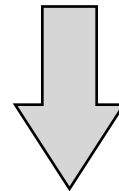


Le langage PL/SQL

Objectifs PL/SQL

SQL est un langage BD limité

- ☹ Pas de fonctionnalités procédurales
 - ☹ Individualité des requêtes
 - ☹ Pas de gestion de contexte
- ☹ Ne répond pas au besoin du transactionnel



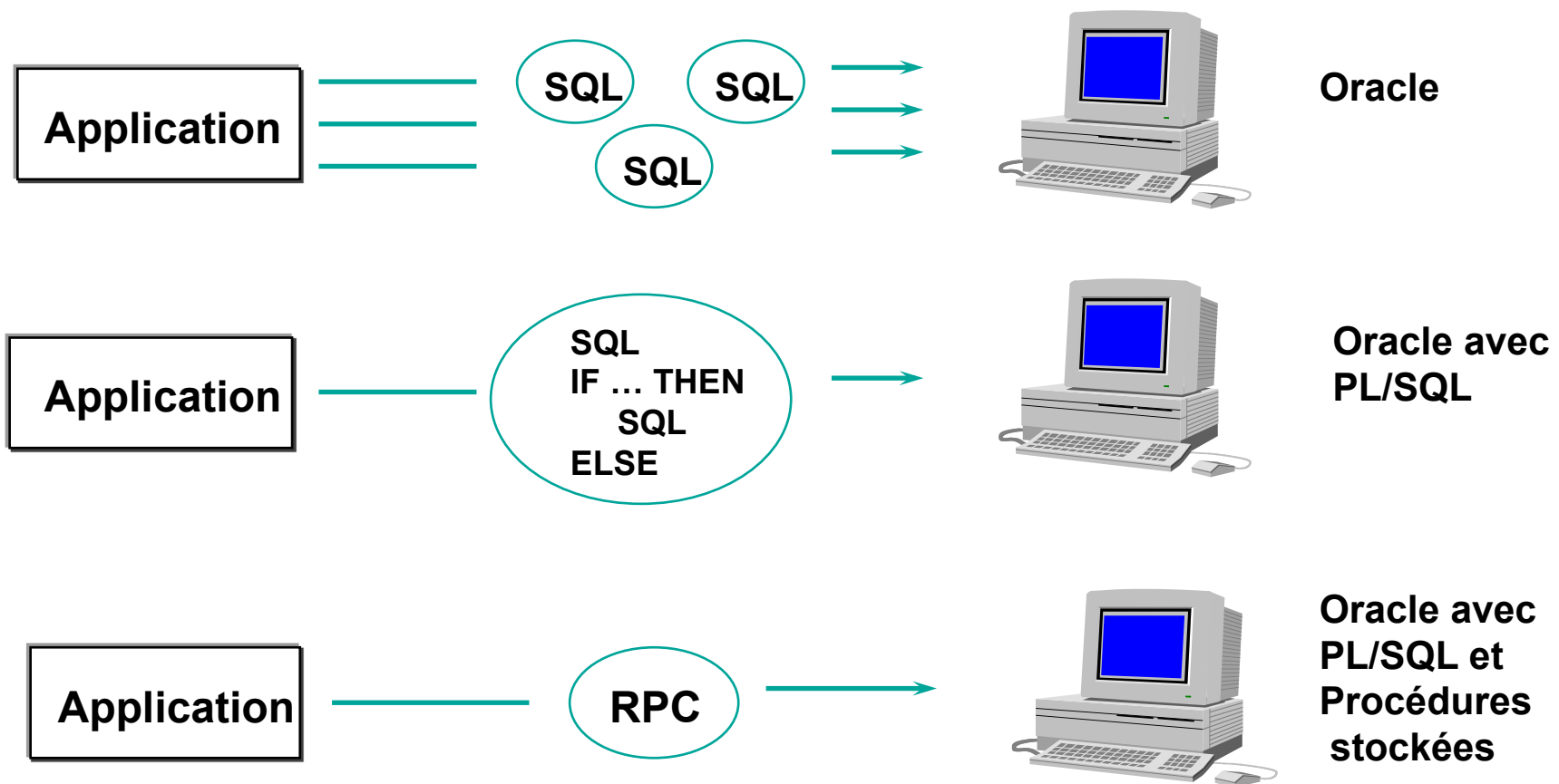
**Besoin d'étendre le langage
PL/SQL : Procedural Language SQL**

Avantages de PL/SQL (1)

- ◆ PL/SQL est un langage complet
 - Ajout de fonctionnalités procédurales à SQL
- ◆ Meilleure productivité des applications
- ◆ Intégration complète avec ORACLE utilisable dans :
 - SQL*MENU, SQL*PLUS, SQL*DBA,

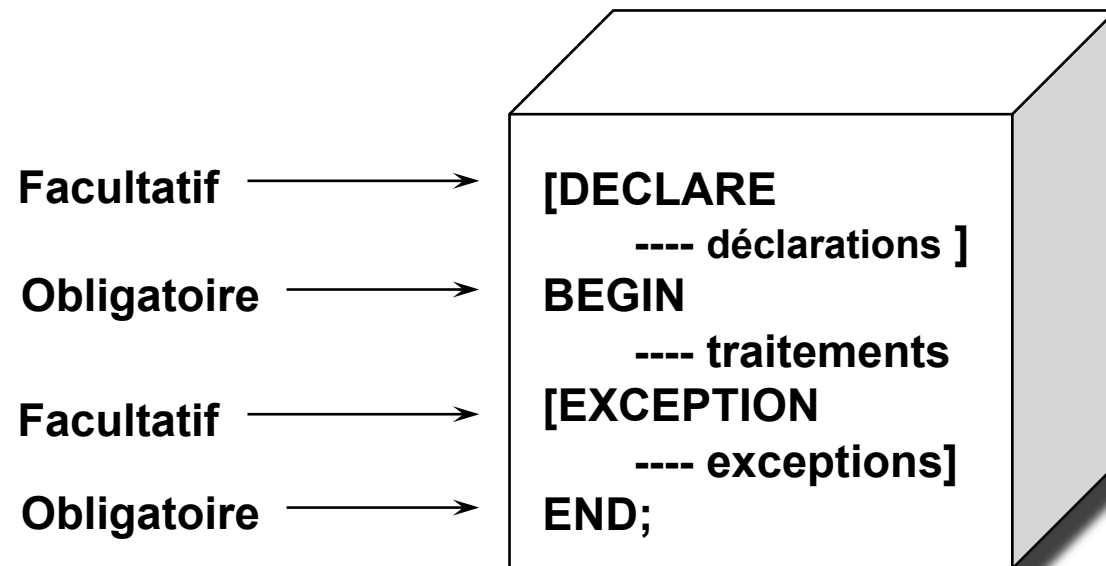
Avantages de PL/SQL (2)

Bonnes performances dans un environnement réseau
Réduction du trafic



Structure d'un Programme PL/SQL

- ♦ PL/SQL est structuré en blocs
- ♦ Le bloc est l'unité de base de tout programme PL/SQL
- ♦ Un bloc peut contenir d'autres blocs



Un Exemple de Programme PL/SQL

DECLARE

```
nombre_vol    NUMBER(11,2);  
Nb_max        INTEGER NOT NULL := 300;
```

BEGIN

```
SELECT COUNT (numvol) INTO nombre_vol  
FROM vol  
WHERE ville_depart='Lyon';  
IF nombre_vol < Nb_max THEN  
    INSERT INTO vol  
    VALUES ('AF320', TO_DATE('12:30', 'HH24:MI'),  
            TO_DATE('13:15','HH24:MI'), 'Paris', 'Lyon');
```

ELSE

```
    NULL;
```

```
END IF;
```

```
COMMIT;
```

```
END;
```

Limitations PL/SQL

- ◆ PL/SQL ne supporte pas :
 - Les commandes de LDD :
 - CREATE, MODIFY, GRANT ...
 - Les commandes de contrôle de session
 - SET ROLE, ALTER SESSION
 - Les commandes de contrôle système
 - ALTER SYSTEM
- ◆ Toutes les commandes de LMD sont honorées
 - Sauf EXPLAIN PLAN

Le Langage PL/SQL

- ◆ Notions de base
- ◆ Les curseurs
- ◆ Gestion des erreurs
- ◆ Procédures, fonctions et packages
- ◆ Les Triggers

Notions de base

Le bloc PL/SQL

DECLARE

< variables, constantes, exceptions, curseurs>
-- commentaires

BEGIN [nom du bloc]

<instructions sql et plsql>
/* commentaires.....
.....*/

EXCEPTION

<gestion des erreurs>

END ;

Exercice 1

- ◆ Pour le Debuggage, pas nécessairement de bloc declare :
- ◆ Soit le bloc PL/SQL suivant :

```
BEGIN  
    DBMS_OUTPUT.put_line('Bienvenue au cours Oracle9i-PL/SQL!');  
END;  
/
```

