ORACLE PL/SQL

Langage procédurale

PL/SQL

PL/SQL signifie “Procedural Language extensions to SQL”.

Garder des instructions SQL et de lui rajouter d’autres constructions syntaxiques afin d’en faire un véritable langage de programmation procédural

On y trouve :

- Langage d’Interrogation de Données (LID) : SELECT.
- Langage de Manipulation de Données (LMD) : INSERT, UPDATE,DELETE.
- Gestion des transactions : COMMIT, ROLLBACK, SAVEPOINT.
- Les fonctions : TO_CHAR, TO_DATE, UPPER, SUBSTR, ROUND…

En revanche :

Pas de Langage de Définition de Données (LDD) : CREATE, ALTER…

ni de Langage de Contrôle de Données (LCD) : GRANT, REVOKE…

Instructions spécifiques au PL/SQL

Déclarations de variables (avec utilisation des types SQL)

Instructions conditionnelles

Instructions itératives

Traitements des curseurs

Traitements des erreurs par les exceptions.

Syntaxe d'un bloc PL/SQL

```
[DECLARE
   { < déclaration des variable >
   | < déclaration des constante >
   | < déclaration des exception >
   | < déclaration des curseur >
   | ... } ...]

BEGIN
   { < ordre SQL >
   | < instructions PL/SQL > } ...

[EXCEPTION
   < traitement des exceptions (gestion des erreurs) >]
END ;
/
```

Les différents type de variables

Il existe trois catégories de variable utilisables en PL/SQL :

Les variables de substitution (SQL*plus)

&ma_variable

Les variables externes ou de référence

:ma_variable

Les variables scalaires ou structurées PL/SQL.

ma_variable

Les variables de substitution (SQL*plus)

Pour récupérer une valeur saisie par un utilisateur dans une variable de substitution, on utilise la commande SQL*plus :

ACCEPT nomVariableSubst PROMPT message

On peut utiliser les variables de substitution dans un bloc PL/SQL ou dans une commande SQL, à condition de les préfixer de &.

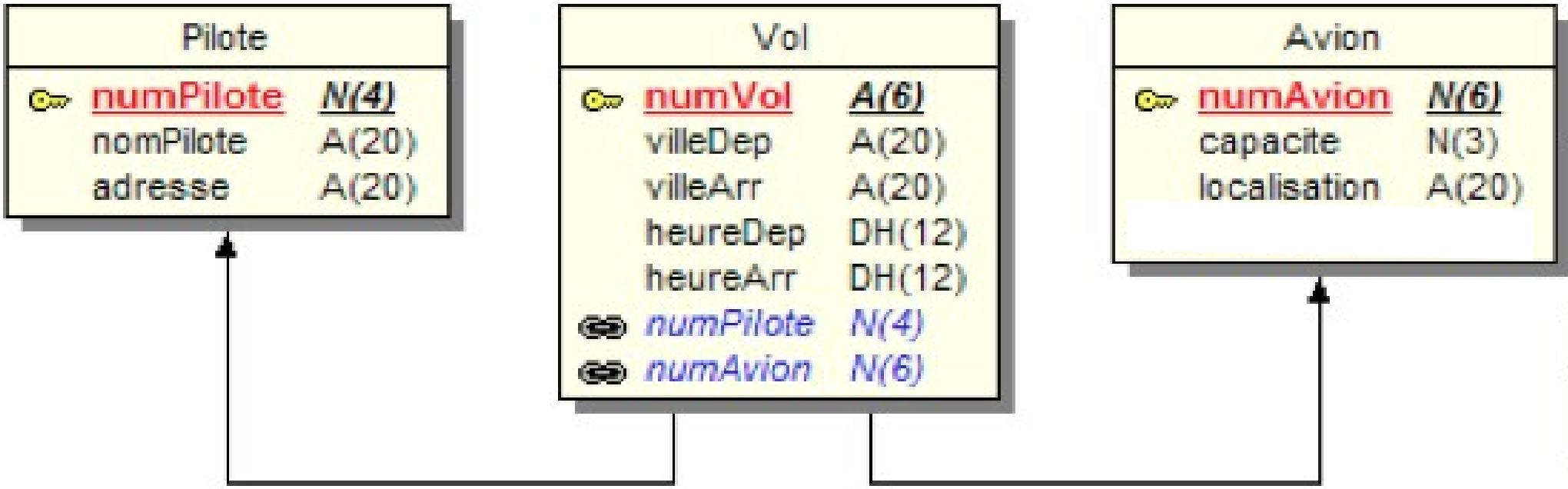
Ce type de variable est appelé "variable de substitution" car Oracle ne fait que substituer la variable par sa valeur.

Exemple :

```
Accept nomTab Prompt 'Table ? ' ;
Select * From &nomTab;
```

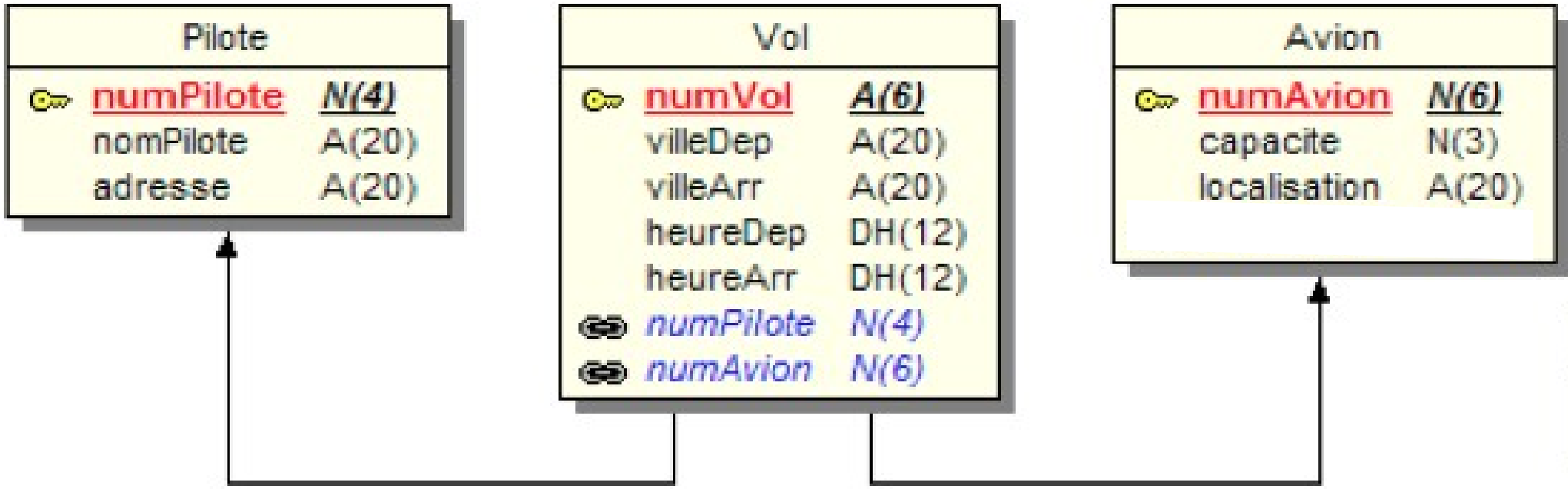
Affiche Table ? Puis une attente de saisie dans la variable nomTab puis un affichage du contenu de la table dont le nom a été saisi dans la variable nomTab.

Les variables de substitution (SQL*plus)



Ecrire un programme qui lit le nom d'un pilote et affiche toutes ses informations.

Les variables de substitution (SQL*plus)

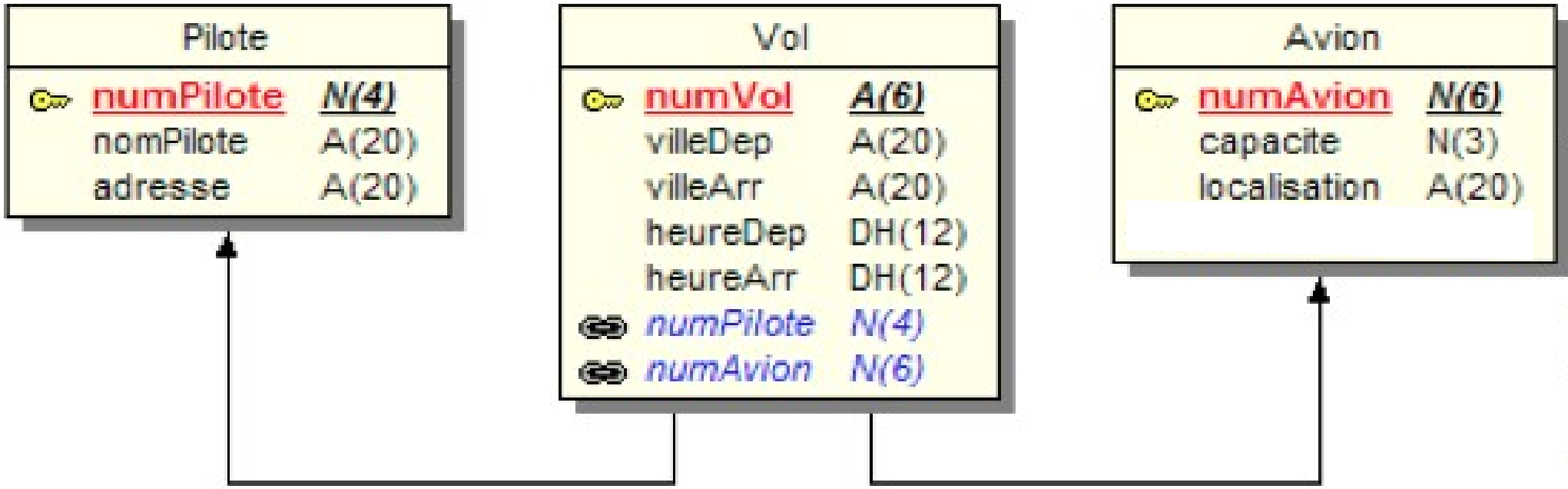


Ecrire un programme qui lit le nom d'un pilote et affiche toutes ses informations.

Accept nomP Prompt 'Donnez le nom ' ;
Select * From Pilote where nomPilote='&nomP';

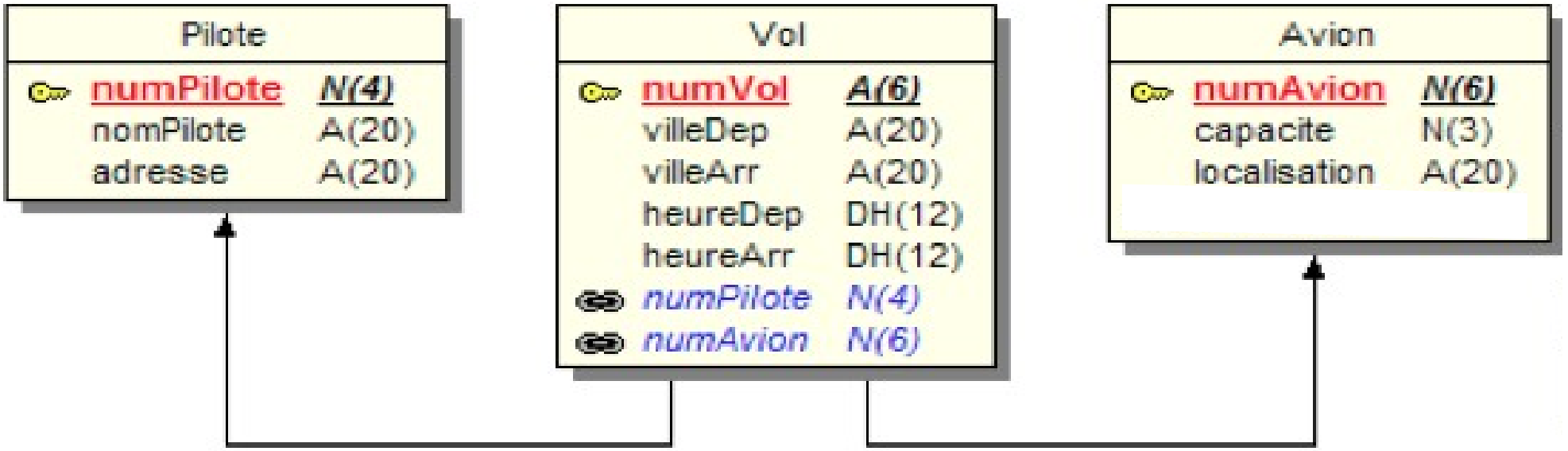
Les 'sont obligatoire car Oralce remplace la variable de substitution par sa valeur donc si on ne met pas les ' on aura : Select * From Pilote where nomPilote=**dupont** ; ce qui est faux syntaxiquement

Les variables de substitution (SQL*plus)



Ecrire un programme qui affiche tous les avions qui ont fait au moins un vol mais qui sont jamais allait à la ville X avec le pilote Y où X et Y sont des variables de sub qui représentent le nom de la ville d'arrivée et le nom du pilote.

Les variables de substitution (SQL*plus)



Ecrire un programme qui affiche tous les avions qui ont fait au moins un vol mais qui sont jamais allait à la ville X avec le pilote Y où X et Y sont des variables de sub qui représentent le nom de la ville d'arrivée et le nom du pilote.

```
Accept nomP Prompt 'Donnez le nom du pilote ' ;
Accept nomV Prompt 'Donnez le nom de la ville ' ;
Select numAvion From Vol MINUS
 (select numAvion from (
                                (select numPilote from Pilote where nomPilote='&nomP')
                                                natural join
                                        (select * from Vol where villeArr='&nomV')
                        )
) ;
```

Les variables externes ou de référence

En SQL*Plus, pour déclarer une variable GLOBALE, on utilise la commande :

VARIABLE nomVarGlobale { NUMBER | CHAR | CHAR(n) | VARCHAR2(n) } ;

Pour l'afficher sous SQL*Plus :

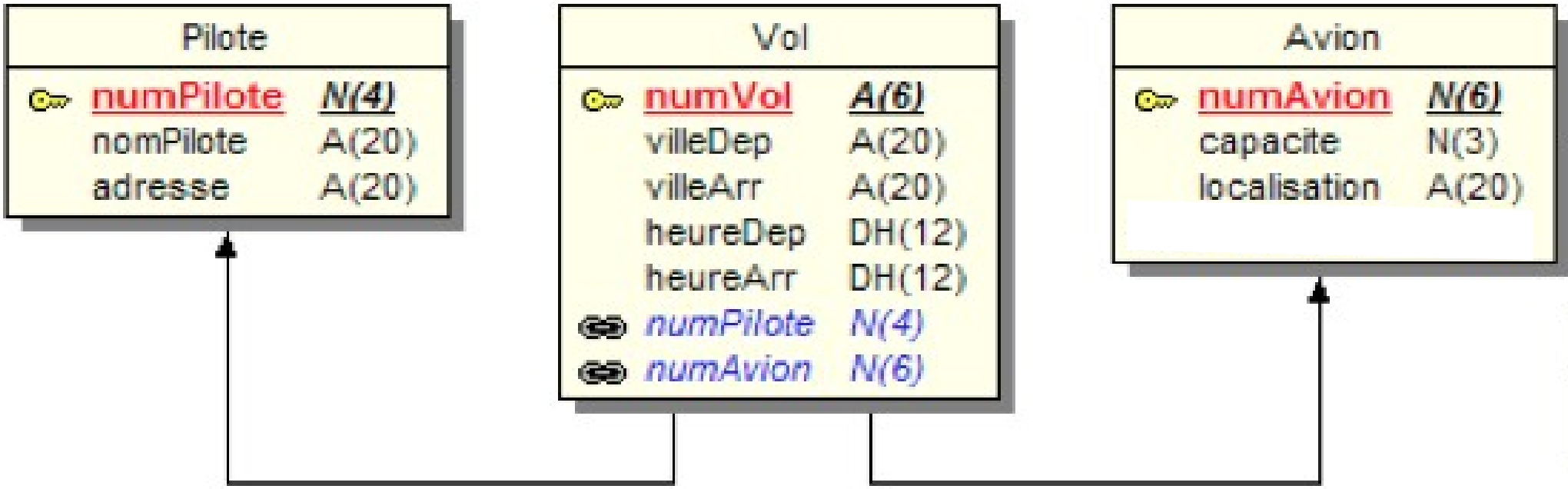
PRINT nomVarGlobale ;

Pour utiliser une variable externe dans un bloc PL/SQL ou dans un ordre SQL, il faut les préfixer par le symbole **:**

