

PL/SQL

Procedural Language /SQL

# Langage PL/SQL

## PLAN

- ❖ Introduction
- ❖ Avantages de PL/SQL
- ❖ Les principales caractéristiques du PL/SQL
- ❖ Les variables et les constantes
- ❖ Les instructions de base
- ❖ Les curseurs
- ❖ Gestion des exceptions
- ❖ Les procédures, les fonctions...
- ❖ Les triggers

# Introduction

- ❖ PL/SQL (Procedural Language /SQL), L'extension procédurale proposée par oracle pour SQL
- ❖ Il permet de combiner des requêtes SQL (SELECT, INSERT, UPDATE et DELETE) et des instructions procédurales (boucles, conditions...),
- ❖ Créer des traitements complexes destinés à être stockés sur le serveur de base de données (objets serveur),
- ❖ Comme on le sait, les structures de contrôle habituelles d'un langage (IF, WHILE...) ne font pas partie intégrante de la norme SQL. Oracle les prend en compte dans PL/SQL

# Introduction

Id_Produit	Libellé	Marque	Prix	ID_Fournisseur
P1	HP600	HP	1000	1
P2	Epson30	Epson	2300	1
P3	IBM 7380	IBM	4600	2
P4	Lexmark310	Lexmark	2000	3

Update produit set prix =2500 where id\_fournisseur=1

Pour le fournisseur N°

Si Moyenne(prix)>2000

Augmenter le prix de 5%

sinon

Augmenter le prix de 3%

**Code PL/SQL**

**Procedure:**

IF .... THEN

ELSEIF

ENDIF

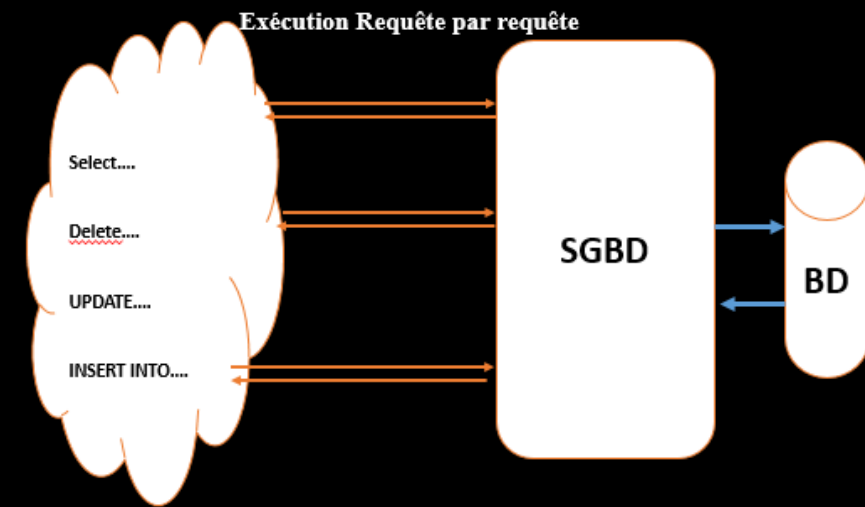
# Avantage du langage PL/SQL

Ce langage propose des performances pour le traitement des transactions et offre les avantages suivantes:

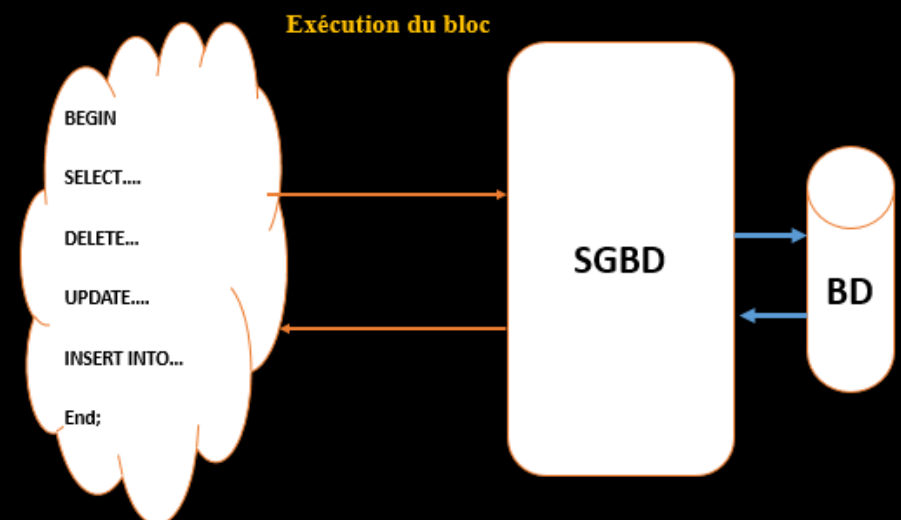
- Intégration complète du SQL
- Prise en charge de la programmation orientés objet (surcharge)
- Parfaite intégration avec Oracle et Java. PL/SQL est un langage propriétaire d'Oracle, on peut lancer des sous-programme PL/SQL à partir de Java et de même appeler des procédures Java à partir d'un bloc PL/SQL

# Avantage du langage PL/SQL

Dans un environnement client serveur, **chaque instruction SQL donne lieu à l'envoi d'un message du client vers le serveur** suivi de la réponse du serveur vers le client



**Un bloc PL/SQL donne lieu à un seul échange sur le réseau entre le client et le serveur.** Les résultat intermédiaires sont traités côté serveur et seul le résultat final est retourné au client



# Langage PL/SQL

Construction de procédures ou fonctions stockées qui améliorent le mode client-serveur par stockage des procédures ou fonctions souvent utilisées au niveau serveur:

- ❖ Gestion des erreurs
- ❖ Construction de triggers (ou déclencheurs)

# Structure d'un bloc PL/SQL

Un programme ou une procédure PL/SQL est un ensemble de un ou plusieurs blocs. Chaque bloc comporte trois sections :

- 1. Section déclaration**
- 2. Section corps du bloc**
- 3. Section traitement des erreurs**



# Section déclaration

- ❖ Elle Contient la description des structures et des variables utilisées dans le bloc
- ❖ Section facultative
- ❖ Commence par le mot clé **DECLARE**

# Section corps du bloc

- ❖ Contient les instructions du programme et éventuellement, à la fin, la section de traitement des erreurs
- ❖ Obligatoire
- ❖ Introduite par le mot clé **BEGIN**
- ❖ Se termine par le mot clé **END**

# Section traitement des erreurs

- ❖ Facultative
- ❖ Introduite par le mot clé **EXCEPTION**

# Syntaxe

**DECLARE**

*déclaration*

**BEGIN**

*corps-du-bloc*

**EXCEPTION**

*traitement-des-erreurs*

**END;**

*/*

# Exemple

**SET SERVEROUTPUT ON**

**DECLARE**

**x** VARCHAR2(10);

**BEGIN**

**x** := 'Bonjour';

**DBMS\_OUTPUT.PUT\_LINE(x);**

**END;**

**/**

# Exemple (2)

**DECLARE**

erreurEx EXCEPTION;

num exemplaire.numExemplaire%TYPE;

film exemplaire.numFilm%TYPE;

pb exemplaire.probleme%TYPE;

**BEGIN**

...

# Exemple (2 suite)

...

**BEGIN**

**SELECT** numExemplaire, numFilm, probleme **INTO** num, film,

pb

**FROM** exemplaire **WHERE** numExemplaire = 1;

**IF** probleme **IS NOT NULL THEN**

**RAISE** erreurEx;

**END IF;**

**DBMS\_OUTPUT.PUT\_LINE**(num || ' OK');

...

# Exemple (2 suite )

```
....  
EXCEPTION  
WHEN NO_DATA_FOUND THEN  
  DBMS_OUTPUT.PUT_LINE('numéro inconnu');  
WHEN erreurEx THEN DBMS_OUTPUT.PUT_LINE(num ||  
'problème');  
END;  
/
```



# Types de variables

- ❖ Variables scalaires
- ❖ Types composés
  - Enregistrement (record)
  - Table

# Variables scalaires

- ❖ Types issus de SQL : **CHAR**, **NUMBER**, **DATE**, **VARCHAR2**
- ❖ Types PL/SQL : **BOOLEAN**, **SMALLINT**, **BINARY\_INTEGER**, **DECIMAL**, **FLOAT**, **INTEGER**, **REAL**, **ROWID**
- ❖ Les variables hôtes sont préfixées par « : » ce sont des variables définies dans l'environnement extérieur au bloc et utilisées dans le bloc

# Déclaration des Variables scalaires

❖ *nom-variable nom-du-type;*

- Exemple :

x VARCHAR2(10);

❖ *nom-variable nom-table.nom-colonne%TYPE;*

- Exemple :

film exemplaire.numFilm%TYPE;

# Example

```
set SERVEROUTPUT ON
DECLARE
var_nom VARCHAR(20);
var_date date;
var_age NUMBER;
BEGIN
var_nom := 'SALAMI';
var_age := 23;
var_date := '01/04/2020';
DBMS_OUTPUT.PUT_LINE('Le nom : ' || var_nom);
DBMS_OUTPUT.PUT_LINE('La date : ' || var_date);
DBMS_OUTPUT.PUT_LINE('L'age : ' || var_age);

END;
```

# Déclaration pour un enregistrement

- ❖ Soit par référence à une structure de table ou de curseur en utilisant la notation **%ROWTYPE** :

*nom-variable*    *nom-table***%ROWTYPE**;

*nom-variable*    *nom-curseur***%ROWTYPE**;

# Déclaration pour un enregistrement

- ❖ Soit par énumération des rubriques qui la composent. Cela se fait en deux étapes :

- Déclaration du type enregistrement

**TYPE** *nom-du-type-record* **IS RECORD** (

*nom-attribut<sub>1</sub>* *type-attribut<sub>1</sub>*,

*nom-attribut<sub>2</sub>* *type-attribut<sub>2</sub>*, ...);

- Déclaration de la variable de type enregistrement

*nom-variable* *nom-du-type-record*;

# Example

**DECLARE**

**TYPE** revenu **IS RECORD** (

Nom pilote.nom%type,

Salaire Number (9,2));

Rev\_pilote revenu;

**BEGIN**

**DBMS\_OUTPUT.PUT\_LINE** (rev\_pilote.nom || ' ' ||  
rev\_pilote.salaire);

**END;**

/

# Exemple

```
Accept nom_entré Prompt ' Entrez le nom du pilote'
DECLARE
  TYPE revenu IS RECORD (
    Nom pilote.nom%type:= '&nom_entré',
    Salaire Number (9,2):= 8000.00);
  Rev_pilote revenu;
BEGIN
  DBMS_OUTPUT.PUT_LINE (' le nom du pilote est: ' ||
    rev_pilote.nom);
END;
```



# Tables

- ❖ Structure composée d'éléments d'un même type *scalaire*
- ❖ L'accès à un élément de la table s'effectue grâce à un indice, ou clé primaire
- ❖ Cet index est déclaré de type **BINARY\_INTEGER** (valeurs entières signées)

# Déclaration pour une table

❖ Deux étapes :

- Déclaration du type de l'élément de la table
- Déclaration de la variable de type table

# Déclaration pour une table

- ❖ Déclaration du type de l'élément de la table :

```
TYPE nom-du-type-table  
IS TABLE OF type-argument  
INDEX BY BINARY_INTEGER;
```

- ❖ Déclaration de la variable de type table :

```
nom-variable nom-du-type-table;
```

# Example

**DECLARE**

TYPE **tabNom** IS TABLE OF VARCHAR2(20)