- ◆ Exécutez ce bloc en prenant soin d'activer au préalable la commande sous SQL*Plus :
 - **SET SERVEROUT ON**

Les variables

- ◆ Variables locales
 - Définies dans la section DECLARE
- ◆ Variables extérieures :
 - SQL*FORMS (préfixées par &)
 - SQL*PLUS (préfixées par :)
- ◆ Types de variables:
 - Types Oracle (CHAR, NUMBER, DATE...)
 - nom_var CHAR
 - Type Booléen
 - nom_var BOOLEAN
 - Types référençant le dictionnaire de données
 - nom_var table.colonne%TYPE

Nommage des Variables et Constantes

- ◆ Le nom d'une variable est une chaîne de caractères qui peut contenir :
 - des lettres majuscules A..Z
 - des lettres minuscules a..z
 - des chiffres : 0..9
 - les symboles \$ # _
- ◆ PL/SQL, ni SQL ne différencie pas les majuscules des minuscules
- ◆ Exemple : Heure_départ = heure_départ

Déclarations Valides

DECLARE

Date_naissance	DATE;
Nombre_vols	SMALLINT := 0;
Nom_pilote	CHAR(10) NOT NULL := 'St-Exupéry';
pi	CONSTANT REAL := 3.14159;
rayon	REAL := 1;
aire	REAL := pi * rayon **2;

BEGIN

-- corps du programme

END;

Déclarations Non Valides

DECLARE

/ Les variables doivent avoir des identifiants distincts */*

Date_naissance DATE;

Date_naissance NUMBER;

/ La clause NOT NULL doit être suivie d'une affectation */*

Nom_pilote CHAR(10) NOT NULL;

/ Une constante doit être initialisée à sa déclaration */*

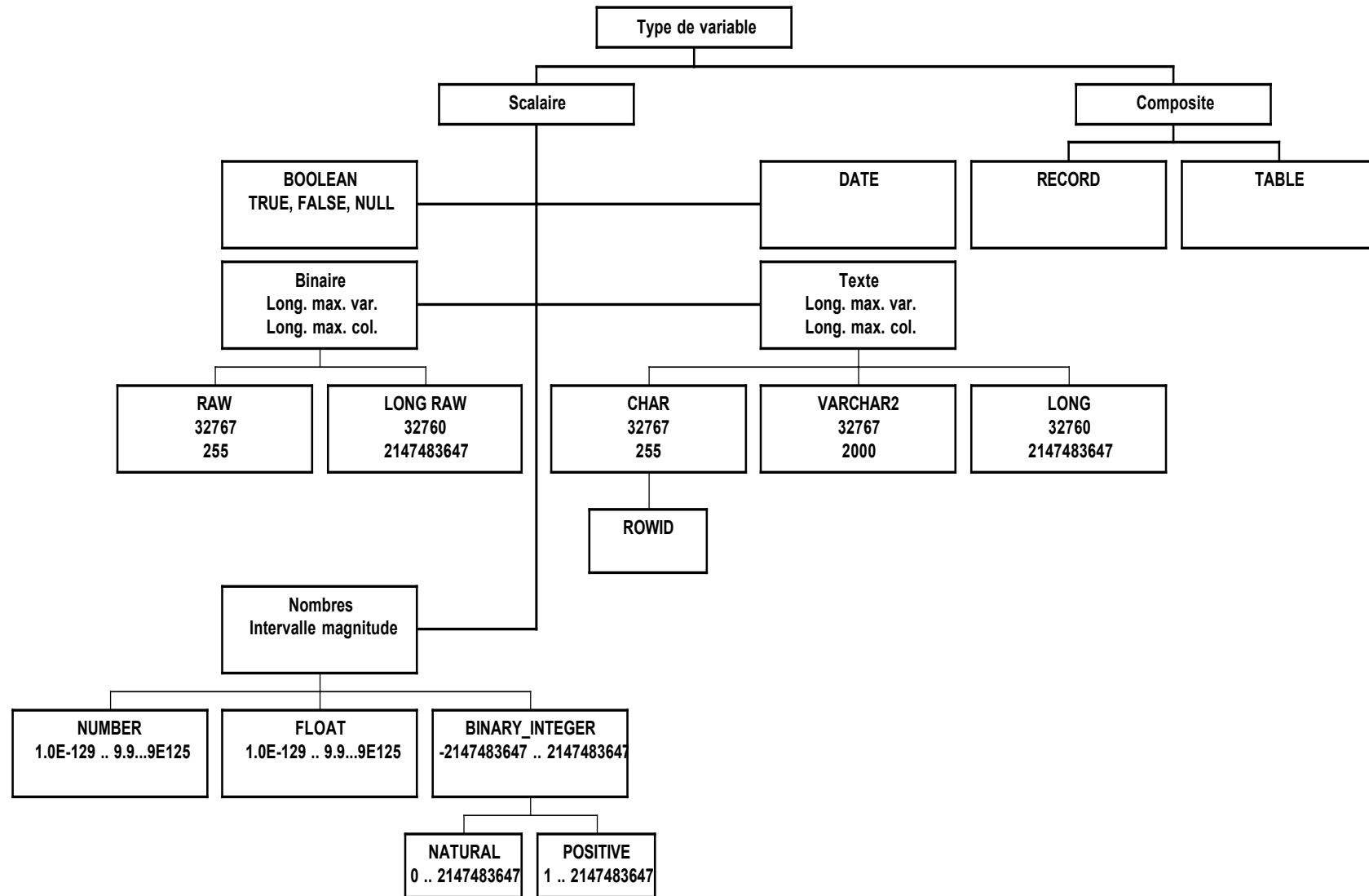
pi CONSTANT REAL;

BEGIN

-- corps du programme

END;

Types de Données PL/SQL



Opérateurs Logiques

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Affectation de valeurs

- ◆ Dans toutes les sections du bloc par ':= '

DECLARE

var1 NUMBER := 0;

BEGIN

var1 := var1+1;

END;

- ◆ Dans la section BEGIN/END par 'INTO'

DECLARE

var1, var2 NUMBER;

BEGIN

SELECT col1, col2 **INTO** var1, var2

FROM table **WHERE** condition;

END;

Compatibilité des Conversions

De A	BINARY-INTEGER	CHAR	DATE	LONG	NUMBER	RAW	ROWID	VARCHAR2
BINARY-INTEGER		YES		YES	YES			YES
CHAR	YES		YES	YES	YES	YES	YES	YES
DATE		YES		YES				YES
LONG		YES				YES		YES
NUMBER	YES	YES		YES				YES
RAW		YES		YES				YES
ROWID		YES						YES
VARCHAR2	YES	YES	YES	YES	YES	YES	YES	

Conversions Explicites

- ◆ PL/SQL permet la conversion explicite des variables d'un type donné en un autre en utilisant des fonctions pré-définies
- ◆ Les fonctions de conversion pré-définies :

De à	CHAR	DATE	NUMBER	RAW	ROWID
CHAR		TO_DATE	TO_NUMBER	HEXTORAW	CHARTOROWID
DATE	TO_CHAR				
NUMBER	TO_CHAR	TO_DATE			
RAW	RAWTOHEX				
ROWID	ROWIDTOCHAR				

Fonctions de Conversion

- ◆ TO_CHAR (*dte* DATE [, *fmt* Varchar2]) Return Varchar2
- ◆ TO_DATE (*str* Varchar2 [, *fmt* Varchar2]) Return Date
- ◆ TO_DATE (*num* Number, *fmt* Varchar2) Return Date

Format	Description	Format	Description
CC,SCC	Siècle (S met un - pour BC)	DDD	Jour de l'année (1-366)
YYYY,SYYYY	Année (S met un - pour BC)	DD	Jour du mois (1-31)
YYY,YY,Y	Derniers trois, deux, un chiffre(s) de l'année	D	Jour de la semaine (1-7)
BC, AD	Indicateurs	DAY	Nom du jour
Q	Trimestre de l'année (1-4)	DY	Nom abrégé du jour
MM	Mois (01-12)	AM, PM	Indicateur méridien
MONTH	Nom du mois	HH,HH12	Heure du jour (1-12)
MON	Nom abrégé du mois	HH24	Heure du jour (0-23)
WW	Semaine de l'année (1-53)	MI	Minute (0-59)
W	Semaine du mois (1-5)	SS	Second (0-59)
		SSSSS	Secondes après minuit (0-86399)

Conversions Implicites

- ◆ PL/SQL peut convertir implicitement le type d'une variable
 - A condition qu'il y ait compatibilité entre les types

- ◆ Exemple :

DECLARE

```
debut      CHAR(5);  
fin        CHAR(5);  
duree      NUMBER(5);
```