Exemple

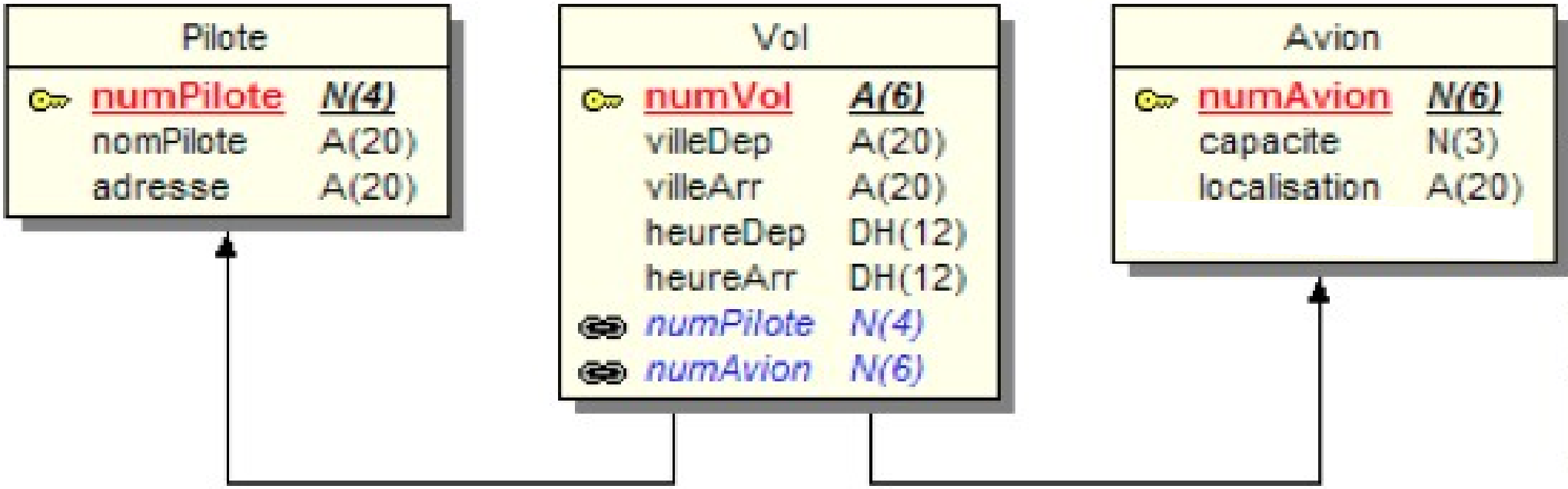
```
accept sal prompt 'Entrez le salaire : ' ;   -- saisi du salaire dans la variable sal

VARIABLE salAnnuel NUMBER ;          -- déclaration de la variable salAnnuel
BEGIN
:salAnnuel := &sal*12;               -- affectation du salaire annuel dans salAnnuel
END;
/
PRINT salAnnuel ;                    -- affichage du salaire annuel
```

Les variables externes



Ecrire un programme qui lit une capacité et lui rajoute 20 % puis l'affiche

Les variables externes

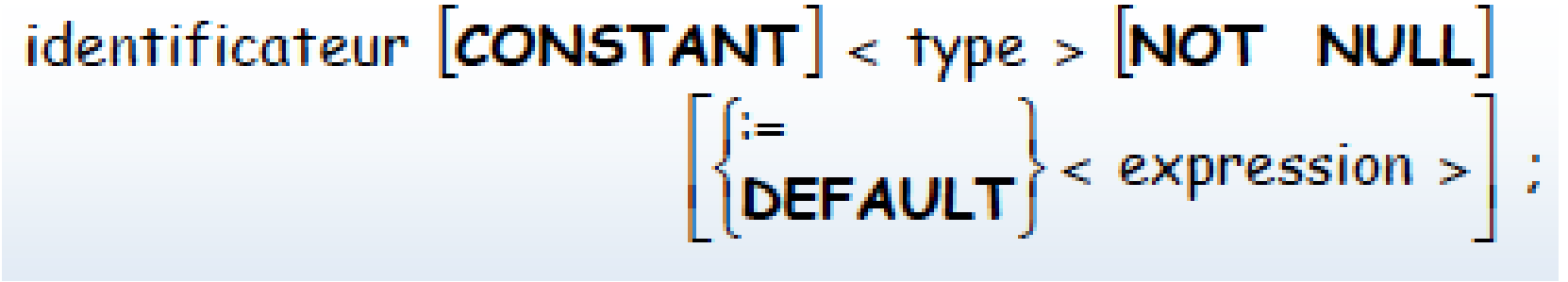


Ecrire un programme qui lit une capacité et lui rajoute 20 % puis l'affiche

```
accept cap prompt 'Entrez la capacite : ' ;   -- saisi

VARIABLE capacitePlus20 NUMBER ;        -- déclaration de la variable
BEGIN
:capacitePlus20 := &cap*1.20;                    -- affectation
END;
/
PRINT capacitePlus20 ;                          -- affichage
```

Les variables scalaires ou structurées

Ce sont des variables locales qui sont définies dans la section DECLARE



Exemple :

```
DECLARE
      var1 Number;
      var2 Number :=2;
```

Attention : un seul identificateur par ligne :

```
DECLARE
var1, var2 Number;
```
ERREUR !!!!

```
DECLARE
var1 Number;
var2 Number;
```
Bonne syntaxe

Type de données

NUMBER[(n[, m])] : pour les nombres fixes et à virgule flottante

n : précision et m : échelle.
Longueur maxi = 38.

CHAR (n) : chaîne de caractères de longueur fixe n.
n est compris entre 1 (valeur par défaut) et 2000 (en Oracle8i)

VARCHAR2 (n) : chaîne de longueur variable n.
n est compris entre 1 et 4000 (en Oracle 8i)
n doit toujours être spécifié (pas de valeur par défaut)

DATE : dates et heures

Type de données : BOOLEAN

Il est utilisé pour stocker l'une des 3 valeurs possibles pour un calcul logique :
TRUE, FALSE et UNKNOWN (absence de valeur).

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

NOT (TRUE)	NOT (FALSE)	NOT (UNKNOWN)
FALSE	TRUE	UNKNOWN

Type de données : BINARY_INTEGER

Ce type permet de mémoriser des données entières signées mais contrairement au type NUMBER, les données de ce type ne nécessitent aucune conversion pour les calculs, d'où des performances plus élevées.

Il est utilisé notamment pour les indices de boucle ou de tableau

Exemple d'un bloc DECLARE avec plusieurs variables et initialisations :

```
DECLARE

    Reponse CHAR(1);

    compteur BINARY_INTEGER := 0;

    dateValide DATE := SYSDATE + 7;   -- date du systeme plus 7 jours

    total NUMBER(9,2) := 0;
```

L'attribut %TYPE

Syntaxe 1 : nomVariable nomTable.nomColonne%TYPE ;

La variable nomVariable aura le même type que la colonne nomColonne de la table nomTable.

Exemple :

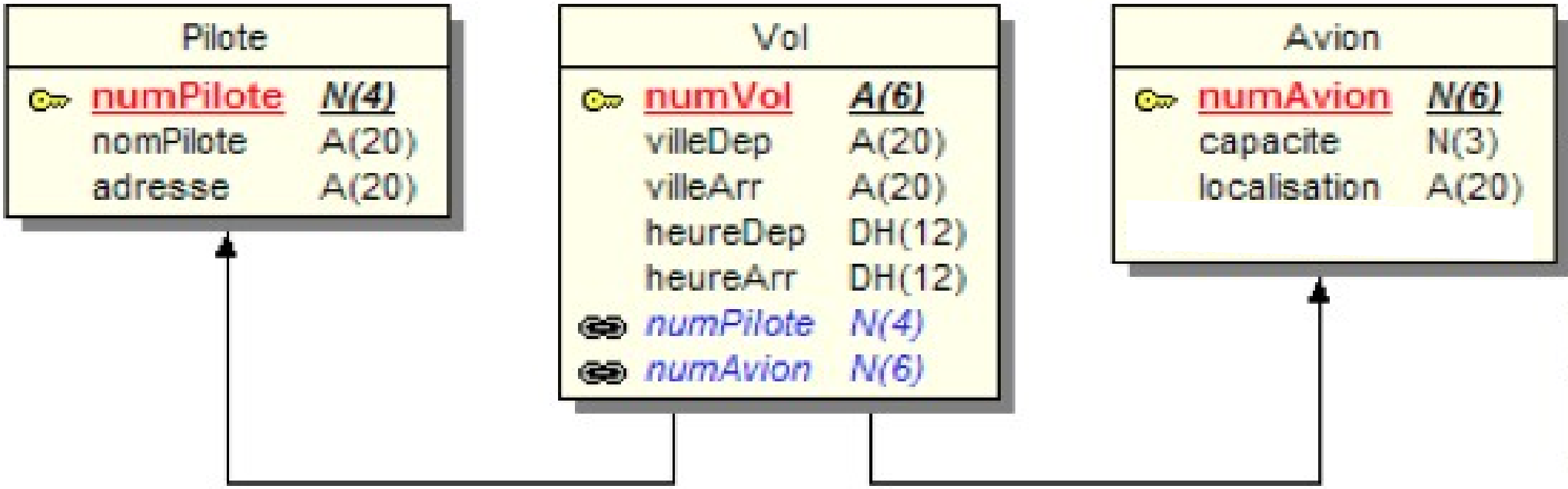
```
DECLARE

    Id Simple.ident%TYPE; -- Id a le même type que la colonne ident de la table Simple
```

Syntaxe 2 : nomVariable2 nomVariable1%TYPE ;

La variable nomVariable2 aura le même type que la variable nomVariable1

```
DECLARE

    salaire NUMBER(7,2);
    commission salaire%TYPE;  -- commission a le même type que salaire
    prime salaire%TYPE := 1000;  -- prime a le même de type que salaire et contient 1000
```

Les variables externes

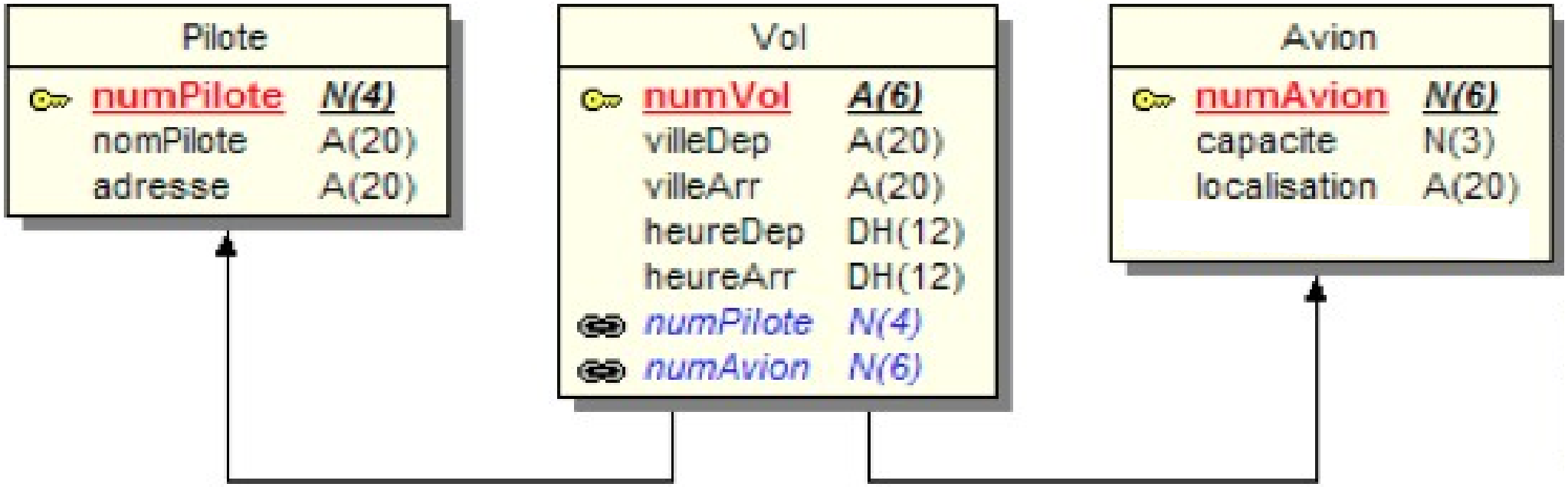


Lire le numéro d'un pilote dans une variable de substitution subP puis déclarez :

- une variable local P qui est du même type que numPilote de la table Vol et initialisée à 0
- une variable local PiloteNN qui est du même type que P et initialisée à 100
- une variable externe Addition qui contient la somme des trois variables ci-dessus

Enfin affichez Addition

Les variables externes




```
accept cap prompt 'Entrez la variable subP : ' ;   -- saisi var de subst
VARIABLE addition NUMBER ;        -- déclaration de la variable externe
DECLARE
      P VOL.numPilote%TYPE:=0 ;
      PiloteNN P%TYPE:=100 ;
BEGIN
:addition := &cap+PiloteNN+P ;   – notez l'absence des ''autour de &cap car nombre
END;
/
PRINT addition ;                        -- affichage
```

Les variable de type structuré : table

Il ne faut pas confondre une table PL/SQL (un tableau à une dimension) avec une table de base de données relationnelle.

Une table PL/SQL est un objet composé de deux éléments : une clé primaire de type BINARY_INTEGER, une colonne de type scalaire (number,...).

La déclaration d'une variable de type TABLE se fait en deux étapes :

Etape 1 : définir un type TABLE PL/SQL

TYPE nomTypeTable IS TABLE OF typeScalaire [NOT NULL] INDEX BY BINARY_INTEGER ;

Etape 2 : déclarer une variable de ce type de données.

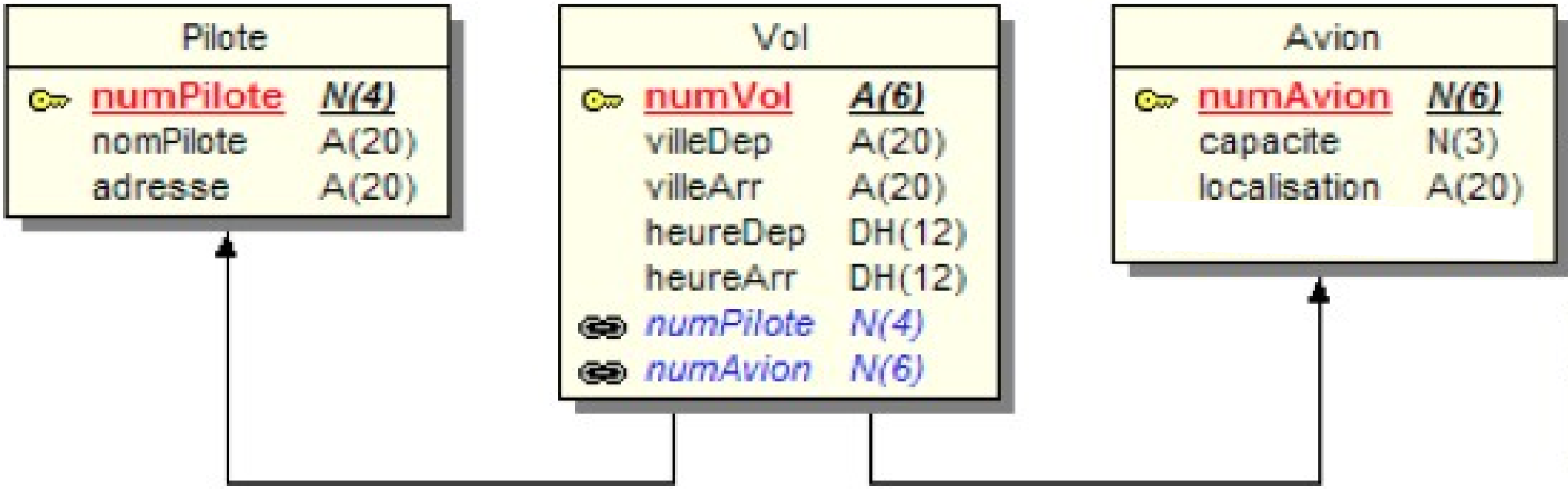
maTablePL nomTypeTable ;

Les variable de type structuré : table

```
DECLARE

    TYPE monType IS TABLE OF VARCHAR2(25) INDEX BY BINARY_INTEGER;

    -- monType est un type de table de chaines de caractères


    table1 monType;

    -- table1 est un tableau de type monType
```

L'accès à ses cases se fait par table1(num_case):= valeur ;