INDEX BY BINARY\_INTEGER;

**tableNom** **tabNom**;

**i** BINARY\_INTEGER;

**BEGIN**

**tableNom**(5) := 'Dupont';

**i** := 10;

**tableNom**(**i**) := 'Dupond';

**END;**

# Variables (scalaires ou composées)

❖ Valeur initiale :

*~~nom-variable~~ nom-type := valeur;*

❖ Constante :

*~~nom-variable~~ nom-type DEFAULT valeur;*

ou

*~~nom-variable~~ CONSTANT nom-type := valeur;*

# Variables (scalaires ou composées)

- ❖ Visibilité : une variable est utilisable dans le bloc où elle a été définie ainsi que dans les blocs imbriqués dans le bloc de définition, sauf si elle est redéfinie dans un bloc interne

# Conversion de type

- ❖ Explicite avec

**TO\_CHAR, TO\_DATE, TO\_NUMBER,  
RAWTOHEX, HEXTORAW**

- ❖ Implicites, par conversion automatique

Les instructions



# Les instructions

1. Affectations
2. Instructions du langage SQL : CLOSE, COMMIT, DELETE, FETCH, INSERT, LOCK, OPEN, ROLLBACK, SAVEPOINT, SELECT, SET TRANSACTION, UPDATE
3. Instructions de contrôle itératif ou répétitif
4. Instructions de gestion de curseurs
5. Instructions de gestion des erreurs

# L'Affectation

- ❖ Opérateur d'affectation **:=**
- ❖ Option **INTO** dans un ordre **SELECT**
- ❖ Instruction **FETCH** avec un curseur

# Example

**DECLARE**

Type **t\_nom** is table of **Char(35)**  
**INDEX BY BINARY\_INTEGER;**

**tableNom** **t\_nom;**

**I BINARY\_INTEGER;**

**BEGIN**

**tableNom(5) := 'Arthur';**

**i := 10;**

**tableNom(i) := 'Arthur';**

**END;**

# Exemple

**DECLARE**

U\_nom pilote.nom%TYPE;

U\_sal pilote.sal%TYPE;

**BEGIN**

**SELECT** nom, sal **INTO** u\_nom, u\_sal **FROM** Pilote

**WHERE** nopilote = '7937';

DBMS\_OUTPUT.PUT\_LINE (u\_nom || ' ' || u\_sal);

**END;**

# Exemple

**DECLARE**

**TYPE** t\_pilote **IS RECORD** (

    nom\_pilote   pilote.nom%TYPE,

    Revenu\_pilote **Number** (5,2));

Employé t\_pilote;

**BEGIN**

Employé.nom\_pilote := 'DUPUY';

Employé.revenu\_pilote := 12345.00;

**END;**

# Exemple

**DECLARE**

```
TYPE t_emprec IS RECORD (  
    r_nom  pilote.nom%TYPE,  
    r_sal  pilote.sal%TYPE);
```

```
Emprec t_emprec;
```

**BEGIN**

```
SELECT nom, sal INTO emprec FROM Pilote WHERE nopilote  
='7937';
```

```
DBMS_OUTPUT.PUT_LINE (emprec.r_nom || ' ' || emprec.r_sal);
```

**END;**

# Structures de contrôle

- ❖ Structure alternative

- ❖ Structure répétitives

# Structures alternatives

**IF *condition* THEN *instructions*;**  
**END IF;**

**IF *condition* THEN *instructions*;**  
**ELSE *instructions*; END IF;**

**IF *condition* THEN *instructions*;**  
**ELSIF *condition* THEN *instructions*;**  
**ELSE *instructions*; END IF;**



# Exemple

```
set SERVEROUTPUT on;  
DECLARE  
var_number NUMBER:= 9;  
BEGIN  
if (var_number>10) then  
dbms_output.put_line(' le numéro est supérieur à 10');  
elsif (var_number <10) then  
dbms_output.put_line(' le numéro est inférieur à 10');  
else  
dbms_output.put_line(' le numéro est égal à 10');  
end if;  
END;
```

# Structures répétitives

```
LOOP instructions; END LOOP;
```

Cette boucle est infinie: il faut utiliser **EXIT** pour en sortir

```
LOOP instructions; ...  
EXIT WHEN condition; ...  
END LOOP;
```

```
LOOP ...  
IF condition THEN EXIT; END IF;  
... END LOOP;
```

# Example

```
set serveroutput On
DECLARE
    i number:=0;
begin
    /*la boucle simple*/
loop
    DBMS_OUTPUT.PUT_LINE(i);
    i:=i+1;
    Exit when (i>5);
end loop;

end;
```

# Structures répétitives

L'instruction **For** contrôle le nombre d'exécutions des instructions de la structure répétitive par incrémentation et test d'une variable indice

**FOR** *variable-indice* **IN** [**REVERSE**] *val-début* .. *val-fin*

**LOOP** *instructions*; **END LOOP**;

- ❖ *variable-indice* est une variable locale (locale à la boucle) **non déclarée**
- ❖ *val-début* et *val-fin* sont des variables locales **déclarées** et initialisées ou alors des constantes
- ❖ le pas est -1 si **REVERSE** est présent, sinon il est égal à +1

# Structures répétitives

L'instruction **While** répète les instructions de la structure répétitive tant que la condition à la valeur est Vrai

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

# Exemple

## Avec la boucle While

```
set serveroutput On
DECLARE
  i number:=0;
begin
  /*la boucle While*/
  While (i>5)
  loop
    DBMS_OUTPUT.PUT_LINE(i);
    i:=i+1;
  end loop;
  |
end;
```

## Avec la boucle For

```
set serveroutput On
begin
  /*la boucle for*/
  for i in 0..5
  loop
    DBMS_OUTPUT.PUT_LINE(i);


  end loop;

  end;
  |
```

# L'interaction avec la base de données

## La clause INTO en PL/SQL

La clause **INTO** permet de passer des valeurs d'une table dans des variables

```
 DECLARE
    nom varchar(30);
    salaire NUMBER(8,2);

begin
    Select nom, sal Into nom, salaire from pilote where nopilote='1333'
end;
```

# L'interaction avec la base de données

## Mettre à jour des données

Ex: Augmenter le salaire de tous les pilotes ayant le salaire entre 19000 et 23000

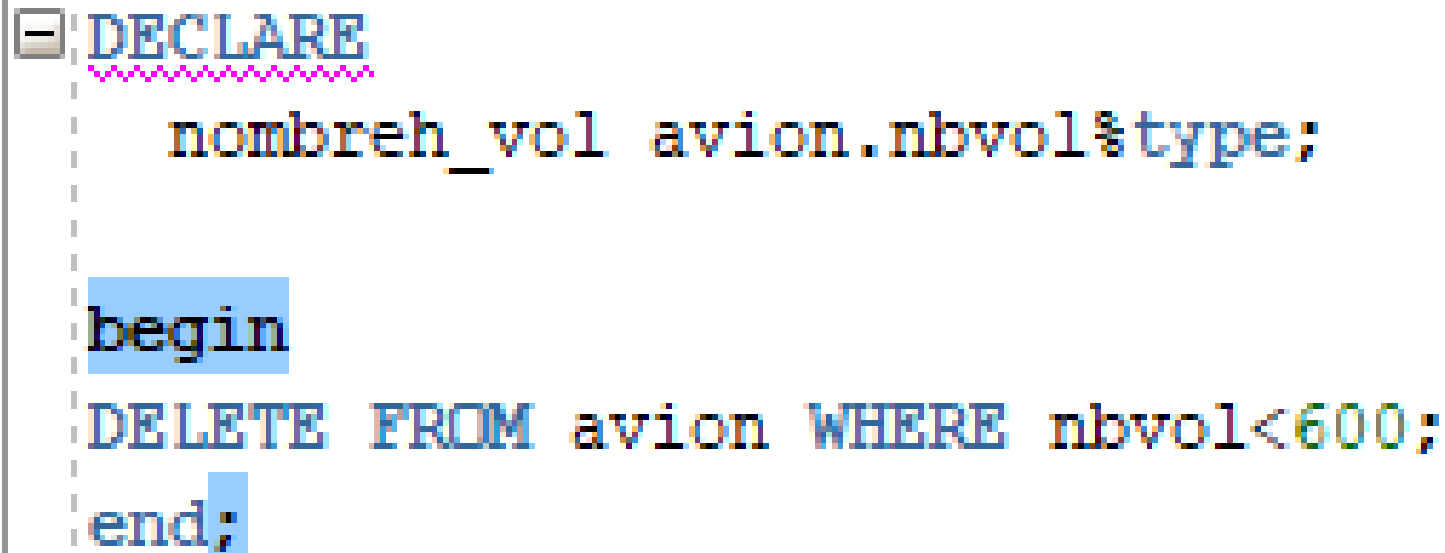
```
set serveroutput
DECLARE
  nom varchar(30);
  salaire pilote.sal%TYPE:=1000;
begin
  Update pilote
  set sal=sal+salaire where sal BETWEEN 19000 and 23000;
end;
```



# L'interaction avec la base de données

## Supprimer des données

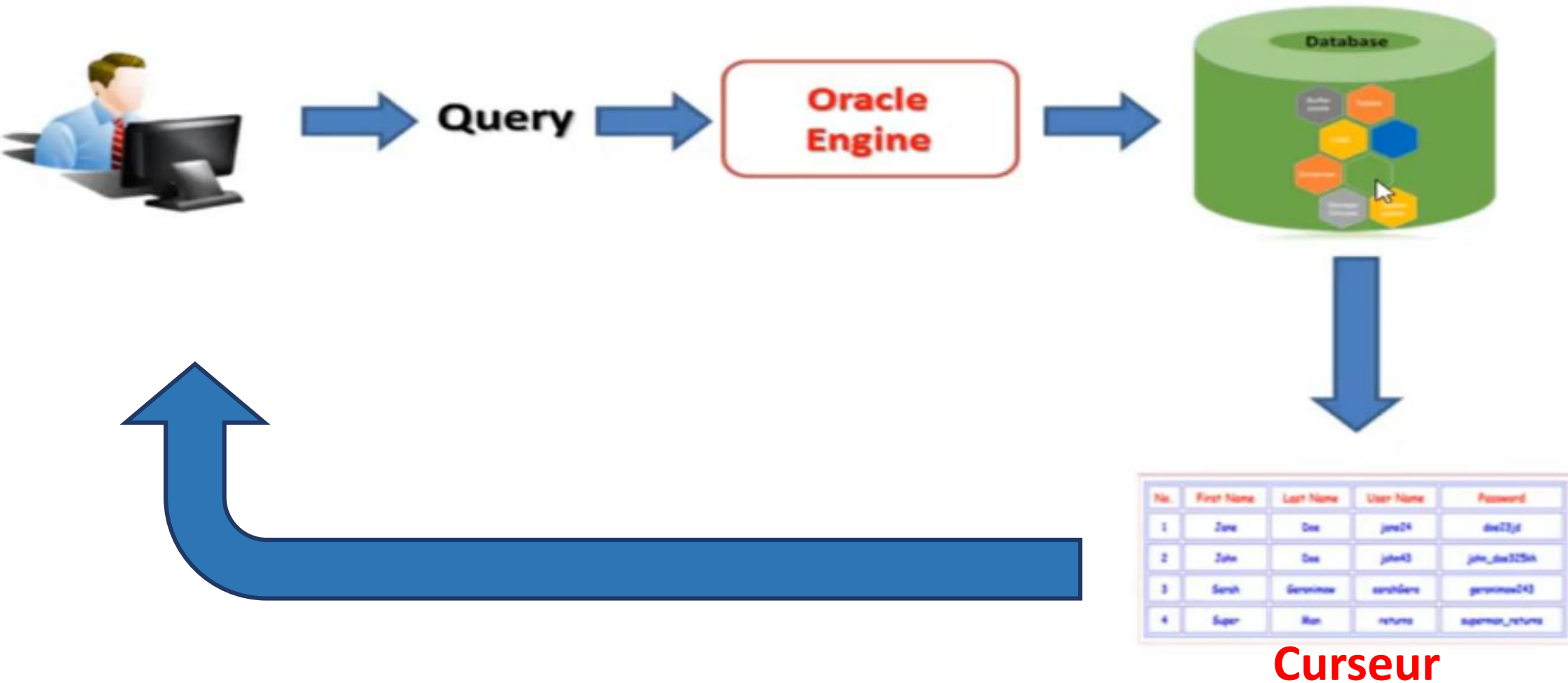
Ex: supprimer les lignes qu'on nombre d'heure de vol est <600 à partir de la table Avion

```
A screenshot of a SQL code editor window. The window has a title bar with a minus, plus, and close button. The code is written in a monospaced font with syntax highlighting: DECLARE is blue, nombreh_vol is black, avion.nbvol&type is black, begin is blue, DELETE FROM avion WHERE nbvol<600; is black, and end; is blue. The code is as follows:  
  
DECLARE  
    nombreh_vol avion.nbvol&type  
  
begin  
DELETE FROM avion WHERE nbvol<600;  
end;
```

