

# **CHAPITRE 2:**

## **Programmation avancée en**

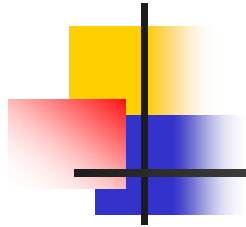
### **BD: PL/SQL**



# Pourquoi PL/SQL ?

---

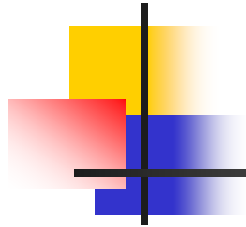
- PL/SQL = PROCEDURAL LANGUAGE/SQL
- SQL est un langage non procédural
- Les traitements complexes sont parfois difficiles à écrire si on ne peut utiliser des variables et les structures de programmation comme les boucles et les alternatives
- On ressent vite le besoin d'un langage procédural pour lier plusieurs requêtes SQL avec des variables et dans les structures de programmation habituelles



# Principales caractéristiques

---

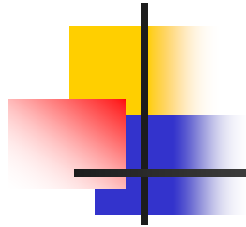
- Extension de SQL : des requêtes SQL cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles)
- Un programme peut être constitué de procédures et de fonctions
- Des variables permettent l'échange d'information entre les requêtes SQL et le reste du programme



# Utilisation de PL/SQL

---

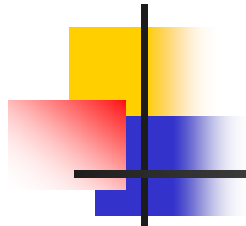
- PL/SQL peut être utilisé pour l'écriture des procédures stockées et des triggers
- Il convient aussi pour écrire des fonctions utilisateurs qui peuvent être utilisées dans les requêtes SQL (en plus des fonctions prédéfinies)
- Il est aussi utilisé dans des outils Oracle
  - Ex : Forms et Report



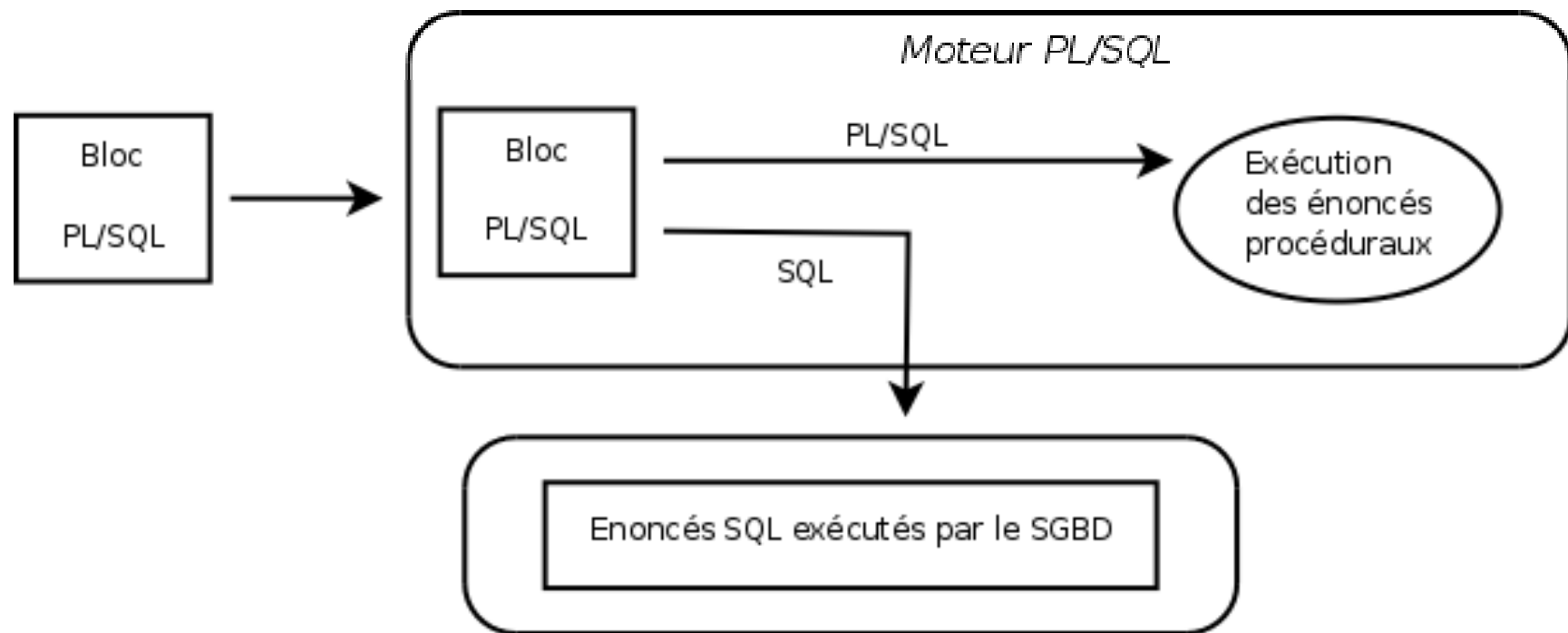
# Normalisation du langage

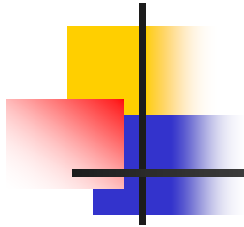
---

- Langage spécifique à Oracle
- Tous les SGBD ont un langage procédural
  - TransacSQL pour SQLServer,
  - PL/pgSQL pour Postgresql
  - Procédures stockées pour MySQL depuis 5.0
- Tous les langages L4G des différents SGBDs se ressemblent
- Ressemble au langage normalisé PSM (Persistant Stored Modules)



# Environnement PL/SQL





# Blocs

---

- Un programme est structuré en blocs d'instructions de 3 types
  - bloc anonymes
  - procédures nommées
  - fonctions nommées
- Un bloc peut contenir d'autres blocs
- Un bloc ne peut être vide. Il doit contenir une instruction (il peut donc contenir l'instruction NULL)

