

# PL/SQL

Université Sultan Moulay Slimane  
Faculté Poly disciplinaire  
Khouribga  
Année universitaire 2021/2022

# Pourquoi PL/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 de PL/SQL

- Extension de SQL : des requêtes SQL cohabitent avec les structures de contrôle habituelles de la programmation structurée (blocs, alternatives, boucles)
- La syntaxe ressemble au langage Ada
- Un programme est 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 triggers (Oracle accepte aussi le langage Java), procédures stockées, ...
- Il convient aussi pour écrire des fonctions utilisateurs qui peuvent être utilisées dans les requêtes SQL (en plus des fonctions prédefinies)
- Il est aussi utilisé dans des outils Oracle, Forms et Report en particulier

# Normalisation du langage

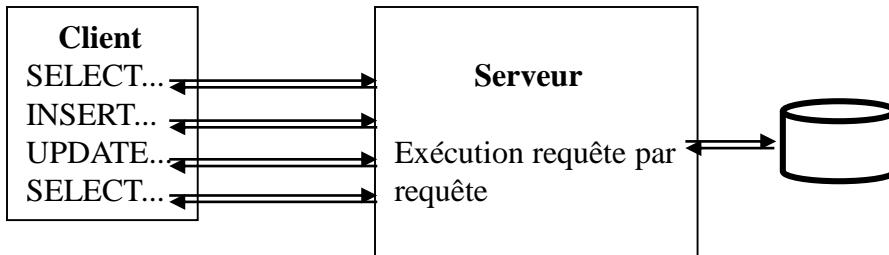
- PL/SQL est un langage propriétaire de Oracle
- PostgreSQL utilise un langage très proche pl/pgsql
- Ressemble au langage normalisé PSM (*Persistent Stored Modules*)
- *Pl/sql est un langage 4g*
- Tous les langages L4G des différents SGBDs se ressemblent

## Définition d'un bloc

- Un programme est structuré en blocs d'instructions de 3 types :
  - 1- procédures anonymes
  - 2- procédures nommées
  - 3- fonctions nommées
- Un bloc peut contenir d'autres blocs

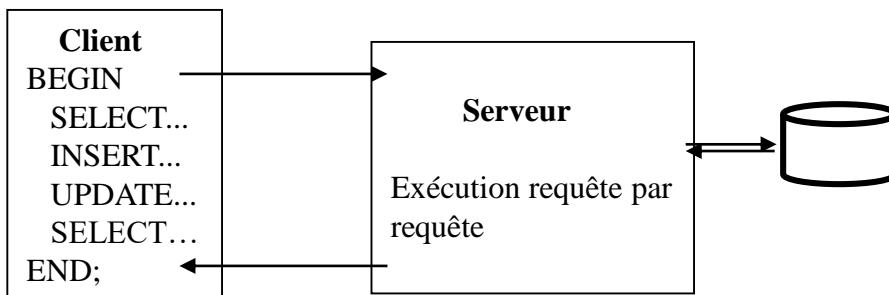
## □ Mode SQL

- Langage de manipulation de donnée(LDD, LMD, LCD)



## □ Mode PL/SQL

- Extension de SQL
- Langage L4G
  - Modulaire (paquetages, sous-programmes)
  - Gestion des erreurs (exceptions)
  - TAD (type abstrait de données)



# Editeurs de code

## □ Oracle SQL Developer

- Oracle SQL Developer is a free, integrated development environment that simplifies the development and management of Oracle Database in both traditional and Cloud deployments. SQL Developer offers complete end-to-end development of your PL/SQL applications, a worksheet for running queries and scripts, a DBA console for managing the database, a reports interface, a complete data modeling solution, and a migration platform for moving your 3rd party databases to Oracle.

## □ The basic command line SQL\*PLUS

- Write the script and save it in a sql extention (ex. myScript.sql)
- Run SQL\*PLUS command line and log in
- Run the following command : start path to myScript +myScript.sql  
OR
- Run : @path/myScript.sql

# Création d'un utilisateur

```
CREATE USER utilisateur IDENTIFIED {BY mot de passe | EXTERNALLY }  
[DEFAULT TABLESPACE tablespace]  
[TEMPORARY TABLESPACE tablespace]  
[QUOTA { entier [K,M] | UNLIMITED } ON tablespace...]  
[PROFILE profile][PASSWORD EXPIRE]
```

Tablespace par défaut : SYSTEM

Quota en Ko ou en Mo dans ce tablespace pour cet utilisateur

Sans profil, l'utilisateur reçoit le profil par défaut, qui est, sauf chgt, sans limitation de ressource

PASSWORD EXPIRE oblige l'utilisateur à changer son mot de passe avant sa prochaine connexion

# Exemple: create user

```
CREATE USER user_irisi  
IDENTIFIED BY azerty12  
DEFAULT TABLESPACE data  
TEMPORARY TABLESPACE temp  
QUOTA UNLIMITED ON data  
QUOTA UNLIMITED ON indx  
PASSWORD EXPIRE;
```

# Créer un tablespace

```
create TABLESPACE "tablespace2"
DATAFILE 'C:\app\admin\oradata\commande\tablespace2.DBF' SIZE 2G
AUTOEXTEND ON NEXT 100M MAXSIZE 5000M
LOGGING ONLINE
PERMANENT BLOCKSIZE 8192
EXTENT MANAGEMENT LOCAL UNIFORM SIZE 10M
SEGMENT SPACE MANAGEMENT AUTO;
```

# Modifier un utilisateur

ALTER **USER** **utilisateur**

IDENTIFIED {BY mot de passe | EXTERNALLY }  
[DEFAULT TABLESPACE tablespace]  
[TEMPORARY TABLESPACE tablespace]  
[QUOTA { entier [K,M] | UNLIMITED } ON tablespace...]  
[PROFILE profile]  
[DEFAULT ROLE { role,...| ALL [EXCEPT role,...]| NONE}]  
[PASSWORD EXPIRE];

# Supprimer un utilisateur

**DROP USER** utilisateur [CASCADE];  
CASCADE supprime les objets créés par l'utilisateur

# Utilisateur prédéfinis

**Il y a deux super-utilisateurs (administrateurs) prédéfinis : sys et system  
Le rôle prédéfini sysdba (ou dba dans d'anciennes versions) peut être octroyé  
par les utilisateurs sys et system**

# Vues du dictionnaire

Table **DBA\_USERS** qui comporte notamment les colonnes :**USERNAME, PROFILE, LOCK\_DATE, EXPIRY\_DATE, DEFAULT\_TABLESPACE, TEMPORARY\_TABLESPACE**

Table **DBA\_TS\_QUOTAS** contient les quotas alloués aux utilisateurs

# Profil Utilisateur

Un **profil** est un ensemble nommé de limitations de ressources permet de contrôler les ressources utilisés par un utilisateur

**CREATE PROFILE** profile LIMIT

**SESSIONS\_PER\_USER** {entier | DEFAULT | UNLIMITED}

*connexions en parallèle par utilisateur*

**CPU\_PER\_SESSION** {entier | DEFAULT | UNLIMITED} *en centièmes de secondes*

**CONNECT\_TIME** {entier | DEFAULT | UNLIMITED} *en secondes*

**IDLE\_TIME** {entier | DEFAULT | UNLIMITED}

*temps max d'inactivité en continu pour une session en secondes*

**LOGICAL\_READS\_PER\_SESSION** {entier | DEFAULT | UNLIMITED}

*nb max de blocs de données lus par session*

**LOGICAL\_READS\_PER\_CALL** {entier | DEFAULT | UNLIMITED}

*nb max de blocs de données lus par appel au noyau*

**COMPOSIT\_LIMIT** {entier | DEFAULT | UNLIMITED}

**PRIVATE\_SGA** {entier [K,M] | DEFAULT | UNLIMITED};

*taille allouée à l'espace privé d'une session dans la SGA*

# Gestion des comptes utilisateur

A partir de la version 8 d'Oracle

Limitation des paramètres d'utilisation des comptes

Options visibles dans la table DBA\_PROFILE

**CREATE PROFILE profile LIMIT**

FAILED\_LOGIN\_ATTEMPTS {entier | DEFAULT | UNLIMITED}

PASSWORD\_LOCK\_TIME {entier | DEFAULT | UNLIMITED}

*nb de jours de blocage d'un compte après un nb d'échecs consécutifs*

PASSWORD\_GRACE\_TIME {entier | DEFAULT | UNLIMITED}

*nb de jours donnés pour changer un mot de passe (après avertissement)*

PASSWORD\_LIFE\_TIME {entier | DEFAULT | UNLIMITED}

PASSWORD\_REUSE\_TIME {entier | DEFAULT | UNLIMITED};

*nb de jours avant que le mot de passe courant puisse être réutilisé*

# Gestion des privilèges

Privilèges système : concernent la connexion à la base Oracle et permettent à un utilisateur un certain nombre d'actions sur la définition d'objets de la base

Privilèges objets : permettent des accès aux objets désignés, accordés par le propriétaire de ces objets

# Privilèges système

```
GRANT {privilège système | rôle }  
[{privilège système | rôle },...]  
TO  
{utilisateur | rôle | public} [, {utilisateur | rôle} ,...]  
[WITH ADMIN OPTION];
```

Avec **WITH ADMIN OPTION**, les privilèges reçus peuvent être transmis à d'autres utilisateurs ou rôles

```
REVOKE privilège FROM {utilisateur|rôle|public};
```

# Privilèges système

Privilèges système	CREATE	ALTER	DROP	Divers
CLUSTER	Oui			
ANY CLUSTER	oui	oui	Oui	
DATABASE	Oui			
DATABASE LINK	Oui			
PUBLIC DATABASE LINK	oui		Oui	
ANY INDEX	oui	oui	Oui	
LIBRARY	oui		Oui	
ANY LIBRARY	Oui		oui	
PROCEDURE	oui			Execute
ANY PROCEDURE	oui	oui	oui	Execute

# Privilèges système

Privilèges système	CREATE	ALTER	DROP	Divers
PROFILE	oui	oui	Oui	
RESOURCE COST		Oui		
ROLE	Oui			
ANY ROLE		oui	oui	Grant
ROLLBACK SEGMENT	Oui	oui	oui	
SESSION	oui	Oui		
SEQUENCE	Oui			
ANY SEQUENCE	oui	oui	Oui	select
SNAPSHOT				
ANY SNAPSHOT				
SYNONYM				
ANY SYNONYM				

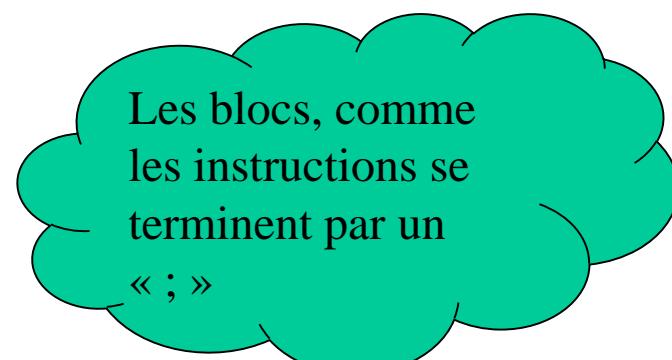
# Structure d'un bloc PL/SQL

## □ Syntaxe d'un bloc anonyme

```
[ DECLARE
    -- Déclaration types, constantes, variables ]
BEGIN
    -- Instructions PL/SQL (et ou SQL-LMD)
[ EXCEPTION
    -- Traitement ou récupération des erreurs ]
END; ;
```

/

**NB: seuls BEGIN et END sont Obligatoires.**



Les blocs, comme les instructions se terminent par un « ; »

# Structure des blocs (3)

Les blocs PL/SQL peuvent être imbriqués:

**DECLARE**

~~...~~  
**BEGIN**

**DECLARE**

~~.....~~  
**BEGIN**

~~.....~~  
**BEGIN**

~~.....~~  
**END ;**

~~.....~~  
**END ;**

~~.....~~  
**END ;**

## Structure des blocs (3)

- Les blocs comme les instructions se terminent par un ‘;’.
- Seuls ‘Begin’ et ‘End’ sont obligatoire.

# Premier programme PL/Sql

```
SQL> SET SERVEROUTPUT ON
```

```
SQL> BEGIN
```

```
2      dbms_output.putline('Bonjour');  
3      END;  
4      /
```

**Bonjour**

- Un programme contient au moins **BEGIN <une instruction> END;**
- **dbms\_output.putline** permet d'écrire sur la console SQLPlus.
- La sortie écran ne marche que si le serveur d'impression est ouvert. Faire **SET SERVEROUTPUT ON** sous SQLPlus pour basculer sur le mode sortie console.
- La ligne 4 commence par un /. C'est la demande d'exécution du programme tapée (donc la fin de saisie)

# Structure d'un bloc PL/SQL

## □ Exemple

ETUDIANTS		
numetudiant	nometudiant	nbformation
1	Jojo	1
2	Mimi	2
3	Polo	0

FORMATIONS	
numformation	nomformation
DW	Data Warehouse
OR	BD Objet-Relationnelles

INSCRIPTIONS			
numformation	numetudiant	dateinscription	nf
DW	2	22/09/2003	
DW	1	19/09/2003	
OR	2	22/09/2003	

## □ « Insertion de Polo »

```
ACCEPT ne PROMPT 'Numero de l''etudiant a inscrire:'  
ACCEPT nf PROMPT 'Numero de l''enseignement:'
```

```
BEGIN  
    INSERT INTO INSCRIPTIONS  
    VALUES (&nf, '&ne', SYSDATE);  
END;  
/
```

```
...  
Numero de l'etudiant a inscrire: 3  
Numero de l'enseignement: DW
```

```
...
```

# Types de données

## □ Types simples habituels correspondants aux types SQL2 :

- CHAR, CHAR (*n*), STRING, VARCHAR2 (*n*), ...
- INTEGER, NATURAL, NUMBER, NUMBER (*n*), FLOAT, REAL, NUMBER (*n, m*), ...
- BOOLEAN
- DATE
- ...

## □ Types composites adaptés à la récupération des colonnes et lignes des tables SQL :

- RECORD
- TABLE
- VARRAY

## □ Types références

- REF <type objet>

## □ Types LOB (« Large Objects »)

- BFILE
- BLOB, CLOB, NCLOB

- **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.**

# Les variables (2)

- Déclaration :

**Nom\_variable type\_variable;**

- Initialisation:

**Nom\_variable := valeur;**

- Déclaration et initialisation :

**Nom\_variable type\_variable := valeur;**

# Les variables

- **var [CONSTANT] datatype [NOT NULL] [:= | DEFAULT expr];**
  
- Adopter des conventions pour nommer des objets.
- Initialiser les constantes et les variables déclarées NOT NULL.
- Initialiser les identifiants en utilisant l'opérateur d'affectation ( := ) ou le mot réservé DEFAULT.
- Déclarer au plus un identifiant par ligne.
- Le type peut être primitif ou objet.

# Commentaires

- Pour une fin de ligne
- /\* Pour plusieurs lignes \*/

# Les types de variables

## □ VARCHAR2

- ❖ Longueur maximale : 32767 octets
- ❖ Syntaxe:

Nom\_variable VARCHAR2(30);

*Exemple:* name  
name

VARCHAR2 (30) ;  
VARCHAR2 (30) := 'toto' ;

## □ NUMBER

Nom\_variable NUMBER(long,dec);

avec Long : longueur maximale

Dec : longueur de la partie décimale

*Exemple:* num\_tel

number (10) ;

toto

number (5,2) = 142.12 ;

# Les types de variables (2)

## □ DATE

Nom\_variable 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 ou NULL

# Les types de variables (3)

- identificateur [CONSTANT] type [:= valeur];
- Exemples :
  - ❖ **age integer;**
  - ❖ **nom varchar(30);**
  - ❖ **dateNaissance date;**
  - ❖ **ok boolean := true;**
- Déclarations multiples interdites :
  - ❖ **i, j integer;**

# Exemples

```
c          CHAR( 1 ) ;
name      VARCHAR2(10) := 'Scott' ;
cpt       BINARY_INTEGER := 0 ;
total     NUMBER( 9, 2 ) := 0 ;
order     DATE := SYSDATE + 7 ;
Ship      DATE ;
pi        CONSTANT NUMBER ( 3, 2 ) := 3.14 ;
done      BOOLEAN NOT NULL := TRUE ;
ID        NUMBER(3) NOT NULL := 201 ;
PRODUIT   NUMBER(4) := 2*100 ;
V_date    DATE := TO_DATE('17-OCT-01','DD-MON-YY') ;
v1        NUMBER := 10 ;
v2        NUMBER := 20 ;
v3        BOOLEAN := (v1>v2) ;
Ok        BOOLEAN := (z IS NOT NULL) ;
```

# Quelques conventions en PL/SQL

- Deux variables peuvent partager le même nom si elles sont dans des portées distinctes.
- Les noms des variables doivent être différents des noms des colonnes des tables utilisées dans un bloc:
  - ❖ **v\_empno** (variable)
  - ❖ **g\_deptno** (globale)
  - ❖ **c\_emp** (CURSOR)
- L'identifiant est limité à 30 caractères, le premier caractère devant **être une lettre**.