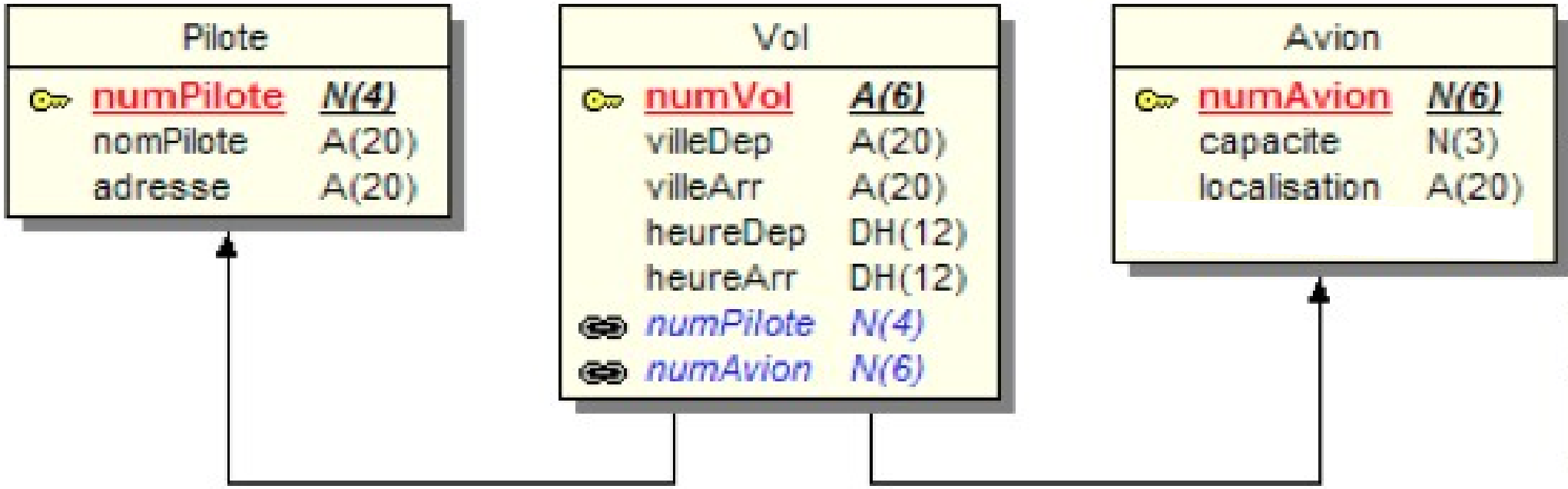
Les tableaux



Définir un tableau dans lequel on mettra les numéro de pilote. Vous devez donc définir le même type que celui des numéros de pilote en utilisant%TYPE

Affecter à la première case de ce tableau la valeur 12

Les tableaux



Définir un tableau dans lequel on mettra les numéro de pilote
Affecter à la première case de ce tableau la valeur 12

```
declare
      type tb is table of Pilote.numPilote%type index by binary_integer;
      T tb;
begin
      T(0):=12;
end;
/
```

type de la colonne numPilote de la table Pilote

Les variable de type structuré : enregistrement (record)

Comme pour les type « table »il faut d'abord créer le type puis déclarer une variable de ce type.

```
TYPE nomTypeRecord IS RECORD
    ( nomChamp1 sonType [ NOT NULL ]
    [, nomChamp2 sonType [ NOT NULL ] ]…
    ) ;
```

déclaration d'un type de record nommé nomTypeRecord

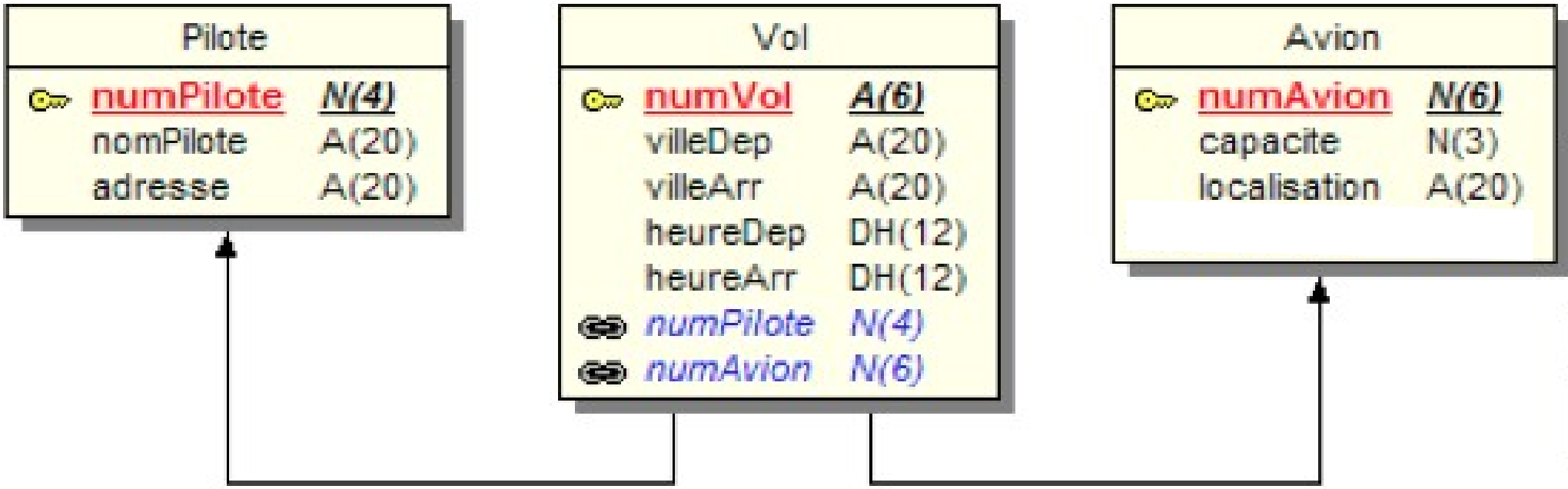
```
MonRecord nomTypeRecord ;
```

déclaration d'un record de type MonRecord

Exemple :

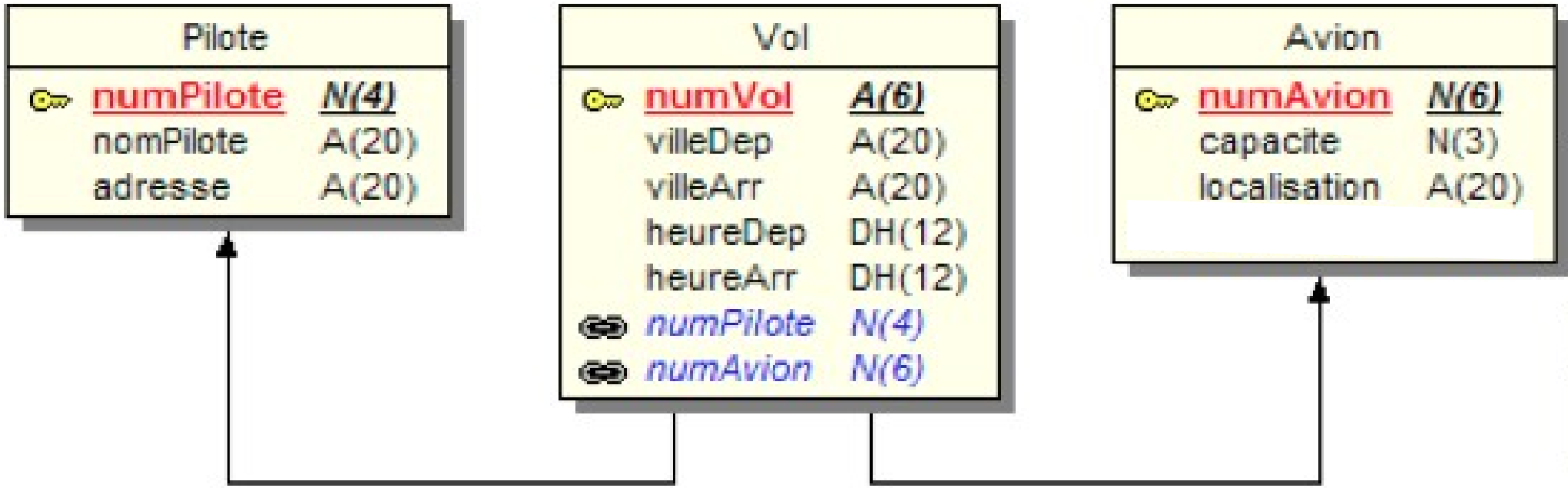
```
DECLARE
    Type Personne is record
        (  id number not null
        ,  nom varchar2(32)
        ,  prenom varchar2(32)
        ,  dateNaiss date
        ) ;
    MonEnregistrement Personne ; -- on peut accéder à MonEnregistrement.id,
                                 --   MonEnregistrement.nom, etc
```

Les enregistrement



Définir un enregistrement monPilote(nomP,adresseP) puis lui affecter Dupont et Nice. On utiisera%TYPE pour la cohérence des types.

Les enregistrement



Définir un enregistrement monPilote(nomP,adresseP) puis lui affecter Dupont et Nice

```
declare
      type Enreg is record ( nomP PILOTE.nomPilote%TYPE,
                             adresseP PILOTE.adresse%TYPE);
      monPilote Enreg;
begin
       monPilote.nomP:='dupont';
       monPilote.adresseP:='nice';
end;
/
```

L’attribut %ROWTYPE

nomVariable nomTable%ROWTYPE ;

La variable nomVariable contient un enregistrement dans lequel chaque cellule contient une colonne de la table et sera accessible via nomVariable.nomColonne. En d'autres termes c'est une ligne de la table nomTable sous forme d'un enregistrement.

Exemple :

Soit la table Facture(id, nom,...).

```
DECLARE
    LigneFact Facture%ROWTYPE;
```

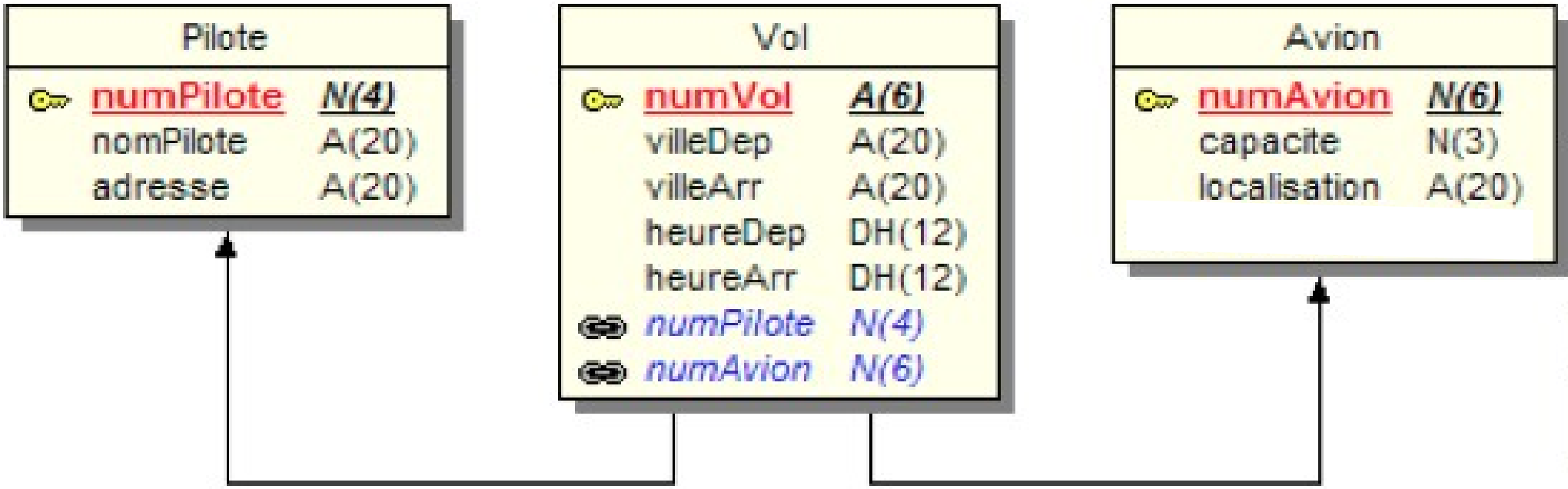
La variable LigneFact est un enregistrement qui contient :

LigneFact.id,

LigneFact.nom,

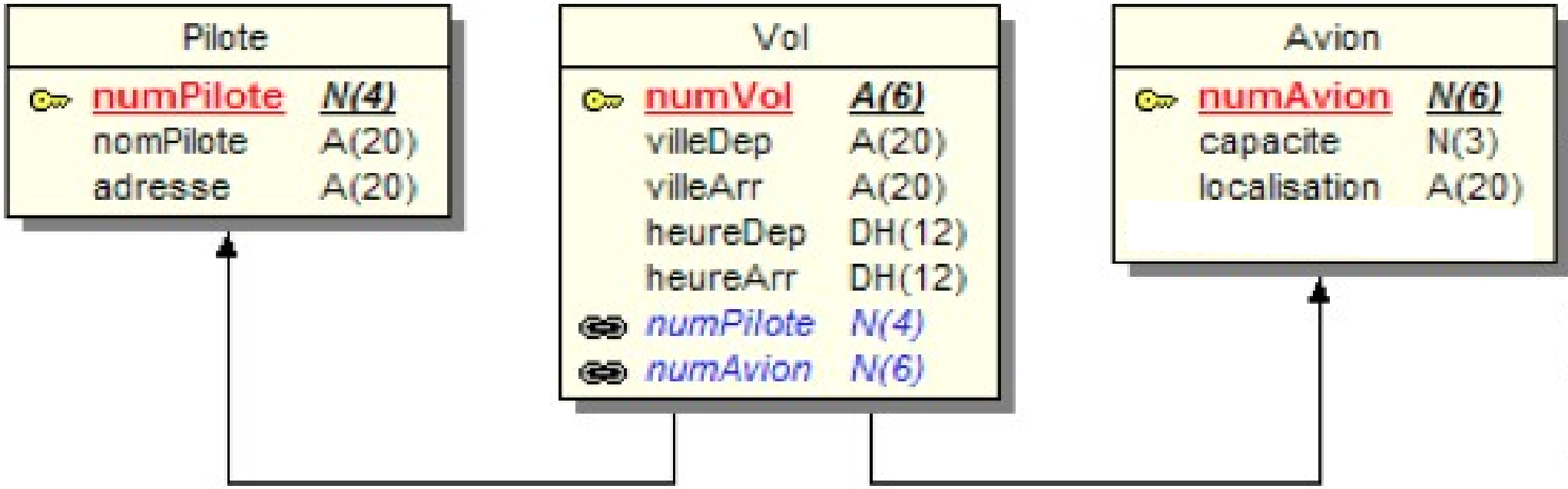
etc...

L'attribut %ROWTYPE



Reprendre l'exemple précédent mais sans définir un enregistrement mais en utilisant directement Rowtype pour avoir une variable monPilote(nomP,adresseP) puis lui affecter Dupont et Nice

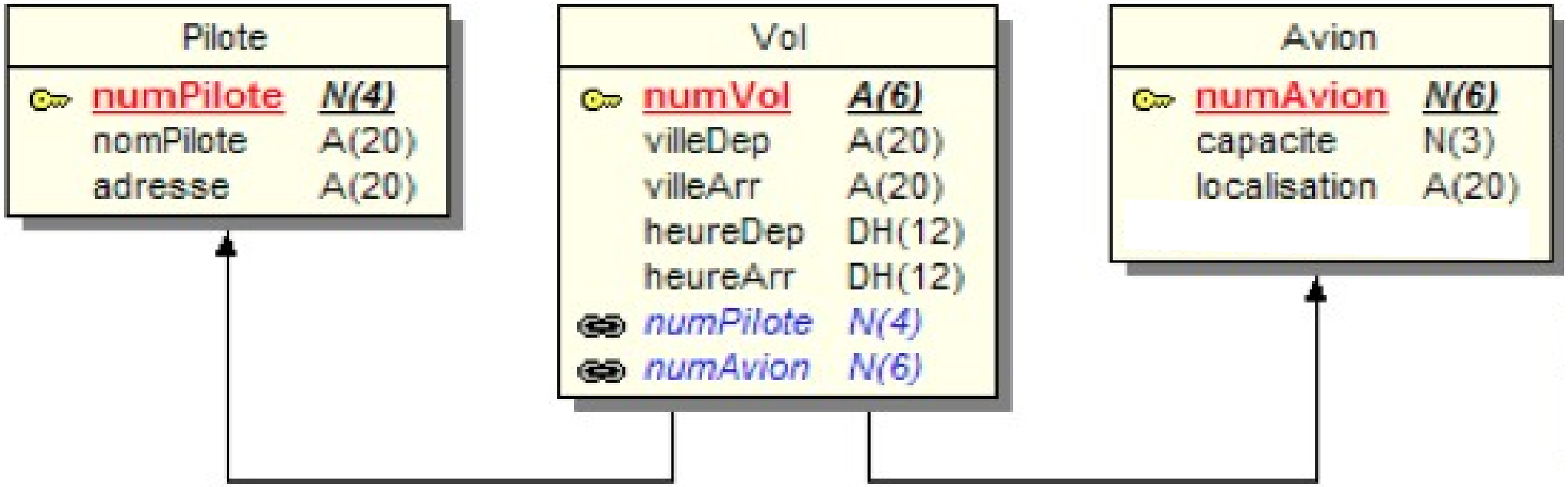
L'attribut %ROWTYPE



Reprendre l'exercice des RECORD mais sans définir un enregistrement mais en utilisant directement ROWTYPE pour avoir une variable monPilote à laquelle on peut lui affecter 1, Dupont et Nice.