# Structure d'un bloc anonyme

Seuls BEGIN et END sont obligatoires

```
DECLARE
    -- définition des variables
BEGIN
    -- code du programme
    (instruction SQL ou PL/SQL)
EXCEPTION
    -- code de gestion des
    erreurs
END ;
```

Comme les instruction SQL, les  
bloc se terminent par un ;





# Déclaration, initialisation des variables

---

- Identificateurs Oracle :
  - 30 caractères au plus
  - commence par une lettre
  - peut contenir lettres, chiffres, \_, \$ et #
  - pas sensible à la casse
- Portée habituelle des langages à blocs
- Doivent être déclarées avant d'être utilisées



# Déclaration, initialisation des variables

- Déclaration et initialisation

Nom variable type variable := valeur;

- Initialisation

Nom variable := valeur;

- Déclaration multiple interdite

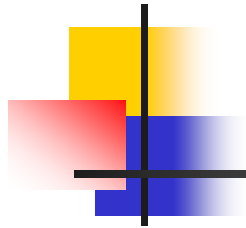
- Exemples :

age integer;

nom varchar(30);

dateNaissance date;

ok boolean := true;



# Initialisation de variables

- Plusieurs façons de donner une valeur à une variable

- **Opérateur d'affectation**

n : =

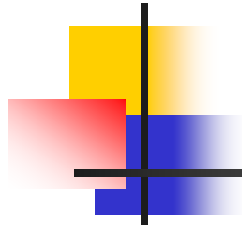
- **Directive INTO** de la requête SELECT

- Exemples :

```
dateNaissance := to_date(' 10/10/2004', 'DD/MM/YYYY');
```

```
SELECT nom INTO v_nom  
FROM emp  
WHERE matr = 509;
```

Pour éviter les conflits de  
nommage, préfixer les  
variables PL/SQL par v\_



# SELECT ... INTO ...

---

- **SELECT** expr1,expr2,... **INTO** var1, var2,...
- Met des valeurs de la BD dans une ou plusieurs variables var1, var2, ...
- Le select ne doit retourner **qu'une seule ligne**
- Avec Oracle il n'est pas possible d'inclure un select sans «into » dans une procédure
- Pour retourner plusieurs lignes, voir la suite du cours sur les curseurs.



# Le type de variables

---

- VARCHAR2
  - Longueur maximale : 32767 octets
  - Syntaxe:  
Nom variable VARCHAR2(30);
  - Exemple:  
Name VARCHAR2 (30);  
Name VARCHAR2 (30) := 'toto' ;
- NUMBER(long,dec)
  - Long : longueur maximale
  - Dec : longueur de la partie décimale
  - Exemple  
num\_tel number(10);  
toto number(5,2)=142.12;



# Le type de variables

---

- DATE
- Par défaut DD-MON-YY (18-DEC-02)
- Fonction TO\_DATE

- Exemple:

```
start _date := to _date(' 29-SEP-2003' , ' DD-MON-  
YYYY' );
```

```
start _date := to _date(' 29-SEP-2003:13:01' , ' DD-  
MON-YYYY:HH24:MI' );
```

- BOOLEAN
- TRUE
- FALSE
- NULL



# Déclaration %TYPE et %ROWTYPE

---

- On peut déclarer qu'une variable est du même type qu'une colonne d'une table (ou qu'une autre variable)

- Exemple :

```
v_nom emp.nom%TYPE;
```

- Une variable peut contenir toutes les colonnes d'une ligne d'une table

- Exemple :

```
v_employe emp%ROWTYPE;
```

déclare que la variable v\_employe contiendra une ligne de la table emp



# Exemple d'utilisation

---

```
DECLARE
    v_employe emp%ROWTYPE;
    v_nom emp.nom%TYPE;

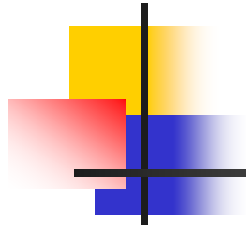
BEGIN
    SELECT * INTO v_employe
    FROM emp
    WHERE matr = 900;

    v_nom := v_employe . nom;
    v_employe.dept := 20;

    INSERT into emp VALUES v_employe;

END;
```



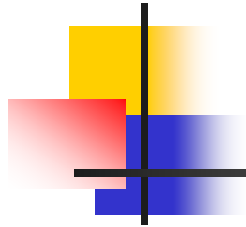


# Commentaires

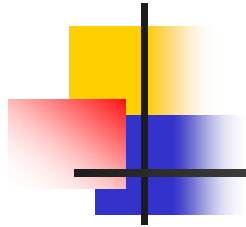
---

-- Pour une fin de ligne

/\* Pour plusieurs lignes \*/



# Les principales commandes



# Test conditionnel

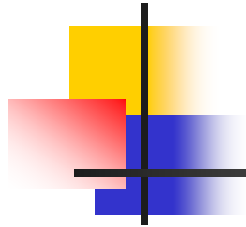
---

- IF-THEN

- **IF** v \_date > '11-APR-03' **THEN**  
    v\_salaire := v\_salaire \* 1.15;  
**END IF;**

- IF-THEN-ELSE

- **IF** v \_date > '11-APR-03' **THEN**  
    v\_salaire := v\_salaire \* 1.15;  
**ELSE**  
    v\_salaire := v \_salaire \* 1.05;  
**END IF;**



# Test conditionnel

---

- IF-THEN-ELSIF

- **IF** v\_nom = 'PAKER' **THEN**  
    v\_salaire := v\_salaire \* 1.15;  
**ELSIF** v\_nom = 'ASTROFF' **THEN**  
    v\_salaire := v \_salaire \* 1.05;  
**END IF**;



# Test conditionnel

Le CASE renvoie une valeur qui vaut  
résultat1 ou résultat2 ou ....

Ce n'est pas une instruction.