# Utilisation des variables

## □ On utilise des variables pour :

- ❖ Le stockage temporaire de données
- ❖ La manipulation de valeur stockées
- ❖ La possibilité de les réutiliser
- ❖ Simplifier la maintenance du code
- ❖ Variable typée dynamiquement au moyen d'attributs spéciaux
  - %ROWTYPE ou %TYPE

## □ Les variables Abstract Data Type et les collections seront abordées ultérieurement

- ❖ Oracle9i est également un SGBD orienté objet

# Bloc PL/SQL Anonyme accédant à la base

```
SQL> SET SERVEROUTPUT ON
```

```
SQL> DECLARE
```

```
    v_name VARCHAR2(10);
```

```
BEGIN
```

```
    SELECT ename INTO v_name
```

```
    FROM emp; -- WHERE empno=7839;
```

```
    DBMS_OUTPUT.PUT_LINE(v_name);
```

```
EXCEPTION
```

```
    WHEN OTHERS
```

```
        THEN NULL;
```

```
END;
```

```
/
```

Affection

```
PL/SQL Procedure successfully completed.
```

# Cas à plusieurs variables hôtes

```
SET SERVEROUTPUT ON
SQL> DECLARE
          v_ename      VARCHAR2(12);
          v_sal       NUMBER(7,2);
BEGIN
    SELECT    ename,sal   INTO
              v_ename,v_sal
        FROM emp WHERE ROWNUM = 1;
    DBMS_OUTPUT.PUT_LINE(v_ename);
    DBMS_OUTPUT.PUT_LINE(v_sal);
EXCEPTION
    WHEN OTHERS
        THEN NULL;
END;
/
```

Optionnel à  
ce stade

# Variables et Constantes

## □ Déclaration dans la partie déclarative d'un bloc PL/SQL

## □ Variables

```
DECLARE
    date_naissance DATE;
    compteur INTEGER:=0;          -- initialisation du compteur à zéro
    compteur INTEGER DEFAULT 0;   -- initialisation du compteur à zéro
    ok BOOLEAN DEFAULT TRUE;
    id CHAR(5) NOT NULL :='ali';
```

## □ Constantes

```
DECLARE
    pi CONSTANT FLOAT:=3.14;
```

- Déclarations multiples interdites : ~~i, j integer;~~
- null est valeur par défaut, si pas d'initialisation (sauf si NOT NULL spécifié)
- Mais les variables les plus intéressantes en PLSQL sont celles qui vont contenir des données de la base. Variable de même type qu'un attribut (colonne) ou de même type qu'un schéma (ligne).



# Les types de variables (4)

- Types composites adaptés à la récupération des colonnes et lignes des tables SQL :  
**%TYPE, %ROWTYPE.**

# Typage dynamique %TYPE

- Déclarer une variable à partir :
  - ❖ D'une autre variable déjà déclarée
  - ❖ D'une définition d'un attribut de la base de données
- Préfixer %TYPE avec :
  - ❖ La table et la colonne de la base de données
  - ❖ Le nom de la variable déclarée précédemment
- PL/SQL évalue le type de donnée et la taille de la variable.
- Inspiré du langage ADA

# Déclaration %TYPE

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

```
nom emp.name%TYPE;
```

# L'Attribut %TYPE - Exemple

**DECLARE**

ename	scott.emp.ename%TYPE ;
job	emp.job%TYPE ;
balance	NUMBER( 7, 2 ) ;
mini_balance	balance%TYPE := 10 ;
rec	emp%ROWTYPE

- Le type de données de la colonne peut être inconnu.
- Le type de données de la colonne peut changer en exécution.
- Facilite la maintenance.

# Variable typée dynamiquement

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
          v_ename          emp.ename%TYPE;
    BEGIN
        SELECT ename
        INTO v_ename
        FROM emp
        WHERE ROWNUM = 1;
        DBMS_OUTPUT.PUT_LINE(v_ename);
    EXCEPTION
        WHEN OTHERS
            THEN NULL;
    END;
/
```

# Déclaration %ROWTYPE

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

```
employe emp%ROWTYPE;
```

- déclare que la variable employe contiendra une ligne de la table emp.

## Exemple d'utilisation

```
employe emp%ROWTYPE;
nom emp.name.%TYPE;
select * INTO employe
from emp
where matr = 900;
nom := employe.name;
employe.dept := 20;
-----
insert into emp
values employe;
```

# Attribut %TYPE & %ROWTYPE

```
SQL> SET SERVEROUTPUT ON
```

```
SQL> DECLARE
```

```
    rec  emp%ROWTYPE;
        address VARCHAR2(64);
        income emp.sal%TYPE; -- rec.sal%TYPE;
```

```
BEGIN
```

```
    SELECT *
```

```
        INTO rec FROM emp
        WHERE ROWNUM = 1;
```

```
    income := rec.sal*12;
```

```
    address := rec.ename || CHR(10) ||
                income || CHR(10) ||
```

```
                TO_CHAR(rec.hiredate,'DD/MM/YYYY');
```

```
    DBMS_OUTPUT.PUT_LINE(address);
```

```
END;
```

```
/
```

SMITH

9600

17/12/1980

Manipulation dans une  
variable PL

# Affectation

- Plusieurs façons de donner une valeur à une variable:

- ❖ :=

- ❖ par la directive INTO de la requête SELECT

- Exemples :

- ❖ **dateNaissance := '10/10/2004';**

- ❖ **select name INTO nom from emp where matr = 509;**

PL/SQL supporte les opérateurs suivants :

- Arithmétique : +, -, \*, /,
- MOD(m,n) modulo
- Concaténation : ||
- Parenthèses (contrôle des priorités entre opérations): ()
- Comparaison : =, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN
- Logique : AND, OR, NOT
- Affectation: :=

# Portée

- **Les instructions peuvent être imbriquées là où les instructions exécutables sont autorisées.**
- **La section EXCEPTION peut contenir des blocs imbriqués.**
- **Les boucles possède chacune une portée**
  - les incrémentations y sont définies
- **Un identifiant est visible dans les régions où on peut référencer cet identifiant :**
  - Un bloc voit les objets du bloc de niveau supérieur.
  - Un bloc ne voit pas les objets des blocs de niveau inférieur.

# Portée

```
DECLARE —————  
      X      INTEGER;  
BEGIN  
      . . .  
      DECLARE —————  
          Y      NUMBER;  
      BEGIN  
          . . .  
      END ; —————  
      . . .  
END ; —————  
/
```

Portée de x

Portée de y

# Règles générales

- Commenter le code**
- Adopter une convention de casse**
- Développer une convention d'appel pour les identifiants et autres objets**
- Indenter le code**

Instruction SQL en majuscules

Mots-clés en majuscules

Type de données en majuscules

Identifiant et paramètres en minuscules

Tables & colonnes en minuscules

# Conventions d'appel possible

Identifiant	Nom	Exemple
Variable	v_name	v_sal
Constante	c_name	c_pi
Cursor	name_cursor	Emp_cursor
Exception	e_name	E_too_many
Table type	name_table_type	sum_table_type
Table	name_table	emp_tot_table
Record Type	name_record_type	emp_record_type
Record	name_record	emp_record
Substitution	p_name	p_sal
Globale	g_name	g_deptno

# REcap

## □ Référence au dictionnaire des données : %TYPE et %ROWTYPE

**DECLARE**

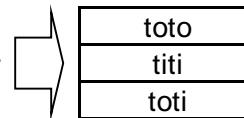
```
date_ins INSCRIPTIONS.dateinscription%TYPE; /*type de la colonne dateinscription  
dans la table INSCRIPTIONS*/  
ligne INSCRIPTIONS%ROWTYPE; /* type composite d 'un enregistrement  
de la table INSCRIPTIONS */
```

## □ Tableau

**DECLARE**

```
-- déclaration du type tableau  
TYPE type_vecteur  
    IS TABLE OF VARCHAR2(10)  
    INDEX BY BINARY_INTEGER;
```

```
-- déclaration d'une variable tableau  
t type_vecteur := ('toto', 'titi', 'toti');
```



n\_etudiant **VARCHAR2(10)**;

**BEGIN**

```
n_etudiant:=t(1);  
t(1):=t(2);  
t(2):=n_etudiant;
```

**END;**



## □ Type structuré

**DECLARE**

```
-- déclaration du type  
TYPE type_structure IS RECORD (numero INTEGER NOT NULL,  
nom ETUDIANTS.nometudiant%TYPE);  
-- déclaration d'une variable  
etudiant type_structure;
```

# Exemple d'utilisation

```
employe    emp%ROWTYPE;
nom        emp.nom%TYPE;
Select *  INTO  employe
              from      emp
                           where      matr = 900;
nom    :=  employe.nom;
employe.dept  :=  20;
...
insert  into  emp  values  employe;
```

# Le paquetage DBMS\_OUTPUT

Les procédures de ce paquetage vous permettent d'écrire des lignes dans un tampon depuis un bloc PL/SQL anonyme, une procédure ou un déclencheur.

Le contenu de ce tampon est affiché à l'écran lorsque le sous-programme ou le déclencheur est terminé.

L'utilité principale est d'afficher à l'écran des informations de trace ou de débogage

La taille maximum du tampon est de 1 million de caractères

La taille maximum d'une ligne est de 255 caractères

La capacité maximum d'une ligne étant de 255 caractères, la procédure DEBUG permet de s'affranchir de cette limitation

# Fonctions et procédures du paquetage

DBMS\_OUTPUT.ENABLE ( taille Tampon IN INTEGER DEFAULT 20000)

Cette procédure permet d'initialiser le tampon d'écriture et d'accepter les commandes de lecture et d'écriture dans ce tampon.

taille Tampon représente la taille maximum en octets allouée au tampon.

Les valeurs doivent être indiquées dans une plage de valeur allant de 2000 (minimum) et 1 million (maximum). Sa valeur par défaut est 20000.

Exceptions générées

ORA-20000: Buffer overflow, limit of (buffer\_limit) bytes.

ORU-10027:

DBMS\_OUTPUT.DISABLE: Cette procédure désactive les appels de lecture et écriture dans le tampon et purge ce dernier.

Procédures d'écriture dans le tampon

DBMS\_OUTPUT.PUT (item IN NUMBER)

DBMS\_OUTPUT.PUT (item IN VARCHAR2)

DBMS\_OUTPUT.PUT (item IN DATE)

DBMS\_OUTPUT.PUT\_LINE (item IN NUMBER)

DBMS\_OUTPUT.PUT\_LINE (item IN VARCHAR2)

DBMS\_OUTPUT.PUT\_LINE (item IN DATE)

DBMS\_OUTPUT.NEW\_LINE

# Type RECORD

- Equivalent à struct en langage C

- Déclaration

TYPE *nomRecord* IS RECORD ( *champ1 type1*, *champ2 type2*, ...);

- Utilisation du type RECORD

```
TYPE emp2 IS RECORD (num_matr integer, nom varchar(30));
```

```
employe emp2;
```

```
employe.num_matr := 500;
```

# Conflits de noms

- Si une variable porte le même nom qu'une colonne d'une table, c'est la colonne qui l'emporte

DECLARE

```
prenom varchar(30) := 'ali';
```

BEGIN

```
delete from emp where prenom = prenom;
```

- Pour éviter ce genre de contradiction, tout simplement ne pas donner de nom de colonne à une variable !

# Visibilité des Variables

- une variable est visible dans le bloc où elle a été déclarée et dans les blocs imbriqués si elle n'a pas été redéfinie.

DECLARE

```
    var1 NUMBER(10);  
    var2 CHAR(10);
```

BEGIN

DECLARE

```
    var1 CHAR(10);  
    var3 DATE;
```

BEGIN

```
    ... var1 CHAR(10)  
    ... var2  
    ... var3
```

END;

```
    ... var1 NUMBER(3)  
    ... var2 CHAR(10)  
    ... var3 CHAR(10) – génère une erreur
```

END;

# Variables de session

- Outre les variables locales, un bloc Pl/sql peut utiliser d'autres variables: les variables de sessions(globales). La directive sql\*plus à utiliser en début de bloc est VARIABLE. Dans le code PL/SQL il faut faire préfixer le nom de la variable de session par « : ».
- L'affichage de cette variable sous SQL\*Plus est réalisé par la directive PRINT.

```
SET SERVEROUTPUT ON;
VARIABLE nombre Number;
DECLARE
    Num number(3) :=0;
BEGIN
    :nombre := Num+1;
    DBMS_OUTPUT.PUT_LINE(:nombre);
END;
/
PRINT :nombre;
```

# Types LOB

- Avec les type Large Object on peut stocker des blocs de données non-structurées (textes, images, clips vidéos, sons) d'une taille pouvant aller jusqu'à 4 Go.

CLOB	Stocke de gros blocs de données de type caractère 8 bits dans la base de données
BLOBE	Stocke de gros objets binaires : photos
BFILE	Stocke de gros objets binaires dans des fichiers du système opérateur à l'extérieur de la base de données

# Structures de contrôle

## □ Instructions conditionnelle

```
IF condition1 THEN  
    liste d'instructions1;  
[ ELSEIF condition2 THEN  
    liste d'instructions2; ]  
...  
[ ELSE  
    liste d'instructionsn; ]  
END IF;
```

```
IF condition THEN  
    instructions1;  
[ ELSE  
    instructionsn; ]  
END IF;
```

```
IF condition THEN  
    instructions  
END IF;
```

## □ Contrôle séquentiel

```
BEGIN  
...  
GOTO label;  
...  
<<label>>  
instructions;  
...  
END;
```

# Structures de contrôle

## □ Nested IF Statements

**DECLARE**

```
sales NUMBER(8,2) := 12100; quota NUMBER(8,2) := 10000; bonus NUMBER(6,2);  
emp_id NUMBER(6) := 120;
```

**BEGIN**

```
IF sales > (quota + 200) THEN bonus := (sales - quota)/4;
```

```
ELSE IF sales > quota THEN bonus := 50; ELSE bonus := 0;
```

```
END IF; END IF;
```

```
UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;  
END; /
```

**Using the IF-THEN-ELSIF Statement**

**DECLARE** sales NUMBER(8,2) := 20000; bonus NUMBER(6,2); emp\_id NUMBER(6)  
:= 120;

**BEGIN IF** sales > 50000 **THEN** bonus := 1500; **ELSIF** sales > 35000 **THEN** bonus :=  
500; **ELSE** bonus := 100; **END IF**;

```
UPDATE employees SET salary = salary + bonus WHERE employee_id = emp_id;  
END; /
```

# Structures de contrôle

## □ Extended IF-THEN Statement

**DECLARE**

**grade CHAR(1);**

**BEGIN** grade := 'B';

**IF** grade = 'A' **THEN** DBMS\_OUTPUT.PUT\_LINE('Excellent');

**ELSIF** grade = 'B' **THEN** DBMS\_OUTPUT.PUT\_LINE('Very Good');

**ELSIF** grade = 'C' **THEN** DBMS\_OUTPUT.PUT\_LINE('Good');

**ELSIF** grade = 'D' **THEN** DBMS\_OUTPUT.PUT\_LINE('Fair');

**ELSIF** grade = 'F' **THEN** DBMS\_OUTPUT.PUT\_LINE('Poor');

**ELSE** DBMS\_OUTPUT.PUT\_LINE('No such grade');

**END IF;**

**END; /**

# Structures de contrôle

## Exemple

```
SET SERVEROUTPUT ON;
ACCEPT Chaine PROMPT 'Choisir votre sexe :F ou M: ';
BEGIN
    IF Upper('&Chaine') = 'F' THEN
        Goto Message1;
    END IF;
    IF Upper('&Chaine') = 'M' THEN
        Goto Message2;
    END IF;
    « Message1 »
    DBMS_OUTPUT.PUT_LINE('Femme');
    « Message2 »
    DBMS_OUTPUT.PUT_LINE('Homme');
END;
/
```



# Choix

```
CASE expression  
WHEN expr1 THEN instructions1;  
WHEN expr2 THEN instructions2;  
...  
ELSE instructionsN;  
END CASE;
```

- **expression peut avoir n'importe quel type simple**

# Les itérations

□ PL/SQL offre la possibilité d'écrire des boucles à l'aide d'instructions LOOP, WHILE ou FOR.

# Les itérations (2)

## 1ere forme:

```
LOOP  
    instructions  
    EXIT WHEN expression  
END LOOP ;
```

## 2eme forme:

La boucle **FOR** permet des boucles sur un ensemble fini comme un ensemble d'entiers ou un ensemble d'enregistrements issus d'une requête.