BEGIN

```
SELECT TO_CHAR(SYSDATE, 'SSSSS') INTO debut FROM sys.dual;
```

```
SELECT TO_CHAR(SYSDATE, 'SSSSS') INTO fin FROM sys.dual;
```

```
duree:= fin - debut;
```

```
INSERT INTO resultat VALUES (... , duree, ...);
```

END;


Fonctions Pré-définies

- ◆ PL/SQL offre différentes fonctions pré-définies :
 - Fonctions pour chaînes de caractères
 - Fonctions pour les dates
 - Fonctions numériques
 - Fonctions de conversions de types

- ◆ Toutes ces fonctions sont contenues dans le package STANDARD d'Oracle

Les Attributs de Type

◆ L'attribut %TYPE

- Affecte à une variable le type d'un champ d'une table ou d'une autre variable
- ex : **nom_pilote pilote.nom%TYPE;**
 - équivalent au type de la colonne **nom** de la table **pilote** `Select nom INTO nom_pilote FROM pilote;`

◆ L'attribut %ROWTYPE

- Affecte à une variable le type des tuples d'une table
- ex : **pilote_rec pilote%ROWTYPE;**
 `Select * INTO pilote_rec FROM pilote;`

Type RECORD (1)

- ◆ Type composé de champs nommés

- ◆ Syntaxe

```
TYPE nom_type IS RECORD (  
    nom_champ { type_champ | variable%TYPE |  
    table.colonne%TYPE | table%ROWTYPE } [ NOT NULL ]  
    [ , nom_champ { type_champ | variable%TYPE |  
    table.colonne%TYPE | table%ROWTYPE } [ NOT NULL ] ] ...);
```

- ◆ Déclaration en 2 temps

- Déclaration du type RECORD
- Déclaration d'une variable de ce type

Type RECORD (2)

DECLARE

TYPE Rec_pilote **IS RECORD**

```
( matricule      pilote.matricule%TYPE,
  nom             pilote.nom%TYPE,
  ville          pilote.ville%TYPE,
  age            pilote.age%TYPE,
  salaire        pilote.salaire%TYPE);

p_rec_1          Rec_pilote;
p_rec_2          Rec_pilote;
p_rec_3          pilote%ROWTYPE;
```

BEGIN

```
SELECT * INTO p_rec_1 FROM pilote;
p_rec_2 := p_rec_1;  /* légal */
p_rec_3 := p_rec_1;  /* illégal */
p_rec_1 := (2, 'Alain Dupond', 'Bordeaux', 35, 144000);  /* illégal */
IF p_rec_1 = p_rec_2 ...  /* illégal */
```

END;

Type TABLE (1)

- ◆ Type composé permettant de créer des tableaux à tailles non contraintes
 - La gestion du nombre de lignes dedans est de la responsabilité du développeur
- ◆ Syntaxe
TYPE *nom_type* **IS TABLE OF**
{ *type_colonne* | *variable*%TYPE | *table.colonne*%TYPE } [NOT NULL]
INDEX BY BINARY_INTEGER
- ◆ Déclaration en 2 temps :
 - Déclaration du type TABLE
 - Déclaration d'une variable de ce type
- ◆ La clé doit être nécessairement de type BINARY_INTEGER

Type TABLE (2)

```
DECLARE
    TYPE noms IS TABLE OF CHAR(5)
        INDEX BY BINARY_INTEGER;

    TYPE noms_pilote IS TABLE OF pilote.nom%TYPE
        INDEX BY BINARY_INTEGER;

    Tab_nom      noms_pilote;
    ...
BEGIN
    Tab_nom(1) := 'Michel Foucault';           /* légal */
    Tab_nom := ('Pascal Seurat', 'Serge Favret'); /* illégal */
    ...
END;
```

Variables Hôtes

- ◆ Une variable hôte est une variable définie par l'environnement d'exécution de PL/SQL
 - Sa déclaration diffère donc selon l'environnement
- ◆ PL/SQL est capable de référencer ces variables en les préfixant par ':'

Exemple : dans l'environnement SQL*Plus

- Une variable peut être déclarée grâce à la commande :
`VARIABLE var type_var`
- Sa valeur peut être affichée grâce à la commande :
`PRINT var`

Exemple : variable hôte

```
variable :somme integer;  
set :somme 10;  
print :somme;
```

```
BEGIN
```

```
    :somme := 0;
```

```
    FOR i IN 1..10 LOOP
```

```
        :somme := :somme + i;
```

```
    END LOOP;
```

```
END;
```

```
/
```

Ambiguïté de Nommage

- ◆ Les noms des colonnes l'emportent toujours sur les noms des variables
- ◆ Les noms des variables l'emportent toujours sur les noms des relations

```
DECLARE
    nom    CHAR(15) := 'Serge Favret';
BEGIN
    DELETE FROM pilote WHERE nom=nom;
END;
```



Supprime tous les pilotes

Exo 2 : Déclaration de Variables et Constantes

◆ Soit le Bloc PL/SQL suivant :

```
DECLARE
    delai                integer Not null
    ville1               CHAR(10) = Paris;
    ville2               CHAR(0);
    Nbre_heure           INTEGER CONSTANT;
BEGIN
    delai = 1;
    ville2= 'Londres'
    INSERT INTO vol VALUES ('RT200', null, null, ville1, ville2);
END;
/
```

Dans les sections DECLARE et BEGIN, corrigez les erreurs de syntaxe commises.

Correction Exercice 2

DECLARE

```
delai      Integer not null :=1;      /* ne pas oublier les points virgule*/
ville1     CHAR(10) := 'Paris';      /* ne pas oublier les ' */
ville2     CHAR(10);
Nbre_heure constant integer :=0;
```

BEGIN

```
delai :=1; /* devient inutile puisque déjà initialisé dans la section DECLARE */
ville2 := 'Londres';
INSERT INTO vol VALUES('RT200', null, null, ville1, ville2);
END;
```

**Remarque : les affectations peuvent se faire indifféremment dans la partie
DECLARE ou BEGIN**

Exercice 3

Fonctions de conversions

Soit le Bloc PL/SQL de l'exercice 3 :

```
DECLARE
    date1          DATE;
    date2          DATE;
    ville1         CHAR(10) := 'Paris' ;
    ville2         CHAR(10) := 'Londres' ;
BEGIN
    date1:= ??      /* 18 heures et 20 minutes */
    date2:= ??      /* 19 heures et 10 minutes */
    INSERT INTO vol VALUES ('RT201', date1, date2, ville1, ville2);
END;
/
```

Travail demandé : Compléter le corps de ce bloc afin que le tuple inséré corresponde à un vol partant de Paris à 18h20 et arrivant à Londres à 19h10

Correction Exercice 3

```
DECLARE
    date1 DATE;
    date2 DATE;
    ville1 CHAR(10) := 'Paris';
    ville2 CHAR(10) := 'Londres';
BEGIN
    date1 := TO_DATE('18:20', 'HH24:MI');
    date2 := TO_DATE('20:10', 'HH24:MI');
    INSERT INTO vol VALUES('RT201', date1, date2, ville1, ville2);
END;
```

3- Test de l'exécution sous SQL*PLUS :

```
SELECT numvol, TO_CHAR(heure_depart, 'HH24:MI') "Heure_depart" ,
           TO_CHAR (heure_arrivee, 'HH24:MI') "Heure_arrivee",
           ville_depart, ville_arrivee
FROM vol;
```

Exercice 4

Utilisation des records

Soit le blocPL/SQL suivant :

```
DECLARE
    num_av          avion.numav%TYPE;
    type_av         avion.type%TYPE;
    ca_av           avion.capacite%TYPE;
    ent_av          avion.entrepot%TYPE;
BEGIN
    SELECT * INTO num_av, type_av, ca_av, ent_av FROM avion WHERE numav = 1;
    /* Attention : Le select ne doit retourner qu'un seul tuple. Voir chapitre sur les Curseurs */
    INSERT INTO avion VALUES (num_av, type_av, ca_av, ent_av);
END;
/
```

Travail demandé :

Analysez ce bloc. Puis, modifier les déclarations et le corps du bloc afin d'utiliser l'attribut %ROWTYPE à la place des attributs %TYPE, puis exécuter le bloc.

Travail complémentaire :

Modifier le bloc en utilisant uniquement une variable de type RECORD

Correction Exercice 4

1°) Ce bloc a pour effet de sélectionner dans la relation avion, le tuple correspondant au numav=1 puis de l'insérer de nouveau dans la base.