- CASE
  - CASE sélecteur

```
WHEN expression1 THEN résultat1
WHEN expression2 THEN résultat2
ELSE résultat3
END ;
```
- Exemple
  - val := CASE city

```
WHEN 'TORONTO' THEN 'RAPTORS'
WHEN 'LOS ANGELES' THEN 'LAKERS'
ELSE 'NO TEAM'
END ;
```



# Les boucles

---

- **LOOP**

*instructions exécutables;*

**EXIT** [**WHEN** condition];

**END LOOP;**

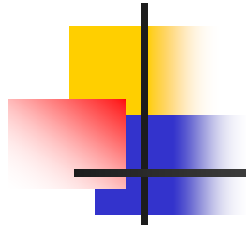
- Obligation d'utiliser la commande **EXIT** pour éviter une boucle infinie.

- **WHILE** condition **LOOP**

*instructions exécutables;*

**END LOOP;**

- -- tant que la condition est vraie les instructions sont répétées



# Les boucles

---

- **FOR** variable **IN** debut..**fin**

**LOOP**

instructions;

**END LOOP;**

- La variable de boucle prend successivement les valeurs de début, debut+1, debut+2, ..., jusqu'à la valeur fin.  
Ne pas déclarer la variable de boucle, elle est déclarée implicitement.
- On pourra également utiliser un curseur dans la clause IN  
(voir plus loin)



# Affichage

---

- Affichage
  - `dbms_output.put_line (chaîne);`
  - Utiliser `||` pour faire une concaténation

```
DECLARE
  v_fname VARCHAR2(25);
BEGIN
  SELECT first_name INTO v_fname
  FROM employees WHERE employee_id=200;
  DBMS_OUTPUT.PUT_LINE(' First Name is : ' || v_fname);
END;
/
```





# Exemple

---

```
DECLARE
```

```
    nb integer;
```

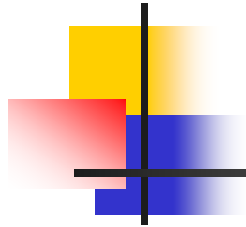
```
BEGIN
```

```
    delete from emp where matr in (600, 610);
```

```
    nb := sql%rowcount;                -- curseur sql
```

```
    dbms_output.put_line('nb = ' || nb);
```

```
END;
```



# Exemple

---

```
DECLARE
```

```
compteur number (3);
```

```
BEGIN
```

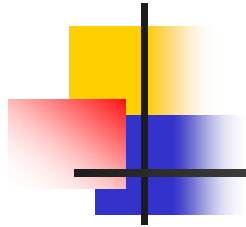
```
    select count(*) into compteur from clients;
```

```
    FOR i IN 1..compteur LOOP
```

```
        dbms_output.put_line('Nombre : ' || i );
```

```
    END LOOP;
```

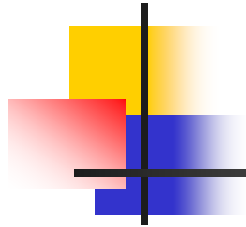
```
END;
```



# Les curseurs

---

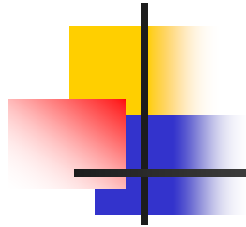
- **Toutes** les requêtes SQL sont associées à un curseur
- Ce curseur représente la zone mémoire utilisée pour *parser* (*analyser*) et exécuter la requête
- Le **curseur** peut être **implicite** (pas déclaré par l'utilisateur) ou explicite
- Les **curseurs explicites** servent à retourner plusieurs lignes avec un select



# Les curseurs

---

- Un curseur est un pointeur vers la zone de mémoire privée allouée par le serveur Oracle. Il est utilisé pour gérer l'ensemble de résultats d'une instruction `SELECT`.
- Il existe deux types de curseur : implicite et explicite.
  - Implicite : créé et géré en interne par le serveur Oracle afin de traiter les instructions SQL
  - Explicite : déclaré explicitement par le programmeur



# Attributs des curseurs

---

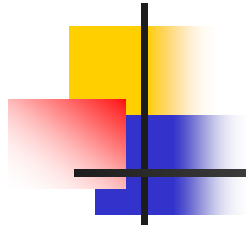
Tous les curseurs ont des attributs que l'utilisateur peut utiliser

**%ROWCOUNT** : nombre de lignes traitées par le curseur

**%FOUND** : vrai si au moins une ligne a été traitée par la requête ou le dernier fetch

**%NOTFOUND** : vrai si aucune ligne n'a été traitée par la requête ou le dernier fetch

**%ISOPEN** : vrai si le curseur est ouvert (utile seulement pour les curseurs explicites)



# Les curseurs implicites

---

- Les curseurs implicites sont tous nommés SQL

## **Exemple :**

```
DECLARE
```

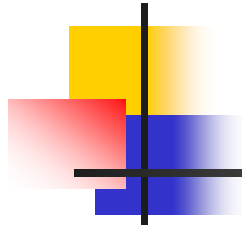
```
nb _lignes integer;
```

```
BEGIN
```

```
    delete from emp where dept = 10;
```

```
    nb_lignes := SQL%ROWCOUNT;
```

```
    ...
```