```
FOR i IN 1..10 LOOP  
    DBMS_OUTPUT.PUT_LINE ( 'iteration ' ||i ) ;  
END LOOP;
```

# Les itérations (3)

## 3eme forme:

Enfin, le troisième type de boucle permet la sortie selon une condition prédefinie.

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

# Les itérations (4)

## □ Exemple:

```
DECLARE
    x NUMBER(3):=1;
BEGIN
    WHILE x<=100 LOOP
        INSERT INTO employe(noemp, nomemp, job, nodept)
        VALUES (x, 'TOTO', 'PROGRAMMEUR', 1);
        x:=x+1;
    END LOOP;
END;
```

# Affichage

## □ Activer le retour écran

❖ set serveroutput on size 10000

## □ Affichage

❖ dbms\_output.put\_line(chaîne);

❖ Utilise || pour faire une concaténation

# Affichage (2)

```
DECLARE
compteur number(3);
i number(3);
BEGIN
select count(*) into compteur from clients;
FOR i IN 1..compteur LOOP
    dbms_output.put_line('Nombre : ' || i );
END LOOP;
END;
```

# Affichage (3)

```
DECLARE
    i number(2);
BEGIN
    FOR i IN 1..5 LOOP
        dbms_output.put_line('Nombre : ' || i );
    END LOOP;
END;
```

# La fonction prédéfini LENGTH

```
SET SERVEROUTPUT ON
DECLARE
    x VARCHAR (5);
    y VARCHAR2 (5);
BEGIN
    x := 'toto'; y := 'toto';
    IF ( x = y) THEN
        DBMS_OUTPUT.PUT_LINE('equals');
    ELSE
        DBMS_OUTPUT.PUT_LINE('e not equals');
    END IF;
    DBMS_OUTPUT.PUT_LINE(LENGTH(x));
    DBMS_OUTPUT.PUT_LINE(LENGTH(y));
END;
/
```

## Autre exemple : CONCAT

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
          nom      VARCHAR2(10) ;
          salaire   NUMBER(7,2) ;
      BEGIN
          SELECT ename,sal INTO nom,salaire
          FROM emp WHERE ROWNUM = 1;
          DBMS_OUTPUT.PUT_LINE(nom || salaire);
          DBMS_OUTPUT.PUT_LINE(
                  CONCAT (nom,salaire));
      END ;
/

```

Une autre façon de concaténer des chaînes de caractères

# Un bloc peut contenir plusieurs requêtes

```
SQL> SET SERVEROUTPUT ON
SQL> DECLARE
          nom          VARCHAR2(10);
          salaire      NUMBER(7,2);
BEGIN
    SELECT ename INTO nom
    FROM emp WHERE ROWNUM = 1;
    DBMS_OUTPUT.PUT_LINE(nom);

    SELECT sal INTO salaire
    FROM emp WHERE ROWNUM = 1;
    DBMS_OUTPUT.PUT_LINE(salaire);
EXCEPTION
    WHEN OTHERS
        THEN NULL;
END;
/
```

# Structures de contrôle: exemple

**DECLARE**

**a number(2) := 10;**

**BEGIN**

**<<loopstart>>**

**dbms\_output.put\_line ('starting');**

**-- while loop execution**

**WHILE a < 30 LOOP**

**dbms\_output.put\_line ('value of a: ' || a);**

**a := a + 1; IF a = 15 THEN a := a + 1;**

**if a=18**

**GOTO loopstart;**

**END IF;**

**END LOOP;**

**END; /**

# Structures de contrôle: exemple

Reprendons le même exemple mais en changant l'emplacement de l'étiquete

**DECLARE**

**a number(2) := 10;**

**BEGIN**

**-- while loop execution**

**WHILE a < 30 LOOP**

**dbms\_output.put\_line ('value of a: ' || a);**

**a := a + 1; IF a = 15 THEN a := a + 1; END IF;**

**IF a=18 THEN**

**GOTO loopstart;**

**END IF;**

**END LOOP;**

**<<loopstart>>**

**dbms\_output.put\_line ('value of a: ' || a);**

**END;**

**/**

# Structures de contrôle: exemple

```
SET SERVEROUTPUT ON;
declare
    n integer;
begin
    select count(*) into n from clients where noclient = 5;
    if n > 0 then
        goto job1;
    else
        goto job2;
    end if;
    <<job1>>
    dbms_output.put_line('job1 – Dept has employees');
    return;
    <<job2>>
    dbms_output.put_line('job2 – Dept has no employees');
    return;
end;
/
```

# Restrictions with GOTO Statement

**GOTO Statement in PL/SQL imposes the following restrictions:**

- A GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement or sub-block.
- A GOTO statement cannot branch from one IF statement clause to another or from one CASE statement WHEN clause to another.
- A GOTO statement cannot branch from an outer block into a sub-block (that is, an inner BEGIN-END block).
- A GOTO statement cannot branch out of a subprogram. To end a subprogram early, either use the RETURN statement or have GOTO branch to a place right before the end of the subprogram.
- A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

# Instructions

## □ Les itérations

**LOOP**

```
    liste d'instructions;  
    ...  
    [ IF condition THEN  
        ...  
        EXIT;  
    END IF; ]  
    ...  
    [ EXIT WHEN condition; ]  
END LOOP;
```

## □ Itération avec label

```
<<sortie>>  
LOOP  
    ...  
    LOOP  
        ...  
        EXIT sortie WHEN ...;      -- sortie de toutes les boucles  
    END LOOP;  
    ...  
END LOOP sortie;
```

# Instructions

**Exemples:**

**DECLARE**

**Compteur NUMBER(4) := 0;**

**BEGIN**

**LOOP**

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

**Compteur := Compteur + 1;**

**EXIT WHEN Compteur > 5;**

**END LOOP;**

**END;**

**/**

**Exemples avec étiquete:**

**DECLARE**

**Compteur NUMBER(4) := 0;**

**BEGIN**

**« exemple »**

**LOOP**

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

**Compteur := Compteur + 1;**

**EXIT exemple WHEN Compteur > 5;**

**END LOOP exemple;**

**END;**

**/**

# Instructions

## □ Itération « Tant que »

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

**WHILE condition LOOP**

    sequence\_of\_statements

**END LOOP;**

Before each iteration of the loop, the condition is evaluated. If it is TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If it is FALSE or NULL, the loop is skipped and control passes to the next statement.

## □ Itération « Pour que »

**FOR compteur INTO [REVERSE] borne\_inf..borne\_sup LOOP**  
    liste d'instructions;  
**END LOOP;**

# Instructions: exemple

## Exemple FOR:

**BEGIN FOR i IN REVERSE 1..3 LOOP -- assign the values 1,2,3 to i  
DBMS\_OUTPUT.PUT\_LINE (TO\_CHAR(i)); END LOOP; END; /**

- Inside a FOR loop, the counter can be read but cannot be changed.

**BEGIN FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i IF i < 3 THEN  
DBMS\_OUTPUT.PUT\_LINE (TO\_CHAR(i)); ELSE i := 2; -- not allowed,  
raises an error END IF; END LOOP; END; /**

# Instructions

## □ Using WHILE-LOOP for Control

```
CREATE TABLE temp (tempid NUMBER(6), tempsal NUMBER(8,2), tempname  
VARCHAR2(25));  
  
DECLARE sal employees.salary%TYPE := 0;  
  
mgr_id employees.manager_id%TYPE;  
lname employees.last_name%TYPE;  
starting_empid employees.employee_id%TYPE := 120;  
  
BEGIN  
  
SELECT manager_id INTO mgr_id FROM employees WHERE employee_id = starting_empid;  
WHILE sal <= 15000 LOOP -- loop until sal > 15000  
SELECT salary, manager_id, last_name INTO sal, mgr_id, lname FROM employees WHERE  
employee_id = mgr_id;  
END LOOP;  
INSERT INTO temp VALUES (NULL, sal, lname); -- insert NULL for tempid  
COMMIT;  
EXCEPTION WHEN NO_DATA_FOUND THEN INSERT INTO temp VALUES (NULL, NULL,  
'Not found'); -- insert NULLs  
COMMIT;  
END;
```

# Instructions

## □ Choix

**CASE expression**

    WHEN expr1 THEN instructions1;

    WHEN expr2 THEN instructions2;

...

    ELSE instructionsN;

END CASE;

expression peut avoir n'importe quel **type simple** (ne peut pas par exemple être un RECORD)

# Instructions: exemple avec case

## □ Using the CASE-WHEN Statement

**DECLARE**

**grade CHAR(1);**

**BEGIN**

**grade := 'B';**

**CASE grade**

**WHEN 'A' THEN DBMS\_OUTPUT.PUT\_LINE('Excellent');**

**WHEN 'B' THEN DBMS\_OUTPUT.PUT\_LINE('Very Good');**

**WHEN 'C' THEN DBMS\_OUTPUT.PUT\_LINE('Good');**

**WHEN 'D' THEN DBMS\_OUTPUT.PUT\_LINE('Fair');**

**WHEN 'F' THEN DBMS\_OUTPUT.PUT\_LINE('Poor');**

**ELSE DBMS\_OUTPUT.PUT\_LINE('No such grade');**

**END CASE;**

**END;**

/

# Instructions: exemple avec case

## □ Using the Searched CASE Statement

```
DECLARE grade CHAR(1);
BEGIN
    grade := 'B';
    CASE WHEN grade = 'A' THEN
        DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN grade = 'B' THEN
        DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN grade = 'C' THEN
        DBMS_OUTPUT.PUT_LINE('Good');
    WHEN grade = 'D' THEN
        DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN
        DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE
        DBMS_OUTPUT.PUT_LINE('No such grade');
    END CASE;
END;
```

-- rather than using the ELSE, in the CASE, could use the following -- EXCEPTION -- WHEN  
CASE\_NOT\_FOUND THEN -- DBMS\_OUTPUT.PUT\_LINE('No such grade'); /

# Instructions: exemple exit

## □ Using an EXIT Statement

**DECLARE**

**credit\_rating NUMBER := 0;**

**BEGIN**

**LOOP credit\_rating := credit\_rating + 1;**

**IF credit\_rating > 3 THEN**

**EXIT; -- exit loop immediately**

**END IF;**

**END LOOP; -- control resumes here**

**DBMS\_OUTPUT.PUT\_LINE ('Credit rating: ' || TO\_CHAR(credit\_rating));**

**IF credit\_rating > 3 THEN**

**RETURN; -- use RETURN not EXIT when outside a LOOP**

**END IF;**

**DBMS\_OUTPUT.PUT\_LINE ('Credit rating: ' || TO\_CHAR(credit\_rating));**

**END;**

# Using the EXIT-WHEN Statement

- The EXIT-WHEN statement lets a loop complete conditionally. When the EXIT statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the next statement after the loop.
- Until the condition is true, the loop cannot complete. A statement inside the loop must change the value of the condition. In the previous example, if the FETCH statement returns a row, the condition is false. When the FETCH statement fails to return a row, the condition is true, the loop completes, and control passes to the CLOSE statement.
- The EXIT-WHEN statement replaces a simple IF statement. For example, compare the following statements:  
IF count > 100 THEN EXIT; ENDIF;  
EXIT WHEN count > 100;

These statements are logically equivalent, but the EXIT-WHEN statement is easier to read and understand.

# Using the EXIT-WHEN Statement

## □ Using EXIT With Labeled Loops

**DECLARE**

**s PLS\_INTEGER := 0;**

**i PLS\_INTEGER := 0;**

**j PLS\_INTEGER;**

**BEGIN**

**<<outer\_loop>>**

**LOOP i := i + 1;**

**j := 0;**

**<<inner\_loop>>**

**LOOP**

**j := j + 1; s := s + i \* j; -- sum a bunch of products**

**EXIT inner\_loop WHEN (j > 5);**

**EXIT outer\_loop WHEN ((i \* j) > 15);**

**END LOOP inner\_loop;**

**END LOOP outer\_loop;**

**DBMS\_OUTPUT.PUT\_LINE('The sum of products equals: ' || TO\_CHAR(s)); END; /**

# SQL: Sequence(générateur de nombre)

- Permet d'obtenir des valeurs incrémentales
- Vérrouillage automatique en cas de concurrence d'accès
- Valeur suivante: **nomSequence.NEXTVAL**
- Valeur courante: **nomSequence.CURRVAL**
- Crédit:

```
CREATE SEQUENCE nomSequence  
    START WITH valeur_départ  
    INCREMENT BY increment;
```

- Insertion:  
**INSERT INTO table VALUES(nomSequence.NEXTVAL,...);**
- Suppression:  
**DROP SEQUENCE nomSequence;**

# SQL: Objet sequence

## □ Exemple:

```
SQL> CREATE TABLE test(numero number, nom varchar2(20));
```

Table créée

```
SQL> CREATE SEQUENCE seqTest START WITH 6 INCREMENT BY 3;
```

Séquence créée.

```
SQL> INSERT INTO test VALUES(seqTest.NEXTVAL, 'jack');
```

```
SQL> select * from test;
```

numero	nom
6	FADILI
9	jack

```
SQL> select seqTest.CURRVAL from dual;
```

CURRVAL
9

# Pseudo-table DUAL

- La table DUAL est une table utilisable par tous (en lecture seulement) et qui appartient à l'utilisateur SYS. Le paradoxe de DUAL réside dans le fait qu'elle est couramment sollicitée, mais les interrogations ne portent jamais sur sa seule colonne (DUMMY définie en VARCHAR2 et contenant un seul enregistrement avec la valeur « X »). En conséquence, DUAL est qualifiée de pseudo-table (c'est la seule qui soit ainsi composée).
- L'interrogation de DUAL est utile pour évaluer une *expression de la manière suivante* : « *SELECT expression FROM DUAL* » (seule l'instruction SELECT est permise sur DUAL). Comme DUAL n'a qu'un seul enregistrement, les résultats fournis seront uniques (si aucune jointure ou opérateur ensembliste ne sont utilisés dans l'interrogation).

# Les types Prédéfinis

Les types BINARY\_INTEGER et PLS\_INTEGER conviennent aux entiers signés (domaine de valeurs de  $-2^{31}$  à  $2^{31}$ , soit -2.147.483.647 à +2.147.483.647). Ces types requièrent moins d'espace de stockage que le type NUMBER.

Les types PLS\_INTEGER et BINARY\_INTEGER ne se comportent pas de la même manière lors d'erreurs de dépassement (*overflow*). *PLS\_INTEGER* déclenchera l'exception ORA-01426 : dépassement numérique. BINARY\_INTEGER ne provoque aucune exception si le résultat est affecté à une variable NUMBER.

PLS\_INTEGER est plus performant au niveau des opérations arithmétiques que les types NUMBER et BINARY\_INTEGER qui utilisent des librairies mathématiques.

# Sous-types

Chaque type de données PL/SQL prédéfini a ses caractéristiques (domaine de valeurs, fonctions applicables...). Les sous-types de données permettent de restreindre certaines de ces caractéristiques à des données. Un sous-type n'introduit pas un nouveau type mais en restreint un existant. Les sous-types servent principalement à rendre compatibles des applications à la norme SQL ANSI/ISO ou plus pertinentes certaines déclarations de variables.

PL/SQL propose plusieurs sous-types prédéfinis et il est possible de définir des sous-types personnalisés.

Sous-type	Type restreint	Caractéristiques
CHARACTER	CHAR	Mêmes caractéristiques.
INTEGER	NUMBER (38, 0)	Entiers sans décimales.
PLS_INTEGER	NUMBER	Déjà étudié.
BINARY_INTEGER	NUMBER	Déjà étudié.
NATURAL, POSITIVE	BINARY_INTEGER	Non négatif.
NATURALN, POSITIVEN		Non négatif et non nul.
SIGNTYPE		Domaine de valeurs {-1, 0, 1}.
DEC, DECIMAL, NUMERIC	NUMBER	Décimaux, précision de 38 chiffres.
DOUBLE PRECISION, FLOAT, REAL		Flottants.
INTEGER, INT, SMALLINT		Entiers sur 38 chiffres.

## Caractères

Les types CHAR et NCHAR permettent de stocker des chaînes de caractères de taille fixe.

Les types VARCHAR2 et NVARCHAR2 permettent de stocker des chaînes de caractères de taille variable (VARCHAR est maintenant remplacé par VARCHAR2).

Les types NCHAR et NVARCHAR2 permettent de stocker des chaînes de caractères Unicode (*multibyte*), *méthode de codage universelle qui fournit une valeur de code unique pour chaque caractère quels que soient la plate-forme, le programme ou la langue*. Unicode est utilisé par XML, Java, JavaScript, LDAP, et WML. Ces types Oracle sont proposés dans le cadre NLS (*National Language Support*).

