
  - ➔ Procédure PL/SQL terminée avec succès

Pour afficher ces valeurs on utilise la commande  
PRINT

- PRINT v\_nom v\_sal

*Exemple avec paramètre IN OUT*

```
CREATE OR REPLACE PROCEDURE format_tel
(no_tel IN OUT VARCHAR2)
IS
BEGIN
    v_phone_no := '(' || SUBSTR(no_tel,1,5)
    ||')'||SUBSTR(no_tel,6,2) ||'-'||SUBSTR(no_tel,8,6);
END format_tel;/
```

# Exemple avec paramètre IN OUT

- Pour exécuter cette procédure il faut créer et initialisé une variable hôte.
  - VARIABLE tel VARCHAR2(16)
  - BEGIN :tel := '0021622232425'; END; /
  - ➔ Procédure PL/SQL terminée avec succès.
- Exécution de la procédure
  - EXECUTE format\_tel('.tel')
  - ➔ Procédure PL/SQL terminée avec succès.
- Affichage du résultat
  - PRINT tel
  - ➔ (00216)22-232425

# Passage de paramètre

On rappelle la déclaration de paramètre :

Nom\_variable [IN | OUT | IN OUT] type\_de\_donnée [DEFAULT valeur]

Trois façon de passer les paramètres :

Méthode	Description
Par position	Les valeurs sont listées dans l'ordre dans lequel sont déclarés les paramètres.
Par association de nom	Les valeurs sont listées dans un ordre arbitraire en associant chacune avec le nom de paramètre correspondant en utilisant une syntaxe spéciale (=>).
Par combinaison	C'est une combinaison des deux méthodes précédentes : les premières valeurs sont listées par position et le reste utilise la syntaxe spéciale de la méthode par association de nom.

# Exemple

```
CREATE OR REPLACE PROCEDURE add_departement(
    num IN NUMBER DEFAULT 0,
    nom IN dept.dname%TYPE DEFAULT 'inconnu',
    loc IN dept.loc%TYPE DEFAULT 'inconnu')
IS
BEGIN
    INSERT INTO dept VALUES (num, nom, loc);
END add_departement;
```

/

# Exemple

```
BEGIN  
    add_departement;  
    add_departement(50, 'MATH', 'TUNIS');  
    add_departement(num=>60, loc=>'SOUSSE', nom=>'INFO');  
    add_departement(70, loc=>'ISSAT');  
end;  
/  
    → Procédure PL/SQL terminée avec succès
```

DEPTNO	DNAME	LOC
...	inconnu	inconnu
0	MATH	TUNIS
50	INFO	SOUSSE
60	inconnu	ISSAT
70		

# Appel de procédure dans un bloc anonyme

```
DECLARE
CURSOR cur IS SELECT empno from emp;
ma_var emp.empno%TYPE;
BEGIN
open cur;
while cur%found loop
fetch cur into ma_var;
augmentation_salaire(ma_var,1.2);
end loop;
close cur;
END;
/
```

# Appel de procédure à partir d'une procédure

```
CREATE OR REPLACE PROCEDURE aug(ident emp.empno%type)
IS
BEGIN
    UPDATE emp SET sal = sal * 2 WHERE empno = ident;
    COMMIT;
END aug;

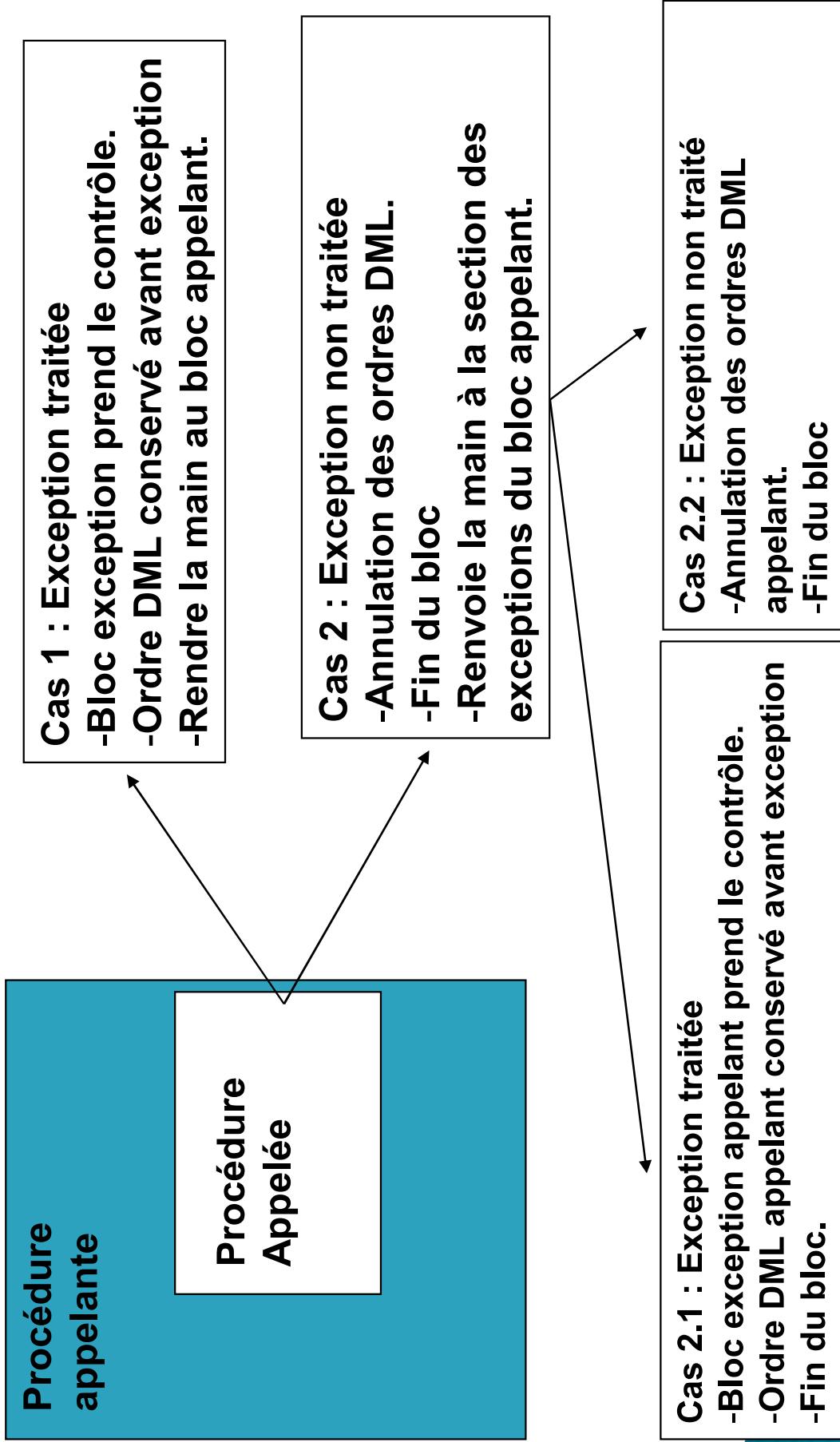
CREATE OR REPLACE PROCEDURE aug2
IS
CURSOR c1 IS SELECT empno FROM emp;
ma_var emp.empno%type;
BEGIN
OPEN c1;
WHILE (c1%found) LOOP
    FETCH c1 INTO ma_var;
    aug(ma_var);
END loop;
COMMIT;
END aug2;
```