# L'interaction avec la base de données

- Initialiser une transaction avec la première instruction LMD suivant COMMIT ou ROLLBACK
- Utiliser les instructions SQL COMMIT et ROLLBACK pour mettre fin explicitement à une transaction

# Les curseurs

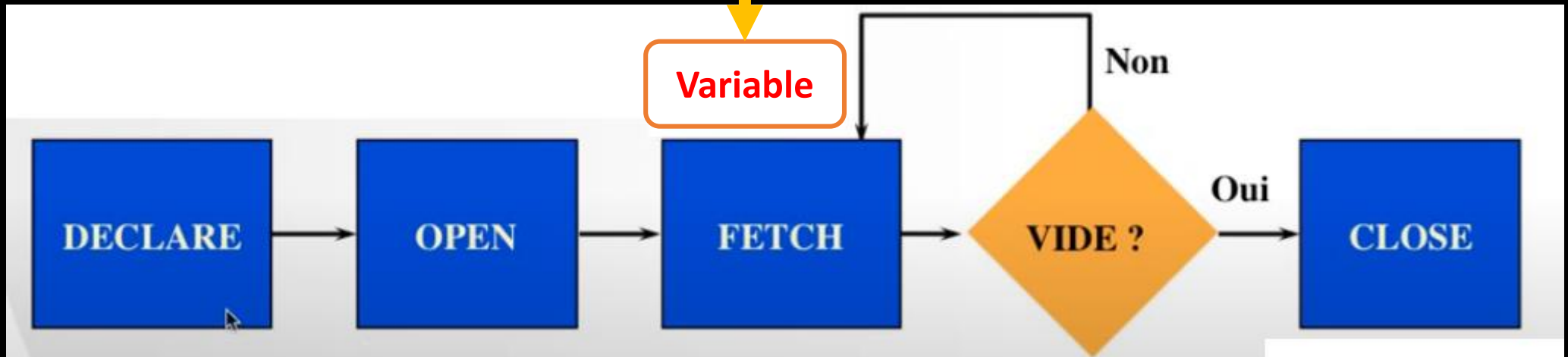


# Les curseurs

- ❖ Dès l'instant où on exécute une requête SQL, il y a création d'un curseur.
- ❖ Un curseur est une variable qui pointe vers le résultat d'une requête SQL,
- ❖ Un curseur est une zone de travail de l'environnement utilisateur qui contient les informations relatives à l'instruction ou requête SQL :
  - Le texte source de l'ordre SQL
  - Le texte «compilé» de l'ordre SQL
  - Un tampon pour une ligne du résultat
  - Le statut (*cursor status*)
  - Des informations de travail et de contrôle

# Curseurs explicites

	N	Nom	Pre	tel
Pointeur Curseur	--	---	---	---
	--	---	---	---
	--	---	---	---



- Créer une zone mémoire SQL

- Exécute
- Identifier l'ensemble actif

- Charger la ligne en cours dans des variables

- Tester l'existence de la ligne
- Si des lignes existent, revenir à FETCH

- Libérer l'ensemble actif

# Curseurs explicites

- ❖ Obligatoires pour un **SELECT** susceptible de produire plusieurs lignes résultat
- ❖ Quatre étapes :
  - 1) Déclaration du curseur
  - 2) Ouverture du curseur
  - 3) Traitement des lignes du résultat
  - 4) Fermeture du curseur

# Déclaration du curseur

- ❖ Association d'un nom de curseur à une requête **SELECT**
- ❖ Se fait dans la section **DECLARE** d'un bloc PL/SQL

**CURSOR** *nom-curseur* **IS** *requête*;

- ❖ Un curseur peut être paramétré :

**CURSOR** *nom-curseur* (*nom-p<sub>1</sub>* *type-p<sub>1</sub>* *[:= val-défaut]*, ...) **IS** *requête*;

# Example

**DECLARE**

**Cursor** C1 **IS** select nom from pilote  
where sal>1000;

**Cursor** C2 (psal Number (7,2), pcom number(7,2)) **IS**  
select ename from emp  
where sal> psal and comm>pcom;



## Ouverture d'un curseur

- ❖ Alloue un espace mémoire au curseur et positionne les éventuels verrous

**OPEN *nom-curseur*;**

ou

**OPEN *nom-curseur* (*liste-paramètres-effectifs*);**

- ❖ Pour les paramètres, association par position ou par nom sous la forme

***paramètre-formel* (*Lors de la déclaration*) $\Rightarrow$**

***paramètre-réel* (*Lors de l'ouverture*)**

# Example

- ❖ OPEN C1;
- ❖ OPEN C2 (1600, 1800); Par position
- ❖ OPEN C2 ( $q \Rightarrow 1800$ ,  $p \Rightarrow 1600$ ); Par nom

# Traitement des lignes

Les lignes obtenues par l'exécution de la requête SQL sont distribuées une à une par l'exécution d'un ordre FETCH . Pour chaque ligne, cette instruction transfère les valeurs des attributs projetés par l'ordre Select dans des variables PL/SQL.

La syntaxe utilisés est la suivante:

**FETCH** *nom-curseur* **INTO** *liste-variables*;

**ou**

**FETCH** *nom-curseur* **INTO** *nom-enregistrement*;

# Example

```
set serveroutput on;  
DECLARE  
    Cursor c_emp  
    IS  
        SELECT first_name from employees where department_id=30;  
    v_name employees.first_name% type;  
BEGIN  
    Open c_emp;  
    loop  
        fetch c_emp into v_name;  
        exit when c_emp%notfound;  
        DBMS_OUTPUT.PUT_LINE('nom: ' || v_name);  
    end loop;  
    close c_emp;  
END;
```

# Exemple 2

```
set serveroutput on;  
DECLARE  
    type t_emp IS Record  
        ( nom employees.first_name%type,  
          sal employees.salary%type);  
    v_emp_rec t_emp;  
    Cursor c_emp  
    IS  
        SELECT first_name, salary from employees where department_id=30;  
BEGIN  
    Open c_emp;  
    loop  
        fetch c_emp into v_emp_rec;  
        exit when c_emp%notfound;  
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.nom||' '||v_emp_rec.sal);  
    end loop;  
    close c_emp;  
END;
```

# Example 3

```
set serveroutput on;
```

```
DECLARE
```

```
Cursor c_emp
```

```
IS
```

```
SELECT * from employees where department_id=30;
```

```
v_emp_rec c_emp% rowtype;-- avec record
```

```
BEGIN
```

```
Open c_emp;
```

```
loop
```

```
fetch c_emp into v_emp_rec;
```

```
exit when c_emp%notfound;
```

```
DBMS_OUTPUT.PUT_LINE(v_emp_rec.first_name||' '||v_emp_rec.salary||' '||v_emp_rec.department_id);
```

```
end loop;
```

```
close c_emp;
```

```
END;
```

# Forme syntaxique condensée avec la boucle FOR

La forme condensée utilise la structure For pour distribuer les lignes résultats selon la construction suivante:

**DECLARE**

**CURSOR** nom\_curseur **IS** requête;

**Begin**

**for** nom\_enregistrement **IN** nom\_curseur  
    [(paramètres effectifs)]

**Loop**

            Traitement;

**end** loop;

**end;**

# Forme syntaxique condensée avec la boucle FOR

```
set serveroutput on;
```

```
DECLARE
```

```
Cursor c_emp
```

```
IS
```

```
SELECT * from employees where department_id=30;
```

```
BEGIN
```

```
For v_emp_rec IN c_emp
```

```
loop
```

```
DBMS_OUTPUT.PUT_LINE(v_emp_rec.first_name||' '||v_emp_rec.salary||' '||v_emp_rec.department_id);
```

```
end loop;
```

```
END;
```



# Statut d'un curseur

Pour Obtenir les informations d'état concernant un curseur:

**NomCurseur%Attribut**

Attribut	Type	Description
%ISOPEN	BOOLEAN	Prend la valeur <b>TRUE</b> si le curseur est ouvert
%NOTFOUND	BOOLEAN	Prend la valeur <b>TRUE</b> si la dernière extraction ne renvoie pas de ligne
%FOUND	BOOLEAN	Prend la valeur <b>TRUE</b> si la dernière extraction renvoie une ligne ; complément de %NOTFOUND
%ROWCOUNT	NUMBER	Prend la valeur correspondant au nombre total de lignes renvoyées jusqu'à présent

# Modification des données

- ❖ Se fait habituellement avec **INSERT**, **UPDATE** ou **DELETE**
- ❖ Possibilité d'utiliser la clause **FOR UPDATE** dans la déclaration du curseur. Cela permet d'utiliser la clause

**CURRENT OF** *nom-curseur*

dans la clause **WHERE** des instructions **UPDATE** et **DELETE**. Cela permet de modifier la ligne du curseur traitée par le dernier **FETCH**, et donc d'accélérer l'accès à cette ligne

# Example

```
set serveroutput on;  
DECLARE  
    Cursor c_emp  
    IS  
        SELECT * from employees where department_id=30 for update;  
  
BEGIN  
    For v_emp_rec IN c_emp  
    loop  
        DBMS_OUTPUT.PUT_LINE(v_emp_rec.first_name||' '||v_emp_rec.salary||' '||v_emp_rec.department_id);  
    end loop;  
  
END;
```

## Dans une autre session

```
update employees set salary =salary+1000;
```

# Example

```
[-] DECLARE
    Cursor c1
    IS
        SELECT * from employees for update ;

    BEGIN

[-]     For r IN c1
        loop
            update employees set salary = salary+1000
            where first_name=r.first_name;
        end loop;

    END;
```

```
[-] DECLARE
    Cursor c1
    IS
        SELECT * from employees for update ;

    BEGIN

[-]     For r IN c1
        loop
            update employees set salary = salary+1000
            where CURRENT of c1;
        end loop;

    END;
```