Les types CLOB et NCLOB permettent de stocker des flots de caractères (exemple : du texte).

# Les collections

- Ces types de données n'existent qu'en PL/SQL et n'ont pas d'équivalent dans la base Oracle. Il n'est pas possible de stocker un enregistrement directement dans la base
- Une collection est un ensemble ordonné d'éléments de même type. Elle est indexée par une valeur de type numérique ou alphanumérique. Elle ne peut avoir qu'une seule dimension ( mais en créant des collections de collections on peut obtenir des tableaux à plusieurs dimensions)  
On peut distinguer trois types différents de collections :
  - Les tables (INDEX-BY TABLES) qui peuvent être indicées par des variables numériques ou alpha-numériques
  - Les tables imbriquées(NESTED TABLES) qui sont indicées par des variables numériques et peuvent être lues et écrites directement depuis les colonnes d'une table
  - Les tableaux de type VARRAY, indicés par des variables numériques, dont le nombre d'éléments maximum est fixé dès leur déclaration et peuvent être lus et écrits directement depuis les colonnes d'une table

# Déclarations et initialisation

## □ Les collections de type NESTED TABLE et INDEX-BY TABLES:

Elles sont de taille dynamique et il n'existe pas forcément de valeur pour toutes les positions.

Déclaration d'une collection de type nested table:

**TYPE nom type IS TABLE OF type élément [NOT NULL] ;**

Déclaration d'une collection de type index by:

**TYPE nom type IS TABLE OF type élément [NOT NULL] INDEX BY  
index\_by\_type ;**

index\_by\_type représente l'un des types suivants :

**BINARY\_INTEGER**

**PLS\_INTEGER(9i)**

**VARCHAR2(taille)**

**LONG**

# Exemple

declare

-- collection de type nested table

TYPE TYP\_NES\_TAB is table of varchar2(100) ;

-- collection de type index by

TYPE TYP\_IND\_TAB is table of number index by binary\_integer ;

tab1 TYP\_NES\_TAB ;

tab2 TYP\_IND\_TAB ;

Begin

tab1 := TYP\_NES\_TAB('Lundi','Mardi','Mercredi','Jeudi' ) ;

for i in 1..10 loop

tab2(i):= i ;

end loop ;

End;

/

# Les collections de type VARRAY

Ce type de collection possède une dimension maximale qui doit être précisée lors de sa déclaration. Elle possède une longueur fixe et donc la suppression d'éléments ne permet pas de gagner de place en mémoire. Ses éléments sont numérotés à partir de la valeur 1

Déclaration d'une collection de type VARRAY:

**TYPE nom type IS VARRAY (taille maximum) OF type élément [NOT NULL] ;**

**declare**

-- collection de type VARRAY

**TYPE TYP\_VAR\_TAB is VARRAY(30) of varchar2(100) ;**

**tab1 TYP\_VAR\_TAB := TYP\_VAR\_TAB('','','','','','','','','','');**

**Begin**

**for i in 1..10 loop tab1(i):= to\_char(i) ; end loop ;**

**End;**

# Type RECORD

- Equivalent à struct en langage C

- Déclaration

TYPE *nomRecord* IS RECORD ( *champ1 type1*, *champ2 type2*, ...);

- Utilisation du type RECORD

```
TYPE emp2 IS RECORD (num_matr integer, nom varchar(30));
```

```
employe emp2;
```

```
employe.num_matr := 500;
```

# Les enregistrements

**TYPE nom type IS RECORD ( nom\_champ type\_élément [[ NOT NULL ] := expression ] [, ....] ) ;**

**Nom\_variable nom\_type ;**

**Comme pour la déclaration des variables, il est possible d'initialiser les champs lors de leur déclaration**

**declare**

**-- Record –**

**TYPE T\_REC\_EMP IS RECORD (Num emp.empno%TYPE, Nom emp.ename%TYPE, Job emp.job%TYPE );**

**R\_EMP T\_REC\_EMP ; -- variable enregistrement de type T\_REC\_EMP**

**Begin**

**R\_EMP.Num := 1 ;**

**R\_EMP.Nom := 'Scott' ; R\_EMP.job := 'GASMAN' ;**

**End;**

**/**

# Exemple

declare

-- Record --

**TYPE T\_REC\_EMP IS RECORD (Num emp.empno%TYPE, Nom emp.ename%TYPE, Job emp.job%TYPE );**

-- Table de records –

**TYPE TAB\_T\_REC\_EMP IS TABLE OF T\_REC\_EMP index by binary\_integer ;**

**t\_rec TAB\_T\_REC\_EMP ; -- variable tableau d'enregistrements**

**Begin**

**t\_rec(1).Num := 1 ;**

**t\_rec(1).Nom := 'Scott' ;**

**t\_rec(1).job := 'GASMAN' ;**

**t\_rec(2).Num := 2 ; t\_rec(2).Nom := 'Smith' ; t\_rec(2).job := 'CLERK' ;**

**End;**

/

# Exemple

Declare

TYPE Temps IS RECORD ( heures SMALLINT, minutes SMALLINT,  
secondes SMALLINT );

TYPE Vol IS RECORD

(

numvol PLS\_INTEGER,

Numavion VARCHAR2(15),

Commandant Employe, -- type objet

Passagers ListClients, -- type nested table

depart Temps, -- type record

arrivee Temps -- type record

);

Begin

...

End ;

# Initialisation des collections

Les collections de type **NESTED TABLE** et **VARRAY** doivent être initialisées avant toute utilisation (à l'exception des collections de type **INDEX-BY TABLE**).

Pour initialiser une collection, il faut se référer à son constructeur. Celui-ci, créé automatiquement par Oracle et porte le même nom que la collection.

**Declare**

-- Déclaration d'un type tableau **VARRAY** de 30 éléments de type  
--Varchar2(100)

**TYPE TYP\_VAR\_TAB is VARRAY(30) of varchar2(100) ;** -- Déclaration et  
initialisation d'une variable de type **TYP\_VAR\_TAB t**  
**ab1 TYP\_VAR\_TAB := TYP\_VAR\_TAB('','','','','','','','','','');**

# Accès aux éléments d'une collection

La syntaxe d'accès à un élément d'une collection est la suivante :

**Nom\_collection( indice )**

Indice peut être un littéral, une variable ou une expression

**Declare**

```
Type TYPE_TAB_EMP IS TABLE OF Varchar2(60) INDEX BY  
BINARY_INTEGER ;  
emp_tab TYPE_TAB_EMP ;  
i pls_integer ;
```

**Begin**

```
For i in 0..10 Loop  
emp_tab( i+1 ) := 'Emp ' || ltrim( to_char( i ) ) ;  
End loop ;
```

**End ;**

**/**

**LTRIM (str): Efface tous les espaces blancs situés en début de chaîne.**

**RTRIM (str): Efface tous les espaces blancs situés en fin de chaîne.**

# Accès aux éléments d'une collection

Declare

```
Type TYPE_TAB_JOURS IS TABLE OF PLS_INTEGER INDEX BY  
VARCHAR2(20) ;
```

```
jour_tab TYPE_TAB_JOURS ;
```

Begin

```
jour_tab( 'LUNDI' ) := 10 ;
```

```
jour_tab( 'MARDI' ) := 20 ;
```

```
jour_tab( 'MERCREDI' ) := 30 ;
```

End ;

/

# Accès aux éléments d'une collection

**Declare**

Type TYPE\_TAB\_EMP IS TABLE OF EMP%ROWTYPE INDEX BY  
BINARY\_INTEGER ;

Type TYPE\_TAB\_EMP2 IS TABLE OF EMP%ROWTYPE INDEX BY  
BINARY\_INTEGER ;

tab1 TYPE\_TAB\_EMP := TYPE\_TAB\_EMP( ... );

tab2 TYPE\_TAB\_EMP := TYPE\_TAB\_EMP( ... );

tab3 TYPE\_TAB\_EMP2 := TYPE\_TAB\_EMP2( ... );

**Begin**

tab2 := tab1 ; -- OK

tab3 := tab1 ; -- Illégal : types différents

...

**End ;**

# Accès aux éléments d'une collection

- Les collections ne peuvent pas être comparées entre elles.
- Exemple:

Declare

```
Type TYPE_TAB_STRING IS TABLE OF Varchar2(10);
tab1 TYPE_TAB_STRING := TYPE_TAB_STRING( '1','2','3');
tab2 tab1%TYPE := TYPE_TAB_STRING( '1','2','3');
```

Begin

```
If tab1 = tab2 Then null ;
End if ;
```

```
End ;
```

/

# Les collections TABLE

## Une table PL/SQL

- c'est une collection ordonnée d'élément du même type
- accessible uniquement en PL/SQL
- stockés en mémoire, ils peuvent grandir dynamiquement
- ils utilisent des index non consécutif
- ils ont le même format que des champs d'une table
- on n'utilise pas sql pour s'en servir
- Définition :

TYPE nom\_tableau\_pl IS TABLE OF datatype [NOT NULL]

INDEX BY BINARY\_INTEGER;

nom\_tableau\_pl nom du tableau

datatype type (curseur, record, variable...)

Declaration :

mon\_tableau nom\_tableau\_pl ;

# Les collections TABLE: exemple

-- table de multiplication par 8 et par 9...

declare

type tablemul is record ( par8 number, par9 number);

type tabledentiers is table of tablemul index by binary\_integer;

ti tabledentiers;

i number;

begin

for i in 1..10 loop

ti(i).par9 := i\*9 ;

ti(i).par8:= i\*8;

dbms\_output.put\_line (i||'\*8='||ti(i).par8||' '||i||'\*9='||ti(i).par9 );

end loop;

end;

/

# Multidimensional arrays

- Ecrire le code en commentaire dans un éditeur de code puis analyses le.

# Fonctions pour les tableaux

□ PL/SQL propose un ensemble de fonctions qui permettent de manipuler des tableaux (également disponibles pour les *nested tables et varrays*). *Ces fonctions sont les suivantes:*

□ Fonction Description

**EXISTS(x)** *Retourne TRUE si le xeme élément du tableau existe.*

**COUNT** Retourne le nombre d'éléments du tableau.

**FIRST / LAST** Retourne le premier/dernier indice du tableau (NULL si tableau vide).

**PRIOR(x) / NEXT(x)** *Retourne l'élément avant/après le xeme élément du tableau.*

**DELETE**

**DELETE(x)**

**DELETE(x,y)** *Supprime un ou plusieurs éléments au tableau.*

□ Il n'est pas possible actuellement d'appeler une de ces fonctions dans une instruction SQL (SELECT, INSERT, UPDATE ou DELETE).

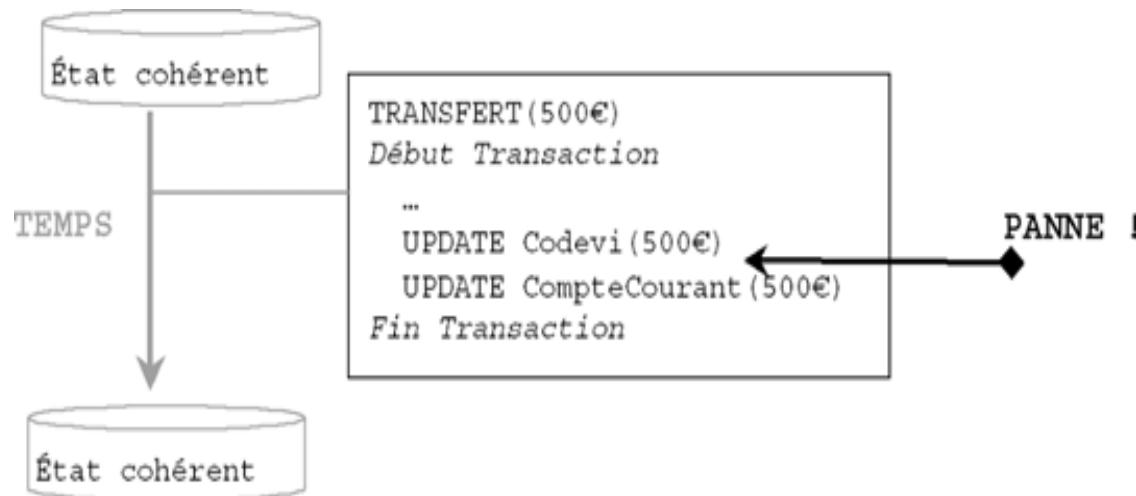
# Méthodes pour les tableaux et varray

nom_table_sql.COUNT	retourne le nombre d'éléments de la table
nom_table_sql.DELETE	supprime toute la table
nom_table_sql.DELETE (n)	supprime l'élément n
nom_table_sql.DELETE (n,m)	supprime les éléments de n à m
nom_table_sql.EXISTS (n)	retourne TRUE si l'élément n existe
nom_table_sql.EXTEND	
nom_table_sql.EXTEND	
nom_table_sql.EXTEND	
nom_table_sql.FIRST	retourne le premier index
nom_table_sql.LAST	retourne la valeur du dernier index
nom_table_sql.TRIM	élément suivant
nom_table_sql.TRIM(n)	
nom_table_sql.NEXT(n)	
nom_table_sql.PRIOR(n)	élément précédent

# Transactions

- Une transaction est un bloc d'instructions LMD faisant passer la base de données d'un état cohérent à un autre état cohérent. Si un problème logiciel ou matériel survient au cours d'une transaction, aucune des instructions contenues dans la transaction n'est effectuée, quel que soit l'endroit de la transaction où est intervenue l'erreur.
- Le modèle le plus simple et le plus frappant d'une transaction est celui du transfert d'un compte épargne vers un compte courant. Imaginez qu'après une panne votre compte épargne a été débité de la somme de 500 € sans que votre compte courant ait été crédité de ce même montant ! Vous ne seriez pas très content, sans doute, des services de votre banque. Le mécanisme transactionnel empêche cet épisode fâcheux en invalidant toutes les opérations faites depuis le début de la transaction si une panne survient au cours de cette même transaction.

# Transactions



## Caractéristiques

Une transaction assure :

- l'atomicité des instructions qui sont considérées comme une seule opération (principe du tout ou rien) ;
- la cohérence (passage d'un état cohérent de la base à un autre état cohérent) ;
- l'isolation des transactions entre elles (lecture consistante, mécanisme décrit plus loin) ;
- la durabilité des opérations (les mises à jour perdurent même si une panne se produit après la transaction).

# Début et fin d'une transaction

- Il n'existe pas d'ordre PL/SQL ou SQL qui marque le début d'une transaction. Ainsi le BEGIN d'un programme PL/SQL n'est pas forcément synonyme de son commencement.
- Une transaction débute à la première commande SQL rencontrée ou dès la fin de la transaction précédente.
- Une transaction se termine explicitement par les instructions SQL COMMIT ou ROLLBACK. Elle se termine implicitement :
  - à la première commande SQL du LDD ou du LCD rencontrée ;
  - à la fin normale d'une session utilisateur avec déconnexion ;
  - à la fin anormale d'une session utilisateur (sans déconnexion).

Exemple:

**COMMIT:**

    Commande SQL (LDD ou LCD)

    Fin normale d'une session.

**ROLLBACK:**

    Fin anormale d'une session.