# Autre Exemple

```
create or replace procedure aug3(ident emp.deptno%type)
is
cursor c1 is select sal, empno from emp where deptno=ident;
begin
for c1_rec in c1 loop
    UPDATE emp SET sal = sal * 1.01 WHERE empno =
c1_rec.empno;
end loop;
commit;
end aug3;

select sal from emp where deptno=20;
execute aug3(20);
select sal from emp where deptno=20;
```

# Gestion des exceptions



# Suppression de procédure

La commande pour supprimer une procédure est :

**DROP PROCEDURE *nom\_procedure***

# Les fonctions

- Une fonction est un bloc PL/SQL nommé pouvant accepter des paramètres et être appelée de la même façon que les procédures.
- Une fonction retourne une valeur.
- Elles sont constituées d'une :
  - Entête
  - Section déclarative
  - Section exécutable
  - Section gestion des erreurs (optionnelle)
- Une fois validée, elles sont stockées dans la BD comme étant des variables de base de donnée pour être utilisées par d'autres applications.

# Syntaxe d'une fonction

```
CREATE [OR REPLACE] FUNCTION nom_fonction
( paramètre1 [mode1] datatype1,
paramètre2 [mode2] datatype2,...)
RETURNS datatype
```

IS/AS  
PL/SQL Block,

→ Seule le mode IN peut être déclaré dans le mode des paramètres.

# Syntaxe d'une fonction

- **RETURN datatype** : type de donnée retourné par la fonction.
- Le bloc PL/SQL commence soit par une section déclarative de variable locale soit par BEGIN et ce termine par END ou END *nom\_fonction*.
- Il doit obligatoirement y avoir au moins une expression RETURN (variable).

# Exemple

```
CREATE OR REPLACE FUNCTION salaire
( matricule IN emp.empno%TYPE )
RETURN NUMBER
IS
    var_sal emp.sal%TYPE:=0;
BEGIN
    SELECT sal INTO var_sal FROM emp WHERE empno=matricule;
    RETURN (var_sal);
EXCEPTION
    WHEN no_data_found THEN dbms_output.put_line('Pas de
        donnée trouvé');
END salaire;
/
```

- Cette fonction retourne le salaire de l'employé dont le numéro est passé comme paramètre.

# Exécution d'une fonction

- ▶ Compiler la fonction
  - **START f1 .sql**  
-- f1.sql est le fichier script contenant la fonction.
- ▶ Il faut créer une variable hôte pour stocker la valeur renournée.
  - **VARIABLE sal NUMBER**
- ▶ Exécution de la fonction
  - **EXECUTE :sal:=salaire(7902)**
- ▶ Affichage du résultat :
  - **PRINT sal**

# Utilisation des fonctions

- Les fonctions PL/SQL peuvent être appelées depuis toute expression SQL dans laquelle on peut utiliser une fonction pré définie :
  - SELECT.
  - Condition de la clause WHERE ou HAVING.
  - Dans les clauses GROUP BY, ORDER BY.
  - Dans la clause VALUES d'un ordre INSERT.
  - Dans la clause SET d'un ordre UPDATE.

# Exemples

```
CREATE OR REPLACE FUNCTION tax
(salaire IN emp.salaire%TYPE )
RETURN NUMBER
IS
BEGIN
RETURN (salaire*0.4);
END tax;

→ SELECT empno, sal, tax(sal) FROM emp;
```

# Exemples

- Trouver les employés ayant le même salaire que l'employé numéro 7934.
  - `SELECT empno, sal, salaire(7934) AS salaire_ali FROM emp WHERE sal=salaire(7934);`
- Insertion d'un nouveau employeur numéro 8080 ayant le même salaire que 7934.
  - `INSERT INTO emp (empno, sal) VALUES (8080, salaire(7934))`

...

# Package DBMS\_OUTPUT

- ▶ il permet aux développeurs de suivre précisément le déroulement d'une fonction ou d'une procédure en envoyant des messages et valeurs au buffer de sortie.
- ▶ C'est une aide précieuse pour le débogage car il permet de suivre les résultats intermédiaires lors de l'exécution.
- ▶ Si on utilise ce package sous SQL\*Plus, il faut s'assurer que l'affichage sur le terminal de sortie (SERVEROUTPUT) est bien défini à ON.  
**SET SERVEROUTPUT ON**

# Exemple :

```
CREATE OR REPLACE FUNCTION moy_h_vol (x_codetype IN
appareil.codetype%TYPE)
RETURN NUMBER
IS
nbhvol_avg NUMBER (8, 2) := 0 /*valeur résultat*/
BEGIN
DBMS_OUTPUT.PUT_LINE ('type avion transmis' || x_codetype);
SELECT AVG(nbhvol)
INTO nbhvol_avg
FROM avion
WHERE type = x_codetype;
DBMS_OUTPUT.PUT_LINE ('valeur calculée' || nbhvol_avg);
RETURN (nbhvol_avg);
END moy_h_vol;
```

# Suppression d'une fonction

- ▶ Pour supprimer une fonction la commande est :

**DROP FUNCTION *nom\_fonction***

# Récapitulatif

- ▶ Les procédures sont créées pour stocker une série d'actions à exécuter ultérieurement.
  - ▶ Une procédure peut accepter ou non des paramètres, qui ne sont pas limités en nombre et peuvent être transférés du et vers l'environnement appelant.
- ▶ Une procédure ne retourne pas nécessairement une valeur.
- ▶ Les fonctions sont créées pour calculer une valeur, elles doivent retourner une valeur à l'environnement appelant. Une fonction peut accepter ou non des paramètres qui sont transmis de l'environnement.
- ▶ Une fonction ne peut retourner qu'une seule valeur et ne peut pas accepter de paramètre OUT ou IN OUT.