2°) Utilisation de '%ROWTYPE'

```
DECLARE
    rec_av    avion%ROWTYPE;
BEGIN
    SELECT * INTO rec_av FROM avion WHERE numav = 1;
    INSERT INTO avion
        VALUES (rec_av.numav, rec_av.type, rec_av.capacité, rec_av. entrepot);
END;
```

2°) Utilisation de RECORD

```
DECLARE
    TYPE rec_avion IS RECORD
        (num_av avion.numav%TYPE, type_av avion.type%TYPE,
         ca_av avion.capacite%TYPE, ent_av avion.entrepot%TYPE);
    rec_av    rec_avion;
BEGIN
    SELECT * INTO rec_av FROM avion WHERE numav = 1;
    INSERT INTO avion
        VALUES (rec_av.num_av, rec_av.type_av, rec_av.ca_av, rec_av. ent_av);
END;
```

Instructions conditionnelles

```
IF condition1 THEN traitement1  
  ELSIF condition2 THEN traitement2  
  ELSE traitement3;  
END IF;
```

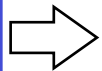
Dans les conditions, tous les opérateurs de comparaison SQL sont utilisés :

(=, >, <, >=, <=, IS NULL, IS NOT NULL, BETWEEN, LIKE.....)

Contrôles Itératifs

♦ PL/SQL propose trois structures de contrôles itératifs :

- Les clauses LOOP et EXIT
- La clause WHILE LOOP
- La clause FOR LOOP



Le choix est fonction du contexte et des habitudes de programmation

Clause LOOP

- ◆ Notion de boucle infinie, la séquence d'instruction est exécutée à chaque itération
- ◆ Besoin d'une instruction de sortie de boucle
 - EXIT
 - EXIT-WHEN



OBLIGATOIRE

- ◆ Syntaxe

```
BEGIN
  LOOP nom_boucle
    instructions;
    EXIT nom_boucle WHEN condition;
  END LOOP;
END;
```


Clause EXIT-WHEN

```
IF count > 100 THEN  
    EXIT;  
END IF;
```



```
EXIT WHEN count > 100;
```

```
LOOP  
    ...  
    EXIT WHEN count > 100;  
    ...  
END LOOP;
```

Clause WHILE-LOOP

- ◆ C'est une boucle conditionnée

- ◆ Syntaxe

```
WHILE condition LOOP  
    séquence d'instructions;  
END LOOP;
```

- ◆ Condition évaluée à chaque début d'itération :

- TRUE : exécution de la séquence d'instructions
- FALSE ou NULL : arrêt de la boucle

```
WHILE x <= 2000 LOOP  
    somme := x + somme;  
    x := x + 1;  
END LOOP;
```

Clause FOR - LOOP (1)

- ◆ C' est une boucle conditionnée (nombre d'itérations connu)
- ◆ Syntaxe
FOR *compteur* **IN** [**REVERSE**] *borne_inf..borne_sup* **LOOP**
séquence d'instructions;
END LOOP;
- ◆ Les bornes sont évaluées une fois pour toute au début de la boucle et jamais réévaluées
- ◆ Le compteur doit être traité comme une constante à l'intérieur de la boucle

```
FOR numero IN 1..10 LOOP  
    INSERT INTO pilote VALUES (numero, NULL, NULL, NULL, NULL);  
END LOOP;
```

Clause FOR - LOOP (2)

- ◆ Utilisation possible de variables pour les bornes

```
SELECT COUNT(matricule) INTO nbre_pilote  
FROM pilote;  
FOR i IN 1..nbre_pilote LOOP  
    ...  
END LOOP;
```

- ◆ Sortie prématurée de la boucle : clause EXIT

- ◆ Imbrication de boucles possibles

```
FOR i IN 1..10 LOOP  
    ...  
    FOR j IN 1..5 LOOP  
        ...  
        EXIT WHEN ...  
    END LOOP;  
END LOOP;
```

Exercice 5

Les Contrôles Itératifs

Travail demandé :

Afin de maîtriser la syntaxe et l'utilisation des contrôles itératifs sous PL/SQL, il vous est demandé d'écrire un bloc PL/SQL qui calcule le factoriel de 10 (10!) en utilisant une boucle FOR.

Travail complémentaire :

Réécrire ce bloc en utilisant successivement une boucle LOOP puis une boucle WHILE.

Correction Exercice 5

1°) Solution avec la boucle FOR

```
DECLARE
    Factoriel    INTEGER := 2;
BEGIN
    FOR i IN 3..10 LOOP
        Factoriel := Factoriel * i;
    END LOOP;
    DBMS_OUTPUT.put_line('Factoriel de 10 = ' || factoriel);
END;
```

2°) Solution avec la boucle LOOP

```
DECLARE
    i            INTEGER := 2;
    Factoriel    INTEGER := 2;
BEGIN
    LOOP
        i := i + 1;
        Factoriel := Factoriel * i;
        EXIT WHEN i = 10;
    END LOOP;
    DBMS_OUTPUT.put_line('Factoriel de 10 = ' || Factoriel);
END;
```

Correction Exercice 5 (suite)

3°) Solution avec la boucle WHILE

DECLARE

i **INTEGER := 2;**

Factoriel **INTEGER := 2;**

BEGIN

WHILE i < 10 LOOP

i := i + 1;

Factoriel := Factoriel * i;

END LOOP;

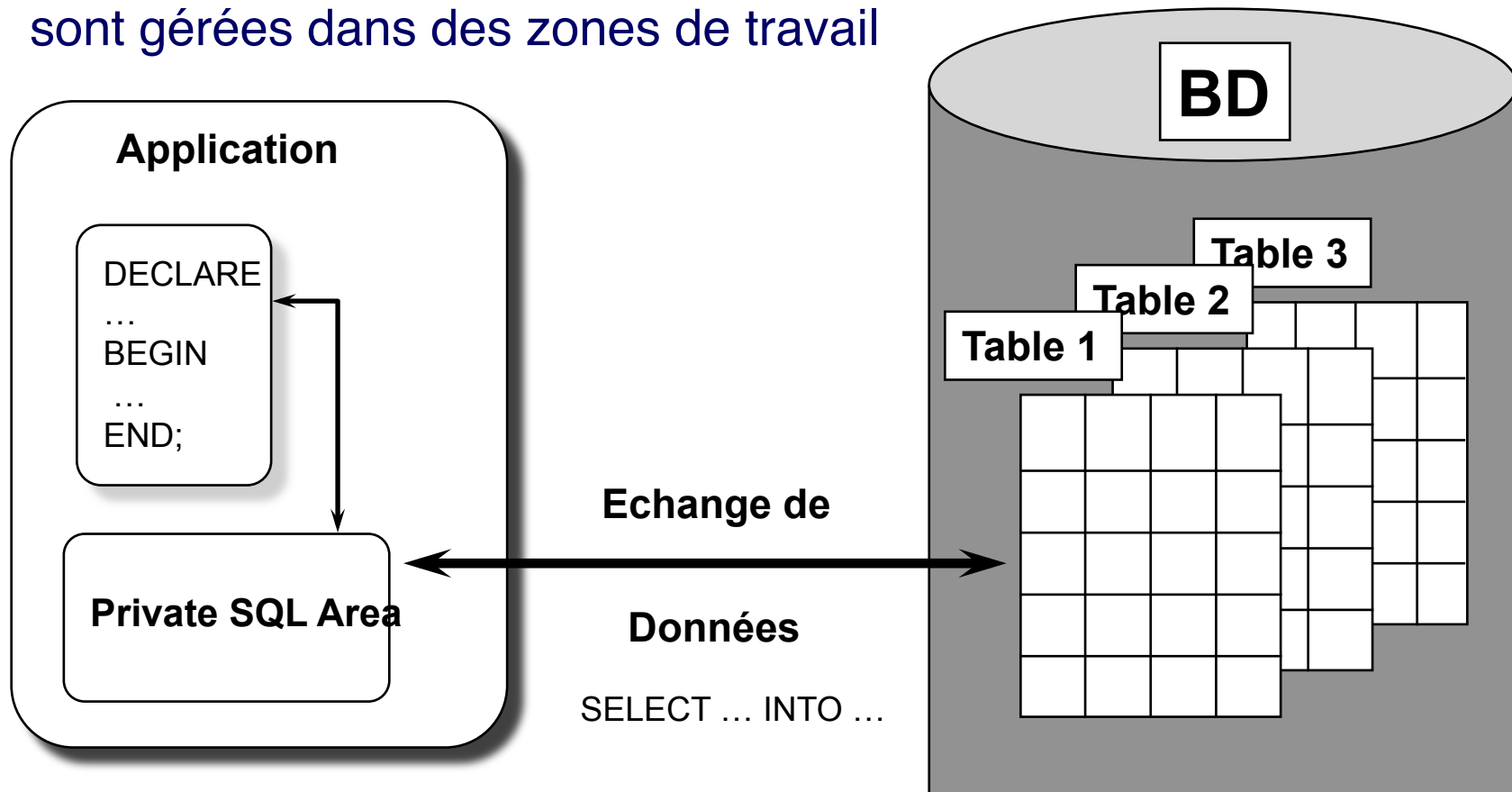
DBMS_OUTPUT.put_line('Factoriel de 10 = ' || Factoriel);