# Gestion des erreurs (erreurs standard)

Le langage PL/SQL offre au développeur un mécanisme de gestion des exceptions. Il permet de préciser la logique du traitement des erreurs survenues dans un bloc PL/SQL. Il s'agit donc d'un point clé dans l'efficacité du langage qui permettra de protéger l'intégrité du système. Il existe deux types d'exception:

- **Interne:** les exceptions internes sont générées par le moteur du système (division par zéro, connexion non établie, table inexistante, privilèges insuffisants, mémoire saturée,...),
- **Externe:** les exceptions externes sont générées par l'utilisateur

# Gestion des erreurs (erreurs standard)

Les erreurs Oracle générées par le noyau sont numérotées (ORA-xxxxx). Il a donc fallu établir une table de correspondance entre les erreurs ORACLE et des noms d'exceptions

Voici quelques exemple d'exceptions prédéfinis et des codes correspondants:

Nom d'exception	Erreur ORACLE	SQLCODE
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
ZERO_DIVIDE	ORA-01476	-1476

# Gestion des erreurs (erreurs standard)

```
set SERVEROUTPUT on;
```

```
DECLARE
```

```
    v_l_name employees.last_name%type;
```

```
BEGIN
```

```
    select last_name into v_l_name from employees
```

```
    where first_name='&first_name';
```

```
    dbms_output.put_line(v_l_name);
```

```
Exception
```

```
    when no_data_found then
```

```
    dbms_output.put_line('le nom de l''employée saisi n''existe pas dans la base de données');
```

```
    when too_many_rows then
```

```
    dbms_output.put_line('plusieurs employés ont le même nom dans la base de données');
```

```
END;
```

## Gestion des erreurs (erreurs standard)

- ❖ La nature d'une erreur peut être connue par appel aux fonctions **SQLCODE** et **SQLERRM**
- ❖ **SQLCODE** renvoie le statut d'erreur de la dernière instruction SQL exécutée (0 si n'y a pas d'erreur)
- ❖ **SQLERRM** renvoie le message d'erreur correspondant à **SQLCODE**



# Gestion des erreurs (erreurs standard)

```
set SERVEROUTPUT on;  
DECLARE  
BEGIN  
    insert into departments (department_id, department_name) values (200, null);  
Exception  
    when no_data_found then  
        dbms_output.put_line('le nom de l''employée saisi n''existe pas dans la base de données');  
    when too_many_rows then  
        dbms_output.put_line('plusieurs employés ont le même nom dans la base de données');  
    when others then  
        dbms_output.put_line('autres erreurs');  
        dbms_output.put_line(sqlcode);  
        dbms_output.put_line(sqlerrm);  
END;
```

# Gestion des erreurs (erreurs standard)

```
set SERVEROUTPUT on;
DECLARE
  ex_insert exception;
  pragma exception_init(ex_insert, -1400);
BEGIN
  insert into departments (department_id, department_name) values (200, null);
  Exception
    when no_data_found then
      dbms_output.put_line('le nom de l''employée saisi n''existe pas dans la base de données');
    when too_many_rows then
      dbms_output.put_line('plusieurs employés ont le même nom dans la base de données');
  When ex_insert then
    DBMS_OUTPUT.PUT_LINE('vous ne pouvez pas laissez ce champs vide');
    when others then
      dbms_output.put_line('autres erreurs');
      dbms_output.put_line(sqlcode);
      dbms_output.put_line(sqlerrm);
END;
```

# Exception utilisateur

**DECLARE**

*Nom\_erreur* **EXCEPTION**;

**BEGIN**

...

**IF** (anomalie) **THEN**

**RAISE** *Nom\_erreur*;

**END IF**;

...

**EXCEPTION**

**WHEN** *Nom\_erreur* **THEN** *traitement*;

....

**End**;

# Example

```
set SERVEROUTPUT on;
DECLARE
    v_comm employees.commission_pct%type;
    ex_null exception;
BEGIN
    select commission_pct into v_comm from employees
    where employee_id=&emp_id;
    if v_comm is not null then
        dbms_output.put_line(v_comm);
    else
        raise ex_null;
    end if;
Exception
    when ex_null then
        dbms_output.put_line('l'employé n'a pas de commission');
END;
```

# Erreurs anonymes

- ❖ Pour les codes d'erreur n'ayant pas de nom associé, il est possible de définir un nom d'erreur (code entre -20000 et -20999)

# Erreurs anonymes

```
set SERVEROUTPUT on;
DECLARE
  v_comm employees.commission_pct%type;
BEGIN
  select commission_pct into v_comm from employees
  where employee_id=&emp_id;
  if v_comm is not null then
    dbms_output.put_line(v_comm);
  else
    raise_application_error(-20001,'l''employé n''a pas de commission');
  end if;
Exception
  when others then
    dbms_output.put_line(sqlerrm);
END;
```

# Les procédures

# L'objectif

- ❖ Définition d'une procédure
- ❖ Créer une procédure
- ❖ Faire la distinction entre **les paramètres formels** et **les paramètres réels**
- ❖ Répertorier les fonctions des différents **modes des paramètres**
- ❖ Créer des procédures avec des paramètres
- ❖ Appeler une procédure
- ❖ Traiter des exceptions dans les procédures
- ❖ Supprimer une procédures



# Définition d'une procédure

- ❖ une procédure est un type de sous-programme qui exécute une action
- ❖ Une procédure peut être stockée en tant qu'objet de schéma dans la base de données en vue d'exécutions répétées

# Pourquoi les procédures?

- ❖ **Réduire le trafic sur le réseau** (les procédures sont locales sur le serveur)
- ❖ Masquer la **complexité du code SQL** ( simple appel de procédure avec passage d'arguments)
- ❖ Mieux garantir **l'intégrité des données** (encapsulation des données par les procédures)
- ❖ **Sécuriser l'accès aux données** (accès à certaines tables seulement à travers les procédures)
- ❖ **Optimiser le code** (les procédures sont compilées avant l'exécution du programme et elles sont exécutées immédiatement si elles se trouvent dans la SGA (Zone mémoire gérée par ORACLE). De plus une procédure peut être exécutée par plusieurs utilisateurs.

# Syntaxe pour la création de procédures

```
CREATE [OR REPLACE] PROCEDURE nom_procedure  
[(parameter1 [mode1] datatype1,  
[parameter2 [mode2] datatype2,  
...])]  
IS  
PL/SQL BLOCK;
```

- ❖ *L'option REPLACE indique que, si la procédure existe, elle sera supprimée et remplacée par la nouvelle version créée avec l'instruction*
- ❖ *Le bloc PL/SQL commence par **Begin** ou par la **déclaration de variables** locales et se termine pas **end** ou par **END Procedure\_name***

# Paramètres Formels/Réels

- ❖ *Les paramètres Formels sont des variables déclarées dans la liste de paramètres d'une spécifications de sous-programme*

*Exemple:*

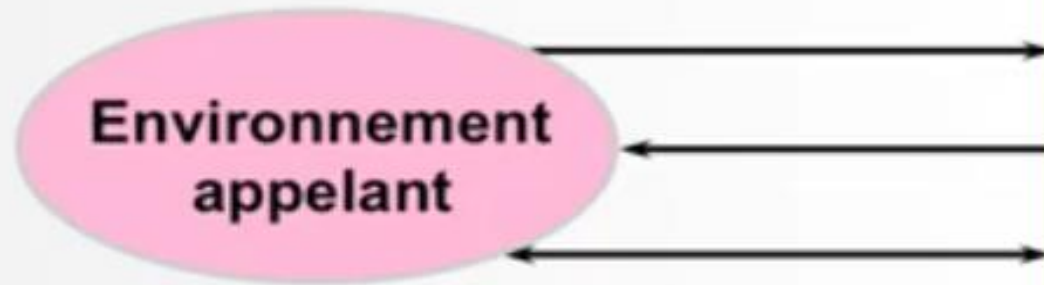
```
CREATE PROCEDURE Moy_sal ( Emp_id NUMBER, Emp_sal NUMBER)
....
END Moy_sal;
```

- ❖ *Les paramètres réels sont des variables ou des expressions référencées dans la liste de paramètres d'un appel de sous-programme*

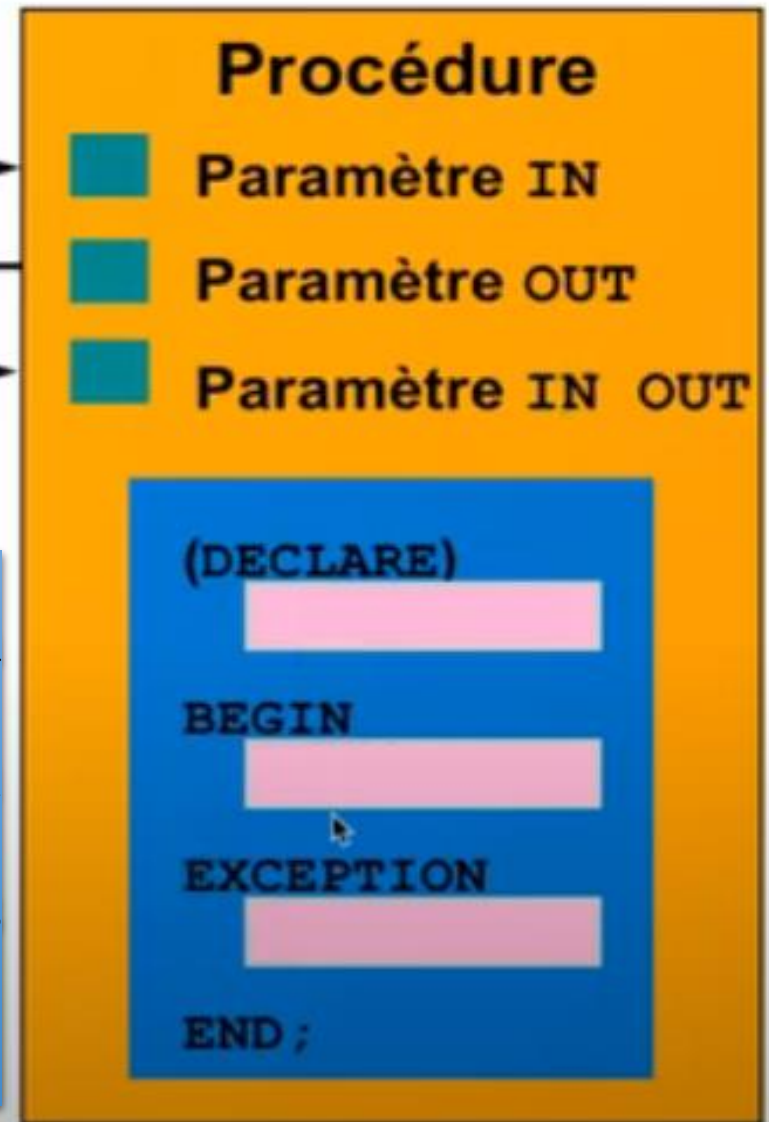
*Exemple:*

```
Moy_sal(v_id, 2000)
```

# Modes des paramètres des procédures



RQ: Datatype ne peut être que la définition %type ou %Rowtype, ou un type de données explicite sans spécifications de taille



# Modes des paramètres des procédures

IN	OUT	IN OUT
Mode par défaut	Doit être indiqué	Doit être indiqué
La valeur est transmise au sous-programme	Est renvoyé à l'environnement appelant	Est transmis à un sous-programme ; est renvoyé à l'environnement appelant
Le paramètre formel se comporte en constante	Variable non initialisée	Variable initialisée
Le paramètre réel peut être un littéral, une expression, une constante ou une variable initialisée	Doit être une variable	Doit être une variable