# Les packages

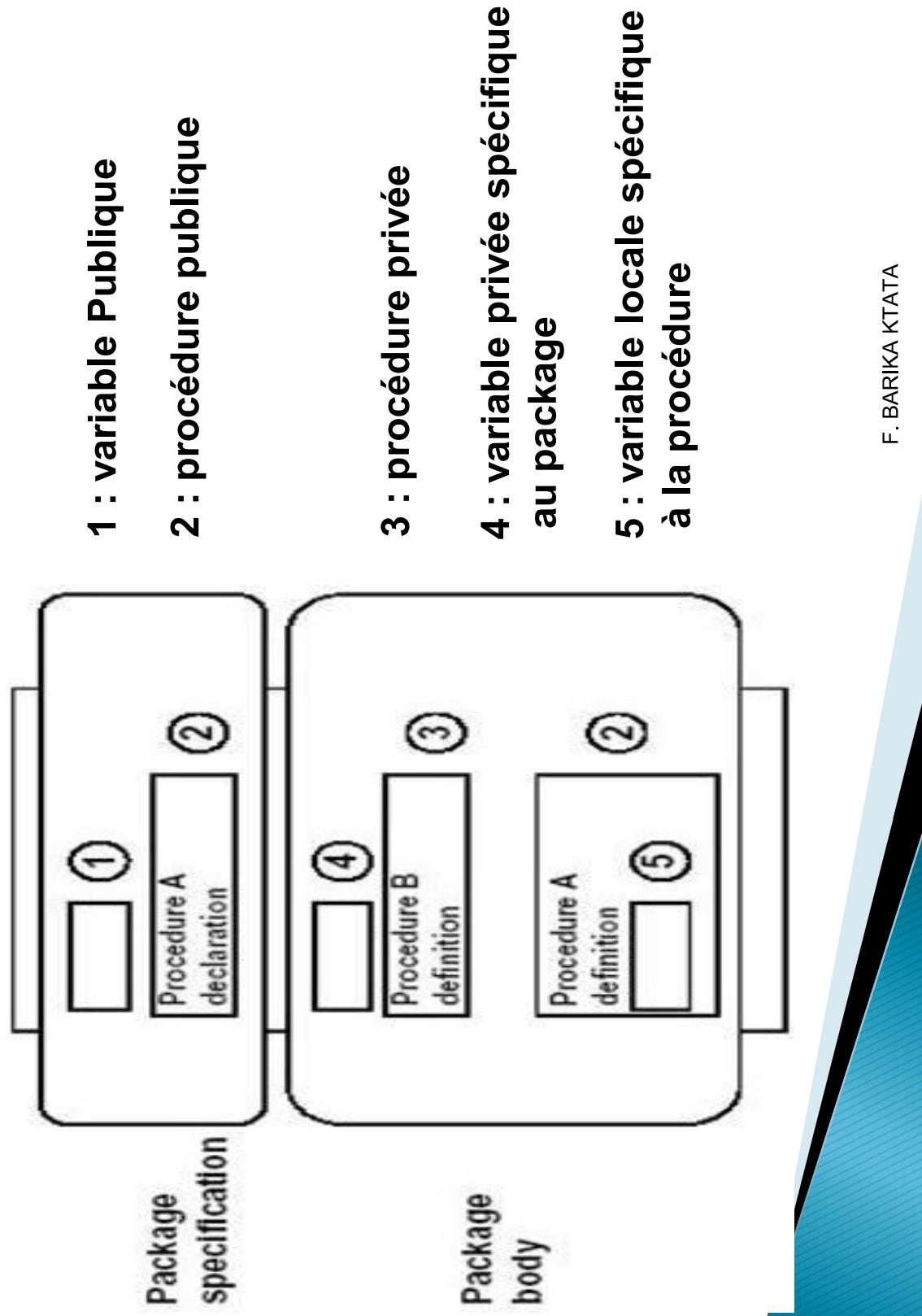
# Les packages

- Ce sont des groupes comprenant des types, des éléments et des sous-programmes PL/SQL logiquement associés.
- Généralement il est constitué de deux parties :
  - La spécification du package.
  - Le corps du package.

# Structure d'un package

- Un package est constitué de deux parties:
  1. *Partie déclarative, ou spécification* : contient la déclaration des procédures, fonctions, variables et traitements d'exception de type public qui sont accessibles de l'extérieur.
  2. *Partie corps, ou body* : contient les définitions de procédures et fonctions de type public déclarées dans la partie spécification, ainsi que les déclarations et définitions des procédures ou fonctions de type privé.

# Structure d'un package



# Types de procédures et fonctions

- ▶ Procédures ou fonctions publiques (PUBLIC): qui peuvent être appelées depuis l'extérieur du package.
- ▶ Procédures ou fonctions privées (PRIVATE): qui sont invisibles de l'extérieur et accessibles uniquement à des procédures du même package.

# Création de package

- ▶ Chaque partie du package doit être déclarée séparément.
- ▶ Partie 1 : *La Spécification*

```
CREATE [OR REPLACE] PACKAGE nom_package  
[IS | AS ]  
{ [déclaration de procédure;]  
[déclaration de fonction;]  
[déclaration de variables;]  
[déclaration de curseur;]  
[déclaration d'exception;] ... }  
END nom_package;
```

# Exemple

```
CREATE OR REPLACE PACKAGE mon_package  
AS  
    PROCEDURE augmentation_salaire (iden IN  
        emp.empno%TYPE, coef IN NUMBER);  
    PROCEDURE aug3 (ident emp.deptno%type);  
    FUNCTION salaire (matricule IN  
        emp.empno%TYPE ) RETURN NUMBER;  
    ma_variable emp.deptno%TYPE;  
    CURSOR c1 IS SELECT sal, empno FROM emp  
    WHERE deptno=ma_variable;  
    mon_exception EXCEPTION;  
END mon_package;
```

# Création de package

## ► Partie 2: *Le CORPS*

```
CREATE OR REPLACE PACKAGE BODY nom_package
[IS | AS]
{ [déclaration de procédures;]
  [déclaration de fonctions;]
  [déclaration de variables;]
  [déclaration de curseurs;]
  [déclaration d'exceptions;]
  [Définition des sous programmes PL/SQL;],...}
END nom_package;
```

# Exemple

```
CREATE OR REPLACE PACKAGE BODY mon_package
AS
    --corps de la procédure augmentation_salaire
    PROCEDURE augmentation_salaire(iden IN empno%TYPE, coef
    IN NUMBER)
    IS
        BEGIN
            UPDATE emp SET sal = sal * coef WHERE empno = iden;
        END augmentation_salaire;
    --corps de la fonction salaire
    FUNCTION salaire(matricule IN emp.empno%TYPE ) RETURN
    NUMBER
    IS
        sal emp.sal%TYPE:=0;
    BEGIN
        SELECT sal INTO sal FROM emp WHERE empno=matricule;
        IF sal<3000 THEN
            RAISE mon_exception;
        END IF;
```

# Exemple suite

```
RETURN (sal);
EXCEPTION
WHEN no_data found THEN dbms_output.put_line('Pas de
donnée trouvée');
WHEN mon_exception THEN dbms_output.put_line('Mon
exception a été générée ==> Salaire inférieur à 3000');
END salaire;
--corps de la procédure aug3
PROCEDURE aug3(ident emp.deptno%type)
IS
BEGIN
FOR c1_rec IN c1 LOOP
UPDATE emp SET sal = sal * 1.01 WHERE empno =
c1_rec.empno;
END LOOP;
commit;
END aug3;
END mon_package;
/
```

# Exécution d'une procédure publique d'un package

- Une procédure publique est déclarée dans la spécification du package et est définie dans le corps du package.
- Par conséquent, on peut l'appeler directement depuis l'environnement SQL\*PLUS en utilisant la commande EXECUTE.