END;

Les Curseurs

PL/SQL : Interaction avec Oracle

- ◆ Les données des bases sont échangées via des CURSEURS et sont gérées dans des zones de travail



- La private Area permet d'obtenir des informations sur l'échange de données

Types de Curseurs

- ◆ PL/SQL propose deux types de CURSEURS :
 - CURSEURS ***Implicites*** gérés et déclarés par PL/SQL
 - CURSEURS ***Explicites*** gérés et déclarés par l'utilisateur

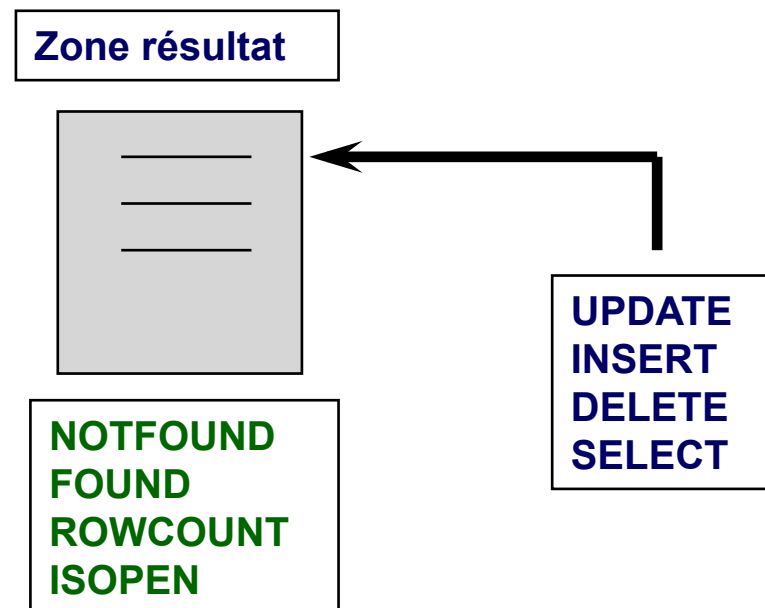
- ◆ PL/SQL utilise un curseur implicite pour :
 - chaque instruction SQL de LMD de mise à jour
 - chaque interrogation retournant un seul tuple

- ◆ Remarque :
 - Une instruction «Select ... From ... Where» qui rend plus d'un tuple doit **obligatoirement** être associée à un curseur explicite.

Curseurs Implicites

- ♦ Toute instruction SQL non associée à un curseur explicite est associée automatiquement à un curseur implicite
- ♦ Un curseur implicite SQL permet d'avoir des informations sur l'exécution de INSERT, UPDATE, DELETE, et SELECT INTO ...
- ♦ Exemple

```
UPDATE ...  
IF SQL%NOTFOUND THEN  
...  
END IF
```



Attributs de Curseurs Implicites (1)

- ◆ L'appel des attributs des curseurs implicites est pré-fixé par la clause SQL:
- ◆ SQL%NOTFOUND = TRUE
 - UPDATE, DELETE ne touche aucun tuple
 - SELECT INTO ne retourne pas de tuples
 - Dans ce cas, l'exception NO_DATA_FOUND est soulevée
- ◆ SQL%FOUND est l'inverse logique de %NOTFOUND
- ◆ SQL%ISOPEN
 - Cet attribut est toujours FALSE car le curseur est immédiatement fermé une fois l'exécution de la commande associée est terminée

Attributs des Curseurs Implicites (2)

- ◆ SQL%ROWCOUNT est mis à :
 - Zéro si aucun tuple n'est concerné ou sélectionné
 - Sinon, au nombre de tuples concernés en cas de INSERT, DELETE, UPDATE
 - Sinon, à un, en cas de SELECT INTO
 - L'exception TOO_MANY_ROWS est soulevée si SELECT INTO retourne plus qu'un tuple

Exemple : curseur implicite

```
DECLARE
    nom_pilote pilote.nom%TYPE := 'Armand Pousin';

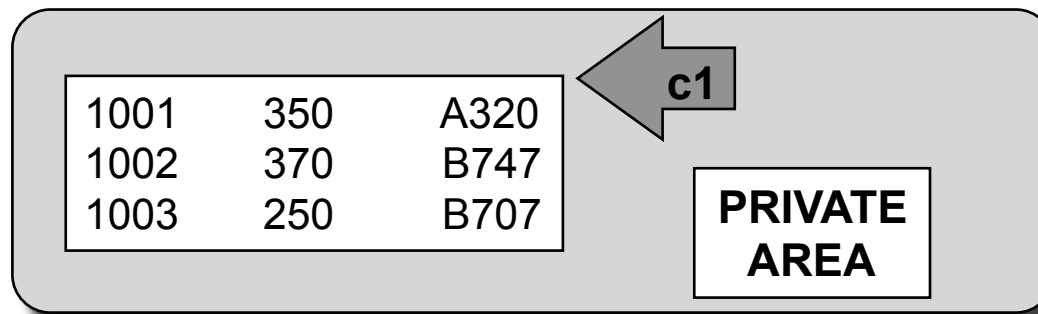
BEGIN
    UPDATE pilote SET nom = 'Armand Poussin'
    WHERE nom = nom_pilote;

    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.put_line('Avertissement : Le pilote '
                               || nom_pilote || ' n"existe pas');
    END IF;
END;
```

Curseurs Explicites (1)

- ◆ Un curseur explicite est défini dans la partie déclarative
 - d'un bloc PL/SQL
 - d'un sous-programme
 - d'un paquetage (package)
- ◆ Exemple

```
DECLARE
  CURSOR c1 IS    SELECT numav, capacite, type
                    FROM avion
                    WHERE capacite < 20;
```



Curseurs Explicites (2)

- ◆ Le nom du curseur n'est pas une variable
 - Il ne peut pas être utilisé dans une expression
 - Il ne peut pas être assigné à une valeur
- ◆ On peut seulement utiliser le nom d'un curseur pour référencer l'interrogation associée
- ◆ Les mêmes règles de portée appliquées aux variables s'appliquent aussi aux curseurs

Curseurs Explicites Paramétrés

- ♦ Un paramètre d'un curseur peut être utilisé dans l'interrogation associée à la place d'une constante
- ♦ Syntaxe :

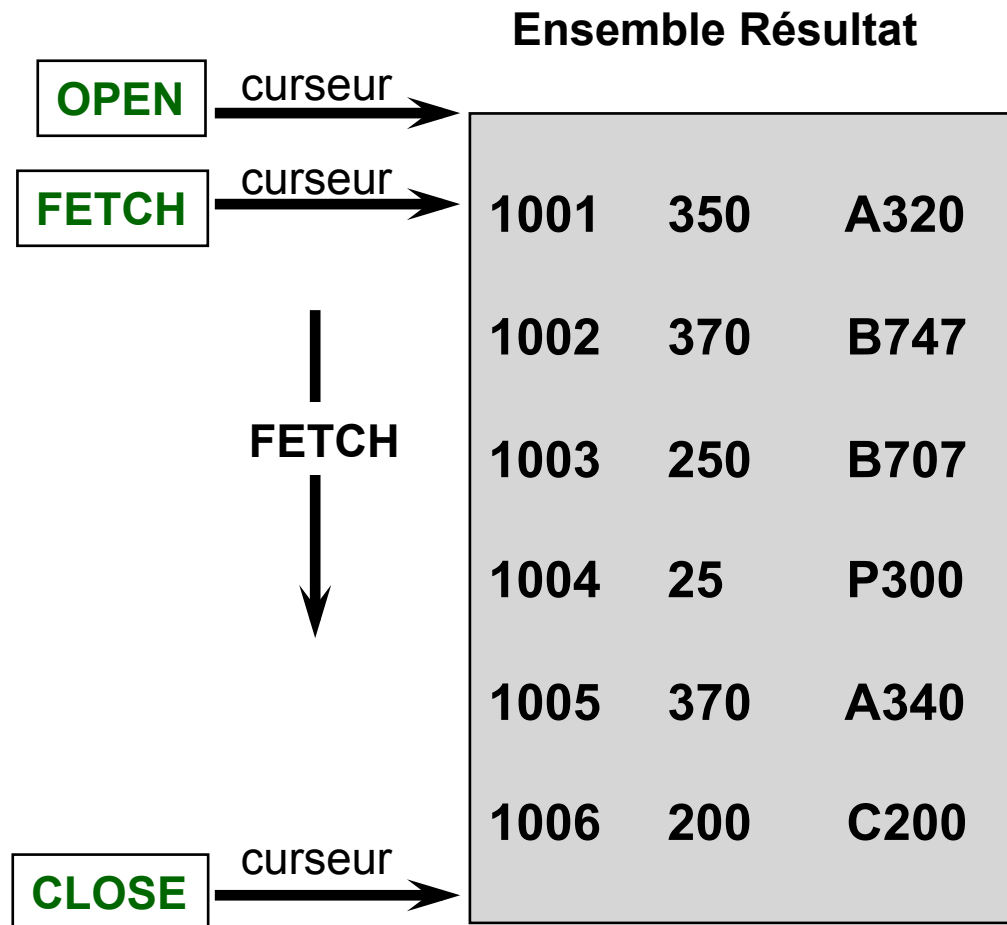
```
CURSOR nom [ ( paramètre [, paramètre ] ... ) ] IS  
SELECT ...
```
- ♦ Syntaxe de *paramètre* :

```
nom_var [ IN ] type_données [ { := | DEFAULT } valeur ]
```
- ♦ Exemple :

```
CURSOR c1(seuil NUMBER DEFAULT 100) IS  
SELECT numav, capacite, type  
FROM avion  
WHERE capacite < seuil;
```
- ♦ Le paramètre formel d'un curseur n'est reconnu que dans la définition du curseur