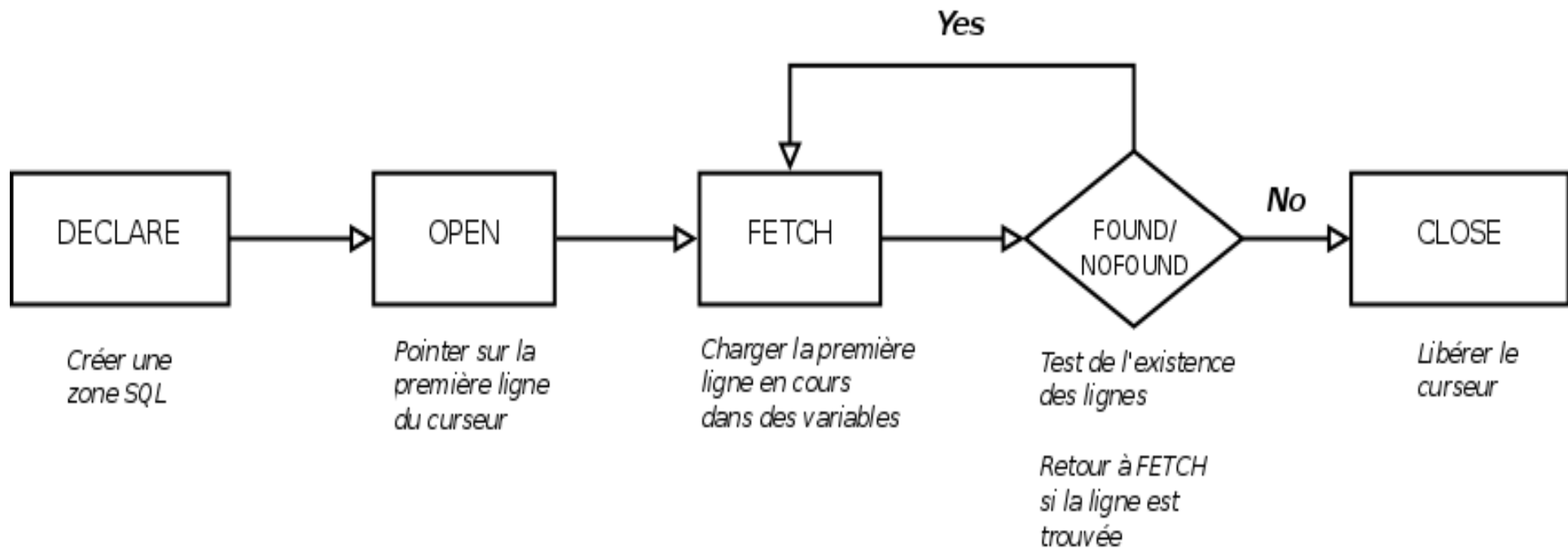


# Les curseurs explicites

---

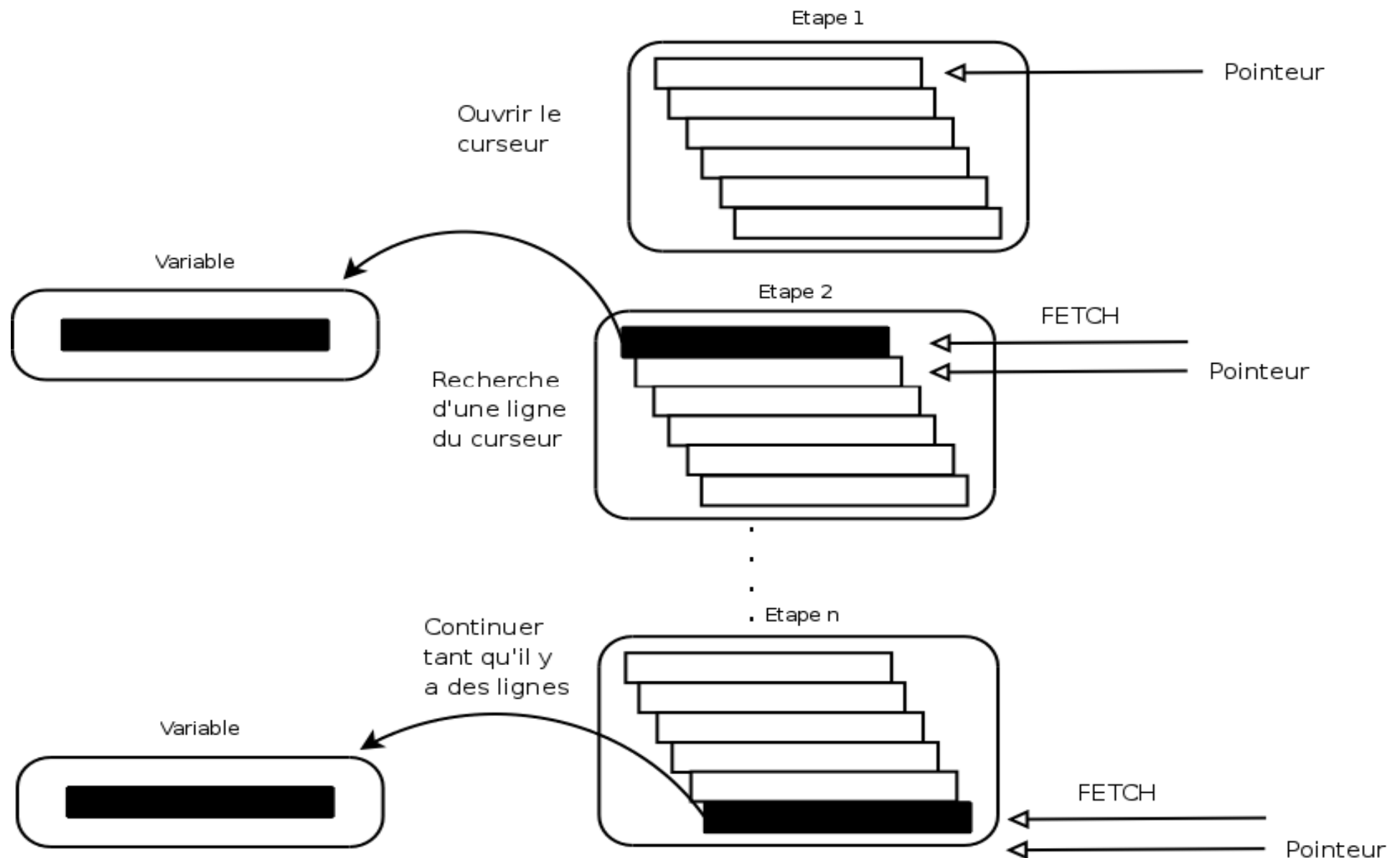
- Servent à traiter les select qui renvoient **plusieurs lignes**
- Les curseurs doivent **être déclarés**
- Le code doit les utiliser explicitement avec les ordres OPEN, FETCH et CLOSE
  - OPEN nomcurseur : ouvre le curseur.
  - FETCH nomcurseur : avance le curseur à la ligne suivante.
  - CLOSE nomcurseur : referme le curseur.
- Le plus souvent on les utilise dans une boucle dont on sort quand l'attribut NOTFOUND du curseur est vrai
- On les utilise aussi dans une **boucle FOR** qui permet une utilisation implicite des instructions OPEN, FETCH et CLOSE