**EXECUTE nom\_package.nom\_procédure  
(paramètre)**

**EXECUTE :var:=nom\_package.nom\_fonction  
(paramètre)**

# Suppression de package

- ▶ Pour supprimer un package on utilise :

**DROP PACKAGE BODY nom\_package;**

**DROP PACKAGE nom\_package;**

# Avantages

- Ils encapsulent des structures logiquement reliées dans un module nommé.
- Ils permettent de cacher des informations aux autres utilisateurs (public/privé).
- Amélioration des performances : lorsque le package est appelé pour la première fois il est entièrement chargé en mémoire ce qui évite les accès disque ultérieur.
- Surcharge des procédures et fonctions : on peut créer de multiples sous-programme avec le même nom dans le même package chacun prenant des paramètres de types différents.

# Exemple de surcharge d'une procédure

```
CREATE PACKAGE pilote_work AS
PROCEDURE valid_pilote (x_pilote_id IN pilote.nopilot%TYPE);
PROCEDURE valid_pilote (x_pilote_name IN pilote.nom%TYPE);
END pilote_work;
CREATE PACKAGE BODY pilote_work AS
PROCEDURE valid_pilote (x_pilote_id IN pilote.nopilot%TYPE);
IS
dif pilote.sal%TYPE := 0;
BEGIN
SELECT sal - comm
INTO dif
FROM pilote
WHERE nopilot = x_nopilot;
IF dif < 0 THEN
RAISE ERR_comm(-20001, 'commission supérieure au salaire!!');
END IF;
```

# Exemple de surcharge d'une procédure

```
PROCEDURE valid_pilote (x_pilote_name IN  
pilote.nom%TYPE);  
IS  
    dif pilote.sal%TYPE := 0;  
BEGIN  
    SELECT sal - comm  
    INTO dif  
    FROM pilote  
    WHERE nom = x_pilote_name;  
    IF dif < 0 THEN  
        RAISE ERR.comm(-20001, 'commission supérieure au  
salaire!');  
    END IF;  
  
END pilote_work;
```

# Initialisation d'un package

- Il est possible d'inclure dans un package un bloc spécifique qui n'est exécuté qu'une seule fois, au premier appel d'un composant du package dans une session.
- Exemple : pour initialiser une variable globale qui contient le nombre moyen d'heures de vol de tous les avions de la base, on utilise la structure suivante :

# Initialisation d'un package

```
CREATE PACKAGE pilote_work AS
g_moy_hvol NUMBER := 0;
END pilote_work;

CREATE PACKAGE BODY pilote_work AS
CREATE OR REPLACE FUNCTION moy_h_vol (x_codetype IN
appareil.codetype%TYPE)
RETURN NUMBER
IS
nbhvol_avg NUMBER (8, 2) := 0 /*valeur résultat*/
BEGIN
SELECT AVG(nbhvol)
INTO nbhvol_avg
FROM avion
WHERE type = x_codetype;
END moy_h_vol;

/*section d'initialisation*/
BEGIN
SELECT AVG (nbhvol)
INTO g_moy_hvol
FROM avion;
END pilote_work;
```

# Initialisation d'un package

```
CREATE PACKAGE pilote_work AS
g_moy_hvol NUMBER := 0;
END pilote_work;

CREATE PACKAGE BODY pilote_work AS
CREATE OR REPLACE FUNCTION moy_h_vol (x_codetype IN
appareil.codetype%TYPE)
RETURN NUMBER
IS
nbhvol_avg NUMBER (8, 2) := 0 /*valeur résultat*/
BEGIN
SELECT AVG(nbhvol)
INTO nbhvol_avg
FROM avion
WHERE type = x_codetype;
RETURN (nbhvol_avg);
END moy_h_vol;

/*section d'initialisation*/
BEGIN
SELECT AVG (nbhvol)
INTO g_moy_hvol
FROM avion;
END pilote_work;
```

# Persistiance des données

- ▶ Tout élément -variable, constante ou curseur- de type privé, c'est-à-dire déclaré dans une procédure (ou fonction) isolée ou dans une procédure d'un corps de package, est créée au moment de l'appel et supprimé à la fin de l'exécution de la procédure.
- ▶ Tout élément -variable, constante ou curseur- de type public, c'est-à-dire déclaré dans la partie spécification du package ou dans la partie corps d'un package mais sans être associé à une procédure ou fonction, est créée lors du premier appel est reste accessible jusqu'à la fin de la session.

# Privilèges requis pour la gestion et l'utilisation d'un package

- ▶ Pour créer un package, une procédure ou une fonction dans son propre compte un utilisateur doit posséder le privilège :

`CREATE PROCEDURE`

- ▶ Pour créer un package, une procédure ou une fonction dans n'importe quel compte un utilisateur doit posséder le privilège :

`CREATE ANY PROCEDURE`

- ▶ Il est possible d'autoriser l'accès à un package, à une procédure ou à une fonction à l'ensemble des utilisateurs en créant un synonyme public par l'ordre :

`CREATE PUBLIC SYNONYM nom`

`FOR {nom_package | nom_procedure | nom_fonction};`

# Trigger

# Définition

- ▶ On appelle déclencheur ou TRIGGER, un traitement déclenché par un événement.
- ▶ Le traitement peut inclure des ordres SQL qui agissent sur des tables auxquelles sont associés ces déclencheurs.
- ▶ Le traitement peut être une requête, séquence de requête SQL, ou une procédure SQL (PL\SQL).

# Syntaxe

```
CREATE TRIGGER <nom>
  // Événement
  { BEFORE | AFTER | INSTEAD OF }
  { INSERT | DELETE | UPDATE [ OF <liste de colonne> ]}
  ON <table> [ORDER <valeur>]
  [REFERENCING { NEW | OLD | NEW_TABLE | OLD_TABLE } AS <nom> ]
  //Condition
  (WHEN (<condition de recherche SQL>)
  //Action
  <Procédure SQL3>
  //Granularité
  [FOR EACH { ROW | STATEMENT }])
```

# Types de Triggers

- ▶ "Row" Trigger ou "Statement" Trigger
  - Row : le trigger est exécuté pour chaque ligne touchée
  - Statement : le trigger est exécuté une fois
- ▶ Exécution avant ou après l'événement
  - "before" : le bloc action est levé avant que l'événement soit exécuté
  - "after" : le bloc action est levé après l'exécution du bloc événement
- ▶ Trois événements possibles
  - UPDATE : certaines colonnes
  - INSERT
  - DELETE

# TRIGGER avec bloc PL/SQL

```
CREATE [OR REPLACE] TRIGGER nom_déclencheur
Séquence
Événement [OR événement]
ON nom_table
Bloc PL/SQL;
```

Où :

Séquence : peut prendre la valeur BEFORE ou AFTER  
Événement : peut prendre une des valeurs INSERT, UPDATE,  
DELETE

Nom\_table : le nom de la table à laquelle est associé le déclencheur

Bloc PL/SQL : décrit le traitement à réaliser.