Utilisation des Curseurs Explicites

- ◆ Trois opérateurs d'utilisation :



Ouverture d'un Curseur

- ◆ L'ouverture d'un curseur signifie l'exécution de la requête associée
- ◆ Ouverture d'un curseur non paramétré
 - `OPEN c1;`
- ◆ Ouverture d'un curseur paramétré
 - `OPEN c1(3, 70);`
 - `OPEN c1(bas, 60);`
 - `OPEN c1(bas, haut);`
- ◆ Un paramètre effectif doit avoir un type de données compatible avec celui du paramètre formel correspondant

Instruction **FETCH**

- ◆ L'instruction **FETCH** extrait un seul tuple à la fois de l'ensemble actif
- ◆ Le curseur avance vers le tuple suivant après chaque appel de **FETCH**
- ◆ Pour chaque colonne retournée par la requête associée au curseur, il faut utiliser une variable de type correspondant dans la liste de **INTO**
- ◆ Exemple
FETCH c1 **INTO** numav_var, capacite_var, type_var;

Fermeture d'un Curseur

- ♦ L'opération de fermeture désactive le curseur et rend l'ensemble actif indéfini
- ♦ Cette opération se fait grâce à l'instruction **CLOSE**
- ♦ Exemple
 - **CLOSE** c1;
- ♦ Une opération autre que l'ouverture sur un curseur fermé soulève l'exception pré-définie **INVALID_CURSOR**

Attributs des Curseurs Explicites (1)

- ◆ L' appel des attributs des curseurs est pré-fixé par le nom du curseur (**ex : C1%Found**)
- ◆ Cursor_name%FOUND
 - Vrai lorsque FETCH rend un tuple avec succès
- ◆ Cursor_name %NOTFOUND
 - L'inverse de l'attribut précédent
- ◆ Cursor_name %ROWCOUNT
 - Zéro à l'ouverture avant toute opération FETCH
 - Augmenté par un à chaque FETCH réussi
- ◆ Cursor_name %ISOPEN
 - Vrai si le curseur est ouvert

Attributs des Curseurs Explicites (2)

		%NOTFOUND	%FOUND	%ROWCOUNT	%ISOPEN
OPEN	Avant	*	*	*	FALSE
	Après	NULL	NULL	0	TRUE
premier FETCH	Avant	NULL	NULL	0	TRUE
	Après	FALSE	TRUE	1	TRUE
FETCH médian	Avant	FALSE	TRUE	suivant données	TRUE
	Après	FALSE	TRUE	suivant données	TRUE
dernier FETCH	Avant	FALSE	TRUE	suivant données	TRUE
	Après	TRUE	FALSE	suivant données	TRUE
CLOSE	Avant	TRUE	FALSE	suivant données	TRUE
	Après	*	*	*	FALSE

Une Séquence Typique

```
DECLARE
    CURSOR C1 is SELECT ...
    ...
BEGIN
    OPEN c1;
LOOP
    FETCH c1 INTO record_var;
    EXIT WHEN c1%NOTFOUND;
        -- Traitement des données extraites
END LOOP;
    CLOSE c1;
END;
```


Boucles FOR à Curseurs (1)

```
DECLARE
    somme_salaires      INTEGER := 0;
    CURSOR c1 IS
        SELECT *
        FROM pilote
        WHERE age > 35;

BEGIN
    FOR tuple IN c1 LOOP
        /* Traitement des données dans la variable tuple */
        somme_salaires := somme_salaires + tuple.salaire;
    END LOOP;
END;
```

La boucle se charge de l'ouverture, du fetch et de la fermeture du curseur.

Boucles FOR à Curseurs (2)

♦ Passage de paramètres

```
DECLARE
    CURSOR c1(seuil INTEGER) IS
        SELECT type, capacite
        FROM avion
        WHERE capacite < seuil;

    ...
BEGIN
    ...
    FOR c1_rec IN c1(50) LOOP
        ...
    END LOOP;
    FOR c1_rec IN c1(100) LOOP
        ...
    END LOOP;
    ...
END;
```

La clause ' CURRENT OF '

- ◆ Permet d' accéder directement en modification à la ligne que vient de ramener le FETCH
- ◆ Au préalable :
 - réservation des lignes lors de la déclaration du curseur par un verrou d ' intention

La clause ' CURRENT OF '

```
DECLARE
  CURSOR c1 IS
    SELECT Nom, Salaire
    FROM pilote
    FOR UPDATE OF Salaire;

BEGIN
  FOR c1_enr IN c1 LOOP
    IF (c1_enr.Salaire < 1500) THEN
      UPDATE pilote SET Salaire = Salaire*1.3
      WHERE CURRENT OF c1;
    END IF;
  END LOOP;
END;
```

Exercice 8

Les Curseurs Implicites

```
DECLARE
    nom_pilote      pilote.nom%TYPE := 'Armand Pousin';
BEGIN
    UPDATE pilote SET nom = 'Armand Poussin' WHERE nom = nom_pilote;
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.put_line('Avertissement : Le pilote ' ||
                               nom_pilote || ' n'existe pas');
    END IF;
END;
```

Travail demandé :

Ajouter une instruction qui permet de connaître le nombre de tuples affectés par le UPDATE.

Correction Exercice 8

```
DECLARE
    nom_pilote      pilote.nom%TYPE := 'Armand Pousin';
    nombre          integer ; /* pour stocker le nombre de pilotes modifiés */
BEGIN
    UPDATE pilote SET nom = 'Armand Collin' WHERE nom=nom_pilote;
    IF SQL%NOTFOUND
        THEN
            DBMS_OUTPUT.put_line('Avertissement : Le pilote ' || nom_pilote ||
                                ' n'existe pas');
        ELSE
            nombre := SQL%ROWCOUNT;
            DBMS_OUTPUT.put_line('Nombre de pilotes modifiés : ' || nombre);
        END IF;
END;
```

Exercice 9

Les Curseurs Explicites

```
DECLARE
    CURSOR c1 IS SELECT salaire FROM pilote;
    /* à compléter */
BEGIN
    /* à compléter */
    LOOP
        /* à compléter */
    END LOOP;
    /* à compléter */
END;
```

Travail demandé :

- 1°) Complétez la structure précédente afin que le bloc calcule le salaire moyen des pilotes.
- 2°) Ecrire la commande SQL qui calcule la même valeur et comparez ces deux résultats

Correction Exercice 9

1°) Solution proposée

```
DECLARE
    CURSOR c1 IS SELECT salaire FROM pilote;
    sal           pilote.salaire%TYPE;
    somme          NUMBER := 0;
    moyenne       NUMBER;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO sal;
        EXIT WHEN c1%NOTFOUND;
        somme := somme + sal;
    END LOOP;
    moyenne := somme / c1%ROWCOUNT;
    CLOSE c1;
    DBMS_OUTPUT.put_line('La moyenne est : ' || moyenne);
END;
```

2°) `SELECT AVG(salaire) FROM pilote;` /* A utiliser car Oracle peut optimiser cette commande */

Exercice 10

Les Boucles FOR à Curseurs

Travail demandé :

Reprendre l'exercice n°9. Modifiez le bloc en utilisant une boucle FOR à curseur pour calculer le salaire moyen.

Remarque :

Outre la différence de programmation, il convient de noter qu'après le END LOOP, les attributs de curseurs ne sont plus utilisables (ex : C1%Rowcount).

Correction Exercice 10

Solution proposée :

```
DECLARE
    CURSOR c1 IS SELECT salaire FROM pilote;
    somme          NUMBER := 0;
    moyenne       NUMBER;
    nombre_pilotes BINARY_INTEGER := 0;
BEGIN
    FOR rec IN c1 LOOP
        nombre_pilotes := nombre_pilotes + 1;
        somme := somme + rec.salaire;    /* rec est une variable de type record */
    END LOOP;

    moyenne := somme / nombre_pilotes;
    DBMS_OUTPUT.put_line('La moyenne est : ' || moyenne);
END;
```

Remarque : C1%Rowcount n'est plus valide au dehors de la boucle puisque celle-ci effectue automatiquement un CLOSE de C1.