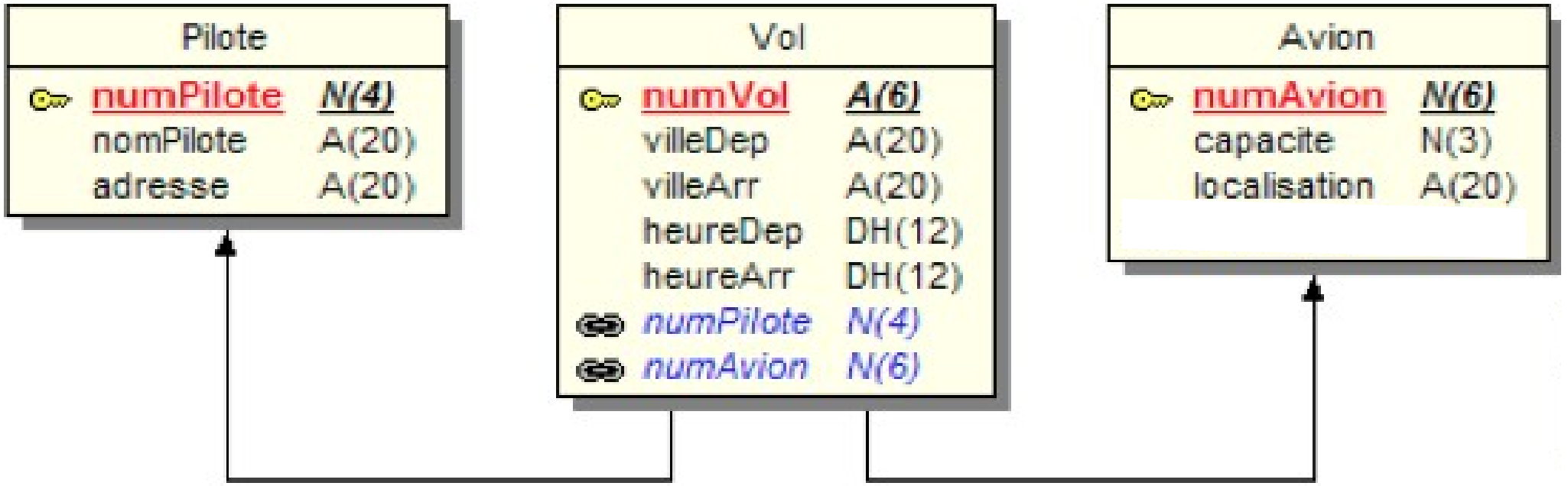
```
declare
      monPilote PILOTE%ROWTYPE ;
begin
       monPilote.numPilote:=1 ;
       monPilote.nomPilote:='dupont';
       monPilote.adresse:='nice';
end;
/
```

Les variables externes



Lire un numéro d'avion, une capacité et une localisation puis l'affecter dans une variable locale de type ROWTYPE

Les variables externes




```
accept num prompt 'Entrez le num avion : ' ;   -- saisi var de subst
accept cap prompt 'Entrez la capacité : ' ;   -- saisi var de subst
accept loc prompt 'Entrez la localisation : ' ;   -- saisi var de subst
DECLARE
      monAvion AVION%ROWTYPE ;
BEGIN
monAvion.numAvion :=&num ;
monAvion.capacite:=&cap ;
monAvion.localisation:='&loc' ;   - notez les ''car chaine de caractère dans localisation
END;
/
```

Affectation/initialisation des variables :=

nomVariable := <expr> ;

nomTABLEPLSQL (indice) := <expr> ;

nomRECORD.nomDuChamp := <expr> ;

Exenple :

```
Begin
    compteur := compteur + 1;
    nom := UPPER(nom);      -- fonction de mise en MAJ
    res := sin(angle);      -- fonction maths
    enreg.emploi := 'FORMATEUR'; --affectation d'une chaine dans un enregistrement
```

L'instruction SELECT en PL/SQL

SELECT <expr1> {, <expr2>}… INTO var1 {, var2}…

FROM nomTable[, nomTable]…

[WHERE <condition>] ;

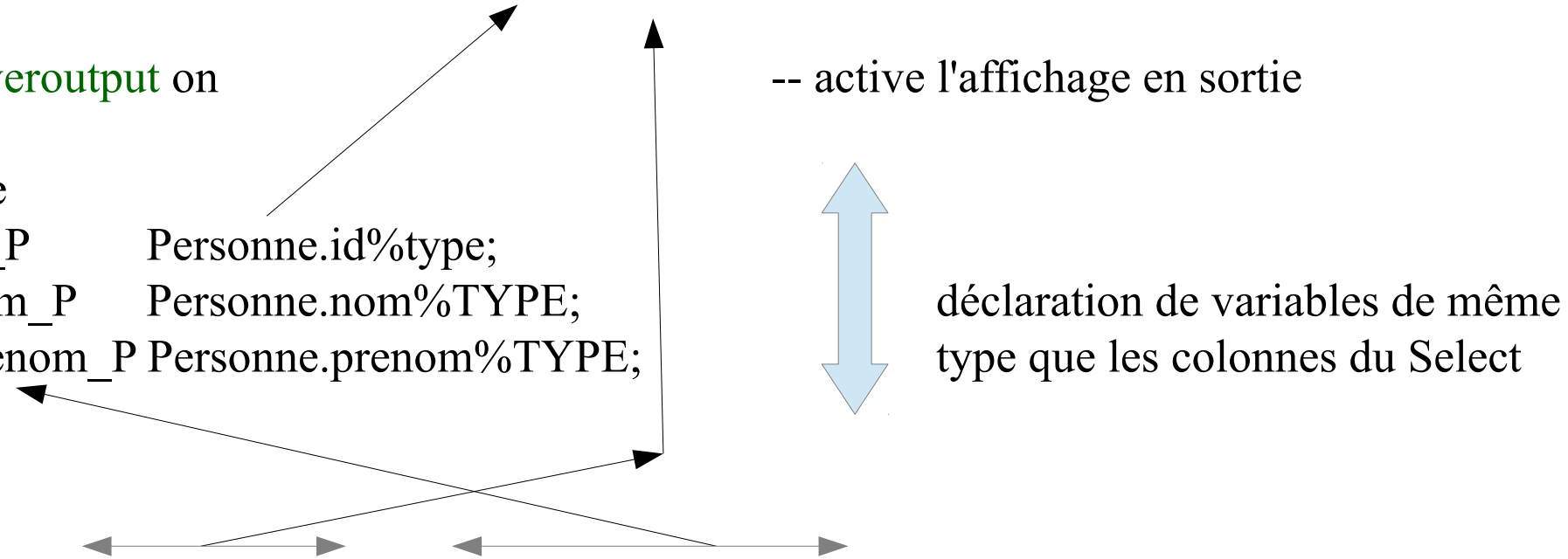
Ne pas oublier la clause INTO (différence avec SQL) ;

Le SELECT doit obligatoirement ne ramener qu'une ligne et une seule, sinon c'est un cas d'erreur.

Les types de var1 et var2 doivent être compatibles avec ceux de expr1 et expr2.

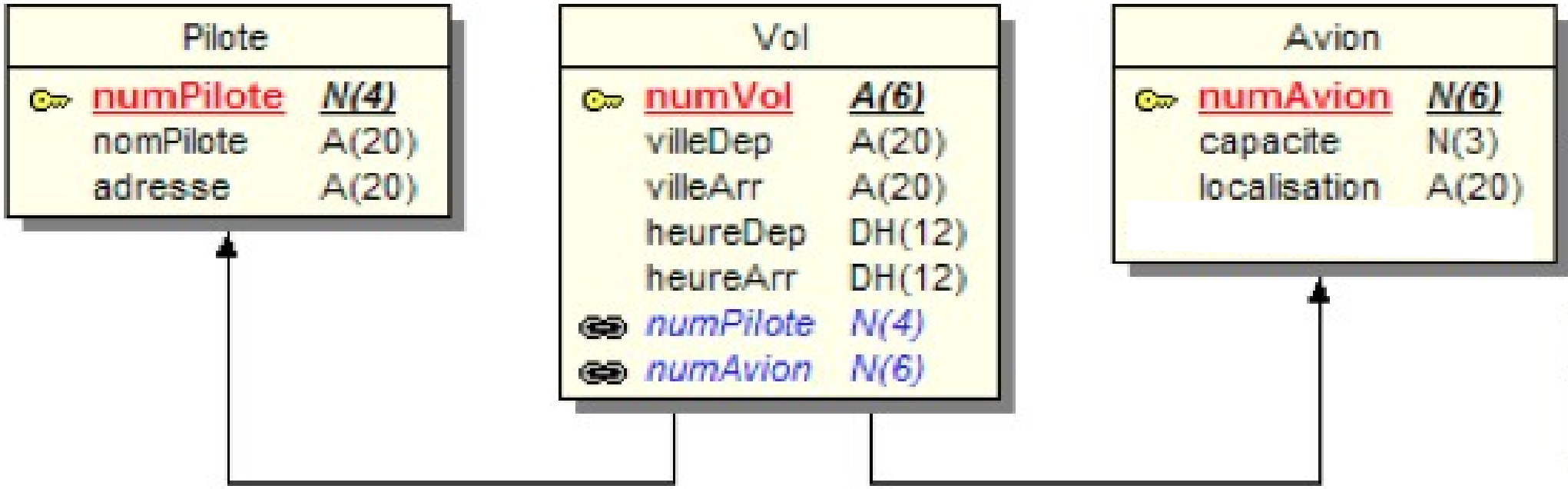
Exemple d'un SELECT en PL/SQL

On reprend la table Personne(id,nom,prenom,…) et on affiche la personne dont l'id est 1.



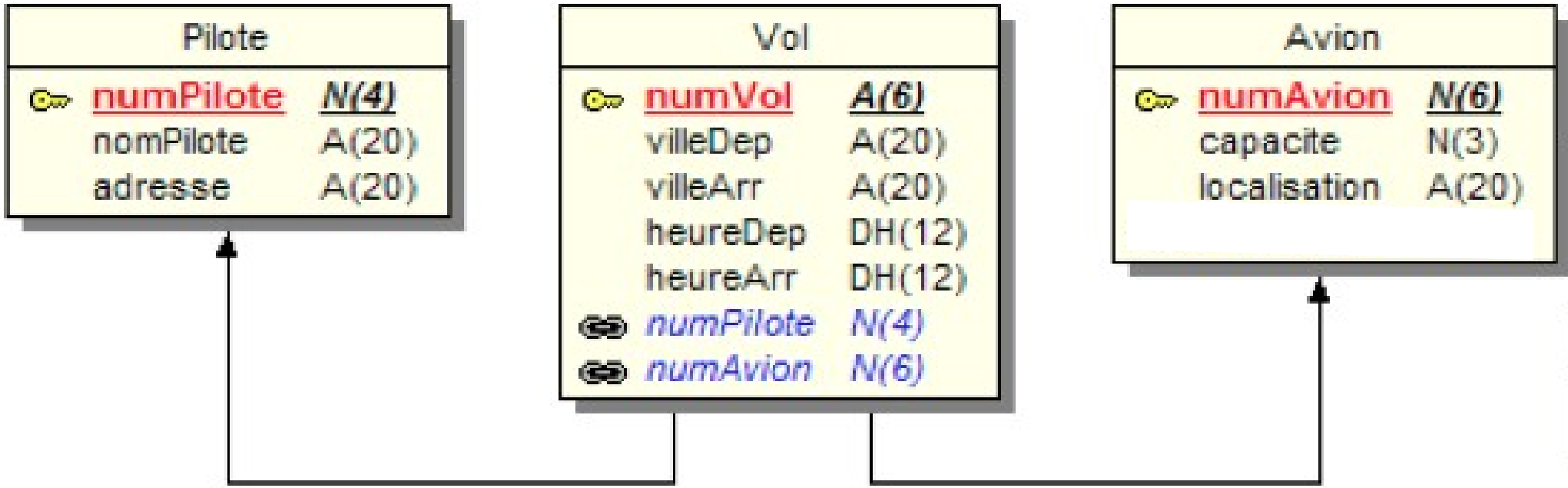
```
set serveroutput on                          -- active l'affichage en sortie

Declare
      id_P        Personne.id%type;
      nom_P       Personne.nom%TYPE;         déclaration de variables de même
      prenom_P Personne.prenom%TYPE;         type que les colonnes du Select

begin

      Select id,nom,prenom into id_P, nom_P, prenom_P from Personne where id=1 ;

      dbms_output.put_line('ID de la personne :'   || id_P);
       -- affichage avec saut de ligne de la chaine 'ID de la personne :' puis concaténation
       -- avec la valeur de variable id_P via ||

      dbms_output.put_line('Nom de la personne : '  || nom_P);
      dbms_output.put_line('Prenom de la personne : '  || prenom_P);

end;
/
```

L'instruction SELECT en PL/SQL



Afficher : lire un numéro de pilote (par exemple 10) puis afficher la phrase :
Le numéro 10 correspond à Dupont

L'instruction SELECT en PL/SQL



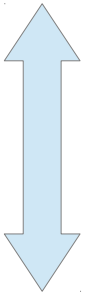
Afficher : lire un numéro de pilote (par exemple 10) puis afficher la phrase :
Le numéro 10 correspond à Dupont

```
accept numP prompt 'donner pilote';  -- pour lire un numéro de pilote dans numP
declare
   nomP Pilote.nomPilote%type;  -- crée nomP de même type que la col nomPilote
begin
 select nomPilote into nomP from Pilote where numPilote=&numP;
 dbms_output.put_line('le numéro  '|| &numP || ' correspond à : ' || nomP);
end;
/
```

Même programme via un enregistrement

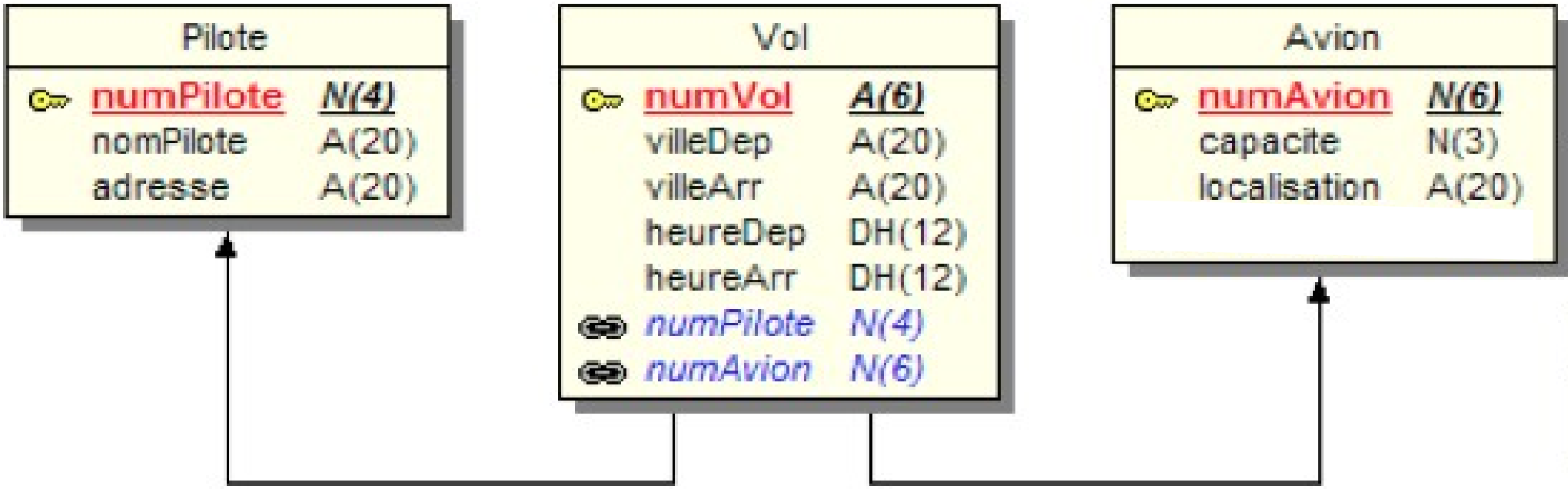
On reprend la table Personne(id,nom,prenom,…).

```
set serveroutput on

Declare

    une_Personne  Personne%ROWTYPE;

begin

    Select * into une_Personne from Personne where id=1 ;

    dbms_output.put_line('ID de la personne :'   || une_Personne.id );

    dbms_output.put_line('Nom de la personne : '  || une_Personne.nom );

    dbms_output.put_line('Prenom de la personne : '  || une_Personne.prenom);

end;
/
```