Par défaut, le paramètre **IN** est transmis par **référence** et pour les paramètres **OUT** et **IN OUT** sont transmis par **valeur**

## Exemple de paramètre IN:

```
CREATE OR REPLACE PROCEDURE modifier_salaire
  (Emp_id IN employees.employee_id%TYPE)
IS
BEGIN
  UPDATE employees
  SET    salaire = salaire * 1.10
  WHERE  employee_id = Emp_id;
END modifier_salaire;
```



## Exemple de paramètre OUT:

```
CREATE OR REPLACE PROCEDURE query_emp
  (p_id      IN    employees.employee_id%TYPE,
   p_nom      OUT   employees.nom%TYPE,
   p_salaire  OUT   employees.salaire%TYPE,
   p_comm     OUT   employees.commission%TYPE)
IS
BEGIN
  SELECT    nom, salaire, commission
  INTO      p_nom, p_salaire, p_comm
  FROM      employees
  WHERE     employee_id = p_id;
END query_emp;
```



# Exemple de paramètre IN OUT:

Environnement appelant



```
CREATE OR REPLACE PROCEDURE format_phone
  (p_phone_no IN OUT VARCHAR2)
IS
BEGIN
  p_phone_no := '(' || SUBSTR(p_phone_no,1,3) ||
                ')' || SUBSTR(p_phone_no,4,3) ||
                '-' || SUBSTR(p_phone_no,7,4) ;
END format_phone;
```

# Déclarer des sous-programmes:

```
CREATE OR REPLACE PROCEDURE leave_emp2
  (p_id IN employees.employee_id%TYPE)
IS
  PROCEDURE log_exec
  IS
  BEGIN
    INSERT INTO log_table (user_id, log_date)
    VALUES (USER, SYSDATE);
  END log_exec;
BEGIN
  DELETE FROM employees
  WHERE employee_id = p_id;
  log_exec;
END leave_emp2;
/
```

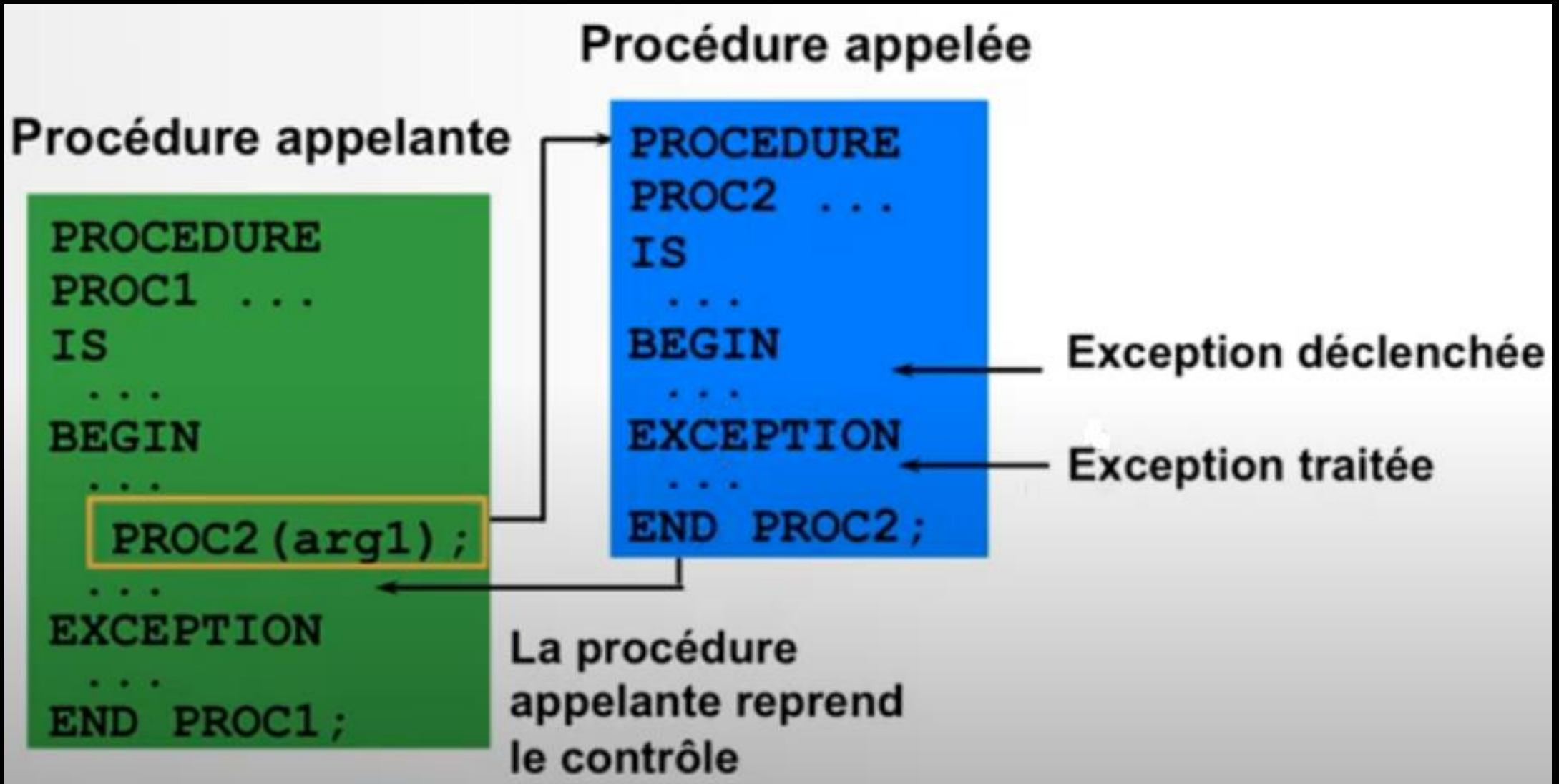
Appeler une procédure depuis un bloc PL/SQL anonyme:

```
DECLARE
    v_id NUMBER := 163;
BEGIN
    Modifier_salaire(v_id);           --Appel procedure
    COMMIT;
    ...
END;
```

## Appeler une procédure depuis une autre procédure:

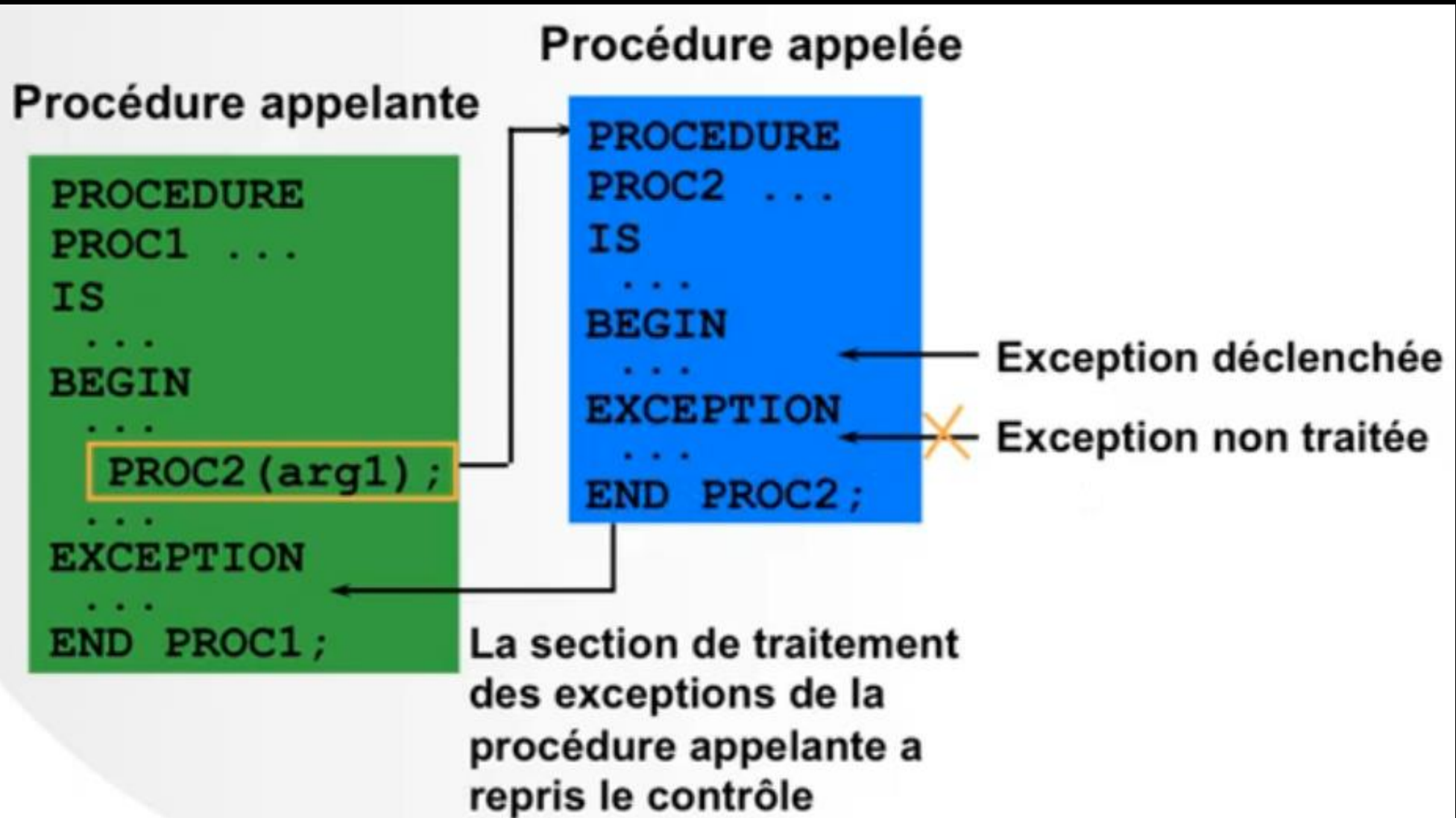
```
CREATE OR REPLACE PROCEDURE process_emps
IS
    CURSOR emp_cursor IS
        SELECT employee_id FROM employees;
BEGIN
    FOR emp_rec IN emp_cursor
    LOOP
        Modifier_salaire(emp_rec.employee_id);
    END LOOP;
    COMMIT;
END process_emps;
/
```

## Appeler une procédure depuis une autre procédure:





# Appeler une procédure depuis une autre procédure:



## Question:

**Créer une procédure qui affiche le nombre des employées qu'on le department\_id=30**

## Exemples:

```
[-] Create Or replace procedure modifier_nom(emp_id IN employees.employee_id%type)
IS
BEGIN
    Update employees set first_name='jad' WHERE employee_id=emp_id;
END modifier_nom;
```

```
----- Appeler la procédure dans un autre fichier PL/SQL
[-] declare
entier integer:=141;
Begin
modifier_nom(entier);
end;
```



## Examples:

```
[-] Create Or replace procedure modifier_comm (emp_id  IN employees.employee_id%type,
                                              v_comm    OUT employees.commission_pct%type)
IS
BEGIN
    select commission_pct into v_comm from employees
    where employee_id=emp_id;
END modifier_comm;
-----Exécution de la procédure-----
Variable x_comm number;
execute modifier_comm(147, :x_comm);
print x_comm;
```

```
----- Appeler la procédure dans un autre fichier PL/SQL
[-] declare
N_commission employees.employee_id%type:=147;
taux employees.commission_pct%type;
Begin
modifier_comm(N_commission, taux);
end;
```