# Contrôle des curseurs explicites





# Contrôle des curseurs explicites





## Déclaration d'un curseur (Synthaxe)

---

`CURSOR nom_du_curseur IS`  
`un énoncé SELECT;`

- Ne pas inclure la clause INTO dans la déclaration du curseur
- Si le traitement des lignes doit être fait dans un ordre spécifique, on utilise la clause ORDER BY dans la requête



## Déclaration d'un curseur (Exemple)

---

```
DECLARE  
CURSOR C1 IS  
    SELECT RefAr t , NomArt , QteArt  
    FROM Article  
    WHERE QteAr t < 500;
```



## Ouverture d'un curseur (Synthaxe)

---

OPEN nom du curseur ;

- Ouvrir le curseur pour exécuter la requête et identifier l'ensemble actif
- Utiliser les attributs des curseurs pour tester le résultat du FETCH

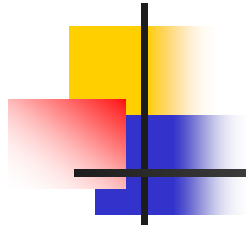


## Recherche des données dans le curseur (Synthaxe)

---

```
FETCHnom du curseur  
INTO [variable1 , [variable2 ,      . . . ];
```

Rechercher les informations de la ligne en cours et les mettre dans des variables.



## Fermeture d'un curseur (Synthaxe)

---

CLOSE nom du curseur;

- Fermer le curseur après la fin du traitement des lignes
- Rouvrir le curseur si nécessaire
- On ne peut pas rechercher des informations dans un curseur si ce dernier est fermé



# Les curseurs explicites

---

```
DECLARE
```

```
    CURSOR c_emp_cursor IS
```

```
        SELECT employee_id, last_name FROM employees
```

```
        WHERE department_id =30;
```

```
    v_empno employees.employee_id%TYPE;
```

```
    v_lname employees.last_name%TYPE;
```

```
BEGIN
```

```
    OPEN c_emp_cursor;
```

```
    LOOP
```

```
        FETCH c_emp_cursor INTO v_empno, v_lname;
```

```
        EXIT WHEN c_emp_cursor%NOTFOUND;
```

```
        DBMS_OUTPUT.PUT_LINE( v_empno || ' ' || v_lname);
```

```
    END LOOP;
```

```
    CLOSE c_emp_cursor;
```

```
END;
```

```
/
```



# Les curseurs explicites

---

On peut déclarer un type « row » associé à un curseur

```
DECLARE
```

```
    cursor c IS
```

```
        select matr, nome, sal from emp;
```

```
    employe c%ROWTYPE;
```

```
BEGIN
```

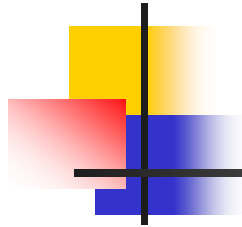
```
    open c;
```

```
    fetch c into employe;
```

```
    if employe.sal is null then ...
```

```
    END;
```

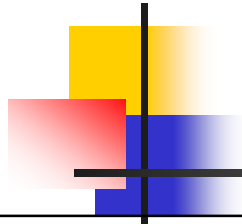




# Boucle FOR pour un curseur

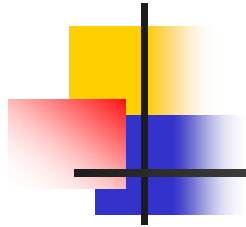
---

- Elle simplifie la programmation car elle évite d'utiliser explicitement les instruction OPEN, FETCH, CLOSE
  - En plus elle déclare implicitement une variable de type ROW associée au curseur



# Boucle FOR pour un curseur

```
DECLARE
  CURSOR c_nom_clients IS
  SELECT nom, adresse FROM clients;
  BEGIN
    FOR le _client IN c _nom _clients
    LOOP
      dbms_output.put_line('Employé : ' ||
        UPPER(le _client.nom) || ' Ville : ' ||
        le_client. adresse);
    END LOOP;
  END;
```



# Curseurs paramétrés

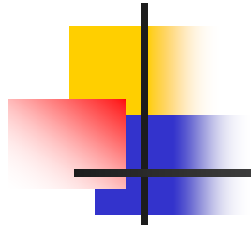
---

- Un curseur paramétré peut servir plusieurs fois avec des valeurs des paramètres différentes
- On doit fermer le curseur entre chaque utilisation de paramètres différents (sauf si on utilise « for » qui ferme automatiquement le curseur)



# Curseurs paramétrés

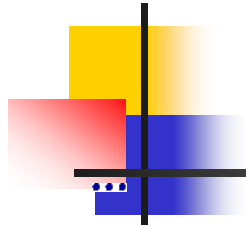
```
DECLARE
  CURSOR c(p_dept integer) is
    select dept, nome from emp where dept = p_dept;
BEGIN
  FOR employe in c(10) LOOP      Instantiation du paramètre
    dbms_output .put _line (employe .nome);
  END LOOP;
  FOR employe in c(20) LOOP
    dbms_output .put _line (employe .nome);
  END LOOP;
END;
```



# Les Exceptions

---

- Une exception est une erreur qui survient durant une exécution
- 2 types d'exception :
  - prédéfinie par Oracle (déclenchées implicitement)
  - définie par le programmeur (déclenchées explicitement)
- Saisir une exception
  - Une exception ne provoque pas nécessairement l'arrêt du programme si elle est saisie par un bloc (dans la partie « EXCEPTION »)
  - Une exception non saisie remonte dans la procédure appelante (où elle peut être saisie)



# Syntaxe de la clause exception

EXCEPTION

WHEN *exception1* [OR *exception2* . . .] THEN

*statement1*;

*statement2*;

. . .