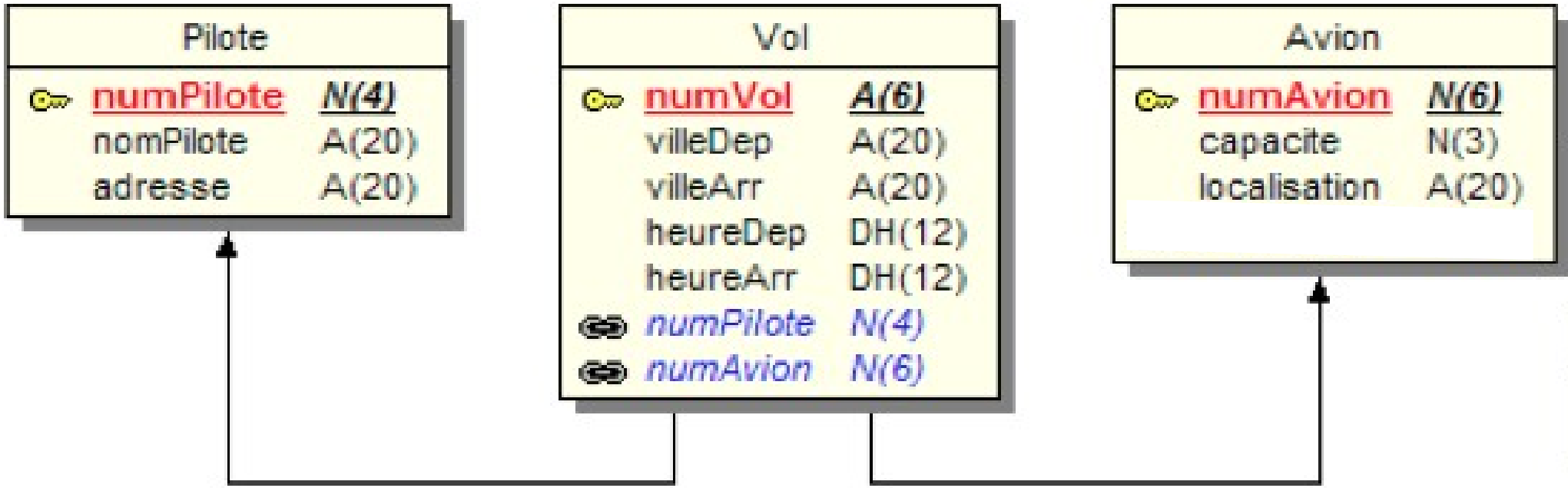
déclaration d'un enregistrement
sur toutes les colonnes de Personne

L'instruction SELECT en PL/SQL



Reprendre l'exercice précédent mais avec un ROWTYPE
Afficher : lire un numéro de pilote (par exemple 10) puis afficher la phrase :
Le numéro 10 correspond à Dupont

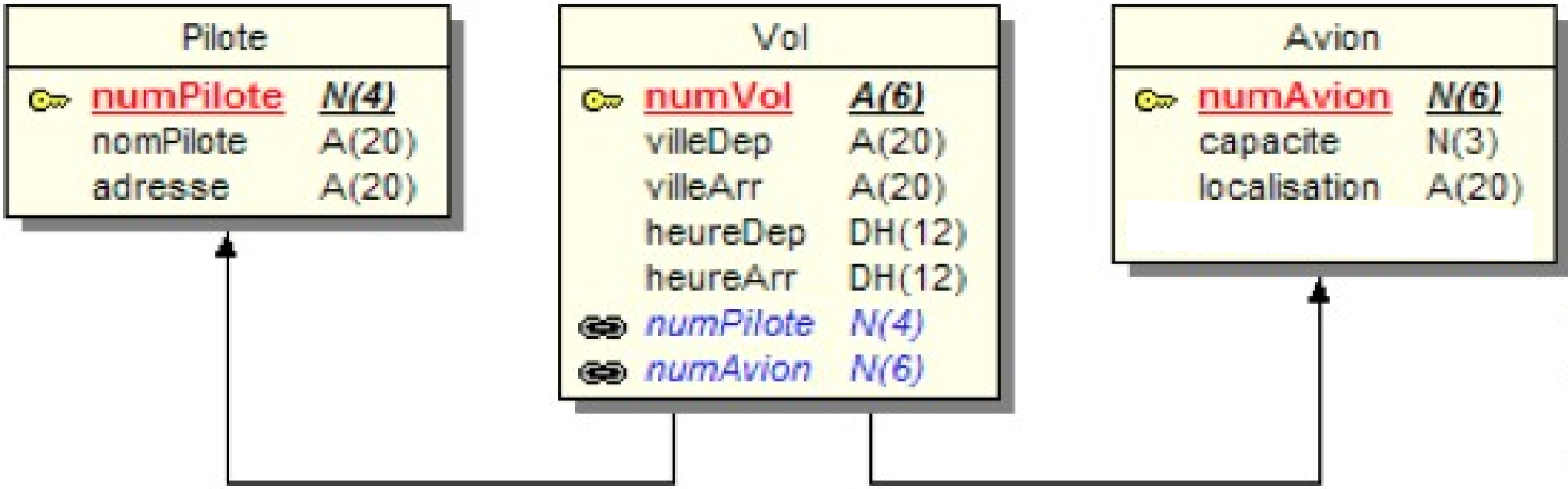
L'instruction SELECT en PL/SQL



Afficher : lire un numéro de pilote (par exemple 10) puis afficher la phrase :
Le numéro 10 correspond à Dupont

```
accept numP prompt 'donner pilote';  -- pour lire un numéro de pilote dans numP
declare
   monPilote Pilote%rowtype;  -- crée monPilote un enregistrement de ligne sur Pilote
begin
 select *  into    monPilote from Pilote where numPilote=&numP;
 dbms_output.put_line('le numéro  '|| &numP || ' correspond à : ' || monPilote.nomPilote);
end;
/
```

L'instruction SELECT en PL/SQL



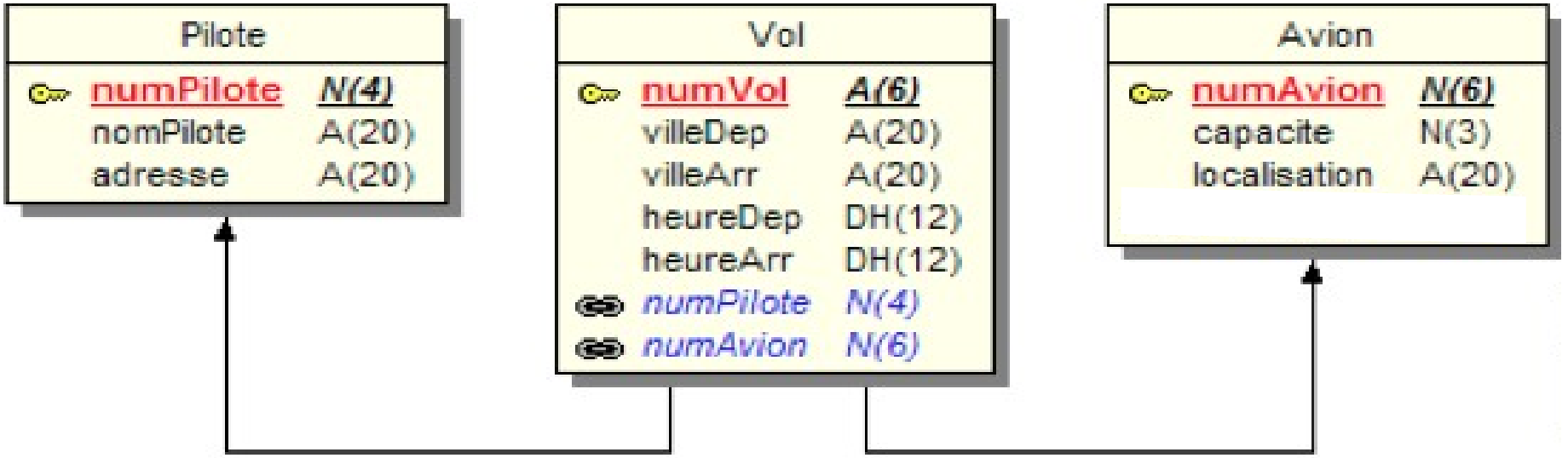
Lire un numpilote X et une ville Y puis trouver où habite X et enfin afficher la phrase :

Le [nom du pilote X] qui habite [adresse de X] a fait Z vols vers la ville Y.

Par exemple :

Le pilote Dupont qui habite Orleans a fait 5 vols vers la ville Nice.

L'instruction SELECT en PL/SQL



Lire un numpilote X et une ville Y puis trouver où habite X et enfin afficher la phrase :
Le [nom du pilote X] qui habite [adresse de X] a fait Z vols vers la ville Y.

```
accept numP prompt 'donner pilote';  -- pour lire un numéro de pilote dans numP
accept nomV prompt 'donner une ville';  -- pour lire un nom  de ville numV
declare
   monPilote Pilote%rowtype;  -- crée un enregistrement monPilote de ligne sur Pilote
   nbVol number ;
begin
 select count(numVol) into nbVol from Vol where numPilote=&numP and villeArr='&nomV' ;
 Select *  into monPilote from Pilote where numPilote=&numP ;
 dbms_output.put_line('le pilote  '|| monPilote.nomPilote  || ' qui habite  ' ||
                          monPilote.adresse || ' a fait   ' || nbVol || ' vols vers' || '&nomV ' );
end;
/
```

La condition IF

IF <condition1> THEN

une ou plusieurs instructions

[ELSIF <condition2> THEN

une ou plusieurs instructions]…

[ELSE

une ou plusieurs instructions]

END IF ;

Les opérateurs utilisés dans les conditions sont les mêmes que dans SQL :

= ,<, >, !=, >=, ≤

IS NULL, IS NOT NULL, BETWEEN,

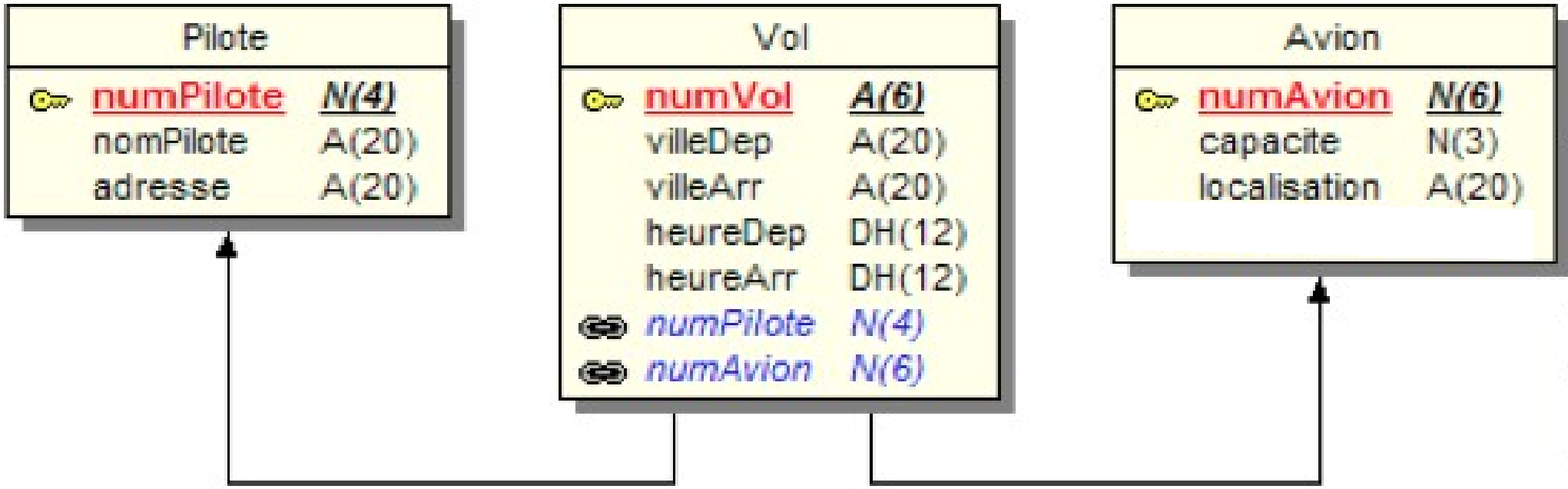
LIKE, AND, OR, etc.

La condition IF

On reprend la table Personne(id,nom,prenom,salaire) et on effectue des testes selon le salaire de l'employé id=1

```
DECLARE
      salaire_P   Personne.salaire%type;
      mes CHAR(30);
BEGIN

SELECT salaire INTO salaire_P FROM Personne WHERE id=1;

IF salaire_P IS NULL THEN
      mes := ' pas de salaire ';

ELSIF salaire_P <1200  THEN
      Mes := ' sous payé ';

ELSE
      mes :=' correctement payé ';
END IF;
dbms_output.put_line(mes);
END;
/
```

La condition IF



Pour le pilote numéro 1 :

afficher ‘Senior’ s’il a fait plus de 50 vols

Afficher ‘Confirmé’ s’il a fait entre 20 et 50 vols

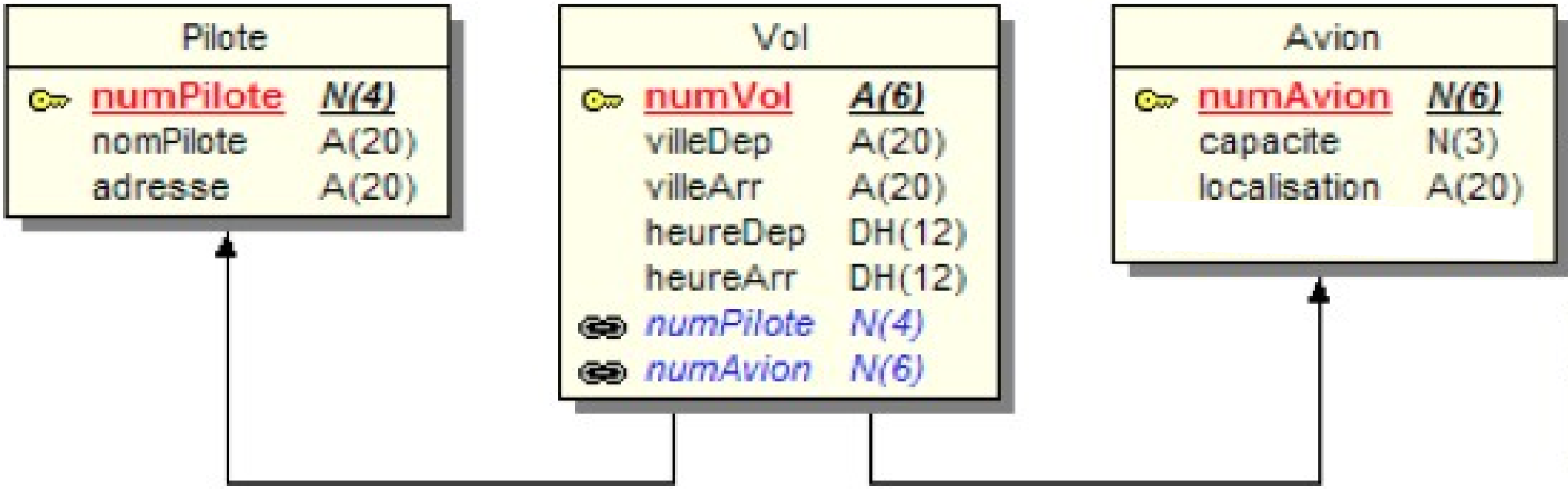
Afficher ‘débutant’ s’il a fait moins de 20 vols.

s’il a fait aucun vol on affiche : en attente de vol

La condition IF

```
DECLARE
      nb number ;
      mes varchar(32) ;
BEGIN

SELECT count(*) INTO nb from Vol where numPilote=1 ;

IF nb=0 THEN
      mes := ' en attente de vol ';

ELSIF nb<20 THEN
      mes := ' debutant ';

ELSIF nb<50  THEN
      mes := 'confirme';

ELSE
      mes :=' senior ';
END IF;
dbms_output.put_line(mes);
END;
/
```

La condition IF

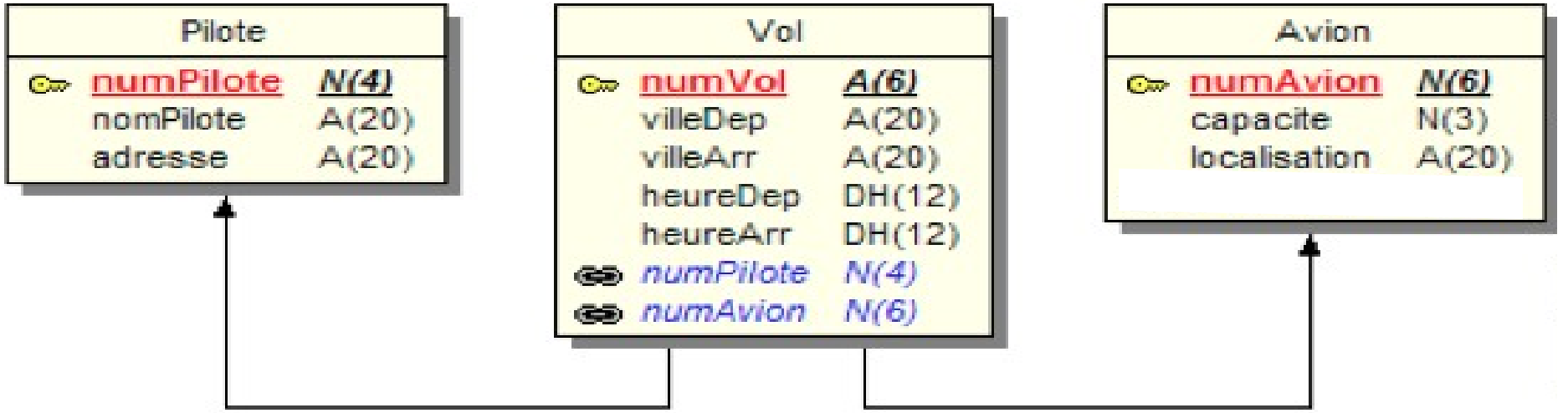


Y a t il de pilotes sans adresse ? (réponse oui ou non)