# Transaction: Point de validation

- Vous pouvez tester rapidement une partie de ces caractéristiques en écrivant le bloc suivant qui insère une ligne dans une de vos tables :

**COMMIT;**

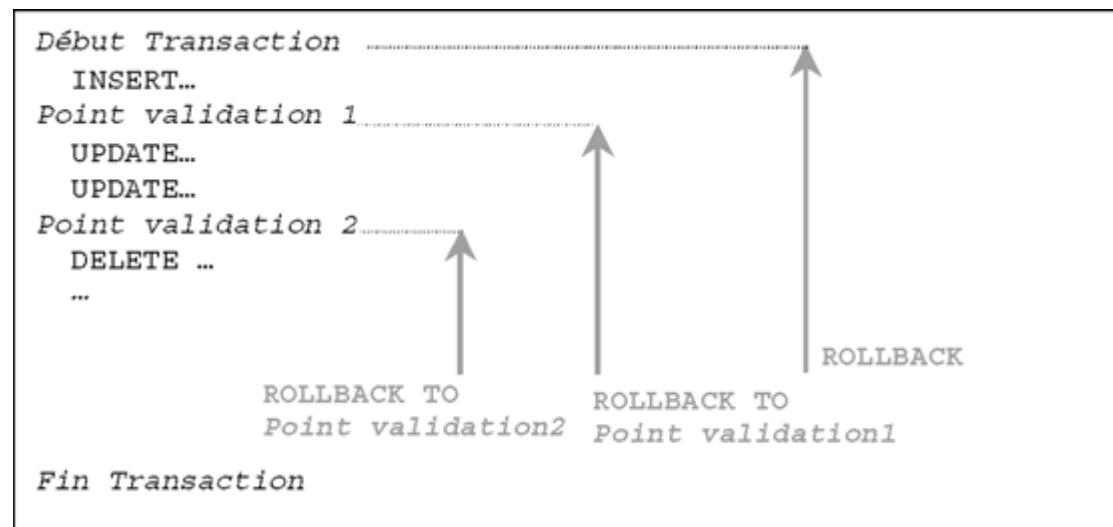
**BEGIN**

**INSERT INTO *TableàVous* VALUES (...);**

**END;**

/

**SELECT \* FROM *TableàVous*;**



# Transaction: Exemple

Code PL/SQL	Commentaires
BEGIN	Première partie de la transaction.
INSERT INTO Compagnie VALUES ('C2', 2, 'Place Brassens', 'Blagnac', 'Easy Jet');	
SAVEPOINT P1;	Deuxième partie de la transaction.
UPDATE Compagnie SET nrue = 125 WHERE comp = 'AF';	
UPDATE Compagnie SET ville = 'Castanet' WHERE comp = 'C1';	
SAVEPOINT P2;	Troisième partie de la transaction.
DELETE FROM Compagnie WHERE comp = 'C1';	
-- ROLLBACK TO P1;	Première partie à valider.
-- ROLLBACK TO P2;	Deuxième partie à valider.
-- ROLLBACK TO P3	Troisième partie à valider.
ROLLBACK;	Tout à invalider.
COMMIT;	Valide la ou les sous-parties.
END;	

# Exemple: savepoint

**Begin**

```
insert into EMP(empno, ename, job) values( 9991, 'Dupontont', 'CLERK' );
insert into EMP(empno, ename, job ) values( 9992, 'Duboudin', 'CLERK' );
SAVEPOINT mise_a_jour ;
Update EMP Set sal = 2500 Where empno > 9990 ;
ROLLBACK TO SAVEPOINT mise_a_jour ;
Commit ;
```

**End ;**

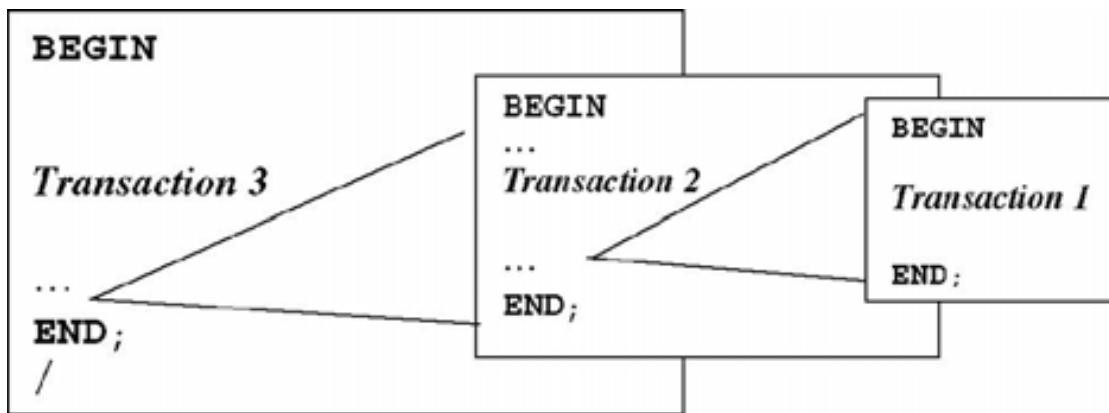
/ Procédure PL/SQL terminée avec succès.

```
SQL> Select * From EMP Where empno > 9990 ;
EMPNO ENAME JOB MGR HIREDATE SAL COMM DEPTNO
```

---

9991	Dupontont	CLERK
9992	Duboudin	CLERK

# Transactions imbriquées



# **BASES DE DONNÉES AVANCÉES: LES CURSEURS**

# Introduction

## □ Manipulation de tuples:

- ❖ Les directives INSERT INTO, UPDATE et DELETE FROM peuvent être utilisées sans restriction avec des variables PL/SQL (scalaires, %ROWTYPE)

## □ Lecture de tuples Chargement d'une variable à partir de la lecture d'un unique enregistrement dans la base (exception si 0 ou plusieurs enregistrements en réponse)

```
DECLARE
    heure_depart Vols.depart%TYPE;
BEGIN
    SELECT Vols.depart INTO heure_depart
    FROM Vols WHERE Vols.id = 'AF3517';
END;
```

# Curseurs

- Oracle crée des zones de travail pour exécuter les ordres SQL, stocker leurs résultats et les utiliser: Curseur
- Toutes les requêtes SQL sont associées à un curseur:
  - Un curseur est un pointeur vers un résultat d'une requête
  - Le curseur peut être implicite (pas déclaré par l'utilisateur) ou explicite.
  - Les curseurs explicites permettent de manipuler l'ensemble des résultats d'une requête.

# Pourquoi utiliser les curseur

- Les instructions de type **SELECT ... INTO ...** manquent de souplesse, elles ne fonctionnent que sur des requêtes retournant une et une seule valeur. Ne serait-il pas intéressant de pouvoir placer dans des variables le résultat d'une requête retournant plusieurs lignes ?

# Fonctionnalités

- Toutes les requêtes SQL sont associées à un curseur.
- Ce curseur représente la zone mémoire utilisée pour *parser 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.

# 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)

# Types de curseurs

## □ Curseurs implicites:

- Créer automatiquement par Oracle lorsque la clause **INTO** accompagne le **select**
- Pour toute requête SQL du LMD et pour les interrogations qui retournent un seul enregistrement

## □ Curseurs explicite

- Déclaré et manipulés par le programmeur pour pouvoir traiter le résultat de requêtes retournant plus d'un tuple
- L'utilisation d'un curseur explicite nécessite les étapes suivantes:
  1. Déclaration du curseur: section **DECLARE**
  2. Ouverture du curseur: section **BEGIN**
  3. Traitement des lignes: section **BEGIN**
  4. Fermeture du curseur: section **BEGIN** ou **EXCEPTION**

# Curseurs

## □ Définitions

- les échanges entre l'application et la base de données sont réalisés grâce à des curseurs: se sont des zones de travail capables de stocker plusieurs enregistrements et de gérer l'accès à des enregistrements
- Stocke le résultat d'une requête retournant plusieurs enregistrements
- Permet un traitement séquentiel des enregistrements retournés

## □ Déclaration

**□ Utilisation**

```
CURSOR nomcurseur IS requête;
OPEN nomcurseur;
LOOP
    FETCH nomcurseur INTO variable;
    EXIT WHEN condition;
    instructions;
    ...
END LOOP;
CLOSE nomcurseur;
```

# Curseur implicite: exemple

```
BEGIN
```

```
    FOR x IN (SELECT * FROM emp WHERE sal < 100) LOOP
        dbms_Output.Put_Line(x.eName ||' '||x.sal||'... should REALLY be
                             raised :D');
    END LOOP;
END; /
```

# Curseurs

## □ Attributs associés aux curseurs

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

**%FOUND** est égal à vrai (« TRUE ») si la commande FETCH retourne un enregistrement

**%NOTFOUND** est égal à faux (« FALSE ») si la commande FETCH retourne un enregistrement

**%ROWCOUNT** retourne le nombre d'enregistrements constituant le curseur

**%ISOPEN** indique si le curseur est dans un état ouvert (utilisable)

## □ Gestion automatique d'un curseur

~~-OPEN~~ ...

**LOOP**

~~-FETCH~~ ...

EXIT WHEN ...

...

**END LOOP;**

~~-CLOSE~~ ...

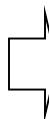
**FOR variable INTO nomcurseur LOOP**

...

EXIT WHEN ...

...

**END LOOP;**



...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

...

# Curseurs implicites

- Nom: **SQL**; actualisé après chaque requête LMD et chaque select non associé à un curseur explicite
- A travers ses attributs, permet au programme PL/SQL d'obtenir des infos sur l'exécution des requêtes:
  - **SQL%FOUND**: TRUE si la dernière requête LMD a affecté au moins 1 enregistrement
  - **SQL%NOTFOUND**: TRUE si la dernière requête LMD n'a affecté aucun enregistrement
  - **SQL%ROWCOUNT**: nombre de lignes affectées par la requête LMD

**DELETE FROM pilote WHERE age >= 55;**

**DBMS\_OUTPUT.PUT\_LINE(SQL%ROWCOUNT ||'pilotes partis à la retraite');**

# Curseurs explicites

## □ Les étapes de la vie d'un curseur

Les étapes d'utilisation d'un curseur explicite pour traiter un ordre select suivantes:

- ✓ Déclaration du curseur: CURSOR ... IS ...
- ✓ Utilisation du curseur:
  - ouverture du curseur: OPEN
  - Traitement des lignes: FETCH ou FOR
  - Fermeture du curseur: CLOSE

# Curseur explicite: Déclaration

- Tout curseur explicite utilisé dans un bloc PL/SQL doit être déclaré dans la section DECLARE du bloc, en précisant son nom et l'ordre sql associé.
- La syntaxe de déclaration d'un curseur explicite est la suivante:

CURSOR NOM\_DU\_CURSEUR IS REQUETE;

La requête peut contenir tous les ordres SQL d'intérrogation de données, y compris les opérateurs ensembliste UNION, INTERSECT ou MINUS.

**DECLARE**

**CURSOR Liste\_des\_Pilotes IS Select \* from pilote;**

**NB: Les expressions, calcul ou fonction SQL, dans la clause select de la requête du curseur, doivent comporter un alias pour pouvoir être référencées.**

# Curseur explicite exemple

- La table utilisée est: employe (... , nomemp, sal, ville, ...)

**DECLARE**

**CURSOR emp\_rabat IS select nomemp, sal FROM employe WHERE  
ville='rabat';**

**nom employe.nomemp%type; salaire employe.sal%type;**

**BEGIN**

**OPEN emp\_rabat;**

**FETCH emp\_rabat INTO nom, salaire;**

**WHILE emp\_rabat%found LOOP**

**....**

**END LOOP;**

**END;**

**/**

- La table utilisée est: employe (... , nomemp, sal, ville, ...)

**DECLARE**

**CURSOR emp\_rabat IS select nomemp, sal FROM employe WHERE  
ville='rabat';**

**nom employe.nomemp%type; salaire employe.sal%type;**

**BEGIN**

**OPEN emp\_rabat;**

**FETCH emp\_rabat INTO nom, salaire;**

**WHILE emp\_rabat%found LOOP**

**IF salaire>3000 THEN**

**INSERT INTO resultat VALUES (nom,salaire);**

**END IF;**

**FETCH emp\_rabat INTO nom, salaire;**

**END LOOP;**

**END;**

# Curseur explicite: utilisation

```
OPEN Nom curseur; -- ouverture du curseur
LOOP
    FETCH nom curseur INTO variable; -- Traitement des lignes d'un curseur
    EXIT WHEN conditions;
Instructions;
```

...

```
END LOOP;
CLOSE nom curseur; -- fermeture du curseur
```

Remarque:

L'utilisation de **FOR LOOP** remplace **OPEN,FETCH and CLOSE.**

Lorsque le curseur est invoqué, un enregistrement est automatiquement créé avec les mêmes éléments de données que ceux définies dans l'instruction **SELECT**

# Curseur explicite: Attributs

- Pour chaque exécution d'un ordre de manipulation du curseur, le noyau renvoie une information appelée statut, qui indique si l'ordre a été exécuté avec succès ou non. Cette information est disponible dans le programme par l'intermédiaire de 4 attributs rattachés à chaque curseur.

<b>%FOUND</b>	Est égal à vrai(« true ») si la commande FETCH retourne un enregistrement
<b>%NOTFOUND</b>	Est égal à faux(« FALSE ») si la commande FETCH retourne un enregistrement
<b>%ISOPEN</b>	Indique si le curseur est en état ouvert(utilisable)
<b>%ROWCOUNT</b>	Retourne le nombre d'enregistrements constituant le curseur

**La syntaxe de consultation d'un attribut est : **NOM\_CURSOR%Attribut;****

# Curseur explicite: FETCH

- La commande FETCH ne retourne qu'un enregistrement. Pour récupérer l'ensemble des enregistrements de l'ordre SQL, il faut prévoir une boucle.

```
FETCH NOM_CURSOR INTO {NOM_ENREGISTREMENT| NOM_VARIABLE[,...]};
```

```
DECLARE
```

```
    Nom Pilote.plnom%TYPE;
```

```
    CURSOR Liste_pilote IS Select plnom From Pilote order by plnom;
```

```
    -- Enreg Liste_pilote%Rowtype;
```

```
BEGIN
```

```
    OPEN liste_pilote;
```

```
LOOP
```

```
    FETCH liste_pilote into nom; -- fetch liste_pilote into enreg;
```

```
    Exit when liste_pilote%notfound;
```

```
    dbms_output.put_line(nom); -- traitement ou affichage
```

```
    --dbms_output.put_line(Enreg.plnom);
```

```
    End loop;
```

```
End liste_pilote; End; /
```

# Curseur implicite: FOR

- La boucle FOR déclare implicitement la variable de parcours, ouvre le curseur, réalise les FETCH successifs et ferme le curseur.

```
FOR nom_enregistrement in NOM_CURSEUR LOOP
    INSTRUCTIONS;
END LOOP;
SET SERVEROUTPUT ON;
DECLARE
    CURSOR liste_pilote IS Select plnom from pilote order by plnom;
BEGIN
    FOR v_pilote IN liste_pilote LOOP
        DBNS_OUTPUT.PUT_LINE(v_pilote.plnom); -- traitement ou affichage
    END LOOP;
END
/
```

# Les 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)

**CURSOR nomCurseur (param1, param2,...) IS ...;**

```
SET SERVEROUTPUT ON;
DECLARE
CURSOR c(Numpilote Integer) IS SELECT plnom from pilote WHERE npilote = Numpilote;
BEGIN
FOR v_pilote in c(1) LOOP
    DBMS_OUTPUT.PUT_LINE(v_pilote.plnom);
END LOOP;
FOR v_pilote in c(2) LOOP
    DBMS_OUTPUT.PUT_LINE(v_pilote.plnom);
END LOOP;
END;
/
```

# Curseurs et Verrouillage

- Objectifs: Lorsqu'un curseur est ouvert: verrouiller l'accès aux colonnes référencées des lignes retournées par la requête afin de pouvoir les modifier.

CURSOR CursorName[(parametres)] IS SELECT listeColonnes1 FROM tableName WHERE condition

FOR UPDATE [OF listeColonnes2] [NOWAIT|WAIT NB\_SECONDES]

- Absence de OF: toutes les colonnes sont verrouillées
- NOWAIT demande à Oracle de verrouiller les enregistrements correspondants immédiatement si les enregistrements sont déjà verrouillés; alors l'ouverture du curseur provoque une erreur.
- WAIT demande à Oracle de verrouiller les enregistrements correspondants si les enregistrements sont déjà verrouillés; alors le programme attend NB\_SECONDES pour le déverrouillage, si non l'ouverture du curseur provoque une erreur

# Modification des lignes verrouillées

- Restrictions: DISTINCT, GROUP BY, opérateurs ensemblistes et fonctions de groupe ne sont pas utilisables dans les curseurs FOR UPDATE
- La clause CURRENT OF permet d'accéder directement en modification ou en suppression à la ligne qui vient de ramener l'ordre FETCH ou FOR

**DECLARE**

```
CURSOR C1 IS SELECT plnom, sal FROM pilote  
FOR UPDATE OF sal WAIT 100;  
prime pilote.sal%type := 1000;
```

**BEGIN**

```
FOR salaireActuel in C1 LOOP  
    update pilote SET sal = sal + prime  
    WHERE CURRENT OF C1;  
END LOOP;
```

**END;**

/

# Les curseurs dynamiques

- Un curseur dynamique n'est pas lié à une requête comme un curseur statique.
- Une variable de type curseur permet au curseur d'évoluer au cours du programme en lui associant diverses clauses SQL.