[WHEN *exception3* [OR *exception4* . . .] THEN

*statement1*;

*statement2*;

. . .]

[WHEN OTHERS THEN

*statement1*;

*statement2*;

. . .]



# Exceptions prédéfinies

---

- NO \_DATA \_FOUND
  - Quand Select into ne retourne aucune ligne
- TOO\_MANY\_ROWS
  - Quand Select into retourne plusieurs lignes
- VALUE \_ERROR
  - Erreur numérique
- ZERO\_DIVIDE
  - Division par zéro
- OTHERS
  - Toutes erreurs non interceptées



# Exemple d'exceptions prédéfinies

---

```
BEGIN
```

```
...
```

```
EXCEPTION
```

```
    WHEN NO_DATA_FOUND THEN
```

```
        DBMS_OUTPUT.PUT_LINE (TO_CHAR ( etudno ) || 'Non valide' );
```

```
    WHEN TOO_MANY_ROWS THEN
```

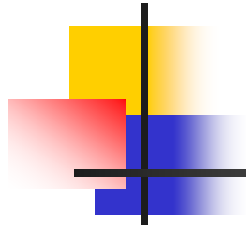
```
        DBMS_OUTPUT.PUT_LINE ( 'Donnees invalides' );
```

```
    WHEN OTHERS THEN
```

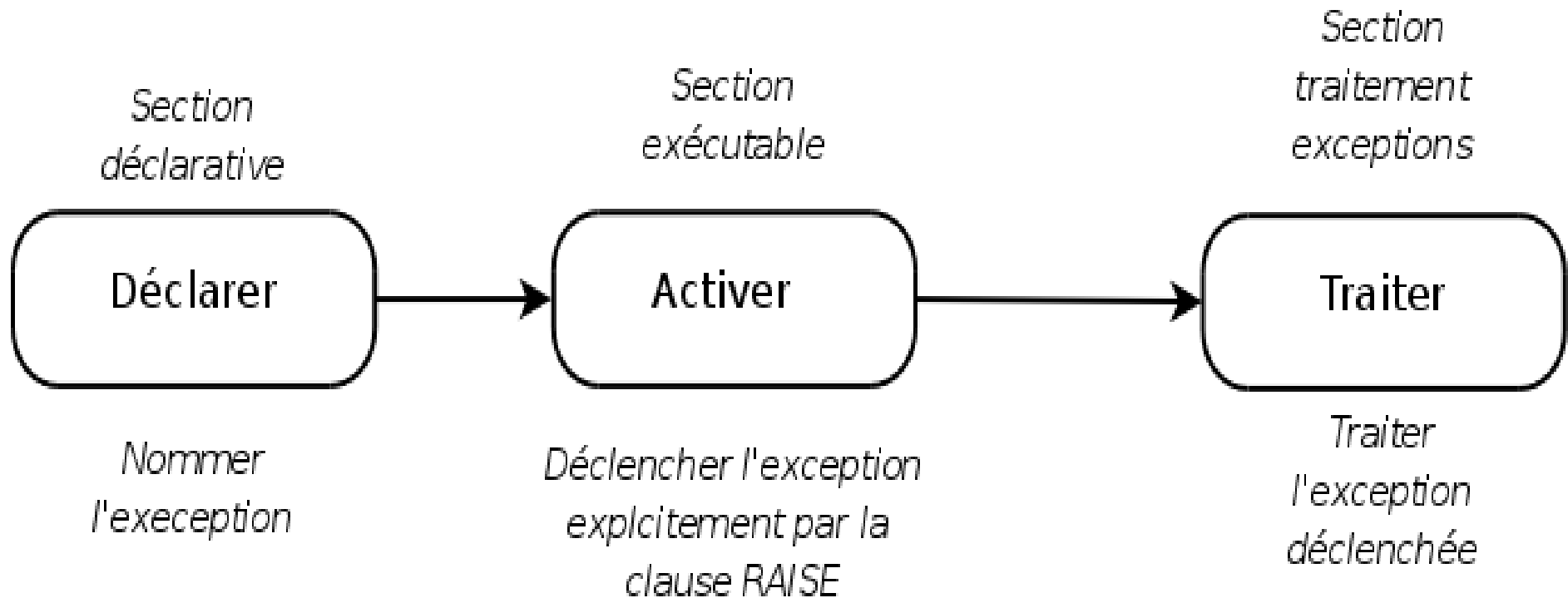
```
        DBMS_OUTPUT.PUT_LINE ( 'Autres erreurs' );
```

```
.....
```





# Exceptions utilisateur

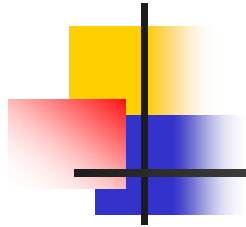




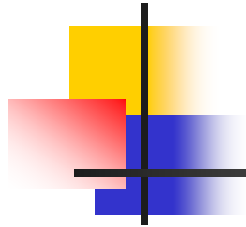
# Exemple d'exception utilisateur

---

```
DECLARE
  salaire numeric(8,2);
  salaire_trop _ bas EXCEPTION;
BEGIN
    select sal into salaire from emp where matr = 50;
    if salaire < 300 then
        RAISE salaire _trop_bas;
    end if;
EXCEPTION
    WHEN salaire_ trop _ bas THEN
        dbms_output.put_line ('Salaire trop bas' );
    WHEN OTHERS THEN
        dbms_output.put_line (SQLERRM) ;
```



# Procédures et fonctions



# Bloc anonyme ou nommé

---

- Un bloc anonyme PL/SQL est un bloc « DECLARE – BEGIN – END » comme dans les exemples précédents
- On peut exécuter directement un bloc PL/SQL anonyme en tapant sa définition
- Le plus souvent, on crée plutôt une procédure ou une fonction nommée pour réutiliser le code
- Une procédure ou une fonction est un bloc PL/SQL nommé puis compilé et stocké dans la base



# Syntaxe pour une procédure

---

- Utilisez la syntaxe `CREATE PROCEDURE` suivie du nom, de paramètres facultatifs et du mot-clé `IS` ou `AS`.
- Ajoutez l'option `OR REPLACE` afin de remplacer une procédure existante.
- Ecrivez un bloc PL/SQL contenant des variables locales, un mot-clé `BEGIN` et un mot-clé `END` (ou `END procedure_name`).