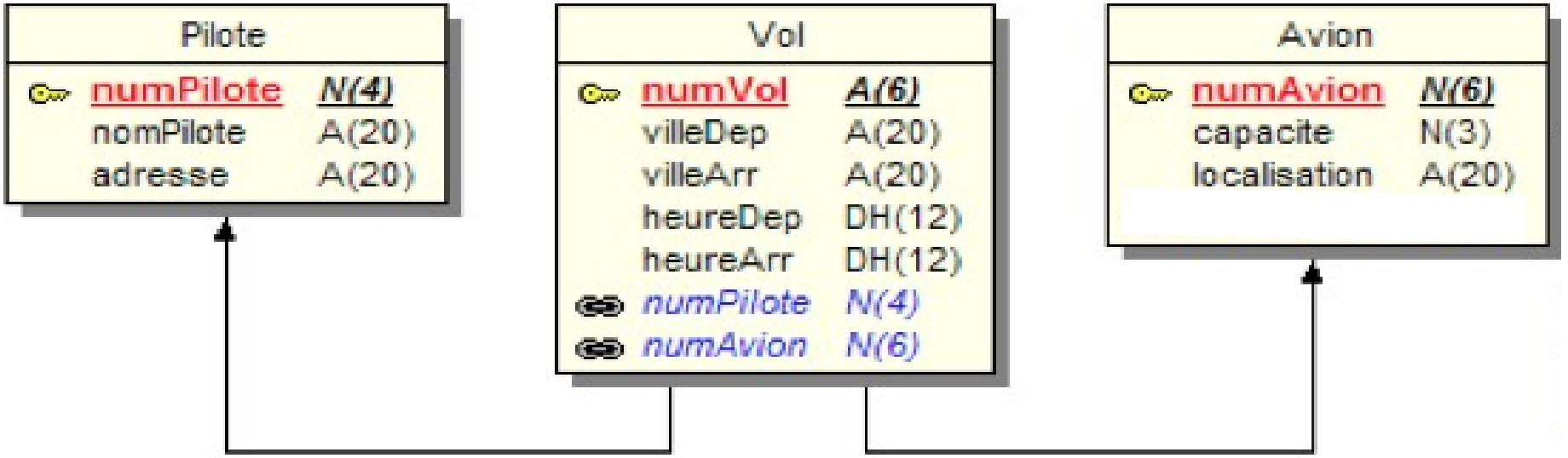
La condition IF

Y a t il de pilotes sans adresse ? (réponse oui ou non)

```
DECLARE
      nb number ;
BEGIN
 SELECT count(*) INTO nb from Pilote where adresse IS NULL ;
 IF nb=0 THEN dbms_output.put_line(‘ non ’);
 ELSE dbms_output.put_line(‘ oui ’);
 END IF ;
END;
/
```



La condition IF



Créez une table piloteNonActif qui contient le numéro et nom du pilote

Lire un numpilote X. Si le pilote n'a fait aucun vol alors le rajouter dans la table piloteNonActif puis écrire :

- soit « pilote inactif, il y a donc au total x pilote inactif » où x est le nombre de pilote inactif

- soit « pilote actif ».

La condition IF

```
Create table piloteNonActif (numPilote number(4) primary key
                                    ,nomPilote varchar2(20)
                                    ) ;
accept numP prompt 'donner pilote';  -- pour lire un numéro de pilote dans numP

declare
   nbVol number ;  -- pour calculer le nb de vol
   NbInActif number ; -- pour calculer le nombre de pilote inactif
   monPilote Pilote%ROWTYPE ;

begin
  select count(numVol) into nbVol from Vol where numPilote=&numP ;
  IF nbVol= 0 THEN
      select * into monPilote from Pilote where numPilote=&numP ;
      insert into piloteNonActif(numPilote,nomPilote)
                values(monPilote.numPilote,monPilote.nomPilote) ;
      select count(*) into nbInActif from piloteNonActif ;
      dbms_output.put_line('Pilote  inactif, il y a donc au total  '|| nbInActif || ' pilotes inactifs') ;
 ELSE
      dbms_output.put_line('Pilote  actif ') ;
 END IF ;
end;
/
```

Les boucles : LOOP

LOOP

une ou plusieurs instructions

EXIT [WHEN CONDITION] ;

une ou plusieurs instructions

END LOOP ;

Fonctionnement : dès que le mot EXIT est trouvé :

- s'il y a un bloc [WHEN CONDITION] :

  - si la condition est vraie on va après le end loop

  - sinon on continue les instructions qui sont après la ligne du EXIT et on remonte au Loop quand on atteint le End Loop

- sinon on va après le end loop

Les boucles : LOOP

Remplissage de la table resultat(nbre) avec des ligne allant de 0 à 10.

Les boucles : LOOP

Remplissage de la table resultat(nbre) avec des ligne allant de 0 à 10.

```
DECLARE
    nbre NUMBER := 0;
BEGIN
    LOOP
        INSERT INTO resultat VALUES (nbre); -- insertion de nbre dans la table résultat
        nbre := nbre + 1;
        EXIT WHEN nbre = 11;     -- quitter la boucle quand nbre=10
    END LOOP;
END;
/
Select * from resultat ;   -- affichage du contenu de la table résultat
```

Les boucles : WHILE

WHILE condition LOOP

une ou plusieurs instructions

END LOOP ;

Les boucles : WHILE

```
WHILE  condition LOOP

   une ou plusieurs instructions

END LOOP ;
```

```
DECLARE
      nbre NUMBER := 0;

BEGIN

      WHILE nbre<=10 LOOP

            INSERT INTO resultat VALUES (nbre); -- insertion de nbre dans la table résultat

            nbre := nbre + 1;

      END LOOP;
END;
/
Select * from resultat ;   -- affichage du contenu de la table résultat
```

Les boucles : FOR

FOR indice IN [REVERSE] <exp1> .. <exp2> LOOP

une ou plusieurs instructions

END LOOP ;

Très simple à combiner avec un curseur (cf. chapitre curseur)

L'indice de la boucle est déclaré binary_integer dans le bloc DECLARE

Le mot REVERSE force un recul de l'indice de la boucle de la valeur exp1 à exp2.

Exemple : for indice in reverse 1..10 loop → indice commence à 10 et termine à 1.

Les boucles : FOR

On reprend l'exemple de la boucle LOOP mais en version FOR.

Les boucles : FOR

On reprend l'exemple de la boucle LOOP mais en version FOR.

On note qu'il nous faut beaucoup moins de ligne pour arriver au même résultat.

```
DECLARE
    nbre BINARY_INTEGER ;   -- l'indice de la boucle est déclaré binary_integer
BEGIN
    FOR nbr in 0..10 LOOP
        INSERT INTO resultat VALUES (nbre); -- insertion de nbre dans la table résultat
    END LOOP;
END;
/
Select * from resultat ;   -- affichage du contenu de la table résultat
```

CURSOR

Les curseurs

Un curseur est une zone de mémoire de taille fixe, utilisée par le noyau d'Oracle pour analyser et interpréter tout ordre SQL. Il en existe deux types : implicite et explicite.

Curseur implicite

Généré et géré par le noyau pour chaque ordre SQL d'un bloc (SELECT qui ramène une ligne et une seule, UPDATE, INSERT, DELETE).

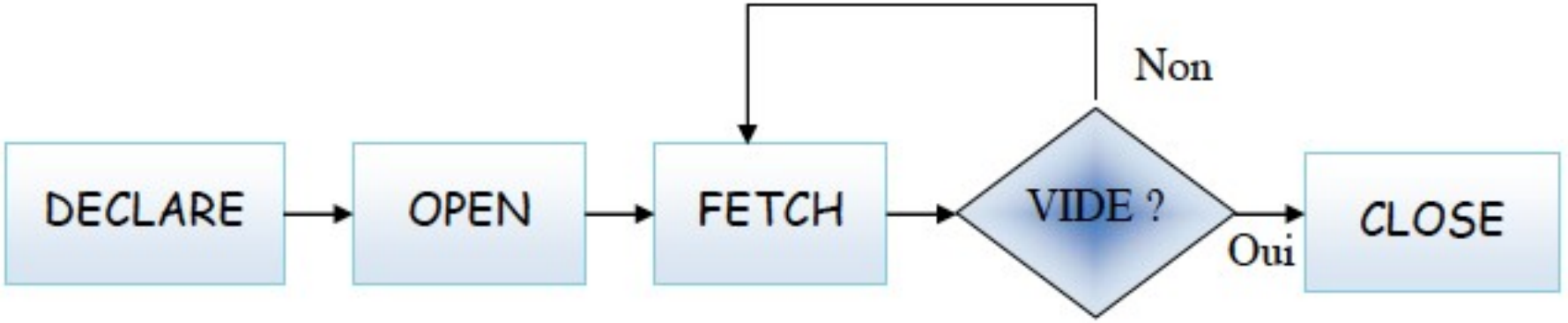
Quand un curseur implicite, dans le cas d'un SELECT, ne ramène pas de lignes (pas de données), l'exception NO_DATA_FOUND est levée (cf. chapitre exception).

Quand le SELECT ramène plus d'une ligne, l'exception TOO_MANY_ROWS est levée.

Le curseur explicite

Curseur SQL généré et géré par l'utilisateur pour traiter un ordre SELECT qui ramène plusieurs lignes.

Le schéma général de traitement d'un curseur est le suivant :



Pour mettre en place ce schéma on utilise au choix :

- soit un passage classique par les instructions OPEN, FETCH, CLOSE

- soit un passage par une forme très courte de la boucle FOR

Le curseur explicite version 1

CURSOR nomCurseur IS ordreSelect ;
Se fait pour déclarer le curseur dans le bloc DECLARE.

Variable_ligne nomCurseur%ROWTYPE ;
Se fait également dans le bloc DECLARE pour déclarer une ligne curseur

OPEN nomCurseur ;
Se fait avant l'entrée en boucle pour ouvrir le curseur.

FETCH nomCurseur INTO Variable_ligne
Se fait dans la boucle pour passer d'une ligne du curseur à une autre

EXIT WHEN nomCurseur%NOTFOUND
Se fait dans la boucle LOOP

CLOSE nomCurseur ;
Se fait après la sortie de la boucle.

Le curseur explicite version 1

Une entreprise dont les employés sont enregistrés dans la table :

Drh(id, nom, prenom, salaire, grade)

souhaite enregistrer dans une table promotion(id, salaire) une éventuelle promotion des employés qui ont le grade 3 selon la règle suivante :

- 30 % si le salaire est inférieur à 1200

- 10 % sinon

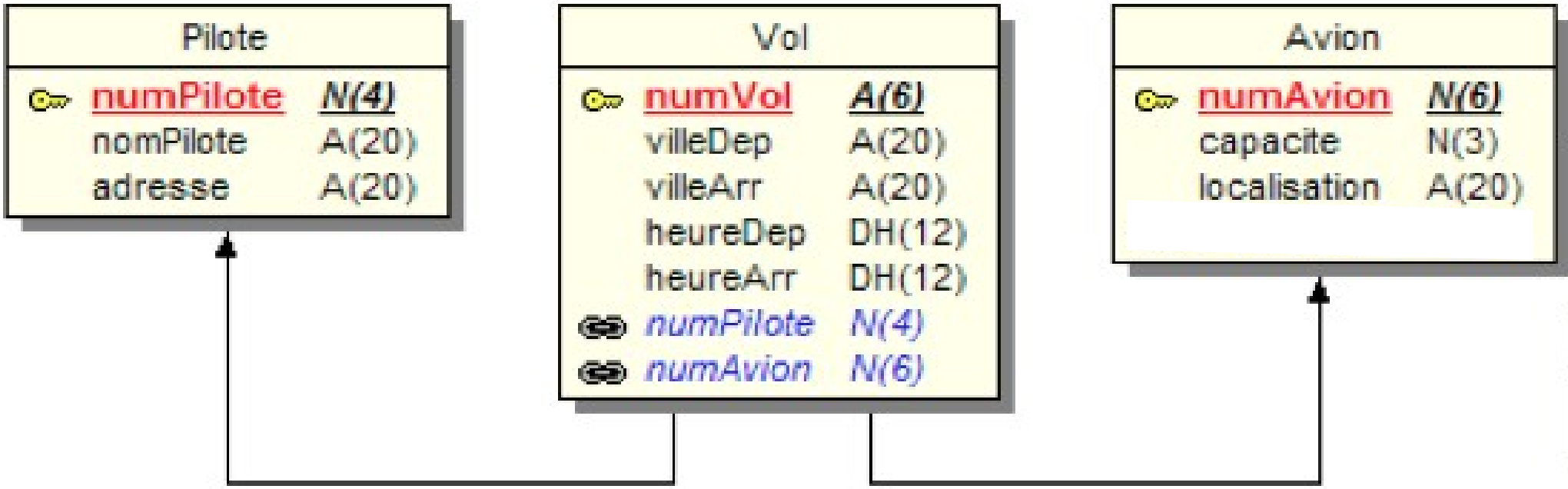
Une fois la table promotion remplie, l'entreprise fera une étude budgétaire puis décidera ou pas de réaliser ces promotions.

Nous allons utiliser un curseur et un parcours par boucle LOOP pour remplir la table des promotions à partir de la table Drh.

Le curseur explicite version 1

```
DECLARE

    CURSOR C1 IS SELECT id,salaire FROM drhWHERE grade=3 ;
    Emp C1%ROWTYPE ;  -- enregistrement qui contient les champs du curseur

BEGIN
     OPEN C1;
     LOOP

          FETCH C1 INTO Emp ;
          EXIT WHEN C1%NOTFOUND;
          IF Emp.salaire < 1200 THEN  Emp.salaire:=Emp.salaire*1.3 ;
          ELSE Emp.salaire:=Emp.salaire*1.1 ;
          END IF ;
          INSERT INTO promotion VALUES (Emp.id, Emp.salaire);

     END LOOP;
     CLOSE C1;
END;
/
```

Le curseur explicite version 1



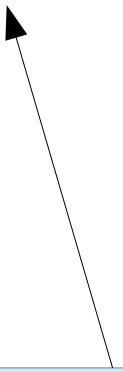
On veut licencier tous les pilotes qui ont fait moins de 10 vols !
Faire un bloc PL/SQL qui pour chaque pilote va afficher :

Pilote numPilote a fait nbVol vols – numPilote est le numéro du pilote et nbVol est le
– nombre de vols de ce pilote

Puis va effacer le pilote de la table Pilote si ce dernier a fait moins de 10 vols.

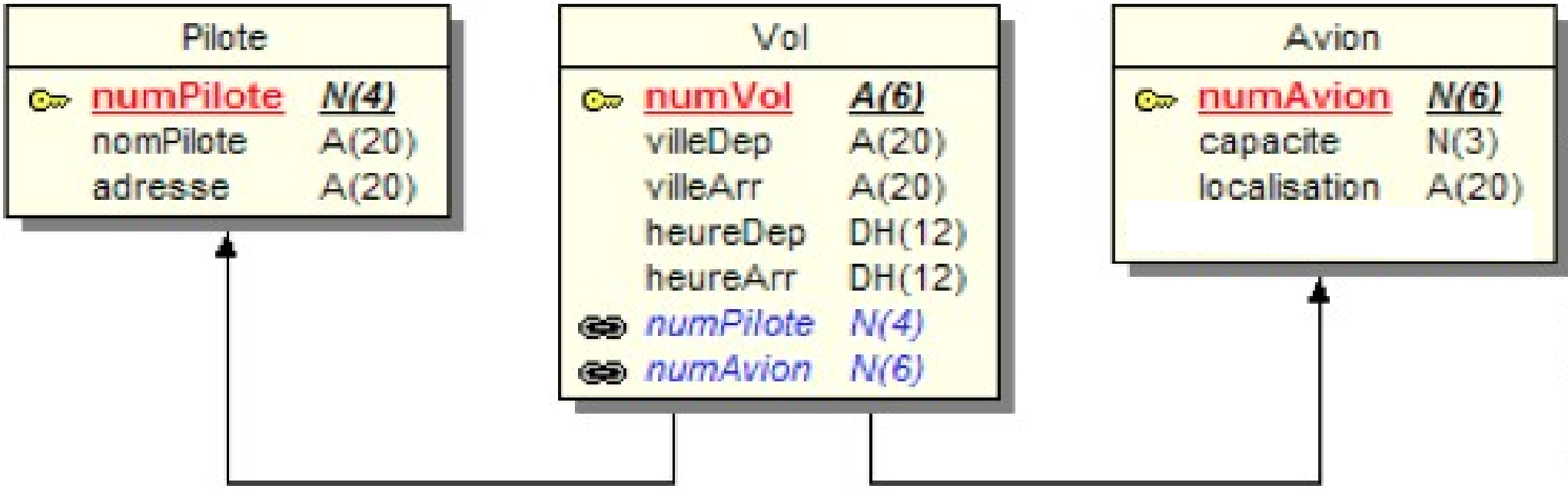
Le curseur explicite version 1

```
Declare
    cursor cx is select * from Pilote;          -- déclaration d'un curseur
    ligne cx%rowtype;                           -- déclaration d'une ligne pour le curseur
    nb number;                                  -- contiendra le nombre de vol par pilote

Begin

    open cx;                                    -- ouverture du curseur
    loop
        fetch cx into ligne;                    -- éclatement du curseur dans la variable ligne
        exit when cx%notfound;
        select count(*) into nb from Vol where numPilote=ligne.numPilote;
        dbms_output.put_line(ligne.numPilote || ' a fait ' || nb || 'vols' );
        if nb<10 then
            delete from Pilote where numPilote=ligne.numPilote;
        end if;
    end loop;

End;
/
```



Calcul du nombre de vol pour chaque pilote du curseur et le mettre dans nb

Le curseur explicite version 1



Pour chaque avion : si l'avion a fait plus de 100 vols à la date d'aujourd'hui alors on va annuler tous les prochaines vols qui sont sur cet avion et supprimer cet avion de la table Avion.