Déclaration

```
Declare  
    TYPE nomTypeCurDyn IS REF CURSOR;  
    nomCurseur nomTypeCurDyn;  
BEGIN  
    ... OPEN nomCurseur FOR 'requête dynamique';  
    LOOP  
        FETCH nomCurseur INTO ...  
        EXIT WHEN ...  
        ...  
    END LOOP;  
    CLOSE nomCurseur;  
END;  
/
```

Utilisation

# Curseurs RECAP

## □ Parcours complet

```
DECLARE
    CURSOR liste_etudiants IS
        SELECT numetu, nometu FROM ETUDIANTS;
BEGIN
    FOR enreg INTO liste_etudiants LOOP
        DBMS_OUTPUT.PUT_LINE(enreg.numetu || ' - '
                             || enreg.nometu);
    END LOOP;
END;
```

-- déclaration du curseur  
-- parcours du curseur  
-- traitement (affichage)

## □ Parcours personnalisé

```
DECLARE
    CURSOR liste_etudiants IS
        SELECT numetu, nometu FROM ETUDIANTS;
    enreg liste_etudiants%ROWTYPE;
BEGIN
    OPEN liste_etudiants;
    LOOP
        FETCH liste_etudiants INTO enreg;
        EXIT WHEN liste_etudiants%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(enreg.numetu || ' - '
                             || enreg.nometu);
    END LOOP;
    CLOSE liste_etudiants;
END;
```

-- déclaration du curseur  
-- ouverture du curseur  
-- consommation d'une ligne  
-- test de sortie de boucle  
-- traitement (affichage)  
-- fermeture du curseur

# CURSOR RECAP

DECLARE

    TYPE curseurDyn IS REF CURSOR; -- type of cursor is dynamique

    ListePilote curseurDyn; -- declaration of cursor

    Num pilote.npilote%TYPE;

    Nom pilote.plnom%TYPE;

BEGIN

    OPEN liste\_pilote FOR SELECT npilote, plnom FROM pilote;

    LOOP

        FETCH liste\_pilote INTO num, nom;

        EXIT WHEN liste\_pilote%NOTFOUND;

        DBMS\_OUTPUT.PUT\_LINE(num||'-'||nom);

    END LOOP;

    CLOSE liste\_pilote;

END;

/

# Curseurs Récapitulatif

## Exemple de curseur dynamique

```
DECLARE
    TYPE curseur_dynamique IS REF CURSOR; -- type curseur
    dynamique
    liste_etudiants curseur_dynamique;      -- déclaration du
    curseur
    enreg liste_etudiants%ROWTYPE;
BEGIN
    OPEN liste_etudiants FOR
        -- ouverture du
    curseur
    SELECT numetu, nometu FROM ETUDIANTS;
    LOOP
        FETCH liste_etudiants INTO enreg; -- consommation
        d'une ligne
        EXIT WHEN liste_etudiants%NOTFOUND; -- test de sortie
        de boucle
        DBMS_OUTPUT.PUT_LINE(enreg.numetu || ' - ' --
        traitement (affichage)
                                || enreg.nometu);
    END LOOP;
    CLOSE liste_etudiants; -- fermeture du curseur
END;
```

# Exception

## □ Principe

- Un identificateur PL/SQL produit au cours de l'exécution suite à une erreur oracle  
Elle c'est produite suite à des erreurs interne d'oracle ou bien suite à des anomalies dues au programme
- Alors chaque erreur à l'exécution déclenche une exception
- Intérêts :
  - Traitement systématique des erreurs
  - Traitement groupé / séparé d'erreurs

## □ Fonctions PL/SQL

- SQLCODE : code de la dernière exception levée
- SQLERRM : message d 'erreur associé à une exception

## □ Comment on traite les exceptions:

- On l'intercepte avec un programme de traitement des exceptions
- on la transmet à l'environnement extérieur

# EXCEPTIONS

## □ Structure:

### **EXCEPTION**

```
WHEN nomexception1 THEN instructions1;  
WHEN nomexception2 THEN instructions2;  
...  
[ WHEN OTHERS THEN instructionsn; ]
```

# Exemple: programme complet

```
SET SERVEROUTPUT ON;
```

```
ACCEPT nv PROMPT 'Numéro du vol'
```

```
BEGIN
```

```
    insert into vol values ( '&nv',to_date('01-05-2009 15:35:00','DD-MM-YYYY  
HH24:MI:SS'),to_date('01-05-2009 19:00:00','DD-MM-YYYY HH24:MI:SS'),  
'TOKYO','BEJING' );
```

```
    DBMS_OUTPUT.PUT_LINE('Données Enregistrées');
```

```
EXCEPTION
```

```
    WHEN DUP_VAL_ON_INDEX THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Doublon interdit dans une clé primaire');
```

```
    WHEN OTHERS THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Autres types d'erreurs détectées');
```

```
END;
```

```
/
```

Exception Name	Reason	Error Number
CURSOR_ALREADY_OPEN	When you open a cursor that is already open.	ORA-06511
INVALID_CURSOR	When you perform an invalid operation on a cursor like closing a cursor, fetch data from a cursor that is not opened.	ORA-01001
NO_DATA_FOUND	When a SELECT...INTO clause does not return any row from a table.	ORA-01403
TOO_MANY_ROWS	When you SELECT or fetch more than one row into a record or variable.	ORA-01422
ZERO_DIVIDE	When you attempt to divide a number by zero.	ORA-01476

# Exception

## □ Quelques exceptions prédéfinies

	<b>Nom</b>	<b>Code d'erreur</b>	<b>Code SQL</b>
	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	-1403
	NOT_LOGGED_ON	ORA-01012	-1012
	PROGRAM_ERROR	ORA-06501	-6501
	STORAGE_ERROR	ORA-06500	-6500
	TIMEOUT_ON_RESOURCE	ORA-00051	-51
(*)	TOO_MANY_ROWS	ORA-01422	-1422
	VALUE_ERROR	ORA-06502	-6502
	ZERO_DIVIDE	ORA-01476	-1476

# Exception

## □ Exemple

ETUDIANTS		
numetu	nometu	nbf
1	Jojo	1
2	Mimi	2
3	Polo	0

FORMATIONS	
numf	nomf
OR11	Oracle 11g
BDR	BD Objet-Relationnelles

INSCRIPTIONS		
numf	numetu	datei
OR11	2	22/09/2003
BDR	1	19/09/2003
OR11	2	22/09/2003

```
SET SERVEROUTPUT ON; -- Affichages visibles sur le poste client
```

```
ACCEPT ne PROMPT 'Numero de l''etudiant a ajouter:'
```

```
ACCEPT nm PROMPT 'Nom de l''etudiant:'
```

```
BEGIN
```

```
    INSERT INTO etudiant VALUES (&nf, '&nm', SYSDATE);
```

```
EXCEPTION
```

```
    WHEN DUP_VAL_ON_INDEX THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Violation clé primaire');
```

```
    WHEN OTHERS THEN
```

```
        DBMS_OUTPUT.PUT_LINE('Erreur détectée');
```

```
END;
```

```
/
```

# Exception définies par l'utilisateur

## 1. - Déclaration (Segment de déclaration, nommer l'exception)

**DECLARE**

nomexception **EXCEPTION**;

## 2. - Déclenchement de l'exception( produire: segment exécutable)

...

**IF** condition **THEN**  
    **RAISE** nomexception;  
**END IF**;

...

## 3. - Traitemennt de l'exception( segment traitement des exceptions): traiter l'exception produite

**EXCEPTION**

**WHEN** nomexception **THEN** instructions;

# Exception: Exemple

```
SET SERVEROUT ON;

ACCEPT ne PROMPT 'Numero de l''etudiant a enregistrer:'
ACCEPT nme PROMPT 'Numero de l''etudiant a enregistrer:'

DECLARE
    n INTEGER;
    except_cle EXCEPTION;
BEGIN
    SELECT COUNT(*) INTO n
    FROM ETUDIANTS WHERE numetu = &ne;
    IF (n>0) THEN
        RAISE except_cle;
    ELSE
        INSERT INTO ETUDIANTS VALUES (&ne, '&nme', 0);
    END IF;
EXCEPTION
    WHEN except_cle THEN
        DBMS_OUTPUT.PUT_LINE('Ce numero d''etudiant existe déjà!');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Autres Erreurs détectées');
END;
/
```

ETUDIANTS		
numetu	nometu	nbf
1	Jojo	1
2	Mimi	2
3	Polo	0

# EXCEPTION: Exemple

DECLARE

    CURSOR liste\_pilote IS select nom FROM pilote;

BEGIN

    OPEN liste\_pilote;

    FOR v\_pilote IN liste\_pilote LOOP

        DBMS\_OUTPUT.PUT\_LINE(v\_pilote.plnom);

    END LOOP;

EXCEPTION

    WHEN CURSOR\_ALREADY\_OPEN THEN

        DBMS\_OUTPUT.PUT\_LINE('le curseur est déjà ouvert');

    WHEN OTHERS THEN

        DBMS\_OUTPUT.PUT\_LINE('autres erreurs c'est produite');

END;

/

# Les Exceptions et les blocs imbriqué

Il n'est pas possible de déclarer la même exception deux fois dans le même bloc.  
Toutefois, dans le cas de blocs imbriqués, vous pouvez déclarer la même exception dans la section EXCEPTION de chaque bloc

**DECLARE**

**EXC1 Exception ;**

**BEGIN**

**DECLARE**

**EXC1 Exception ;**

**BEGIN**

**... EXCEPTION**

**WHEN EXC1 Then ...**

**END ;**

**EXCEPTION**

**WHEN EXC1 Then ...**

**END ;**

## **EXCEPTION**

**WHEN NO\_DATA\_FOUND Then ...**

**WHEN OTHERS THEN**

If     SQLCODE = ...

Then ...

*Elsif SQLCODE ≡ ...*

Then .....  
End if ;

**END;**

# Fonctions SQLCODE et SQLERRM

**Le code erreur numérique Oracle ayant généré la plus récente erreur est récupérable en interrogeant la fonction SQLCODE.**

**Le libellé erreur associé est récupérable en interrogeant la fonction SQLERRM**

**Declare**

**LC\$Chaine varchar2(10) ;**

**Begin**

**LC\$Chaine := rpad('a', 30, 'b') ;**

**Exception**

**when others then**

**dbms\_output.put\_line( 'Code erreur : ' || to\_char( SQLCODE ) ) ;**

**dbms\_output.put\_line( 'libellé erreur : ' || to\_char( SQLERRM ) ) ;**

**End ;**

**/**

**Code erreur : -6502**

**libellé erreur : ORA-06502:**

**PL/SQL: numeric or value error: character string buffer too small Procédure PL/SQL terminée avec succès.**

# RPAD and LPAD

**LPAD(char1,n,[char2]):** Fait précéder la chaîne de caractères char1 de la chaîne de caractères char2 (espace si char2 non spécifié) jusqu'à la longueur n. Si n est inférieur à la longueur de char1, tronque char1 à la longueur n.

**LPAD('Juin', 9,'=')**= '=====Juin'

**LPAD('Juin',2,'=')**='Ju'

```
begin
    for i in 1 .. 15
        loop
            dbms_output.put_line( lpad('string', i) || '<' );
        end loop;
    end;
/

```

# Associer un code ERR à une var Exception

On peut associer un code erreur Oracle à nos propres variables exception à l'aide du mot clé **PRAGMA EXCEPTION\_INIT**, dans le cadre de la section déclarative de la façon suivante :

```
Nom_exception EXCEPTION ;  
PRAGMA EXCEPTION_INIT(nom_exception, -code_error_oracle);
```

Exemple :

Lorsque l'on tente d'insérer plus de caractères dans une variable que sa déclaration ne le permet, Oracle déclenche une erreur **-6502**. Nous allons "nommer" cette erreur en **LE\$trop\_long** et l'intercepter dans la section exception

# Associer un code ERR à une var Exception

**Declare**

```
LC$Chaine varchar2(10) ;  
LE$trop_long exception ;  
pragma exception_init( LE$trop_long, -6502 ) ;
```

**Begin**

```
LC$Chaine := rpad( ' ', 30) ;
```

**Exception**

```
when LE$trop_long then  
dbms_output.put_line( 'Chaîne de caractères trop longue') ;
```

**End ;**

/

**Chaîne de caractères trop longue**

**Procédure PL/SQL terminée avec succès.**

**SQL>**

# Poursuite de l'exécution après l'interception d'une exception

**Une fois dans la section EXCEPTION, il n'est pas possible de retourner dans la section exécution juste après l'instruction qui a généré l'erreur.**

**Par contre il est tout à fait possible d'encadrer chaque groupe d'instructions voire même chaque instruction avec les mots clé**

**BEGIN ... EXCEPTION ... END;**

**Cela permet de traiter l'erreur et de continuer l'exécution**

# Poursuite de l'exécution après l'interception d'une exception

Declare

```
LC$Ch1 varchar2(20) := 'Phrase longue';
LC$Chaine varchar2(10) ;
LE$trop_long exception ;
pragma exception_init( LE$trop_long, -6502 ) ;
```

Begin

Begin

```
LC$Chaine := LC$Ch1;
```

Exception

when LE\$trop\_long then

```
LC$Chaine := Substr( LC$Ch1, 1, 10 ) ;
```

End ; -- poursuite du traitement –

```
dbms_output.put_line(LC$Chaine ) ;
```

End ;

/ Phrase lon

**Procédure PL/SQL terminée avec succès.**

# Utiliser votre propre message d'erreur

Vous pouvez également définir vos propres messages d'erreur avec la procédure RAISE\_APPLICATION\_ERROR

`DBMS_STANDARD.raise_application_error(numero_erreur, message[, {TRUE | FALSE}])`

numero\_erreur représente un entier négatif compris entre -20000 et -20999

message représente le texte du message d'une longueur maximum de 2048 octets

TRUE indique que l'erreur est ajoutée à la pile des erreurs précédentes

FALSE indique que l'erreur remplace toutes les erreurs précédentes

# Sous-programmes: Procédures cataloguées

## Principe

- Bloc PL/SQL nommé stocké dans la BD
- Utilisation par plusieurs utilisateurs
- Exécution à partir des applications ou d'autres procédures cataloguées

## Structure

```
CREATE [ OR REPLACE ] PROCEDURE nomprocedure
    [ (parametre1,..., parametren) ]
    [AUTHID { CURRENT_USER | DEFINER }]
    [IS]
        [ variables_internes ]
BEGIN
    instructions
    [EXCEPTION ...]
END [ nomprocedure ];
/
nomvariable [IN | OUT | IN OUT] type [{ := | DEFAULT} valeur]
```

# Procédures

- CREATE indique que l'on veut créer une procédure stockée dans la base  
La clause facultative OR REPLACE permet d'écasser une procédure existante portant le même nom  
nom procédure est le nom donné par l'utilisateur à la procédure  
AUTHID indique sur quel schéma la procédure s'applique : CURRENT\_USER  
Indique que la procédure utilise les objets du schéma de l'utilisateur qui appelle la procédure  
DEFINER(défaut): Indique que la procédure utilise les objets du schéma de création de la procédure

# Procédures

IN: indique que le paramètre n'est pas modifiable par la procédure

OUT: indique que le paramètre est modifiable par la procédure

IN OUT: indique que le paramètre est transmis par le programme appelant et renseigné par la procédure

Par défaut, les paramètres sont transmis par copie, c'est à dire qu'un espace mémoire est créé pour recevoir une copie de la valeur

# Sous-Programmes

## □ Exemple de procédure

```
CREATE OR REPLACE PROCEDURE insere_etudiant
    ( ne NUMBER,           ←-----→ ( ne ETUDIANTS.numetu%TYPE ,
        nme ETUDIANTS.nometu%TYPE)  IS
    n INTEGER;
    except_cle EXCEPTION;
BEGIN
    SELECT COUNT(*) INTO n
    FROM ETUDIANTS WHERE numetu = ne;
    IF (n>0) THEN
        RAISE except_cle;
    ELSE
        INSERT INTO ETUDIANTS VALUES (ne, nme, 0);
    END IF;
EXCEPTION
    WHEN except_cle THEN
        DBMS_OUTPUT.PUT_LINE('Numero d''etudiant existant!');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Erreur detectee');
END insere_etudiant;
/
```

# Procédures cataloguées: Exemple

```
CREATE OR REPLACE PROCEDURE insert_pilote
(num in Number) IS NumVol vol.npilote%TYPE;
Excep_key Exception;
BEGIN
    Select count(*) INTO numVol FROM vol WHERE nvol = num;
    IF numVol >0 THEN  RAISE Except_key;
    ELSE
        Insert INTO vol values(num,1,2,'rak','paris','0816','1021',SYSDATE);
        DBMS_OUTPUT.PUT_LINE('Données Enregistrées');
    END IF;
EXCEPTION
    WHEN Excep_key THEN
        DBMS_OUTPUT.PUT_LINE('Num de vol existant');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('ERREUR détectée');
END insertPilote;
/
```

# Sous-programmes: Fonctions cataloguées

## - Principe

- Programme PL/SQL stocké dans la BD retournant un résultat
- Même fonctionnement que les procédures

## - Structure

```
CREATE [ OR REPLACE ] FUNCTION nomprocedure
    [ (parametre1,..., parametren) ]
[AUTHID { CURRENT_USER | DEFINER }]
    RETURN typeresultat
    [ IS ]
        [ variables_internes ]
BEGIN
    instructions
    [ EXCEPTION ...]
END [ nomprocedure ];
/
```

nomvariable [IN|OUT|IN OUT] type [{ := | DEFAULT } valeur]

# Sous-programmes: fonctions

## □ Exemples de fonction

```
CREATE OR REPLACE FUNCTION nb_etudiants() RETURN INTEGER IS
    n INTEGER;
BEGIN
    SELECT COUNT(*) INTO n
    FROM ETUDIANTS;
    RETURN n;
END nb_etudiants;
/
```

## □ Récursivité

```
CREATE OR REPLACE FUNCTION factorielle ( n INTEGER ) RETURN INTEGER IS
BEGIN
    IF n = 1 THEN
        RETURN 1;
    ELSE
        RETURN n * factorielle(n-1);
    END IF;
END factorielle;
/
```

# Sous-programmes: Exemple 2

```
CREATE OR REPLACE FUNCTION moy_salaire
```

```
    RETURN FLOAT
```

```
IS
```

```
    v_moy FLOAT;
```

```
BEGIN
```

```
    SELECT AVG(sal) INTO v_moy FROM pilote;
```

```
    RETURN v_moy;
```

```
END moy_salaire:
```

```
/
```

- Utilisation en SQL\*PLUS:

```
select plnom from pilote where (sal>moy_salaire());
```

- utilisation en PL/SQL:

```
select moy_salaire from dual;
```

# Compilation

Pour compiler ces sous-programmes à partir de l'interface SQL\*Plus, il faut rajouter le symbole / en première colonne après chaque dernier END. Si le message suivant apparaît: Avertissement : *Fonction/Procédure créée avec erreurs de compilation*, deux techniques peuvent être utilisées pour visualiser les erreurs de compilation :

- faire SHOW ERRORS sous SQL\*Plus ;
- interroger la vue USER\_ERRORS (SELECT LINE,POSITION,TEXT FROM USER\_ERRORS WHERE NAME='*nomFonction/nomProcédure*');).