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [(parameter1 [mode] datatype1,
    parameter2 [mode] datatype2, ...)]
IS|AS
  [local_variable_declarations; ...]
BEGIN
  -- actions;
END [procedure_name];
```



# Procédures sans paramètre

---

**create or replace procedure list\_nom\_clients**

**IS**

CURSOR c\_nom\_clients IS select nom, adresse from clients;

BEGIN

FOR le\_client IN c\_nom\_clients LOOP

    dbms\_output.put\_line('Employé : ' || UPPER(le\_client.nom) || ' Ville : '  
    || le\_client.adresse);

END LOOP;

END;



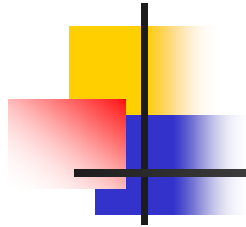
# Procédures avec paramètre

---

```
create or replace procedure list_nom_clients
  (ville IN varchar2,
   result OUT number)
IS

  CURSOR c_nb_clients IS
    select count(*) from clients where adresse=ville;
  BEGIN
    open c_nb_clients;
    fetch c_nb_clients INTO result;
    Close c_nb_clients;

  END;
```



# Fonctions stockées

---

```
CREATE [OR REPLACE] FUNCTION nom fonction
  [(liste parametres formels)]
  RETURN typeRetour AS j IS
  [partie declaration]
BEGIN
  ...
  RETURN valeurRetour
  ...
  [EXCEPTION ...]
END [nom fonction];
```





# Fonctions sans paramètres

---

create or replace **function** nombre\_clients

**return number**

IS

    i number;

    CURSOR get\_nb\_clients IS select count(\*) from clients;

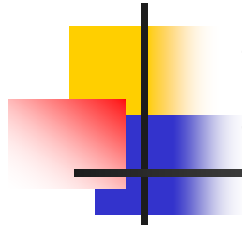
BEGIN

    open get\_nb\_clients;

    fetch get\_nb\_clients INTO i;

**return i;**

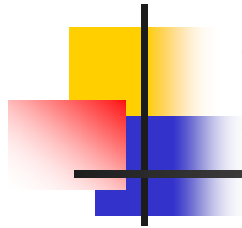
END;



# Fonctions avec paramètres

---

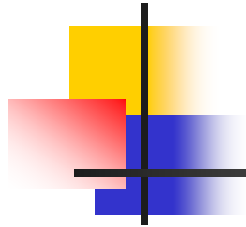
```
create or replace function euro_to_fr(somme IN number)  
return number  
IS  
    taux constant number := 6.55957;  
BEGIN  
    return somme * taux;  
END;
```



# Fonctions avec paramètres

---

```
CREATE OR REPLACE FUNCTION solde (no INTEGER)
RETURN REAL IS
    le_solde REAL ;
BEGIN
    SELECT solde INTO le_solde FROM clients
    WHERE  no_cli=no ;
    RETURN le_solde;
END;
```



# Procédures et fonctions

---

- Suppression de procédures ou fonctions
  - `DROP PROCEDURE nom _procedure`
  - `DROP FUNCTION nom_fonction`
- Table système contenant les procédures et fonctions :  
`user_source`



# Compilation, exécution et utilisation

---

- Compilation
  - Sous SQL\*PLUS, il faut taper une dernière ligne contenant « / » pour compiler une procédure ou une fonction
- Exécution
  - Sous SQL\*PLUS on exécute une procédure PL/SQL avec la commande EXECUTE :
  - EXECUTE *nomProcédure(param 1, ...);*
- Utilisation
  - Les procédures et fonctions peuvent être utilisées dans d'autres procédures ou fonctions ou dans des blocs PL/SQL anonymes
  - Les fonctions peuvent aussi être utilisées dans les requêtes SQL



# Execution de procédure dans SQLPLUS

---

- Déclarer une variable

SQL> variable nb number;

Une variable globale  
s'utilise avec le préfixe :

- Exécuter la procédure

SQL> execute list\_nom\_clients('alger',:nb)

- Visualisation du résultat

SQL> print (ou print nb)

- Description des paramètres


SQL> desc nom\_procedure



# Execution de fonction dans SQLPLUS

---

- Déclarer une variable  
SQL> variable x number;
- Exécuter la fonction  
SQL> execute :x := nombre\_clients
- Visualisation du résultat  
SQL> print       (ou print x)



## Exemple d'appel de procedure dans un bloc PL/SQL

---

...

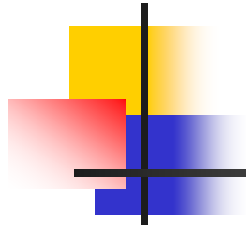
**BEGIN**

**list\_nom\_clients;**

**END;**

**/**

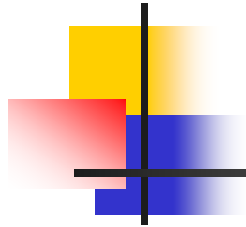




# Les déclencheurs (trigger)

---

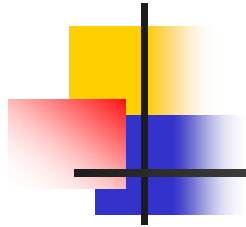
- Programme se déclenchant automatiquement suite à un événement ou au passage à un état donné de la base de données.
- Règles de gestion non modélisables par des contraintes d'intégrité
- Porte sur des tables ou des vues
- Trois parties : a) description de l'évènement, b) conditions éventuelles de déclenchement sur l'évènement, c) action à réaliser



# Les déclencheurs (trigger)

---

- Un trigger est un programme PL/SQL qui s'exécute automatiquement avant ou après une opération LMD (Insert, Update, Delete)
- Contrairement aux procédures, un trigger est déclenché automatiquement suite à un ordre LMD



# Les déclencheurs (trigger)

---

- Automatiser des actions lors de certains événements du type :  
AFTER ou BEFORE et  
INSERT, DELETE ou UPDATE
- Syntaxe :  