Le curseur explicite version 1

```
Declare
    cursor cx is select * from Avion;          -- déclaration d'un curseur
    ligne cx%rowtype;                          -- déclaration d'une ligne pour le curseur
    nb number;                                 -- contiendra le nombre de vol par avion

Begin

    open cx;                                   -- ouverture du curseur
    loop
        fetch cx into ligne;                   -- éclatement du curseur dans la variable ligne
        exit when cx%notfound;
        select count(*) into nb from Vol where numAvion=ligne.numAvion and
                                               heureDep<=SYSDATE;
        if nb>100 then
            delete from Avion where numAvion=ligne.numAvion;
            delete from Vol where numAvion=ligne.numAvion and  and
                                               heureDep>SYSDATE
        end if;
    end loop;

End;
/
```

L'attribut %FOUND

Avec un curseur implicite : IF SQL%FOUND THEN ...

TRUE (vrai)	INSERT UPDATE DELETE	→ traite au moins une ligne.
	SELECT … INTO	→ ramène une et une seule ligne.

Avec un curseur explicite : IF nomCurseur%FOUND THEN ...

TRUE → Le dernier FETCH a ramené une ligne.

Nombre de ligne déjà traitées : %ROWCOUNT

Avec un curseur implicite : IF SQL%ROWCOUNT =3 THEN ...

INSERT
UPDATE → nombre de lignes traitées
DELETE

SELECT...INTO qui ne ramène aucune ligne → 0 (erreur)
SELECT...INTO qui ramène exactement 1 ligne → 1
SELECT...INTO qui ramène plus d'1 ligne → 2 (erreur)

Avec un curseur explicite : IF nomCurseur%ROWCOUNT=3 THEN

L'attribut %ISOPEN

En curseur explicite il sert à savoir si le curseur est ouvert : IF nomCurseur%ISOPEN THEN

En implicite il est est toujours à FALSE.

Le curseur explicite version 2 (la plus simple)

CURSOR nomCurseur IS ordreSelect ;
Se fait pour déclarer le curseur dans le bloc DECLARE.

FOR Une_variable in nomCurseur loop

On travaille directement avec Une_Variable.nom_un_champ_du_curseur

End Loop ;

Pas besoins de gérer les : Open, Fetch, Close, **ni même de déclarer la variable Une_variable !**

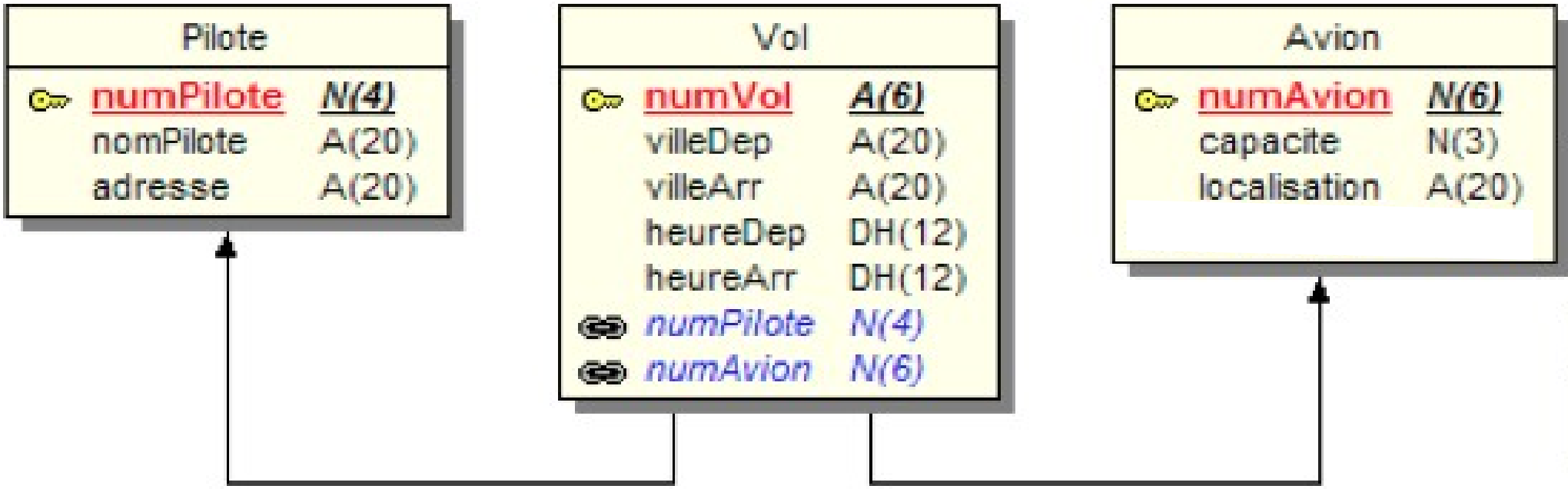
ATTENTION : on ne peut pas utiliser nomCurseur%ROWCOUNT après le end loop

Le curseur explicite version 2

```
DECLARE

    CURSOR C1 IS SELECT id,salaire FROM drhWHERE grade=3 ;


BEGIN

    FOR Emp in C1 LOOP

        IF Emp.salaire < 1200 THEN  Emp.salaire:=Emp.salaire*1.3 ;
        ELSE Emp.salaire:=Emp.salaire*1.1 ;
        END IF ;

    INSERT INTO promotion VALUES (Emp.id, Emp.salaire);

    END LOOP;

END;
/
```


Vous remarquez qu'on a pas déclaré la variable Emp

Le curseur explicite version 2



Reprenons l'exercice précédent mais avec un curseur explicite version 2
On veut licencier tous les pilotes qui ont fait moins de 10 vols !
Faire un bloc PL/SQL qui pour chaque pilote va afficher :

Pilote numPilote a fait nbVol vols
- numPilote est le numéro du pilote et nbVol est le
- nombre de vols de ce pilote

Puis va effacer le pilote de la table Pilote si ce dernier a fait moins de 10 vols.

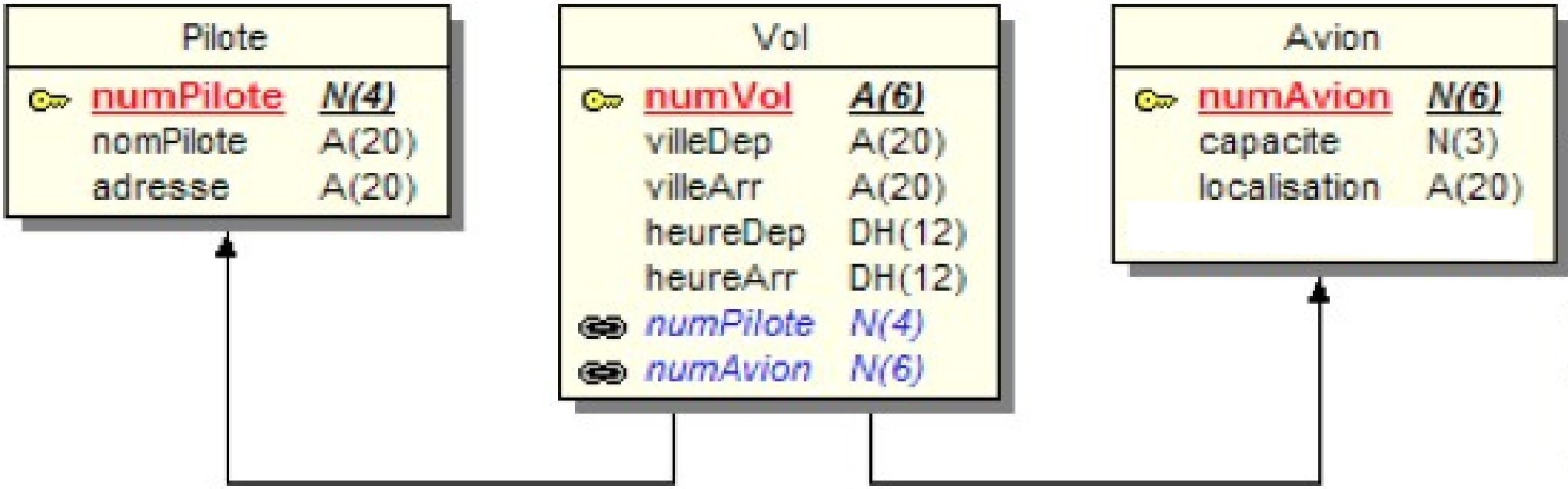
Le curseur explicite version 2

```
Declare
    cursor cx is select * from Pilote;          -- déclaration d'un curseur

    nb number;                                  -- contiendra le nombre de vol par pilote

Begin

    For ligne in cx loop
        select count(*) into nb from Vol where numPilote=ligne.numPilote;
        dbms_output.put_line(ligne.numPilote || ' a fait ' || nb || 'vols' );
        if nb<10 then
            delete from Pilote where numPilote=ligne.numPilote;
        end if;
    end loop;

End;
/
```

Pas besoin de déclarer la variable « ligne »

Calcul du nombre de vol pour chaque pilote du curseur et le mettre dans nb

Le curseur explicite version 2



Reprenons l'exercice précédent mais avec un curseur explicite version 2 :

Pour chaque avion : si l'avion a fait plus de 100 vols à la date d'aujourd'hui alors on va annuler tous les prochaines vols qui sont sur cet avion et supprimer cet avion de la table Avion.

Le curseur explicite version 2

```
Declare
    cursor cx is select * from Avion;          -- déclaration d'un curseur

    nb number;                                 -- contiendra le nombre de vol par avion

Begin

   For ligne in cx loop
        select count(*) into nb from Vol where numAvion=ligne.numAvion and
                                               heureDep<=SYSDATE;
        if nb>100 then
            delete from Avion where numAvion=ligne.numAvion;
            delete from Vol where numAvion=ligne.numAvion and  and
                                               heureDep>SYSDATE
        end if;
    end loop;

End;
/
```


Pas besoin de déclarer la variable « ligne »

Le curseur explicite version 2

Si le SELECT contient un champ qui n'est pas simplement le nom d'une colonne (attribut calculé ou une fonction, etc), alors chaque champ de ce type doit être renommé pour pouvoir être récupéré dans l'enregistrement lié au curseur dans la boucle FOR.

Reprenons l'exemple précédent et appliquons un test sur les salaires nets : 70 % du salaire brut. On remarque que salaire*0.7 n'est pas un nom de colonne, on doit donc le renommer en salaire_net ce qui permettra de le manipuler dans l'enregistrement Emp via Emp.salaire_net

```
DECLARE
    CURSOR C1 IS SELECT id, salaire*0.7 as "salaire_net" FROM drh
BEGIN
    FOR Emp in C1 LOOP
        IF Emp.salaire_net < 1200 THEN ...
```

Le curseur paramétré

Un curseur paramétré peut servir plusieurs fois avec des valeurs des paramètres différentes dans le même esprit que les fonctions avec paramètres.

DECLARE

CURSOR nomCurseur (para1 TYPE, para2 TYPE,…) IS ordreSelect_qui_utilise_para_i

BEGIN

FOR nomStructure IN nomCurseur (val1, val2, ...) LOOP

Le curseur paramétré

Reprenons la table Drh(id, nom, prenom, salaire, grade).

Nous allons utiliser un curseur paramétré pour récupérer les id des employées de grade initial 1 puis ceux de grade terminal 10.

```
Declare

  Cursor C (grd number) is select id from Drh where grade=grd;
  -- le curseur C est paramétré par le grade

Begin

  For Empl in C(1) Loop                -- on passe 1 en parametre donc on cherche les gradé 1
        dbms_output.put_line(' Employé de grade initial :  Id= ' || Empl.id);
  End Loop;

  For Empl in C(10) Loop               -- on passe 10 en parametre donc on cherche les gradé 10

        dbms_output.put_line(' Employé de grade terminal :  Id= ' || Empl.id);
  End Loop;

End;
```

La clause CURRENT OF

Permet d'accéder directement en modification ou en suppression à la ligne que vient de ramener l'ordre FETCH.

Il faut au préalable réserver les lignes lors de la déclaration du curseur par un verrou d'intention à la fin de la déclaration du curseur :

- FOR UPDATE OF colonne1,…,colonne_n : réserve les colonnes 1 à n

- FOR UPDATE : réserve toutes les colonnes

Pour modifier les lignes réservées il faut rajouter

WHERE CURRENT OF nomCurseur

à la fin de la requête update ou delete.

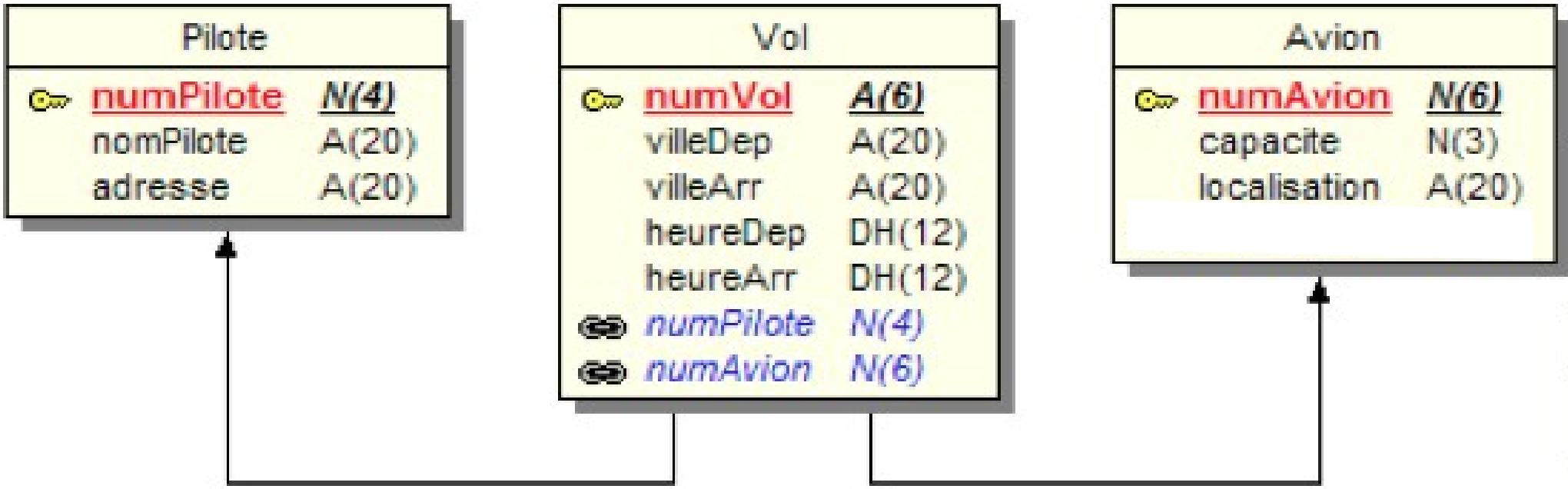
La clause CURRENT OF

Reprenons la table Drh(id, nom, prenom, salaire, grade).