## Exemples:

```
[-] Create Or replace procedure changer_tel(phone IN OUT varchar2)
is
Begin
    dbms_output.put_line('le numéro de téléphone avant: ' || phone);
    phone:='(' || Substr(phone,1,3) || ') ' || Substr(phone, 4,3) || '-' || Substr(phone, 7,4);
    dbms_output.put_line('le numéro de téléphone avant: ' || phone);
end changer_tel;
-----Exécution de la procédure-----
Variable tel varchar2;
execute tel:='212666666600';
print tel;
execute changer_tel(tel);
```

----- Appeler la procédure dans un autre fichier PL/SQL

```
[-] declare
tel varchar2(15) :='212666559256';
Begin
changer_tel(tel);
end;
```

# Exemples avec exception

```
[-] Create Or replace procedure modifier_comm (emp_id  IN employees.employee_id%type,
                                             v_comm   OUT employees.commission_pct%type)
IS
ex_null exception;
BEGIN
    select commission_pct into v_comm from employees
    where employee_id=emp_id;
    if v_comm is not null then
        dbms_output.put_line('la commision est:' || v_comm);
    else
        raise ex_null;
    end if;
exception
    when ex_null then
        dbms_output.put_line('l''employé n''a pas de commision');
END modifier_comm;
-----Exécution de la procédure-----
Variable x_comm number;
execute modifier_comm(141,:x_comm);
```

----- Appeler la procédure dans un autre fichier PL/SQL

```
[-] declare
N_commission employees.employee_id%type:=147;
taux employees.commission_pct%type;
Begin
modifier_comm(N_commission, taux);
end;
```

# Supprimer une procédure dans la base de données:

**Syntaxe:**

```
DROP PROCEDURE procedure_name;
```

**Exemple:**

```
DROP PROCEDURE Modifier_salaire;
```

# Les fonctions

## Les objectifs:

- Décrire les différentes utilisations des fonctions
- Créer des fonctions stockées
- Appeler une fonction
- Supprimer une fonction
- Faire la comparaison entre une procédure et une fonction

# Introduction

- ❖ Une fonction est un bloc PL/SQL nommé qui **renvoie une valeur**
- ❖ Une fonction peut être stockée en tant **qu'objet de schéma** dans la base de données en vue d'exécutions répétées
- ❖ Une fonction est appelée dans une expression

# Syntaxe de création de la fonction

```
CREATE [OR REPLACE] FUNCTION nom_fonction  
[(argument [IN] type, ...)]  
RETURN type-retour  
[ IS | AS]  
bloc-fonction;
```

- ❖ Les paramètres sont forcément en entrée (**IN**)
- ❖ Dans le *bloc-fonction* :

```
RETURN nom-variable;
```



# Example

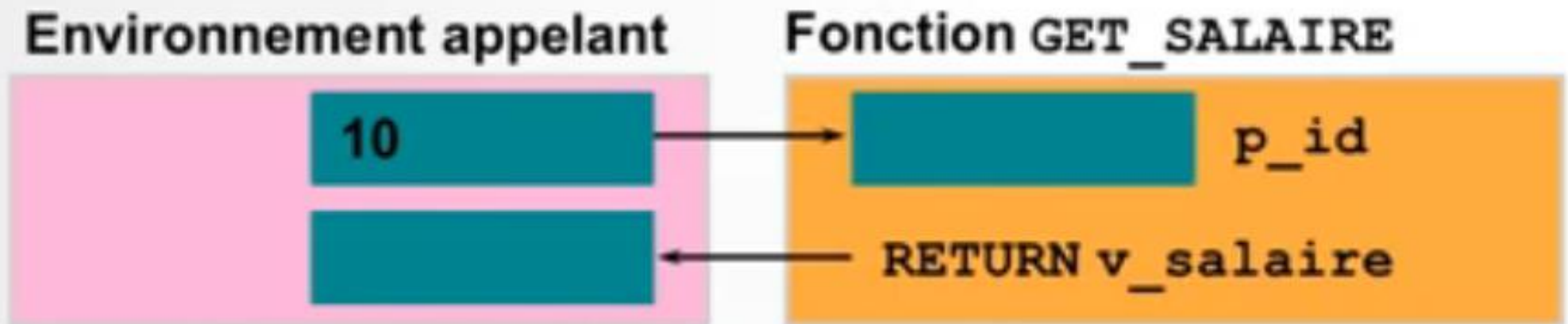
```
create or replace function get_salaire  
    (p_id IN employees.employee_id%type)  
    return number  
  
is  
    v_salaire employees.salary%type:=0;  
  
BEGIN  
    select salary into v_salaire from employees  
    where employee_id=p_id;  
    return v_salaire;  
END get_salaire;
```

# Exécution des fonctions

- ❖ Appeler une fonction dans une expression PL/SQL
- ❖ Créer une variable destinée à recevoir la valeur renvoyée
- ❖ Exécuter la fonction
- ❖ La valeur renvoyée par l'instruction RETURN sera placée dans la variable

Remarque: **Evitez d'utiliser les modes OUT et IN OUT avec les fonctions**

# Exécution des fonctions



- 1 → Charger et exécuter le fichier `get_salary.sql` pour créer la fonction
- 2 → 

```
VARIABLE salaire NUMBER
```
- 3 → 

```
EXECUTE :salaire := get_salaire(10)
```
- 4 → 

```
PRINT salaire
```

# Les avantages des fonctions définies par l'utilisateur dans les expressions SQL

- ❖ Elles **complètent le langage SQL** en permettant de réaliser des traitements qui seraient trop complexes, voire impossibles en SQL
- ❖ **Utilisées dans la clause WHERE** pour filtrer les données, elles peuvent s'avérer plus efficaces qu'un filtrage au sein de l'application
- ❖ Elles permettent de **manipuler les chaînes de caractères**

# Example

```
create or replace function tax
    (p_value IN number)
    return number
is
BEGIN
    return (p_value*0.08);
END tax;
```

```
----- Appel fonction
```

```
select employee_id, first_name, salary, tax(salary)
from employees where department_id=100;
```

# Exemple

```

Create Or replace procedure affiche_info (n integer)
IS
    var_nom employees.last_name%type;
    var_prenom employees.first_name%type;
BEGIN

    if tax(n)<500 then
        select first_name, last_name into var_prenom, var_nom from employees where employee_id=n;
        dbms_output.put_line( var_nom || ' ' || var_prenom);
    else
        dbms_output.put_line( 'tous les employés ont une taxe supérieure à 500');
    end if;
end affiche_info;
```

```

----- Appel la fonction
set SERVEROUTPUT on;
accept id prompt 'entrer le code de l''employé'
declare
emp_id employees.employee_id%type:=&id;
Begin
    dbms_output.put_line('début de la procédure');
affiche_info(emp_id);
    dbms_output.put_line('fin de la procédure');

end;
```

# L'emplacement d'appel des fonctions définies par l'utilisateur

- ❖ Liste de sélection d'une commande **SELECT**
- ❖ Condition des clauses **WHERE** et **HAVING**
- ❖ Clause Values de la commande **INSERT**
- ❖ Clause SET de la commande **UPDATE**

# Restrictions relatives à l'appel des fonctions à partir d'expression SQL

Pour pouvoir être appelée depuis des expressions SQL, une fonction définie par l'utilisateur doit:

- ❖ être une **fonction stockée** (ce n'est pas le cas pour les procédures stockées)
- ❖ En tant que paramètres, accepter uniquement **des types de données SQL valides** (et non des types spécifiques au langage PL/SQL)
- ❖ **Renvoyer des types de données SQL** valide et non des types spécifiques au langage PL/SQL
- ❖ Les fonctions appelées depuis des expressions SQL (SELECT, UPDATE, DELETE en parallèle) **ne peuvent modifier** aucune tables de la BDD



# Supression d'une procédure/fonction stockée

**DROP FUNCTION** *nom\_fonction*;

**EXAMPLE:**

**DROP FUNCTION** get\_salaire;

# Comparer les procédures et les fonctions

Procédures	Fonctions
S'exécutent en tant qu'instruction PL/SQL	Sont appelées dans une expressions
Ne contiennent pas de clause RETURN dans l'entête	Doivent contenir une clause RETURN dans l'entête
Peuvent transférer zéro, une ou plusieurs valeurs	Doivent renvoyer une seule valeur
Peuvent contenir une instruction RETURN	Doivent contenir au moins une instruction RETURN

Soit le schéma relationnel suivant :

- Client (NC, Nom, Prenom, age, ville)
- Produit (NP, marque, prix, #NC)

**Table Client**

NC	Nom	Prenom	Age	ville
1	Nejjar	Ahmed	19	Fès
2	Samah	Kawtar	20	Casa
3	Aourir	Tarik	34	Settat
4	Ahmadi	Adil	40	Rabat
5	jawad	Sofia	40	Marrakech

**Table Produit**

NP	Marque	Prix	NC
13	Lexmark	1200	4
14	Compaq	10000	5
11	Epson	3200	1
12	HP	2300	2
16	HP	2500	3
15	MAC	15000	1
17	MAC	10000	2
18	SUMSUNG	1000	4

## Exercices

1. Ecrire une procédure qui insère le total des achats de chaque client dans une table déjà créée Table\_CA\_Client (numeroclient number, total number)
2. Ecrire une fonction qui calcule le total des achats d'un client donné
3. Ecrire une procédure qui supprime les clients qui n'ont pas réalisé un total achat>3000dhs
4. Ecrire une fonction qui renvoie le nombre des produits achetés par un client donné (le numéro du client est entré comme paramètre)
5. Ecrire une procédure qui stocke les noms des clients qui ont acheté au minimum 2 produits, dans un tableau indexé par des entiers

La procédure doit utiliser :

- Un curseur pour parcourir les noms des clients
- Un tableau contenant les noms des clients qu'ont acheté au minimum 2 produits
- Les exceptions pour gérer les erreurs (des données inexistantes, curseur incorrect,...).