```
CREATE OR REPLACE TRIGGER nom _trigger  
BEFORE/AFTER Événement [OF liste colonnes] ON  
nom_table  
[FOR EACH ROW [WHEN (condition) ]]  
Corps du trigger
```



## Composants d'un trigger

---

- A quel moment se déclenche le trigger ?
- BEFORE : le code dans le corps du triggers s'exécute avant les événements de déclenchement LMD
- AFTER : le code dans le corps du triggers s'exécute après les événements de déclenchement LMD



# Composants d'un trigger

---

- Les événements du déclenchement :
- Quelles sont les opérations LMD qui causent l'exécution du trigger ?
  - ✓ INSERT
  - ✓ UPDATE
  - ✓ DELETE
  - ✓ La combinaison des ces operations

UPDATE peut être suivi optionnellement par OF <liste d'attributs> (c'est-à-dire que la modification concerne que certains attributs).



# Composants d'un trigger

---

- Un déclencheur sur instruction :
  - s'exécute une fois pour l'événement déclencheur
  - est le type de déclencheur par défaut
  - s'exécute une fois même si aucune ligne n'est affectée
- Un déclencheur sur ligne :
  - s'exécute une fois pour chaque ligne affectée par l'événement déclencheur
  - n'est pas exécuté si l'événement déclencheur n'affecte aucune ligne
  - est défini à l'aide de la clause `FOR EACH ROW`



## Composants d'un trigger

---

- `<condition>` : toute condition booléenne SQL (elle est optionnelle). `WHEN` permet de limiter l'action du trigger aux lignes qui répondent à une certaine condition.



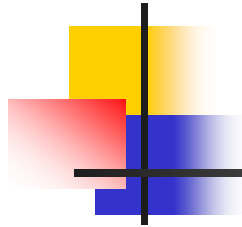
## Composants d'un trigger

---

- Le corps d'un déclencheur :
  - détermine l'action exécutée
  - est un bloc PL/SQL ou un appel (`CALL`) d'une procédure
- Le corps du trigger dans le cas d'un bloc PL/SQL anonyme

```
[DECLARE]
BEGIN
[EXEPTION]
END;
```

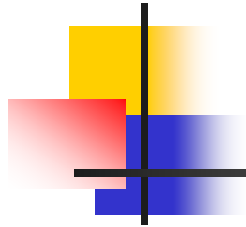




# Accès aux valeurs modifiées

---

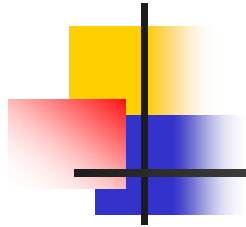
- Utilisation de **new** et **old**
- Si nous ajoutons un client avec un nouveau nom alors nous récupérons ce nom grâce à la variable **:new.nom**
- Dans le cas de suppression ou modification, les anciennes valeurs sont dans la variable **:old.nom**



# Exemple

---

- Archiver le nom de l'utilisateur, la date et l'action effectuée (toutes les informations) dans une table LOG\_CLIENTS lors de l'ajout d'un clients dans la table CLIENTS
- Créer la table LOG\_CLIENTS avec la même structure que CLIENTS
- Ajouter 3 colonnes USERNAME, DATEMODIF, TYPEMODIF



# Exemple

---

create or replace trigger logadd  
after insert on clients  
for each row

BEGIN

    insert into log\_clients values

    (:new.nom,:new.adresse,:new.reference,:new.nom\_piece,  
    :new.quantite,:new.prix,:new.echeance, USER ,SYSDATE,  
    'INSERT');

END;



## ■ Exemple

---

Etudiant (Matr\_Etu , Nom, . . . , Cod\_Cla )

Classe (Cod\_Cla , Nbr\_Etu )

Trigger mettant a jour la table classe suite a une insertion d'un nouvel étudiant

Create or Replace Trigger MajNbEtud

After Insert On Etudiant

For Each Row

Begin

Update Classe Set Nbr Etud = Nbr Etud+1

Where Cod\_Cla =:New. Cod\_Cla ;

End ;

/



# Les prédicats inserting, updating et deleting

---

## ■ Inserting:

- True: Le trigger est déclenché suite à une insertion
- False: Sinon

## ■ Updating:

- True: le trigger est déclenché suite à une mise à jour
- False: sinon

## ■ Deleting:

- True: le trigger est déclenché suite à une suppression
- False: sinon



## Exemple

---

```
Create Or Replace Trigger MajNbEtud
After Insert Or Delete On Etudiant
For Each Row
Begin
    If Inserting Then
        Update Classe Set Nbr Etud = Nbr Etud+1
        where Cod_Cla =:New. Cod_Cla ;
    End If ;
    If Deleting Then
        Update Classe Set Nbr Etud = Nbr Etud-1
        Where Cod_Cla =:Old . Cod_Cla ;
    End If ;
End ;
/
```



## Exemple

---

Film (Titre, Année, metrage, studio, catégorie)

Ecrire un trigger qui permet de ne pas autoriser les tentatives de diminuer le métrage d'un film :

```
CREATE TRIGGER metrage
```

```
    BEFORE UPDATE OF métrage ON FILM
```

```
    FOR EACH ROW
```

```
BEGIN
```

```
    IF old.métrage > new.métrage
```

```
        THEN raise_application_error (n°erreur, "vous ne pouvez pas  
diminuer le métrage »)
```

```
    ENDIF ;
```

Raise\_application\_error est une procédure Oracle qui affiche un message d'erreur et provoque l'échec du bloc du trigger.



## ■ Manipulation des triggers

---

- Activer ou désactiver un Trigger:

Alter Trigger <Nom Trigger> [ Enable |  
disable] ;

- Supprimer un Trigger:

Drop Trigger <Nom\_Trigger >;

- Déterminer les triggers de votre BD:

Select Trigger\_Name From User\_Triggers