Une fois que le message *Fonction/Procédure créée.* apparaît, le sous-programme est correctement compilé et stocké en base.

# Sous-programmes

## □ Appels des sous-programmes

- A partir de SQL\*Plus

```
EXECUTE nomprocedure [ (parametre1,..., parametren) ] ;
```

- A partir de PL/SQL

```
DECLARE res typeresultat;
```

```
BEGIN
```

```
...
```

```
nomprocedure [ (parametre1,..., parametren) ] ;
```

```
res := nomfonction [ (parametre1,..., parametren) ] ;
```

```
...
```

```
END;
```

```
/
```

## □ Suppression de sous-programmes

```
DROP PROCEDURE nomprocedure;
```

```
DROP FUNCTION nomfonction;
```

# Fonction: exemple

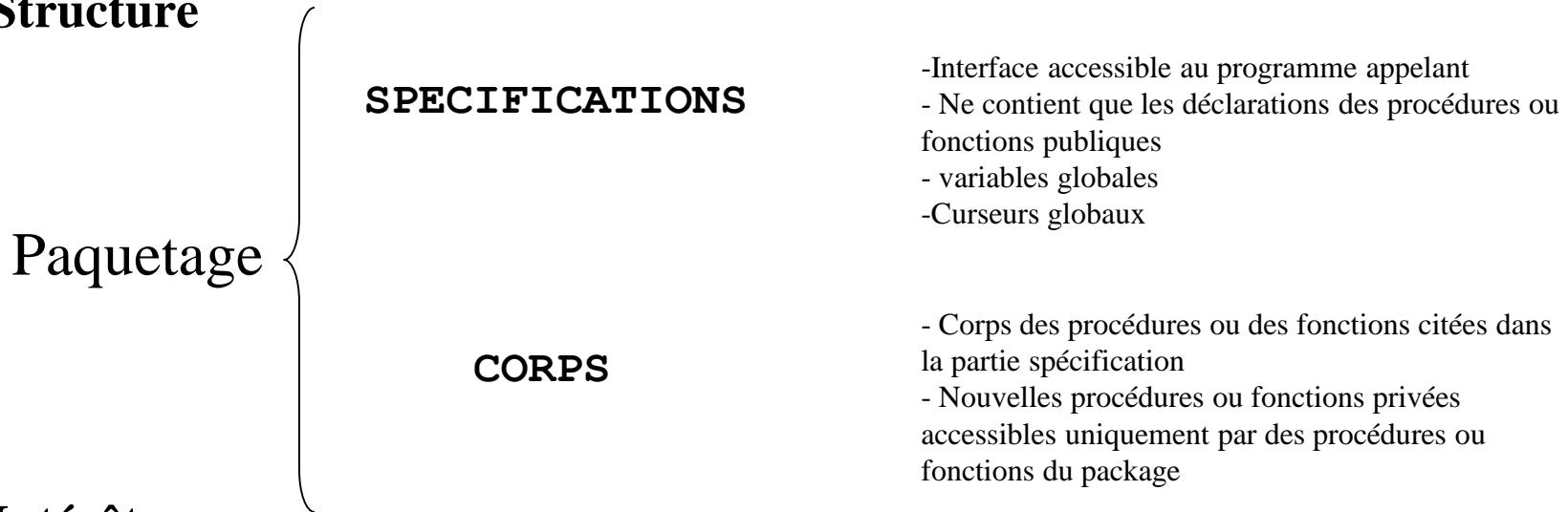
```
CREATE OR REPLACE PROCEDURE PlusExpérimenté (pcomp IN OUT VARCHAR2, pnomPil OUT VARCHAR2,
                                             pheuresVol OUT NUMBER)
IS p1 NUMBER;
BEGIN
    IF (pcomp IS NULL) THEN
        SELECT COUNT(*) INTO p1 FROM Pilote WHERE nbHVol = (SELECT MAX(nbHVol) FROM Pilote);
    ELSE
        SELECT COUNT(*) INTO p1 FROM Pilote WHERE nbHVol = (SELECT MAX(nbHVol) FROM Pilote
                                                       WHERE comp = pcomp) AND comp = pcomp;
    END IF;
    IF p1 = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Aucun pilote n''est le plus expérimenté');
    ELSIF p1 > 1 THEN
        DBMS_OUTPUT.PUT_LINE('Plusieurs pilotes sont les plus expérimentés');
    ELSE
        IF (pcomp IS NULL) THEN
            SELECT nom, nbHVol, comp INTO pnomPil, pheuresVol, pcomp FROM Pilote WHERE nbHVol = (SELECT
                MAX(nbHVol) FROM Pilote);
        ELSE
            SELECT nom, nbHVol INTO pnomPil, pheuresVol FROM Pilote WHERE nbHVol = (SELECT
                MAX(nbHVol) FROM Pilote WHERE comp = pcomp) AND comp = pcomp;
        END IF;
    END IF;
End PlusExpérimenté;
```

# Paquetages(packages)

## □ Principe

- Regroupement d'un ensemble applicatifs
- Contient : fonctions, procédures, variables, constantes, curseurs, types, exceptions

## □ Structure



## □ Intérêts

- Encapsulation données/programmes, accès uniquement aux composants publics
- Persistance des variables pour la session
- Modularité : modifications du corps sans toucher aux spécifications
- Performant : un appel charge le paquetage en mémoire et exécution dans le noyau serveur

# Paquetages

## □ Spécification

```
CREATE PACKAGE nompaquetage [IS]  
    -- types, variables, constantes, curseurs publics  
    -- spécifications de sous-programmes publics  
END [ nompaquetage ];  
/
```

## □ Corps

```
CREATE PACKAGE BODY nompaquetage [IS]  
    -- types, variables, constantes, curseurs privés  
    -- corps de sous-programmes publics & privés  
END [ nompaquetage ];  
/
```

## □ Suppressions

**DROP PACKAGE BODY** nompaquetage; -----> corps

**DROP PACKAGE** nompaquetage; -----> corps + spécification

# Paquetages: Exemple1

## █ Spécification

```
CREATE PACKAGE gestion_etudiants IS

    TYPE t_inscriptions IS RECORD (numetu ETUDIANTS.numetu%TYPE,
                                    numf   FORMATIONS.numf%TYPE,
                                    datei  DATE);

    CURSOR C_inscrits RETURN t_inscriptions;

    PROCEDURE enregistrement(ne ETUDIANTS.numetu%TYPE,
                           nm ETUDIANTS.nometu%TYPE);

    PROCEDURE exclusion(ne ETUDIANTS.numetu%TYPE);

    PROCEDURE inscription(ne ETUDIANTS.numetu%TYPE,
                           nf FORMATIONS.numf%TYPE);

END gestion_etudiants;
/
```

# Paquetages: Exemple1

## □ Exemple

### ■ Corps

```
CREATE PACKAGE BODY gestion_etudiants IS

CURSOR C_inscrits RETURN t_inscriptions IS
    SELECT * FROM INSCRIPTIONS ORDER BY datei;

PROCEDURE enregistrement(ne ETUDIANTS.numetu%TYPE,
                         nm ETUDIANTS.nometu%TYPE) IS
BEGIN
    INSERT INTO ETUDIANTS VALUES (ne, nm, 0);
END enregistrement;

PROCEDURE exclusion(ne ETUDIANTS.numetu%TYPE) IS
BEGIN
    DELETE FROM INSCRIPTIONS WHERE numetu = ne;
    DELETE FROM ETUDIANTS WHERE numetu = ne;
END exclusion;

PROCEDURE inscription(ne ETUDIANTS.numetu%TYPE,
                       nf FORMATIONS.numf%TYPE) IS
BEGIN
    INSERT INTO INSCRIPTIONS VALUES (nf, ne, SYSDATE);
END inscription;

END gestion_etudiants;
/
```

# Paquetages: Exemple2

```
CREATE OR REPLACE PACKAGE gestionPilote AS
```

```
    -- Objets
```

```
    nbreVol integer;
```

```
    CURSOR accesPilotes(salPilote pilote.sal%TYPE)
```

```
    RETURN pilote%ROWTYPE;
```

```
    -- FONCTION
```

```
    FUNCTION nbreVolAvion(TypeAvion IN Avion.typeAv%TYPE)
```

```
    RETURN integer;
```

```
    -- Procédure
```

```
    PROCEDURE AccorderPrime(prime float);
```

```
END gestionPilote;
```

/

# Paquetages: Exemple2

```
CREATE OR REPLACE PACKAGE BODY gestionPilote AS
```

```
    -- curseur
```

```
    CURSOR accesPilote(salPilote pilote.sal%TYPE)
```

```
    RETURN pilote%ROWTYPE
```

```
    IS SELECT * FROM pilote WHERE sal = salPilote;
```

```
    -- fonction
```

```
    FUNCTION nbreVolAvion(TypeAvion IN Avion.typeAv%TYPE)
```

```
    RETURN integer IS
```

```
        Begin
```

```
            select count(*) into nbreVol from avion a, vol v
```

```
            where a.navion = v.navion and typeav = typeAvion;
```

```
            return nbreVol;
```

```
        End nbreVolAvion;
```

```
    -- procédure
```

```
    PROCEDURE accorderPrime(prime float) IS
```

```
        Begin
```

```
            UPDATE pilote SET sal=sal+prime WHERE npilote in select npilote from vol group by npilote having  
            count(*) >=4;
```

```
End Accorder_prime;
```

```
END gestionPilote;
```

```
/
```

# Référence au contenu d'un package

- seuls les objets et sous programmes publics peuvent être référencés depuis l'extérieur
- syntaxe:

```
nomPackage.nomObjet  
nomPackage.nomSousProgramme(...)
```

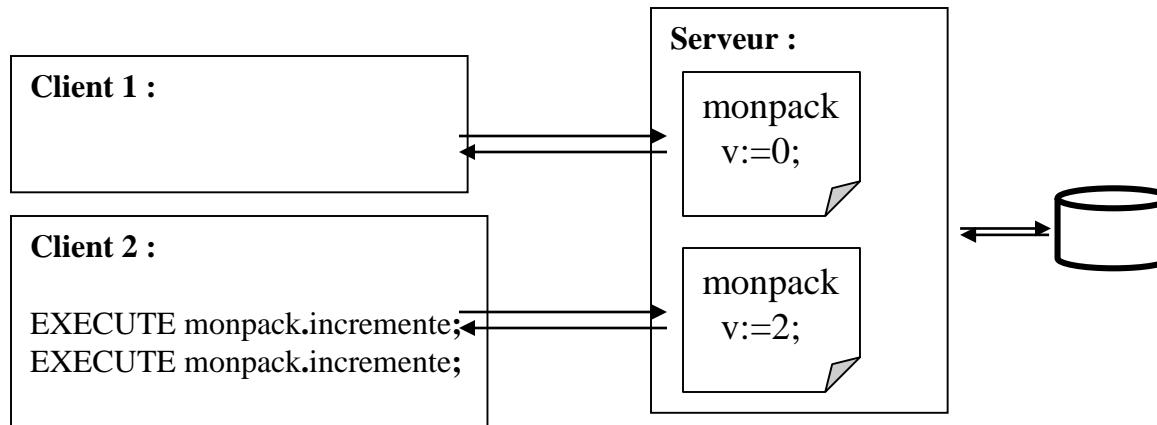
```
set serveroutput on;  
Accept v_sal prompt'Entrer un salaire'  
BEGIN  
    for enreg in gestionPilote.accesPilotes(&v_sal) LOOP  
        DBMS-OUTPUT.PUT_LINE(enreg.plnom);  
    end loop;  
End;  
/  
Select gestionPilotes.nbreVolAvion('A300') from dual;  
Execute gestionPilotes.accorder_Prime(200);
```

# Paquetages

## □ Persistance des variables

```
CREATE OR REPLACE PACKAGE monpack IS
    v INTEGER:=0;
    PROCEDURE incremente;
END monpack ;
/
```

```
CREATE OR REPLACE PACKAGE BODY monpack IS
    PROCEDURE incremente IS
        BEGIN
            v:=v+1;
        END incremente;
END monpack ;
/
```



# Les Déclencheurs

- Un déclencheur est un bloc PL/SQL associé à une vue ou une table, qui s'exécutera lorsqu'une instruction du langage de manipulation de données (DML) sera exécutée
- Un déclencheur définit une action qui doit s'exécuter quand survient un événement dans la base de données.

Il peut servir a :

- ajouter des contraintes sur les valeurs des attributs d'une table,
- enregistrer des changements (suivi)
- ajouter des règles de gestion,...
- Les déclencheurs (*triggers*) existent depuis la version 6 d'Oracle. Ils sont compilables depuis la version 7.3 (auparavant, ils étaient évalués lors de l'exécution).
- À la différence des sous-programmes, l'exécution d'un déclencheur n'est pas explicitement opérée par une commande ou dans un programme, c'est l'événement de mise à jour de la table (ou de la vue) qui exécute automatiquement le code programmé dans le déclencheur. On dit que le déclencheur « se déclenche » (l'anglais le traduit mieux : *fired trigger*).

# Déclencheurs : définition

## □ Quand et comment les utiliser

Un déclencheur peut être déclenché :

- a la création, suppression ou modification d'un objet (CREATE, DROP, ALTER),
- a la connexion, déconnexion d'un utilisateur,
- démarrage ou arrêt d'une instance...

## □ Durée d'activité

L'action associée à un déclencheur est un bloc PL/SQL enregistré dans la base.

Un déclencheur est opérationnel jusqu'à la suppression de la table à laquelle il est lié.

## □ Le traitement mentionné dans un trigger peut s'effectuer :

- pour chaque ligne concernée par l'événement,(trigger de niveau ligne)
- une seule fois pour l'ensemble des lignes concernées par l'événement(trigger de niveau table).

## □ Ligne

FOR EACH ROW : trigger de niveau ligne, sinon de niveau table.

WHEN condition : pour chaque ligne, trigger déclenché si vraie.

# À quoi sert un déclencheur ?

- Un déclencheur permet de :
- Programmer toutes les règles de gestion qui n'ont pas pu être mises en place par des contraintes au niveau des tables. Par exemple, la condition : *une compagnie ne fait voler un pilote que s'il a totalisé plus de 60 heures de vol dans les 2 derniers mois sur le type d'appareil du vol en question, ne pourra pas être programmée par une contrainte et nécessitera l'utilisation d'un déclencheur.*
- Déporter des contraintes au niveau du serveur et alléger ainsi la programmation client.
- Renforcer des aspects de sécurité et d'audit.
- Programmer l'intégrité référentielle et la réPLICATION dans des architectures distribuées avec l'utilisation de liens de bases de données (*database links*).