# Anciennes et nouvelles valeurs

- ▶ Pour les row trigger (triggers lignes) : accès aux valeurs des colonnes pour chaque ligne modifiée
- ▶ Deux variables : :NEW.colonne et :OLD.colonne

	Ancienne valeur :OLD.colonne	Nouvelle valeur :NEW.colonne
INSERT	NULL	Nouvelle valeur
DELETE	Ancienne valeur	NULL
UPDATE	Ancienne valeur	Nouvelle valeur

# Exemple TRIGGER avec bloc PL/SQL

Limiter le droit d'ajouter un nouveau pilote à l'utilisateur Ali :

```
CREATE TRIGGER ajout_pilote
BEFORE INSERT ON pilote
BEGIN
IF USER != 'Ali'
THEN RAISE_APPLICATION_ERROR (-20001, 'utilisateur
non autorisé');
END IF;
END;
```

# TRIGGER ligne

Un TRIGGER ligne est exécuté pour chacune des lignes concernées par l'exécution de l'événement  
CREATE [OR REPLACE] TRIGGER nom\_déclencheur  
Séquence

Événement [OR événement]

ON nom\_table

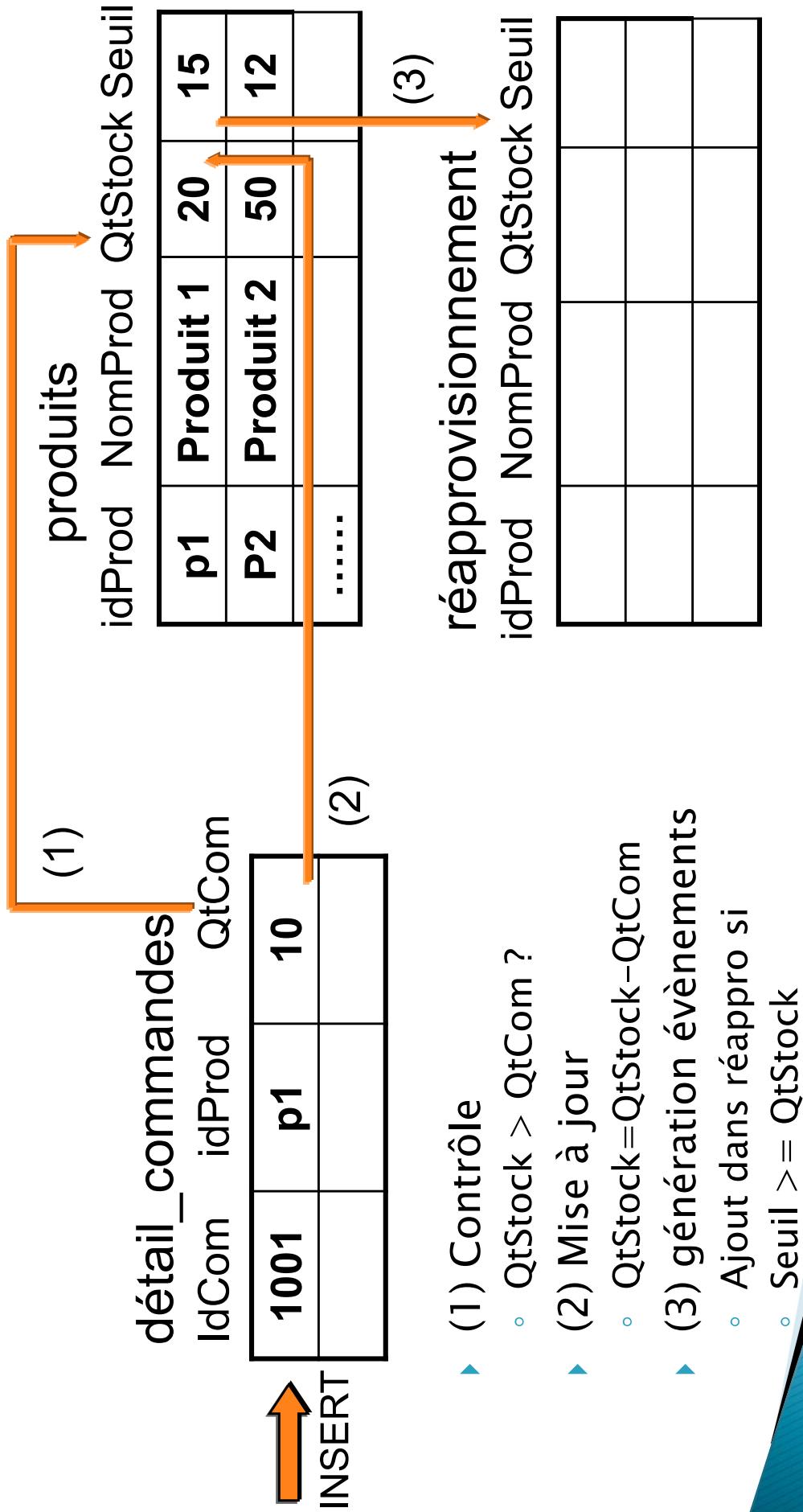
REFERENCING { [OLD [AS] ancien] | [NEW [AS]  
nouveau] } ]

FOR EACH ROW

[WHERE condition]

Bloc PL/SQL;

## Exemples de row trigger : prise de commande



# Prise de commande :

## (1) contrôle quantité en stock ?

```
CREATE TRIGGER t_b_i_detail_commandes  
BEFORE INSERT ON détail_commandes  
FOR EACH ROW  
DECLARE  
    v_qtstock NUMBER;  
BEGIN  
    SELECT qtstock INTO v_qtstock FROM produits  
    WHERE idprod = :NEW.idprod;  
    IF v_qtstock < :NEW.qtcom THEN  
        RAISE_APPLICATION_ERROR  
        (-20001, 'stock insuffisant' );  
    END IF;  
END;
```

## Prise de commande :

### (2) Mise à jour quantité en stock

```
CREATE TRIGGER t_a_i_detail_commandes  
AFTER INSERT ON detail_commandes  
FOR EACH ROW  
BEGIN  
UPDATE produits p  
SET p.qtstock = p.qtstock - :NEW.qtcom  
WHERE idprod = :NEW.idprod;  
END;  
/
```