Exercice 11

Les Curseurs en Mise à Jour

```
DECLARE
    CURSOR c1 IS SELECT salaire FROM pilote;
    /* à compléter */
BEGIN
    /* à compléter */
END;
```

Travail à réaliser :

1°) Complétez ce bloc PL/SQL pour qu'il permette d'augmenter de 5000F les salaires de tous les pilotes et ce de façon itérative.

2°) Quelle est la commande SQL qui permet d'obtenir le même résultat de façon ensembliste ?

Correction Exercice 11

1°) Solution proposée :

```
DECLARE  
    CURSOR c1 IS SELECT salaire FROM pilote FOR UPDATE of salaire;  
BEGIN  
    FOR rec IN c1 LOOP  
        UPDATE pilote SET salaire = rec.salaire + 5000  
        WHERE CURRENT OF c1;  
    END LOOP;  
    -- COMMIT;  
END;  
/
```

2°) Une seule commande SQL utilisant un curseur implicite

```
UPDATE pilote SET salaire = salaire + 5000;
```

La gestion des erreurs

Gestion des erreurs

◆ La section EXCEPTION

- permet d' affecter un traitement approprié aux erreurs survenues lors de l' exécution d' un bloc PL/SQL

◆ Deux types d' erreurs :

- erreurs internes Oracle
- erreurs programmes (dues à l' utilisateur)

Erreurs utilisateurs

```
DECLARE
    ....
    nom_erreur EXCEPTION;
    ....
BEGIN
    ....
    IF anomalie THEN RAISE nom_erreur ;
    ....
EXCEPTION

    WHEN nom_erreur THEN traitement;

END;
```

Ex : Gestion d' une erreur utilisateur

```
DECLARE
    pb_sal      EXCEPTION;
    sal         pilote.salaire%TYPE;

BEGIN
    SELECT MAX(salaire) INTO sal FROM pilote;
    IF sal > 100 000
        THEN RAISE pb_sal;
    END IF;

EXCEPTION
    WHEN pb_sal THEN
        INSERT INTO resultat VALUES ( ' plafond dépassé ' );
END;
```


Erreurs Oracle

```
DECLARE
    ....
    nom_erreur EXCEPTION;
    PRAGMA EXCEPTION_INIT (nom_erreur, code);
    ....
BEGIN
    ....
    ....
EXCEPTION
    WHEN nom_erreur THEN traitement1;
    [WHEN OTHERS THEN traitement2;]
END;
```

Erreurs Oracle

-1 DUP_VAL_ON_INDEX

-1001 INVALID CURSOR

-1476 ZERO_DIVIDE

.....

Erreurs Oracle - Exemple

- ◆ Gestion de l'erreur Oracle
DUP_VAL_ON_INDEX, pour détecter les
doublons sur la clé primaire dans la table :
Pilote(Num, Nom, Ville, Salaire)
- ◆ Enregistrer les tuples ne pouvant pas être
insérés dans la table doublons(Num, Nom)

Erreurs Oracle - Exemple

BEGIN

.....

INSERT INTO pilote VALUES (num, nom, ville, salaire);

.....

EXCEPTION

WHEN DUP_VAL_ON_INDEX THEN

INSERT INTO doublons VALUES (num, nom);

END;

Fonctions PL/SQL

◆ SQLCODE

- Renvoie le code (valeur numérique) de l'erreur courante

◆ SQLERRM [<code_erreur>]

- Renvoie le libellé de l'erreur courante ou le libellé correspondant au code erreur s'il est spécifié

Procédures, fonctions et packages

Procédures et fonctions

◆ Procédure :

- unité de traitement qui peut contenir des instructions SQL (sauf LDD), des instructions PL/SQL, des variables, des constantes, des curseurs et un gestionnaire d'erreurs

◆ Fonction :

- procédure qui retourne une valeur

Procédures et fonctions

- ◆ Stockées sous forme compilée dans la base de données, de la même façon qu'un objet de la base
 - soumise aux mécanismes de sécurité et de confidentialité
- ◆ Partagées par plusieurs applications et utilisateurs
 - à condition d'avoir le privilège EXECUTE

Package

- ◆ Encapsulation au sein d ' une même unité logique nommée de :
 - Fonctions
 - Procédures
 - Curseurs
 - Variables
 - Constantes

Utilité

- ◆ Augmenter la productivité
- ◆ Gérer la sécurité et la confidentialité des données
- ◆ Augmenter les performances

Création d'une procédure/fonction

CREATE [OR REPLACE] PROCEDURE

```
[nom_user.]nom_procedure [(liste d ' arguments)]  
{IS | AS}  
bloc PL/SQL
```

CREATE [OR REPLACE] FUNCTION

```
[nom_user.]nom_fonction [(liste d ' arguments)]  
RETURN type  
{IS | AS}  
bloc PL/SQL
```

Arguments d ' une procédure/fonction

(liste d ' arguments) ::=

nom argument 1 {IN | OUT | IN OUT} type,

nom argument 2 {IN | OUT | IN OUT} type,

.....

nom argument n {IN | OUT | IN OUT} type,

Exemples

```
CREATE PROCEDURE
    modif_prix (id IN NUMBER, taux IN NUMBER) IS
BEGIN
    UPDATE Article a
    SET a.prix_unitaire = a.prix_unitaire*(1+taux)
    WHERE a.id_article = id;
END;
```

```
CREATE FUNCTION
    chiffre_affaire (id IN NUMBER) RETURN NUMBER IS
    ca NUMBER;
BEGIN
    SELECT SUM(montant) INTO ca
    FROM vendeurs
    WHERE id_vendeur = id;
    RETURN ca;
END;
```

Modification/Suppression d ' une procédure/fonction

- ◆ Pour recompiler les procédures et fonctions en cas de modification du schéma de la base de données
 - ALTER {PROCEDURE I FUNCTION} nom COMPILE;
- ◆ Pour supprimer :
 - DROP {PROCEDURE I FUNCTION} nom;

Les packages

- ◆ Introduits à partir de la version 7 d ' Oracle
- ◆ Permettent d ' encapsuler des procédures, fonctions, curseurs, variables comme une unité logique dans la base de données

Les packages

- ◆ Structuration et organisation des traitements
- ◆ Facilité de gestion des privilèges
- ◆ Facilité de gérer la sécurité par la spécification
 - d ' une partie privée du package
 - d ' une partie publique

Création d ' un package

- ◆ Deux étapes :
 - Création des spécifications du package
 - Création du corps du package

Création des spécifications d' un package

- ◆ Définition des éléments du package accessibles par le public

```
CREATE [OR REPLACE] PACKAGE  
    [num_user.]nom_package {IS | AS}  
    spécification PL/SQL ::=  
        déclaration de variables  
        déclaration d ' enregistrements  
        déclaration d ' exception  
        déclaration de curseurs  
        déclaration de procédures  
        déclaration de fonctions  
  
    END nom_package;
```

Création du corps du package

- ◆ Définition des éléments privés du package

CREATE [OR REPLACE] PACKAGE BODY

[num_user.]nom_package {IS | AS}

spécification PL/SQL ::=

déclaration de variables

déclaration d ' enregistrements

déclaration d ' exception

déclaration de curseurs

déclaration de procédures

déclaration de fonctions

END nom_package;

Exemple

```
CREATE PACKAGE traitements_vendeurs IS
```

```
    FUNCTION chiffre_affaire (Id_Vendeur IN NUMBER)  
    RETURN NUMBER;
```

```
    PROCEDURE modif_com (Id IN NUMBER, tx IN NUMBER);
```

```
END traitements_vendeurs;
```

Exemple - suite

```
CREATE PACKAGE BODY traitements_vendeurs IS
```

```
    FUNCTION chiffre_affaire (Id_Vendeur IN NUMBER)  
    RETURN NUMBER      IS
```

```
    ca NUMBER;
```

```
    BEGIN
```

```
        SELECT SUM(montant) INTO ca FROM vendeurs
```

```
        WHERE id_vendeur = id;
```

```
        RETURN ca;
```

```
    END;
```

```
    PROCEDURE modif_com (Id IN NUMBER, tx IN NUMBER) IS
```

```
    BEGIN
```

```
        UPDATE vendeurs v
```

```
        SET v.commission = v.commission*(1+tx)
```

```
        WHERE v.id_vendeur = id;
```

```
    END;
```

```
END traitements_vendeurs ;
```

Conclusion

- ◆ Les langages de type PL/SQL constituent le mode d'utilisation le plus courant des bases de données relationnelles
- ◆ Utilisé pour programmer des procédures stockées ou des triggers
- ◆ Utilisé par les outils de développement de plus haut niveau (SQL*Forms)