# Les Packages

# Présentation des packages

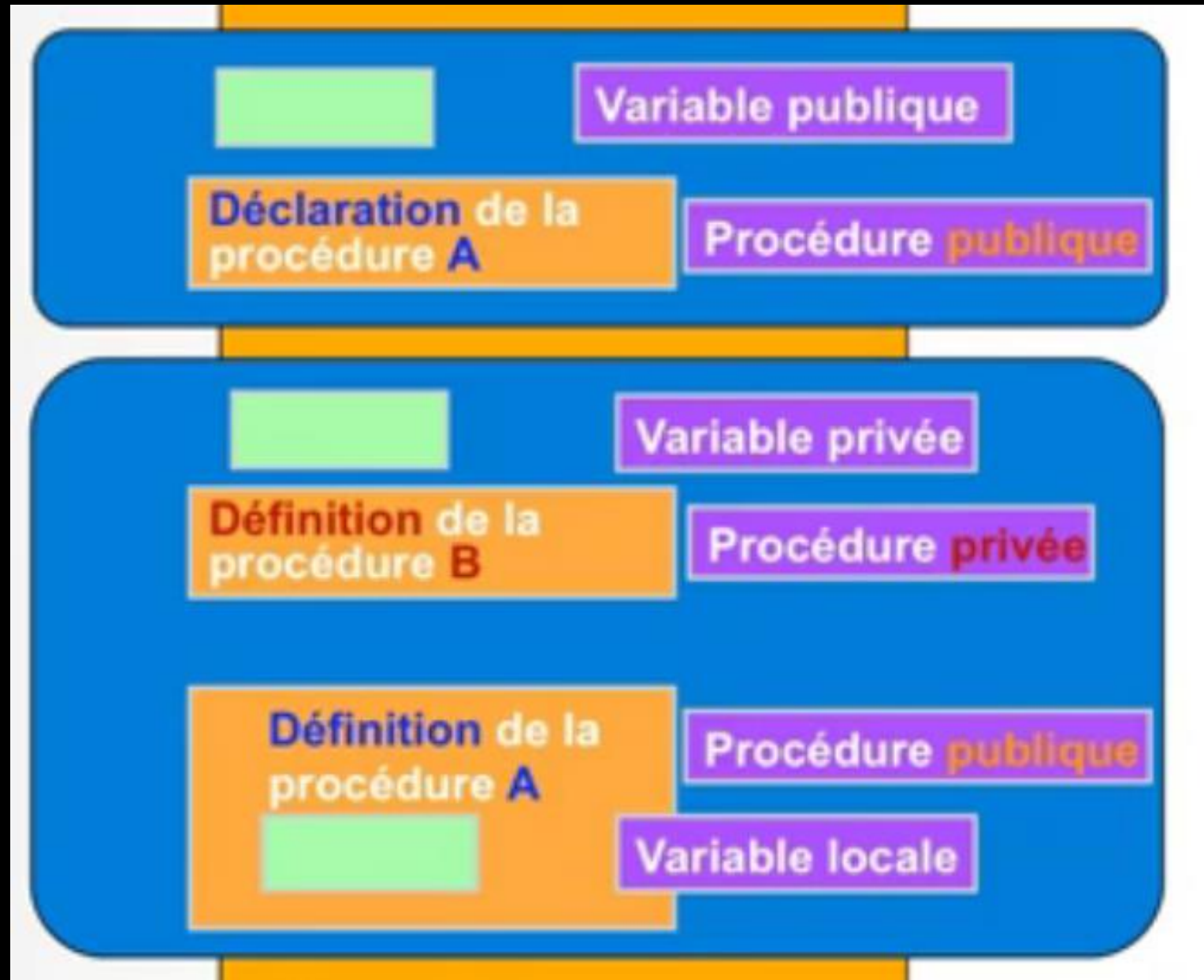
Les packages:

- ❖ Regroupent des types PL/SQL, des éléments et des sous-programmes présentant une relation logique
- ❖ Sont constitués de deux éléments:
  - ➔ **Spécification (Déclaration)**
  - ➔ **Corps (définition)**
- ❖ Ne peuvent pas être appelés, paramétrés ou imbriqués
- ❖ Permettent au serveur Oracle de lire simultanément plusieurs objets en mémoire

# Composants d'un package

## Spécification du package

## Corps du package



# Développer un package

L'enregistrement du texte de l'instruction CREATE PACKAGE dans deux fichiers SQL distincts facilite les modifications ultérieures du package

→ **Spécification: CREATE PACKAGE**

→ **Corps: CREATE PACKAGE BODY**



# Créer la spécification du package

## Syntaxe:

**CREATE** [Or REPLACE] **PACKAGE** package\_name

**IS/ AS**

Définition des types utilisés dans le package;

prototype de toutes les procédures et fonctions du package;

**END** package\_name;

- ❖ L'option Replace supprime et recrée la spécification du package
- ❖ Toutes les structures déclarées dans une spécification de package peuvent être visibles pas les utilisateurs disposant de privilèges sur le package

# Créer la spécification du package

## Spécification du package



# Créer la spécification du package

```
CREATE OR REPLACE PACKAGE PACKAGE_Salaire
IS
    global_v NUMBER := 0.10;  --initialisé à 0.10
    PROCEDURE Public_procedure(p_comm IN NUMBER);
END PACKAGE_Salaire;
/
```

## Spécification du package:

- **G\_var** est une variable globale dont la valeur d'initialisation est 0,10
- **Public\_procedure** est une procédure publique implémentée dans le corps du package

# Créer le corps du package

## Syntaxe:

**CREATE** [Or **REPLACE**] **PACKAGE BODY** package\_name

**IS/ AS**

Déclaration de variables privées;

**END** package\_name;

- ❖ L'option **Relpace** supprime et recrée le corps du package
- ❖ Les identificateurs définis exclusivement dans le corps du package sont des structures privées. Ils ne sont pas visibles à l'extérieur du corps du package
- ❖ Toutes les structures privées doivent être déclarées avant d'être utilisées dans les structures publiques

# Composants d'un package

Spécification du package

Corps du package



# Corps d'un package

## Exemple:

```
CREATE OR REPLACE PACKAGE BODY AGE_PACKAGE
IS
    FUNCTION validate_age (Par_age IN NUMBER)
        RETURN BOOLEAN
    IS
        v_max_age    NUMBER;
    BEGIN
        SELECT      MAX(age)
          INTO      v_max_age
        FROM        client;
        IF    Par_age > v_max_age THEN RETURN(FALSE) ;
        ELSE    RETURN(TRUE) ;
        END IF;
    END validate_age;
    ...
```

# Corps d'un package

## Exemple:

```
PROCEDURE  initialiser_age (p_age    IN  NUMBER)
IS
BEGIN
  IF  validate_age (p_age) <
    THEN    g_age:=p_age;
           DBMS_OUTPUT.PUT_LINE('Age n''est pas valide' );
  END IF;
END  initialiser_age ;
END age_package;
/
```

Appel d'une fonction depuis une procédure du même package.



# Example:

## Spécification du package

```
Create or replace package test
Is
  Var1 Constant number:=10;
  Var2 varchar2(2):= 'ENSA';
  Procedure affichage;
End;
```

## Corps du package

```
Create or replace package body test
Is
  Var3 varchar2(2):= 'FST';
```

```
  Procedure getVar
  Is
  Begin
    DBMS_OUTPUT.PUT_LINE(Var1 );
    DBMS_OUTPUT.PUT_LINE(Var3);
  End;
```

```
  Procedure affichage
  Is
    Var4 varchar2(2):= 'EST';
  Begin
    getVar();
    DBMS_OUTPUT.PUT_LINE(Var2);
    DBMS_OUTPUT.PUT_LINE(Var3);
    DBMS_OUTPUT.PUT_LINE(Var4);
```



# Appeler des structures de package

Appeler une procédure de package depuis SQL\*Plus

**EXECUTE nom\_package.nom\_procedure (valeurs paramètres...)**

Appeler une procédure de package dans un autre schéma

**EXECUTE nom\_utilisateur.nom\_package.nom\_procedure (valeurs paramètres...)**

# Déclarer un package sans corps

```
CREATE OR REPLACE PACKAGE global_var  
IS
```

```
    PI constant Number := 3,14;
```

```
    Age constant Number := 20;
```

```
    R constant Number := 10;
```

```
End globale_var;
```

```
Execute DMNS_output.put_line('age=' || globale_var.age);
```

```
Execute DMNS_output.put_line('Air d'un disque=' ||  
2*globale_var.PI*globale_var.R);
```

# Référencer une variable publique depuis une procédure autonome

```
CREATE OR REPLACE PROCEDURE Air_Disque  
(air OUT NUMBER)  
IS  
BEGIN  
    air:= 2*global_var.PI*global_var.R;  
END;  
/
```

```
VARIABLE air NUMBER  
EXECUTE Air_Disque ( :air)  
PRINT air
```

# Supprimer des packages

La syntaxe de suppression de la spécification et le corps d'un package:

**DROP PACKAGE package\_name;**

La syntaxe de suppression le corps d'un package:

**DROP PACKAGE BODY package\_name;**

# Suppression des packages

Pour afficher les informations sur un package:

```
Select * from user_objects  
where object_name='nom package';
```

Pour afficher le code source d'une spécification

```
Select * from user_source  
Where name='nom package' and type='PACKAGE' ;
```

Pour afficher le code source d'un corps (Body)

```
Select * from user_source  
Where name='nom package' and type='PACKAGE BODY' ;
```

# Avantages liés aux packages

- ❖ **Modularité** : Encapsule les structures associées
- ❖ **Conception simplifiée des applications** : La spécification et le corps sont codés et compilés séparément
- ❖ **Masquage des informations**:
  - Seules les déclarations contenues dans la spécification du package sont **visible et accessibles aux applications**
  - Les structures privées du corps du package sont **masquées et inaccessibles**
  - L'ensemble du code est **masqué dans le corps du package**

# Avantages liés aux packages

## ❖ Performances accrues :

- L'ensemble du package est chargé en mémoire la première fois que celui-ci est référencé
- Une seule copie est chargée en mémoire pour l'ensemble des utilisateurs

## ❖ **Surcharge:** plusieurs sous-programmes portant le même nom

# Exercices

6. Ecrire un package contenant la spécification suivante :
  - Une procédure « add-client » pour insérer les informations d'un client
  - Une procédure « add\_client » pour insérer les informations d'un client sauf la ville
  - Une procédure « get\_client » qui affiche les informations d'un client donné
  - Une fonction « get\_age » qui retourne l'âge d'un client donné
  - Une procédure « delete\_client » qui supprime un client donné
  - Une procédure « delete\_client » pour supprimer les clients dont l'âge est dans la liste {20,30,40}
7. Implémenter le corps de toutes les spécifications



# Les déclencheurs ou Triggers

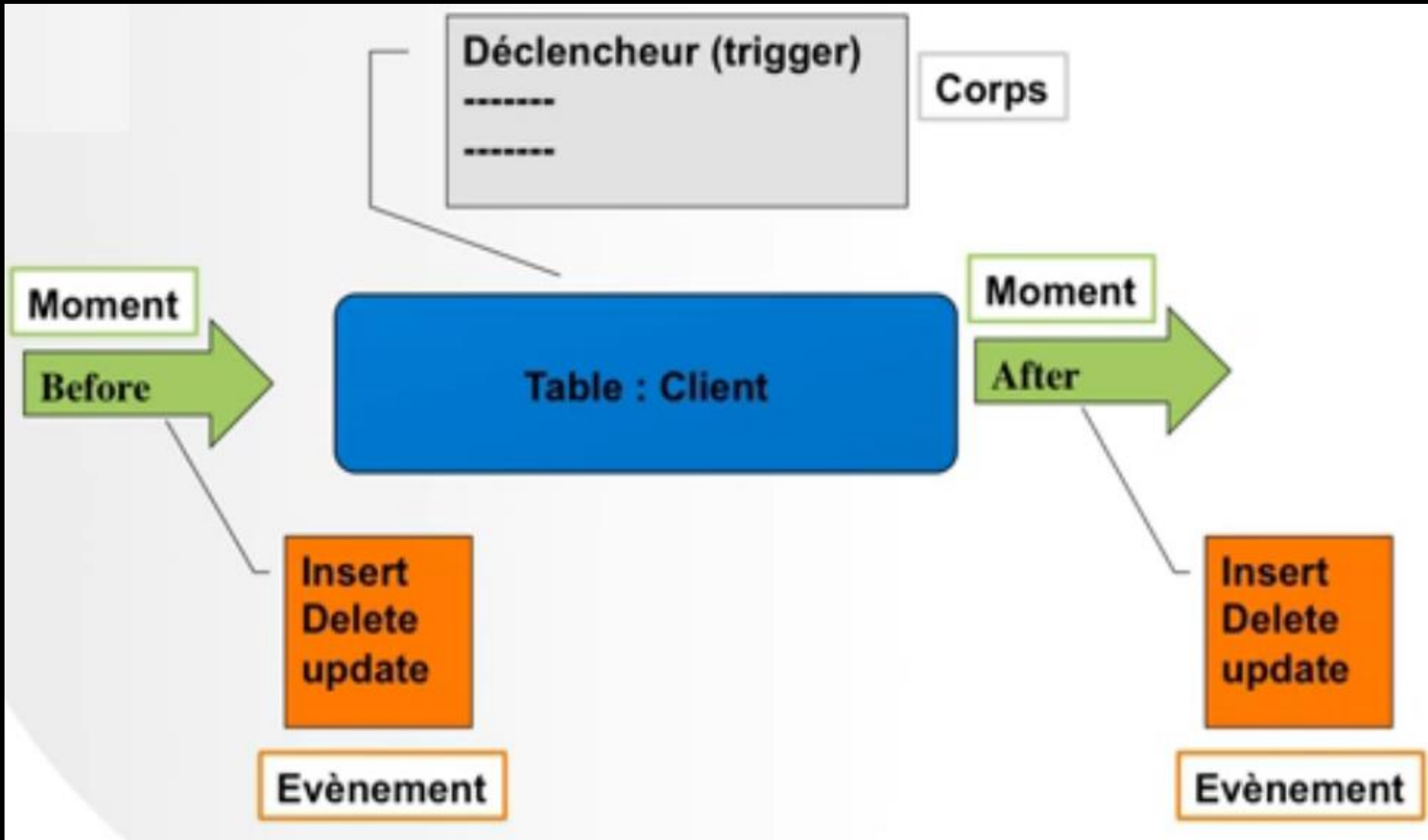
# Les objectifs

- ❖ Décrire différents types de déclencheurs/trigger
- ❖ Décrire les déclencheurs de base de données et leur utilisation
- ❖ Créer des déclencheurs de base de données
- ❖ Décrire les règles d'activation des déclencheurs de base de données
- ❖ Supprimer des déclencheurs de base de données