## Prise de commande :

### (3) génération d'un réapprovisionnement

```
CREATE TRIGGER t_au_produits  
AFTER UPDATE OF qtstock ON produits  
FOR EACH ROW  
BEGIN  
IF :NEW.qtstock <= :NEW.seuil THEN  
INSERT INTO reapprovisionnement VALUES  
(:NEW.idprod, :NEW.nomprod, :NEW.qtstock,  
:NEW.seuil);  
END IF;  
END;  
/
```

# Les triggers d'état ou ‘Statement trigger’

- ▶ Raisonnement sur la globalité de la table et non sur un enregistrement particulier
- ▶ TRIGGER BEFORE : 1 action avant un ordre UPDATE de plusieurs lignes
- ▶ TRIGGER AFTER : 1 action après un ordre UPDATE touchant plusieurs lignes

# Exemple de Statement Trigger

- ▶ Interdiction d'emprunter un ouvrage pendant le week-end

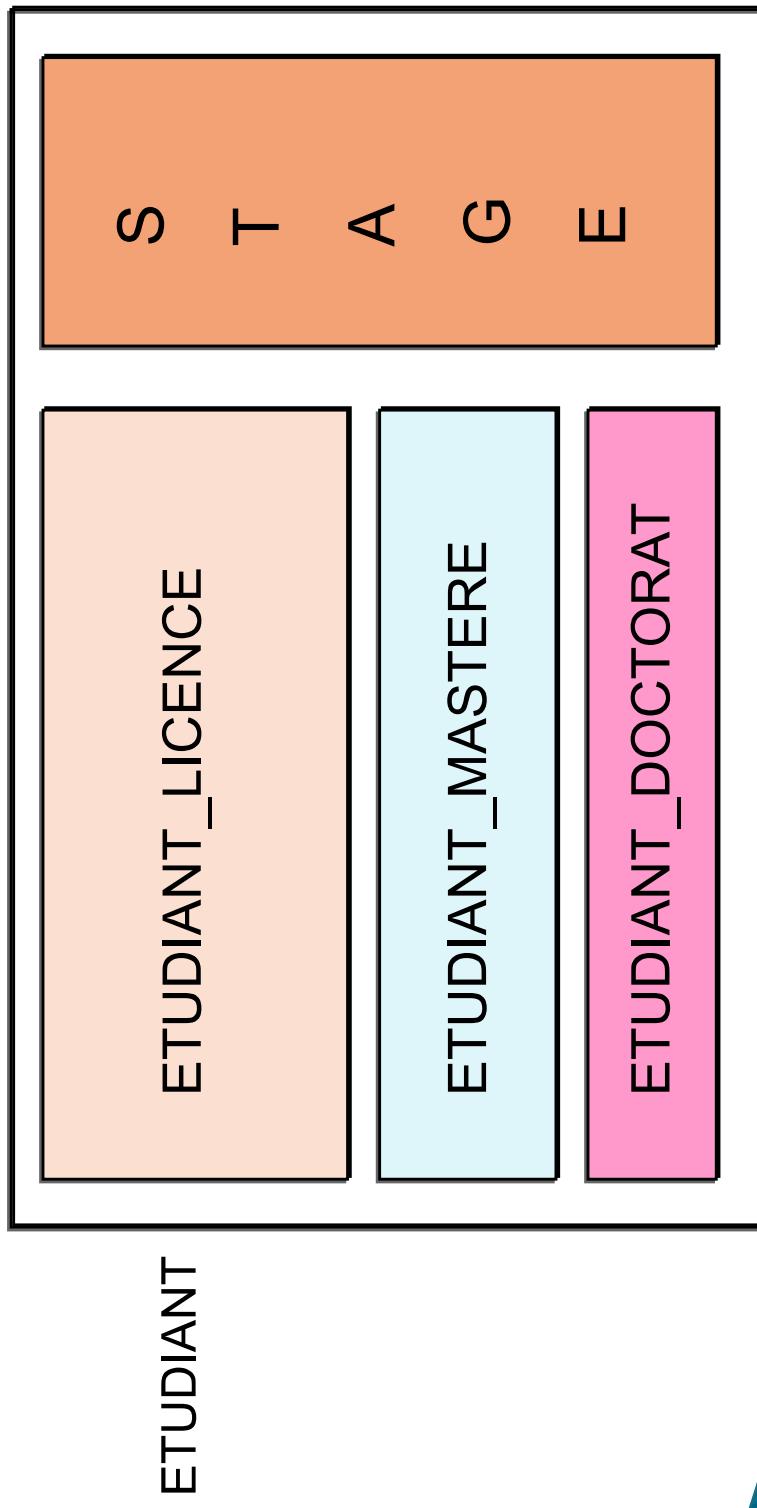
```
CREATE TRIGGER controle_date_emprunt
BEFORE UPDATE OR INSERT OR DELETE ON emprunt
BEGIN
IF TO_CHAR(SYSDATE, 'DY') 
IN ('SAT', 'SUN') THEN
RAISE_APPLICATION_ERROR
(-20102, 'Désolé les emprunts sont
interdits le week-end...') ;
END IF;
END;
```

# Les Triggers ‘INSTEAD OF’

- ▶ Trigger faisant le travail ‘à la place de’ .....
- ▶ Posé sur une vue multi-table pour autoriser les modifications sur ces objets virtuels
- ▶ Utilisé dans les bases de données réparties pour permettre les modifications sur le objets virtuels fragmentés
- ▶ Permet d’assurer un niveau d’abstraction élevé pour les utilisateurs ou développeurs clients : les vraies mises à jour sont faites à leur insu.

# Exemple de trigger 'instead of'

- ▶ Vue étudiant résultant de 4 tables



# Exemple de trigger ‘instead of’ Construction de la vue ETUDIANT

## Colonne virtuelle

```
CREATE VIEW etudiant
(ine, nom, adresse, cycle, nomstage, adstage)
AS SELECT el.ine, el.nom, el.adr, 'L', s.noms, s.ads
FROM etudiant_licence el, stage s
WHERE el.ine=s.ine
UNION
SELECT em.ine, em.nom, em.adr, 'M', s.noms, s.ads
FROM etudiant_mastere em, stage s
WHERE em.ine=s.ine
UNION
SELECT ed.ine, ed.nom, ed.adr, 'D', s.noms, s.ads
FROM etudiant_doctorat ed, stage s
WHERE ed.ine=s.ine;
```

# Exemple de trigger 'instead of' utilisation de la vue : INSERT

```
INSERT INTO etudiant VALUES  
(100, 'Michel', 'Toulouse', 'M', 'Oracle', 'CICT');
```