Les Triggers

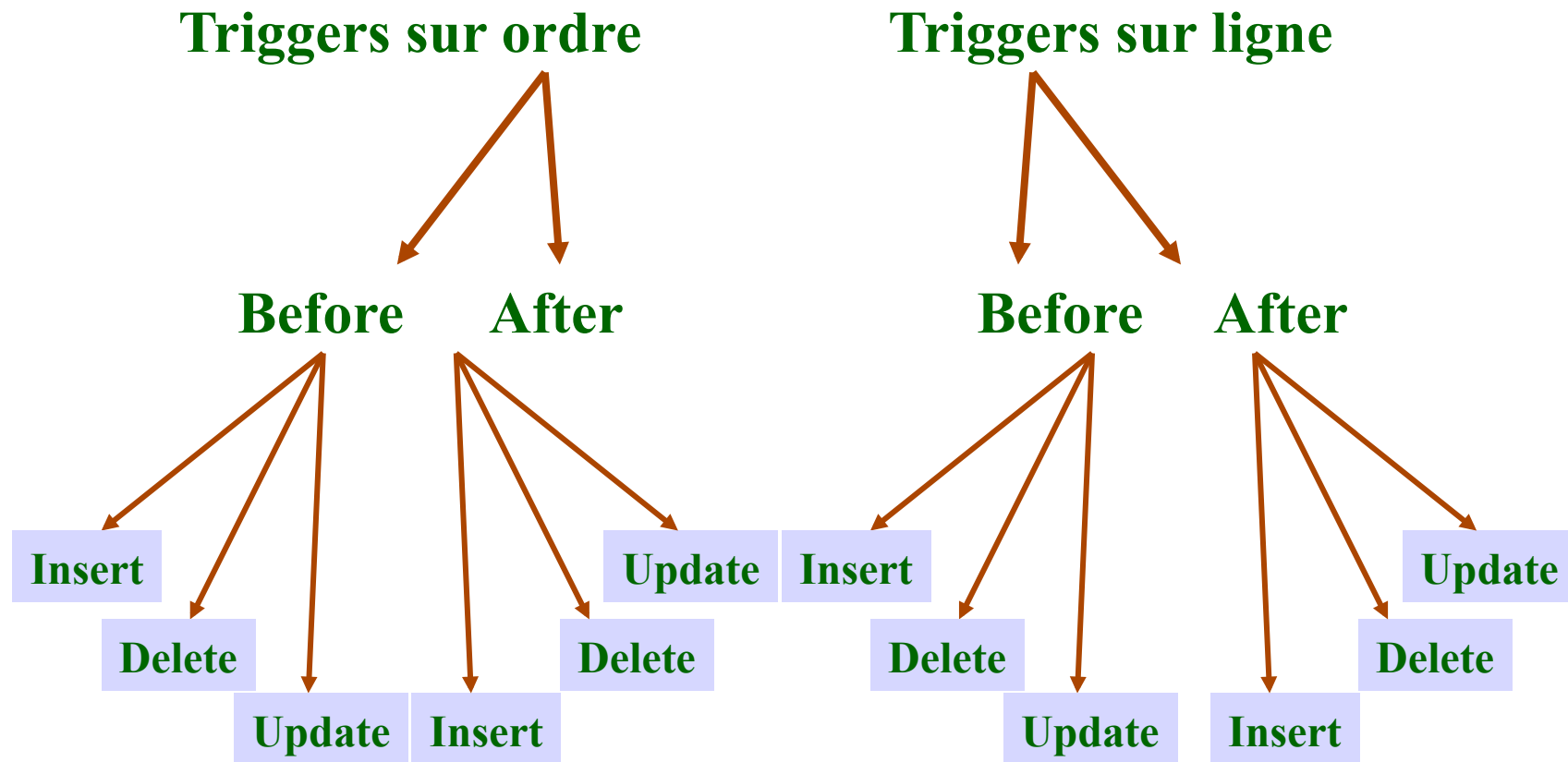
Introduction

- ◆ Un trigger de type Événement/Action (EA) est composé de :
 - un événement qui va le déclencher,
 - une action à exécuter au déclenchement.

Les Triggers d'Oracle 7

- ◆ Un trigger est associé à une relation :
 - L'événement qui le déclenche est une opération de mise à jour sur cette relation (insertion, suppression, modification).
 - L'action à exécuter est spécifiée par un bloc PL/SQL.

Les triggers d' Oracle 7



Trigger Ordre

```
CREATE TRIGGER nom_du_trigger  
{BEFORE | AFTER }
```

```
{  
    UPDATE |  
    INSERT |  
    DELETE |  
    UPDATE OF nom_attribut1 [,om_attribut1...] |  
    UPDATE OR INSERT OR DELETE }
```

```
ON nom_de_la_relation
```

```
Bloc PL/SQL ;
```

Trigger Ordre - Exemple

```
CREATE TRIGGER suppression_impossible
```

```
BEFORE DELETE ON commande
```

```
/* Bloc PL/SQL */
```

```
BEGIN
```

```
    RAISE_APPLICATION_ERROR (-20001,  
                             'Suppression de commande impossible');
```

```
END;
```

- ♦ Le bloc PL/SQL sera exécuté avant chaque instruction de suppression dans la table "commande".

Trigger Ligne

```
CREATE TRIGGER nom_du_trigger  
  {BEFORE | AFTER }
```

```
  {  
    UPDATE |  
    INSERT |  
    DELETE |  
    UPDATE OF nom_attribut1 [,om_attribut1...] |  
    UPDATE OR INSERT OR DELETE }  
  ;
```

```
ON nom_de_la_relation
```

```
FOR EACH ROW
```

```
Bloc PL/SQL ;
```

Trigger ligne - Exemple

```
CREATE TABLE suivi (d DATE, nombre_total_cde INTEGER);  
INSERT INTO suivi (nombre_total_cde) VALUES (0);
```

```
CREATE TRIGGER cpt  
  AFTER INSERT ON commande FOR EACH ROW  
  /* Bloc PL/SQL */  
  DECLARE  
    nb INTEGER;  
  BEGIN  
    SELECT MAX(nombre_total_cde) INTO nb  
    FROM suivi;  
    nb := nb + 1;  
    INSERT INTO suivi VALUES (SYSDATE, nb);  
  END;
```

- ◆ Le bloc PL/SQL sera exécuté après chaque insertion de ligne dans la table "commande".

Les triggers E/C/A

- ◆ Un trigger Événement/Condition/Action est composé de :
 - un événement qui va le déclencher,
 - une condition à vérifier,
 - une action à exécuter au déclenchement si la condition est vérifiée.

Les triggers E/C/A

- ◆ Un trigger est attaché à une relation :
 - l'événement qui le déclenche est une mise à jour sur cette relation,
 - la condition à vérifier est facultative et porte sur le tuple mis à jour,
 - l'action à exécuter est spécifiée par un bloc PL/SQL.

Exemple

```
CREATE TRIGGER controle
  BEFORE INSERT ON commande FOR EACH ROW
  WHEN (new.ClientId in (23, 567, 2993))

  /* Bloc PL/SQL */
  BEGIN
    RAISE_APPLICATION_ERROR (-20003,
      'Client débiteur, commande impossible');

  END;
```

- ◆ La clause WHEN est associée à FOR EACH ROW.
- ◆ La clause WHEN ne peut comporter de requête.
- ◆ Dans une clause WHEN, "new" n'est pas préfixé par ":".

Prédicats INSERTING, DELETING, UPDATING

- ◆ Utilisés dans le bloc PL/SQL du trigger pour les instructions conditionnelles.
- ◆ Permettent de programmer des traitements différenciés

Exemple

```
CREATE TRIGGER traitements_impossibles
  BEFORE INSERT OR UPDATE OR DELETE ON commande

/* Bloc PL/SQL */
BEGIN
  IF INSERTING THEN
    RAISE_APPLICATION_ERROR (-20001, 'Création impossible');
  ELSIF UPDATING THEN
    RAISE_APPLICATION_ERROR (-20002, 'Modification impossible');
  ELSIF DELETING THEN
    RAISE_APPLICATION_ERROR (-20003, 'Suppression impossible');
  END IF;
END.
```

Gestion des Triggers

- ◆ Suppression d'un trigger :
 - DROP TRIGGER nom_du_trigger;
- ◆ Désactivation d'un trigger :
 - ALTER TRIGGER nom_du_trigger DISABLE;
- ◆ Activation d'un trigger :
 - ALTER TRIGGER nom_du_trigger ENABLE;

Conclusion

- ◆ Les triggers permettent de :
 - Maintenir des relations contenant des renseignements sur l'activité de la base.
 - Calculer automatiquement des attributs lors de la mise à jour de tuples ou de tables.
 - Centraliser le contrôle de l'intégrité.
 - Contrôler dynamiquement les manipulations sur la base.