# Déclencheurs: Généralités

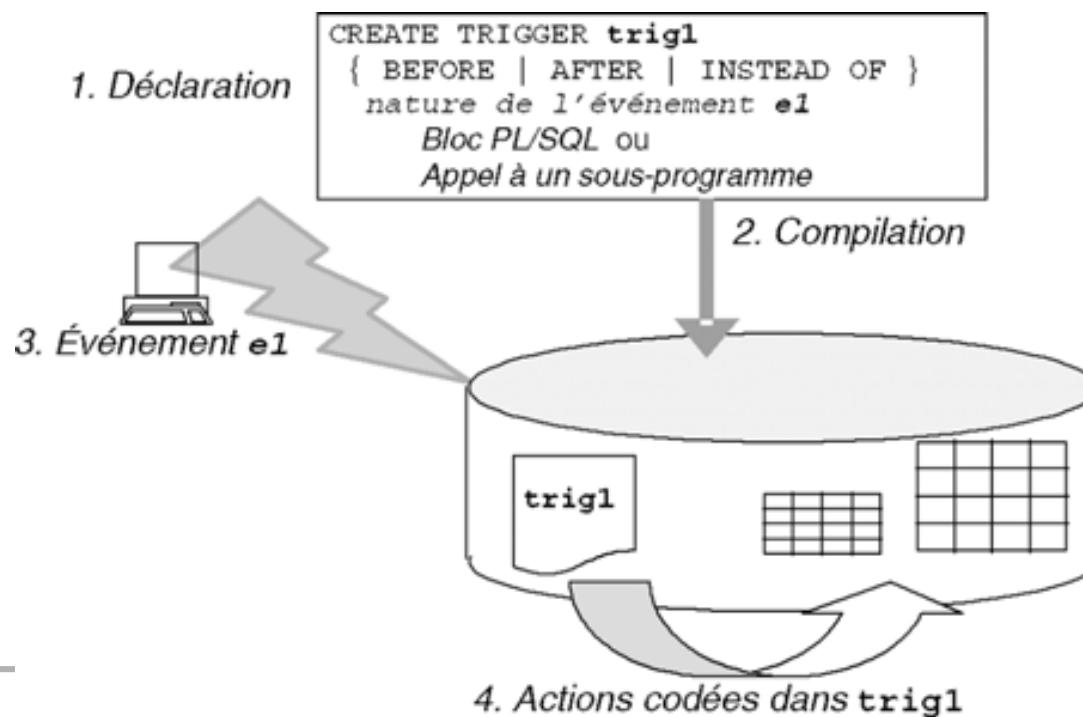
□ Les événements déclencheurs peuvent être :

- une instruction INSERT, UPDATE, ou DELETE sur une table (ou une vue). On parle de déclencheurs LMD ;
- une instruction CREATE, ALTER, ou DROP sur un objet (table, index, séquence, etc.). On parle de déclencheurs LDD ;
- le démarrage ou l'arrêt de la base (*startup ou shutdown*), une erreur spécifique (*NO\_DATA\_FOUND*, *DUP\_VAL\_ON\_INDEX*, etc.), une connexion ou une déconnexion d'un utilisateur.

On parle de déclencheurs d'instances.

# Mécanisme général

- La figure suivante illustre les étapes à suivre pour mettre en oeuvre un déclencheur. Il faut d'abord le coder (comme un sous-programme), puis le compiler (il sera stocké ainsi en base).
- Par la suite, au cours du temps, et si le déclencheur est actif (nous verrons qu'il est possible de désactiver un déclencheur même s'il est compilé), chaque événement (qui caractérise le déclencheur) aura pour conséquence son exécution.



# Syntaxe

- Pour pouvoir créer un déclencheur dans votre schéma, vous devez disposer du privilège CREATE TRIGGER
- Un déclencheur est composé de trois parties : la description de l'événement traqué, une éventuelle restriction (condition) et la description de l'action à réaliser lorsque l'événement se produit. La syntaxe de création d'un déclencheur est la suivante :

# Syntaxe

```
CREATE [OR REPLACE] TRIGGER [schéma.] nomDéclencheur
{ BEFORE | AFTER | INSTEAD OF }
{ { DELETE | INSERT | UPDATE [OF col1 [,col2]...] }
[OR { DELETE | INSERT | UPDATE [OF col1 [,col2]...] }]...
ON { [schéma.] nomTable / nomVue }
[REFERENCING
{ OLD [AS] nomVieux / NEW [AS] nomNew / PARENT [AS] nomParent }
[ OLD [AS] nomVieux / NEW [AS] nomNew / PARENT [AS] nomParent]... ]
[FOR EACH ROW] }
|
{ événementBase [OR événementBase]... /
actionStructureBase [OR actionStructureBase]... }
ON { [schéma.] SCHEMA / DATABASE } }
[WHEN ( condition ) ]
{ Bloc PL/SQL (variables BEGIN instructions END ; )
| CALL nomSousProgramme(paramètres) }
```

# Syntaxe

Les options de cette commande sont les suivantes :

- **BEFORE | AFTER | INSTEAD OF** précise la chronologie entre l'action à réaliser par le déclencheur LMD et la réalisation de l'événement (exemple BEFORE INSERT programmera l'exécution du déclencheur avant de réaliser l'insertion).
- **DELETE | INSERT | UPDATE** précise la nature de l'événement pour les déclencheurs LMD.
- **ON {[schéma.] nomTable | nomVue}** spécifie la table, ou la vue, associée au déclencheur LMD.
- **REFERENCING** permet de renommer des variables.
- **FOR EACH ROW** différencie les déclencheurs LMD au niveau ligne ou au niveau état.
- *événementBase* identifie la nature d'un déclencheur d'instance (*STARTUP* ou *SHUTDOWN* pour exécuter le déclencheur au démarrage ou à l'arrêt de la base), d'un déclencheur d'erreurs (*SERVERERROR* ou *SUSPEND* pour exécuter le déclencheur dans le cas d'une erreur particulière ou quand une transaction est suspendue) ou d'un déclencheur de connexion (*LOGON* ou *LOGOFF* pour exécuter le déclencheur lors de la connexion ou de la déconnexion à la base).
- *actionStructureBase* spécifie la nature d'un déclencheur LDD (*CREATE*, *ALTER*, *DROP*, etc. pour exécuter par exemple le déclencheur lors de la création, la modification ou la suppression d'un objet de la base).
- **ON {[schéma.]SCHEMA | DATABASE}}** précise le champ d'application du déclencheur (de type LDD, erreur ou connexion). Utilisez **DATABASE** pour les déclencheurs qui s'exécutent pour quiconque commence l'événement, ou **SCHEMA** pour les déclencheurs qui ne doivent s'exécuter que dans le schéma courant.
- **WHEN** conditionne l'exécution du déclencheur.

# Déclencheurs LMD

Nature de l'événement	État ( <i>statement trigger</i> ) sans FOR EACH ROW	Ligne ( <i>row trigger</i> ) avec FOR EACH ROW
BEFORE	Exécution une fois avant la mise à jour	Exécution avant chaque ligne mise à jour
AFTER	Exécution une fois après la mise à jour	Exécution après chaque ligne mise à jour

## *Déclencheurs de lignes (row triggers)*

Un déclencheur de lignes est déclaré avec la directive FOR EACH ROW. Ce n'est que dans ce type de déclencheur qu'on a accès aux anciennes valeurs et aux nouvelles valeurs des colonnes de la ligne affectée par la mise à jour prévue par l'événement.

# Quand et quoi...

## □ Quand

- BEFORE et AFTER
- pour trigger de niveau table(statement) : déclenché avant ou après l'événement
- pour trigger de niveau ligne : exécute avant ou après la modification de CHAQUE ligne concernée
- INSTEAD OF : spécifique aux vues.

## □ Quoi

corpsTrigger : bloc PL/SQL effectue quand le trigger est déclenché En plus :

```
IF INSERTING THEN ... END IF;  
IF DELETING THEN ... END IF;  
IF UPDATING THEN ... END IF;
```

# Valeurs des attributs

- Quelles valeurs sont testées?

Dans la clause WHERE ou dans le corps, on peut se référer à la valeur d'un attribut avant ou après que soit effectuée la modification déclenchant le trigger :

:OLD.nomAttribut : la valeur avant la transaction UPDATE ou DELETE

:NEW.nomAttribut : la valeur après la transaction UPDATE ou INSERT

# Tables exemple

- Une table contenant des etudiants (numero, nom, prenom...) avec leur moyenne.
- Des tables reportEtudiant (pour le suivi), Alerte.

Table etudiant			
NumEtudiant	Nom	Prenom	Moyenne
123	DUPONT	JULES	4
234	DUPOND	ALFRED	5
567	DURAND	JULIE	14
598	DURANT	ALFREDINE	16

# Exemple de déclencheur de niveau table

```
CREATE TRIGGER triggerEtudiant
    BEFORE INSERT or UPDATE ON etudiant
BEGIN
    DBMS_OUTPUT.ENABLE(10000);
    IF INSERTING THEN
        IF USER != 'SCOTT'
            THEN RAISE_APPLICATION_ERROR
                (-20001,'Utilisateur non autorisé.');
        END IF;
    END IF;
    IF UPDATING THEN
        DBMS_OUTPUT.PUT_LINE('Mise à jour de l "étudiant.');
    END IF;
END;
```

# Exemple de déclencheur de niveau ligne

```
CREATE TRIGGER auditEtudiant
  AFTER UPDATE OR DELETE ON etudiant
  FOR EACH ROW
BEGIN
  INSERT INTO reportEtudiant
    VALUES (SYSDATE, :OLD.NumEtud, :OLD.nom,
            :OLD.prenom, :OLD:moyenneEtud);
END;
```

# Exemple de déclencheur de niveau ligne avec clause WHERE

```
CREATE TRIGGER moyenneMax
AFTER UPDATE ON etudiant
FOR EACH ROW
WHEN NEW.moyenneEtud > 2*OLD.moyenneEtud
BEGIN
INSERT INTO alerte(datealerte, numEtudiant, message)
VALUES ( SYSDATE, :OLD.nom,
'MOYENNE a plus que double');
END;
```

# Exemple: for each row

Create or replace trigger print\_salary\_changes

Before update on Emp\_tab

For each row

When (new.empno > 0)

Declare

    sal\_diff number;

Begin

    sal\_diff := :new.sal - :old.sal;

    Dbms\_output.put('ancienne:'||:old.sal || 'nouvelle' || :new.sal|| 'difference'||sal\_diff);

End;

/

Ce trigger est lorsque la table emp\_tab est mise à jour. Pour chaque modification(lignes mises à jour), le trigger va calculer puis afficher respectivement l'ancien salaire, le nouveau salaire et la différence entre les deux salaires la condition précisée (when new.empno>0) restreint le déclenchement du trigger(exécuté uniquement si empno > 0)

# Table mutante

- Il faut savoir qu'il n'est pas permis de consulter ou modifier une table mutante. Cette restriction préservera le trigger ligne de lire des données inconsistantes. Donc le trigger ne devrait jamais accéder à la table sur laquelle il porte, autrement que par OLD ou NEW.

- Exemple:

```
Create or replace trigger emp_count
```

```
After delete on Emp_tab
```

```
For each row
```

```
When (new.empno > 0)
```

```
Declare
```

```
    n integer;
```

```
Begin
```

```
    select count(*) into n from Emp_tab;
```

```
    Dbms_output.put('il y'a ' ||n|' employés');
```

```
End;
```

```
/
```

Dans cet exemple la table mutante (celle qui est modifiée par DELETE) est Emp\_tab. Or une clause SELECT porte justement sur cette table. Le risque est d'obtenir un résultat complètement incohérent. Cette consultation est donc interdite

# :OLD & :NEW

- Dans trigger ligne il est possible de modifier les éléments NEW.
- OLD & NEW sont disponible dans BEFORE & AFTER ROW TRIGGERS
- La valeur de NEW peut être assigné dans un TRIGGER ROW BEFOR mais pas dans un trigger AFTER
- Si vous spécifié un TRIGGER ROW que sont déclenchement dépend d'une colonne précise dans une table, il est tout à fait logique que les éléments OLD et NEW ne porte que sur cette colonne
- Parmi les limites des triggers
  - DDL statement are not allowed in the body of a trigger.
  - No transaction control statement are allowed in the trigger: ROLLBACK, COMMIT, SAVEPOINT cannot be used

# Ordre

Ordre d'activation :

Si plusieurs triggers sur une même table, l'ordre d'activation est :

1 BEFORE niveau table

2 BEFORE niveau ligne, aussi souvent que de lignes concernées

3 AFTER niveau ligne, aussi souvent que de lignes concernées

4 AFTER niveau table

Code PL/SQL	Commentaires
<pre><b>CREATE TRIGGER TrigInsQualif</b> <b>BEFORE INSERT ON Qualifications</b> <b>FOR EACH ROW</b></pre>	Déclaration de l'événement déclencheur.
<pre><b>DECLARE</b> v_compteur Pilote.nbHVol%TYPE; v_nom Pilote.nom%TYPE;</pre>	Déclaration des variables locales.
<pre>BEGIN SELECT nbQualif, nom INTO v_compteur, v_nom   FROM Pilote WHERE brevet =:<b>NEW.brevet</b> ; IF v_compteur &lt; 3 THEN UPDATE Pilote SET nbQualif = nbQualif + 1 WHERE brevet =:<b>NEW.brevet</b> ; ELSE    RAISE_APPLICATION_ERROR (-20100, 'Le pilote '    v_nom    ' a déjà 3 qualifications!'); END IF; END; /</pre>	<p>Corps du déclencheur. Extraction et mise à jour du pilote concerné par la qualification.</p> <p>Renvoi d'une erreur utilisateur.</p>

Code PL/SQL	Commentaires
CREATE TRIGGER TrigInsQualif BEFORE INSERT ON Qualifications FOR EACH ROW	Déclaration de l'événement Déclencheur
DECLARE v_compteur Pilote.nbHVol%TYPE; v_nom Pilote.nom%TYPE;	Déclaration des variables locales.
<pre> BEGIN SELECT nbQualif, nom INTO v_compteur, v_nom FROM Pilote WHERE brevet = :NEW.brevet; IF v_compteur &lt; 3 THEN UPDATE Pilote SET nbQualif = nbQualif + 1 WHERE brevet = :NEW.brevet; ELSE RAISE_APPLICATION_ERROR(-20100, 'Le pilote '    :NEW.brevet    ' a déjà 3 qualifications!'); END IF; EXCEPTION WHEN NO_DATA_FOUND THEN RAISE_APPLICATION_ERROR(-20101, 'Pas de pilote de code brevet '    :NEW.brevet); WHEN OTHERS THEN RAISE; END; / </pre>	<p>Corps du déclencheur. Extraction et mise à jour du pilote concerné par la qualification.</p> <p>Renvoi d'une erreur utilisateur.</p> <p>Si erreur au SELECT.</p> <p>Retour de l'erreur courante.</p>

# Quand utiliser la directive :OLD ?

- Chaque enregistrement qui tente d'être supprimé d'une table qui inclut un déclencheur de type DELETE FOR EACH ROW, est désigné par :OLD au niveau du code du déclencheur. L'accès aux colonnes de ce pseudo-enregistrement dans le corps du déclencheur se fait par la notation pointée.

# SQL dynamique

## □ Principe

- Construction de requêtes SQL dynamique dans un bloc PL/SQL
- Exemple : « mise à jour d'une table dont le nom est passé en paramètre »
- Permet d'utiliser les ordres du LDD (CREATE, DROP, ALTER) qui altèrent le schéma de la base

## □ Structure

```
EXECUTE IMMEDIATE requête [ INTO var1,...,varn ]  
[ USING [IN|OUT|IN OUT] parametre1  
[, [IN|OUT|IN OUT] parametre2  
...  
[, [IN|OUT|IN OUT] parametren ]...] ]
```

# SQL dynamique

## □ Exemple d'une requête paramétrée

```
DECLARE
    req VARCHAR2(50);
BEGIN
    req:= 'INSERT INTO ETUDIANTS VALUES (:p1,:p2,o)';
    EXECUTE IMMEDIATE req USING 3, 'Polo';
END;
/
```

## □ Exemple de stockage du résultat

```
DECLARE
    req VARCHAR2(50);
    etud ETUDIANTS%ROWTYPE;
BEGIN
    req:= 'SELECT * FROM ETUDIANTS WHERE numetu = :pid';
    EXECUTE IMMEDIATE req INTO etud USING 3;
END;
/
```

# SQL Dynamique exemples

```
CREATE or REPLACE FUNCTION sql_dyn(v_table IN VARCHAR2,  
clauseWhere IN Varchar2) RETURN integer IS V_requete
```

# SQL dynamique

## □ Exemple d'une requête sans variable

```
BEGIN  
EXECUTE IMMEDIATE 'DELETE FROM ETUDIANTS WHERE numetu = :pid' USING 3;  
END;  
/
```

## □ Exemple d'une requête du LDD

```
BEGIN  
EXECUTE IMMEDIATE 'DROP TABLE ETUDIANTS';  
END;  
/
```