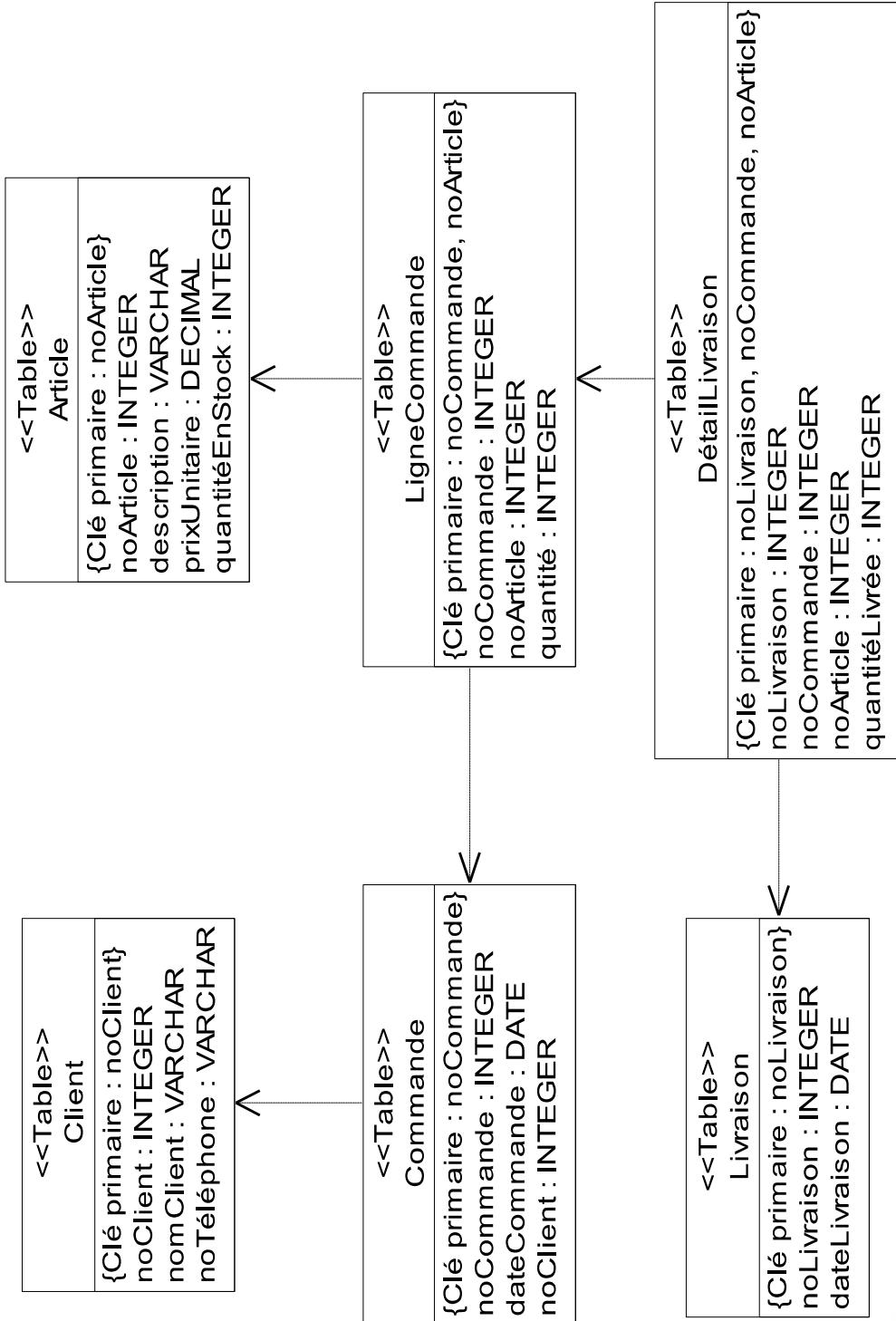
ETUDIANT

100	Michel	Toulouse	100 Oracle CICT

# Exemple de trigger ‘instead of’ trigger pour INSERT

```
CREATE TRIGGER insert_etudiant  
INSTEAD OF INSERT ON etudiant FOR EACH ROW  
BEGIN  
    IF :NEW.cycle='L' THEN  
        INSERT INTO etudiant licence VALUES  
        (:NEW.ine,:NEW.nom,:NEW.adresse);  
        INSERT INTO stage VALUES  
        (:NEW.ine,:NEW.nomstage,:NEW.adstage);  
    ELSEIF :NEW.cycle='M' THEN  
        . . . . .  
        . . . . . Idem pour M et D . . . . .  
    ELSE RAISE APPLICATION_ERROR  
        (-20455,'Entrer M, L ou D');  
    END IF;  
END;
```

# Exercices



# Exercices

- I. Le *prixUnitaire* d'un *Article* ne peut diminuer.
- II. Supposez qu'on ait ajouté à la table *Commande* une nouvelle colonne *totalCommande*. Ce *totalCommande* doit être égal au total des montants de chacune des lignes de la commande. Le montant de chacune des lignes correspond à la quantité commandée multipliée par le *prixUnitaire* de l'*Article*. La modification des *LigneCommandes* et des *prixUnitaire* doit être interdite.

# SQL dynamique

# SQL dynamique

Le SQL dynamique est une technique qui permet de construire des ordres SQL de façon dynamique à l'exécution du code PL/SQL. Le SQL dynamique permet de créer des applications plus souples car les noms des objets utilisés par un bloc PL/SQL peuvent être inconnus au moment de la compilation. Par exemple, une procédure peut travailler avec une table dont le nom est inconnu avant l'exécution de la procédure.

Le SQL est utilisé dans les cas suivants :

- ▶ La commande SQL n'est pas connue au moment de la compilation.
- ▶ Pour exécuter des requêtes construites lors de l'exécution.
- ▶ Pour référencer un objet de la BD qui n'existe pas au moment de la compilation.

L'ordre à exécuter n'est pas supporté par le SQL statique : par l'intermédiaire du SQL dynamique il devient possible d'exécuter des instructions du DDL (CREATE, GRANT, ALTER SESSION, SET ROLE...) à l'intérieur du code PL/SQL ce qui est impossible dans le cas statique.

# SQL dynamique

Le SQL statique englobe des informations qui sont connues au moment de la compilation et les ordres SQL ne changent pas d'une exécution à une autre. Le succès de la compilation est garantit par le fait que les ordres SQL font référence à des objets valides de la BD.

La compilation vérifie aussi que les priviléges nécessaires sont en place pour accéder et exploiter les objets de BD.

Le SQL dynamique ne doit être utilisé que si le SQL statique ne peut pas répondre au besoin, ou bien si la solution en statique est plus compliquée qu'en dynamique.

Le SQL dynamique = plus de flexibilité : on ne connaît pas les ordres SQL qui vont devoir être exécutés par le bloc PL/SQL ou bien si l'utilisateur doit donner des informations qui vont venir construire les ordres SQL à exécuter.