# Règles relatives à la conception de déclencheurs

- ❖ Il est conseillé de concevoir des déclencheurs pour:
  - exécuter des actions associées
  - Centraliser des opérations globales
- ❖ Si le code PL/SQL est très long, **Créer des procédures stockées**, et les appeler dans un déclencheur
- ❖ L'utilisation excessive de déclencheurs peut entraîner des interdépendances complexes dont la gestion peut s'avérer difficile dans les applications volumineuses

# Les éléments d'un déclencheur



# Créer des déclencheurs LMD

Une instruction de déclenchement comporte les éléments suivants:

1. Moment du déclenchement
  - **BEFORE** (Avant), **AFTER** (Après),
2. Événement déclencheur : **Insert, UPDATE ou DELETE**
3. Nom de la table: sur la table ou la vue
4. Type de déclencheur: ligne ou instruction
5. Clause **WHEN**: Condition restrictive par ligne
6. Corps du déclencheur: bloc PL/SQL

# Composants des déclencheurs LMD

## 1. Moment

Moment de déclenchement: à quel moment le déclencheur doit-il s'exécuter?

- ❖ **BEFORE:** exécution du corps du déclencheur avant le déclenchement de l'événement LMD sur une table
- ❖ **AFTER:** exécution du corps du déclencheur après le déclenchement de l'événement LMD sur une table

# Composants des déclencheurs LMD

## 2. Événement

Événement utilisateur déclencheur: quelle instruction LMD entraîne l'exécution du déclencheur?

Vous pouvez utiliser les instructions suivantes

❖ **INSERT**

❖ **UPDATE**

❖ **DELETE**

# Composants des déclencheurs LMD

## 3. Type

**Type de déclencheur:** le corps du déclencheur doit-il s'exécuter une seule fois ou pour chaque ligne concernée par l'instruction?

- ❖ **Instruction:** le corps du déclencheur s'exécute une seule fois pour l'événement déclencheur. Il s'agit du comportement par défaut. Un déclencheur sur instruction s'exécute une fois, même si aucune ligne n'est affectée
- ❖ **Ligne:** le corps du déclencheur s'exécute une fois pour chaque ligne concernée par l'événement déclencheur. Un déclencheur sur ligne ne s'exécute pas si l'événement déclencheur n'affecte aucune ligne



# Composants des déclencheurs LMD

## 4. Corps

**Corps du déclencheur:** Quelle action le déclencheur doit-il effectuer?

Le corps du déclencheur est un bloc PL/SQL ou un appel de procédure (PL/SQL ou Java)

### Remarque:

- ❖ Les déclencheurs sur ligne utilisent des noms de corrélation pour accéder aux anciennes ou nouvelles valeurs de colonne de la ligne en cours de traitement
- ❖ La taille d'un déclencheur est limitée à **32 Ko**

# Séquence d'exécution

## Manipulation concerne une seule ligne

### Instruction LMD

```
INSERT INTO departments (department_id,  
                        department_name, location_id)  
VALUES (400, 'CONSULTING', 2400);
```

### Action de déclenchement

DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID
10	Administration	1700
20	Marketing	1800
30	Purchasing	1700
...		
400	CONSULTING	2400

→ Déclencheur sur  
instruction BEFORE

→ Déclencheur sur ligne BEFORE

→ Déclencheur sur ligne AFTER

→ Déclencheur sur instruction  
AFTER

# Séquence d'exécution

Manipulation concerne une plusieurs lignes

```
UPDATE employees  
  SET salary = salary * 1.1  
  WHERE department_id = 30;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
114	Raphaely	30
115	Khoo	30
116	Baida	30
117	Tobias	30
118	Himuro	30
119	Colmenares	30

→ Déclencheur sur **instruction BEFORE**

→ Déclencheur **sur ligne BEFORE**

→ Déclencheur **sur ligne AFTER**  
...

→ Déclencheur **sur ligne BEFORE**

→ Déclencheur **sur ligne AFTER**  
...

→ Déclencheur sur **instruction AFTER**

# Création de déclencheurs sur une instruction LMD

## Syntaxe:

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing - - BEFORE ou AFTER
    event1 [OR event2 OR event3] - - INSERT, UPDATE, DELETE
    ON table_name
trigger_body
```

**Remarque:** Les noms des déclencheurs doivent être unique au sein d'un même schéma

# Création de déclencheurs sur une instruction LMD

## Syntaxe:

```
CREATE OR REPLACE TRIGGER secure_client
BEFORE INSERT OR DELETE ON client

BEGIN
  IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
     (TO_CHAR(SYSDATE,'HH24:MI')
      NOT BETWEEN '08:00' AND '18:00')
  THEN
    Raise_application_error(-20011,' Vous ne pouvez pas faire une mise
à jour d'un client dans ce temps');
  END IF;
END;
```

**Remarque:** En cas d'échec d'un déclencheur de base de données, l'instruction de déclenchement est annulée



# Prédicats conditionnels

```
CREATE OR REPLACE TRIGGER secure_client
BEFORE INSERT OR DELETE OR UPDATE ON client

BEGIN
  IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
     (TO_CHAR(SYSDATE,'HH24:MI') NOT BETWEEN '08:00' AND '18:00')
  THEN
    IF DELETING THEN
      Raise_application_error(-20011,' Vous ne pouvez pas supprimer un client dans ce
temps');
    ELSIF INSERTING THEN
      Raise_application_error(-20012, ' Vous ne pouvez pas insérer un client dans ce
temps');
    ELSIF UPDATING ('nom') THEN
      Raise_application_error(-20013, Vous ne pouvez pas modifier le nom d'un client dans
ce temps');
    ELSE
      dbms_output.put_line(' Vous pouvez modifier la table client seulement dans l'horaire
normal ');
    END IF;
  END IF;
END;
```

# Création de déclencheurs sur ligne LMD

## Syntaxe:

```
CREATE [OR REPLACE] TRIGGER trigger_name
    timing
    event1 [OR event2 OR event3]
    ON table_name
    [REFERENCING OLD AS old | NEW AS new]
FOR EACH ROW
    [WHEN (condition)]
    trigger_body
```

## Utilisation: New et OLD

- Si nous ajoutons un client dont le nom est Ahmed alors nous récupérons ce nom grâce à la variable: **New.nom**
- Dans le cas de suppression ou modification, les anciennes valeurs sont dans la variable: **OLD.nom**

# Création de déclencheurs sur ligne LMD

## Exemple:

```
CREATE OR REPLACE TRIGGER restrict_salaire
  BEFORE UPDATE OF salaire ON employees
  FOR EACH ROW
  BEGIN
    IF :NEW.salaire > 15000
    THEN
      Raise_application_error (-20001, ' Vous ne pouvez pas modifier le
salaire de cet employé ');
    END IF;
  END;
```

```
UPDATE employees
  SET salaire = 15500
WHERE nom = 'Essoufi';
```

**Vous ne pouvez pas modifier le salaire de cet employé**



# Utilisation des qualificatifs OLD et NEW

```
CREATE OR REPLACE TRIGGER audit_emp_values
  AFTER DELETE OR INSERT OR UPDATE ON employees
  FOR EACH ROW
BEGIN
  INSERT INTO audit_emp_table (user_name, times_user,
    id, Ancien_nom, nouveau_nom, Ancienne_fonction,
    nouvelle_fonction, Ancien_salaire, nouveau_salaire)
  VALUES (USER, SYSDATE, :OLD.employee_id,
    :OLD.nom, :NEW.nom, :OLD.fonction,
    :NEW.fonction, :OLD.salaire, :NEW.salaire );
END;
```

# Restreindre l'action d'un déclencheur sur ligne

```
CREATE OR REPLACE TRIGGER emp_salaire
  BEFORE INSERT OR UPDATE OF salaire ON employees
  FOR EACH ROW
  WHEN (NEW.numero_employe = 1)
BEGIN

  IF UPDATING ('salaire') THEN
    IF :NEW.salaire < :OLD.salaire THEN
      raise_application_error (-20001, 'Attention,
Diminution De salaire ');
    END IF;
  END IF;
END;
```

# Différences entre les déclencheurs et les procédures

Déclencheurs	Procédures
<p>Définis via la commande <b>CREATE TRIGGER</b></p> <p>Le dictionnaire de données contient le code source dans <u><b>USER_TRIGGERS</b></u></p> <p>Appel implicite</p> <p>Les instructions COMMIT, et ROLLBACK ne sont pas autorisées</p>	<p>Définis via la commande <b>CREATE PROCEDURE</b></p> <p>Le dictionnaire de données contient le code source dans <u><b>USER_SOURCE</b></u></p> <p>Appel explicite</p> <p>Les instructions COMMIT, et ROLLBACK sont autorisées</p>

# Tables système

❖ **USER\_TRIGGERS**

❖ **ALL\_TRIGGERS**

❖ **DBA\_TRIGGERS**

# Suppression

**DROP TRIGGER** *nom-déclencheur*;

Exemple:

**Drop TRIGGER secure\_emp;**

Remarque: lorsqu'une table est supprimée, tous ses déclencheurs sont également supprimés

# Activation/Désactivation des déclencheurs

Désactiver ou réactiver un déclencheur de base de données :

❖ **ALTER TRIGGER** *nom\_déclencheur* **DISABLE/ENABLE;**

Désactiver ou réactiver tous les déclencheurs d'une table :

❖ **ALTER TABLE** *nom\_table* **DISABLE /ENABLE ALL TRIGGERS;**

# Restrictions

- Un déclencheur ne peut modifier la valeur d'un attribut déclaré avec l'une des contraintes **PRIMARY KEY**, **UNIQUE** ou **FOREIGN KEY**
- Un déclencheur ne peut pas consulter les données d'une table en *mutation* : une *table en mutation* est une table directement ou indirectement concernée par l'événement qui a provoqué la mise en œuvre du déclencheur