# SQL dynamique

**EXECUTE IMMEDIATE**

Exécuter dynamiquement un ordre SQL dans un bloc PL/SQL :

```
EXECUTE IMMEDIATE chaîne_dynamique  
[INTO {variable, ...} | record]  
[USING [IN | OUT | IN OUT] argument ...]  
[ {RETURNING | RETURN } INTO argument, ...]
```

chaîne\_dynamique : représente l'ordre SQL

Variable : pour stocker la valeur issue d'une colonne sélectionnée

Argument : valeur passée à l'ordre SQL  
INTO ne peut être utilisée que pour les ordres SQL ne retournant qu'une seule ligne de valeurs.

Le mode par défaut des arguments : **IN**

Les arguments du type OUT peuvent être précisés derrière le mot clé **RETURN**.

# SQL dynamique

- Étant donnée la base de données suivante :

```
CLIENTS (NoCli, Nom, Prénom, Adresse, CodePostal, Ville, Tel)
CATEGORIES(CodeCat, LibelleCat)
ARTICLES(RefArt, DesignationArt, Prix, #CodeCat)
STOCKS(#RefArt, Depot, QteStock, StockMini, StockMaxi)
COMMANDES (NoCde, DateCde, TauxRemise, #NoCli, EtatCde)
LIGNESCOMMANDES(#NoCde, NoLig, #RefArt, QteCde)
HISTOFAC(NoFac, DateFac, #NoCde, MontantHT, EtatFac)
```

# SQL dynamique

Exemple 1 :

```
DECLARE
    requete VARCHAR2(200);
    bloc_pl VARCHAR2(200);
    vnocli commandes.nocli%TYPE=8;
    vnocde commandes.nocde%TYPE=6;
    vjour DATE := sysdate();
    vetat char(2) := 'EC';
    vclients clients%ROWTYPE;
    vprix articles.prix%TYPE;
    vrefart articles.refart%TYPE; := '2201';

BEGIN
    --exécution d'un ordre DDL
    EXECUTE IMMEDIATE 'CREATE TABLE clients_fideles (nocli number(6), ca
    number (8,2))';
```

# SQL dynamique

```
--ordre DML avec passage d'arguments en entrée  
requete := 'INSERT INTO commandes (nocde, nocli, datecde, etatcde)  
values (:1,:2,:3,:4);  
EXECUTE IMMEDIATE requete USING vnocde, vnocli, vjour, vetat;  
  
--construction d'un bloc PL/SQL anonyme  
bloc_pl := 'BEGIN update articles set prix = prix *0.9; END;';  
EXECUTE IMMEDIATE bloc_pl;  
  
--utilisation du mot clé INTO  
requete := 'SELECT * FROM clients WHERE nocli =:1';  
EXECUTE IMMEDIATE requete INTO vclients USING vnocli;  
  
--utilisation de la clause RETURNING INTO  
requete := 'UPDATE articles SET prix =200 WHERE refart =:1 RETURNING  
prix INTO :2;  
EXECUTE IMMEDIATE requete USING vrefart RETURNING INTO vprix;  
END;
```

# SQL dynamique

Exemple 2 :

```
CREATE PROCEDURE create_dept ( deptno IN OUT NUMBER, dname IN
                               VARCHAR2, loc IN VARCHAR2) AS
BEGIN
    deptno := seq.NEXTVAL;
    INSERT INTO dept VALUES (deptno, dname, loc);
END;
```

Appel de cette procédure par du code PL/SQL dynamique :

```
DECLARE
    plsql_block VARCHAR2(500);
    new_deptno NUMBER(2);
    new_dname VARCHAR2(14) := 'ADVERT';
    new_loc VARCHAR2(13) := 'SOUSSÉ';
BEGIN
    plsql_block := 'BEGIN create_dept(:a, :b, :c); END;';
    EXECUTE IMMEDIATE plsql_block USING IN OUT new_deptno, new_dname,
    new_loc;
    IF new_deptno > 90 THEN
        ...
    END;
```

# SQL dynamique

## Variable Curseur

Pour paramétrier un ordre SQL travaillant sur plusieurs tuples  
Action à préciser dans la directive OPEN

```
OPEN nom curseur
FOR chaîne caractères décrivant_ordre_SQL_dynamique
[ USING param 1 [, param 2...]]
```

```
DECLARE
  TYPE t_ref_cursor IS REF CURSOR;
  ex_cursor t_ref_cursor;
  v_p1 INT DEFAULT 100;
BEGIN
  OPEN ex_cursor FOR 'SELECT Nom FROM Clients WHERE Debit > :p1 ,
  USING v_p1;
  LOOP ...
```

# SQL dynamique

```
DECLARE
    req VARCHAR2(100);
    TYPE monCurseur IS REF CURSOR;
    v_maVariable maTable%ROWTYPE;
BEGIN
    req := 'SELECT * FROM maTable';
    OPEN monCurseur FOR req;
    LOOP FETCH monCurseur INTO v_maVariable;
    EXIT WHEN monCurseur%NOTFOUND;
    --- ...
END LOOP;
CLOSE monCurseur;
END; /
```

# SQL dynamique

```
Package
CREATE OR REPLACE PACKAGE CURSPKG AS
  TYPE T_CURSOR IS REF CURSOR;
  PROCEDURE OPEN_ONE_CURSOR (N_EMPNO IN NUMBER, IO_CURSOR IN OUT
    T_CURSOR);
  PROCEDURE OPEN_TWO_CURSORS (EMPCURSOR OUT T_CURSOR, DEPCURSOR OUT
    T_CURSOR);
END CURSPKG;

CREATE OR REPLACE PACKAGE BODY CURSPKG AS
  PROCEDURE OPEN_ONE_CURSOR (N_EMPNO IN NUMBER, IO_CURSOR IN OUT
    T_CURSOR)
  IS
    V_CURSOR T_CURSOR;
  BEGIN
    IF N_EMPNO <> 0
    THEN
      OPEN V_CURSOR FOR
        SELECT EMP.EMPNO, EMP.ENAME, DEPT.DEPTNO, DEPT.DNAME
        FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO
        AND EMP.EMPNO = N_EMPNO;
```

# SQL dynamique

```
ELSE
OPEN V_CURSOR FOR
SELECT EMP.EMPNO, EMP.ENAME, DEPT.DEPTNO, DEPT.DNAME
FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO;
END IF;

IO_CURSOR := V_CURSOR;
END OPEN_ONE_CURSOR;

PROCEDURE OPEN_TWO_CURSORS (EMPCURSOR OUT T_CURSOR, DEPCURSOR OUT
T_CURSOR)
IS
V_CURSOR1 T_CURSOR;
V_CURSOR2 T_CURSOR;
BEGIN
OPEN V_CURSOR1 FOR SELECT * FROM EMP;
OPEN V_CURSOR2 FOR SELECT * FROM DEPT;
EMPCURSOR := V_CURSOR1;
DEPCURSOR := V_CURSOR2;
END OPEN_TWO_CURSORS;
END CURSPKG;
```