Nous allons faire une promotion à tous les gradés 1 et les faire passer grade 2.

```
Declare
  Cursor C is select id from Drh where grade=1 FOR UPDATE OF grade;
Begin
  For Empl in C Loop
      UPDATE Drh set grade=2 WHERE CURRENT OF C ;
  End Loop ;
End;
/
Select * from Drh ;
```

La clause CURRENT OF



On souhaite mettre sur Nice tous les avions qui sont à Paris

La clause CURRENT OF

```
Declare

  Cursor C is select * from Avion where localisation=paris FOR UPDATE OF
localisation;

Begin

  For ligne in C Loop

      UPDATE Avion set localisation=’nice’ WHERE CURRENT OF C ;

  End Loop ;

End;
/
```

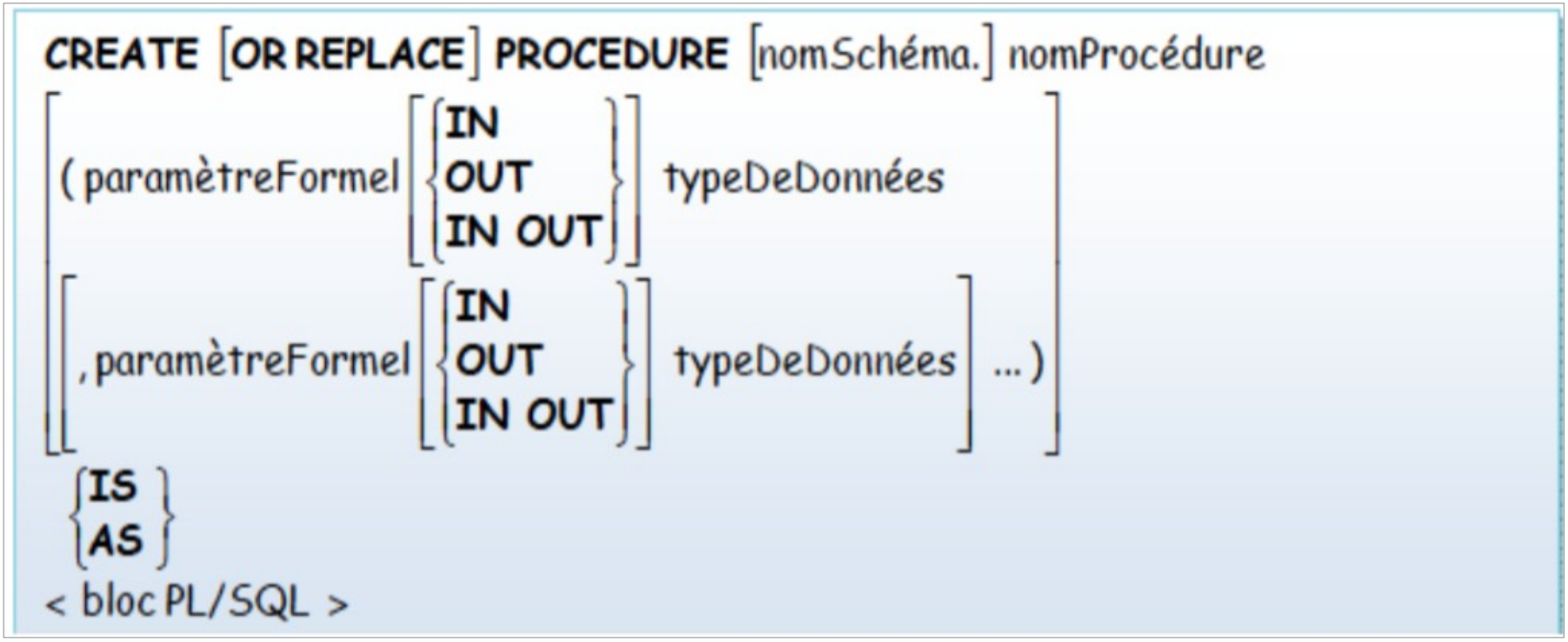
Fonctions et procédures

Fonctions et procédures

Les fonctions et les procédures sont stockées dans la Base de Données, et sont gérées par le noyau du SGBD. On parle de fonctions et de procédures stockées.

Les variables locales seront déclarées après le mot IS (ou bien AS c'est la même chose) mais sans mettre le mot DECLARE.

On supprime le suffixe (X) de la déclaration des paramètres formels. Donc au lieu de mettre varchar2(32) on met juste varchar2.



Fonctions et procédures

Pour les fonctions on doit déclarer le type renvoyé mais sans mettre le suffixe (X).

On doit aussi avoir un RETURN une_expression dans le corps du bloc PL/SQL.

```
CREATE [OR REPLACE] FUNCTION [nomSchéma.] nomFonction
[( paramètreFormel [IN] typeDeDonnées
[, paramètreFormel [IN] typeDeDonnées] ... )]
 RETURN typeDeDonnées
 {IS}
 {AS}
< bloc PL/SQL >
```

Fonctions et procédures

Reprenons la table Drh(id, nom, prenom, salaire, grade) et calculons le total salaires d'un grade donné en paramètre d'une fonction

```
Create or replace function total_salaire_grade (grd IN NUMBER) return NUMBER
As

      nb number ;
Begin

      select sum(salaire) into nb from Drh where grade=grd ;
      return (nb) ;
End;
/
```

Comment on utilise cette fonction ?

Appel d'une fonction

Dans un bloc PLSQL, la valeur de retour est représentée par :

nomFonction(paramètre_1,…,paramètre_n);

On peut alors l'utiliser dans une requête dans un bloc PLSQL :

```
DECLARE

Cursor C is Select * from Drh where salaire=nomFonction(paramètre_1,…,paramètre_n);

BEGIN
…
End ;
```

On peut aussi demander dans un bloc PLSQL un affichage de la valeur de retour d'une fonction :

```
BEGIN
…
DBMS_OUTPUT.PUT_LINE( nomFonction(paramètre_1,…,paramètre_n)   ) ;
…
End ;
```

Appel d'une fonction

Un appel à partir de SQL*PLUS est aussi possible via EXECUTE :

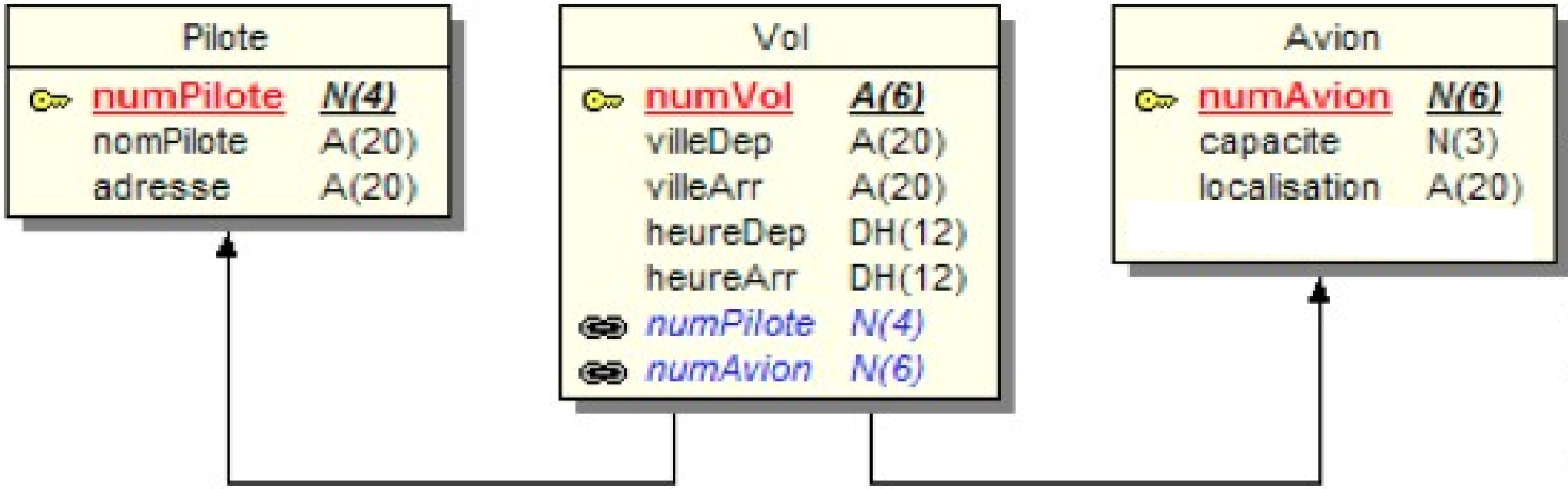
EXECUTE DBMS_OUTPUT.PUT_LINE(nomFonction(paramètre_1, …,paramètre_n)) ;

Qui signifie exécute l'affichage de la valeur de retour de la fonction nomFonction.

Par exemple : EXECUTE DBMS_OUTPUT.PUT_LINE(total_salaire_grade(3)) ;

va afficher la somme des salaires des gradés 3.

Fonctions

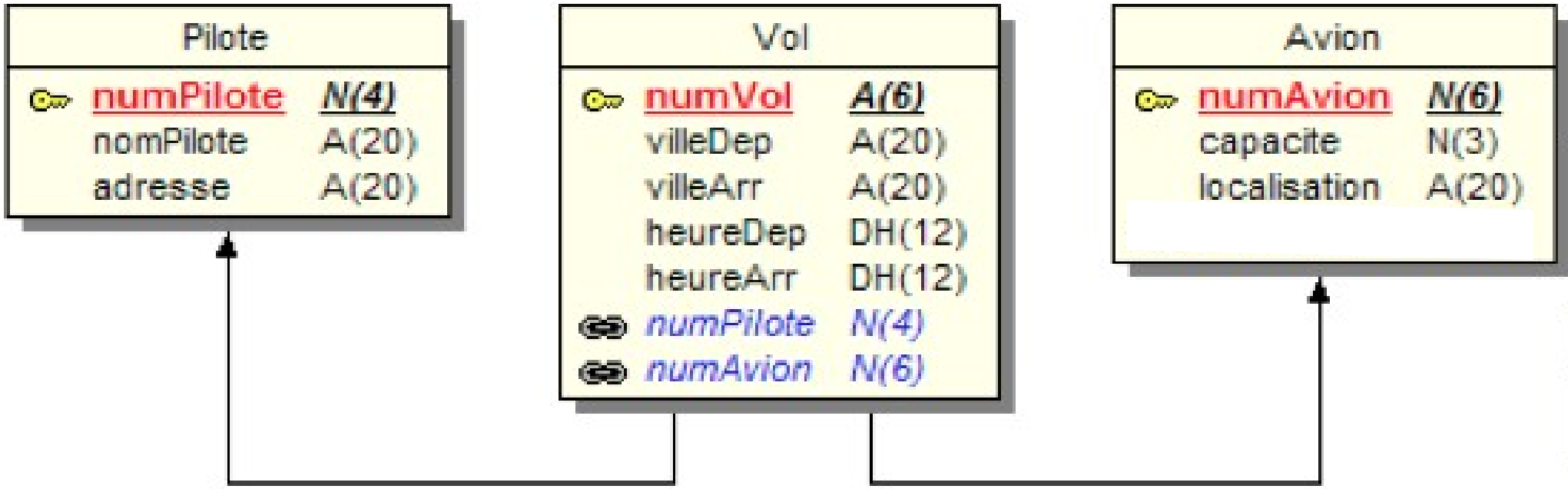


1) Ecrire une fonction qui prend en paramètre le nom d'un pilote et renvoie le nombre de vol qu'il a effectué

2) Ecrire un bloc PL/SQL qui demande un nom de pilote et affiche le nombre de vol qu'il a effectué en utilisant un appel à la fonction précédente sous le format d'affichage suivant :

Le pilote Dupont à 10 vols

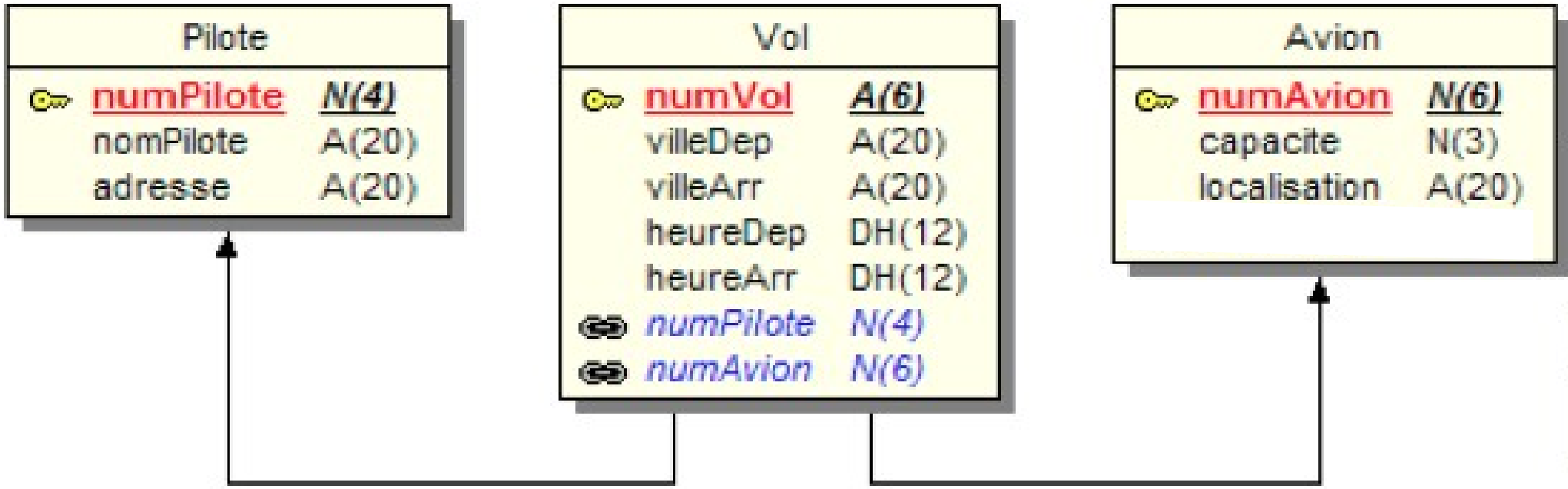
Fonctions



En supposant qu'il n'y a pas d'homonymes, écrire une fonction qui prend en paramètre le nom d'un pilote et renvoie le nombre de vol qu'il a effectué

```
Create or replace function nbVol (nom IN varchar2) return NUMBER
As
      nb number ;
Begin
      select count(*) into nb from Vol where numPilote=(select numPilote from Pilote
                                                         where nomPilote=nom) ;
      return (nb) ;
End;
/
```

Fonctions

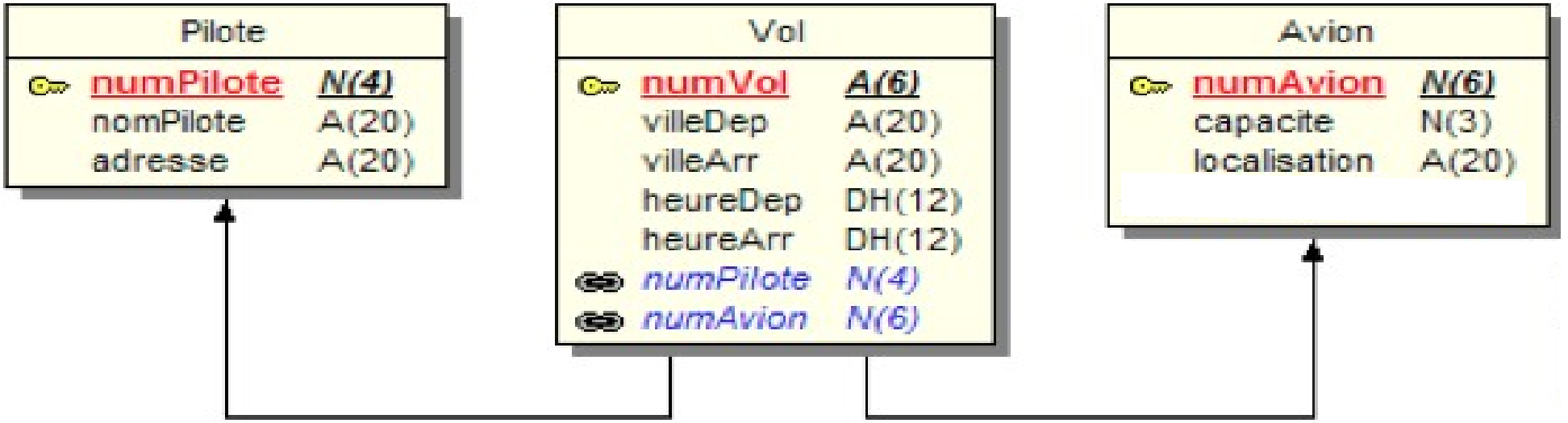


Ecrire un bloc PL/SQL qui demande un nom de pilote et affiche le nombre de vol qu'il a effectué en utilisant un appel à la fonction précédente

```
Accept nomP prompt 'donnez le nom d'un pilote' ;
Begin
      dbms_output.put_line('Le pilote ' || '&nomP' || ' a : ' || nbVol('&nomP') || 'vols' ) ;
End;
/
```

Notez les 'entre &nomP car le paramètre doit être un varchar2 (voir entête de nbVol).

Fonction



Supposons maintenant qu'on puisse avoir des homonymes
Créer une fonction nbVolAvion qui a deux paramètres en entrée : un numéro de pilote et un numéro d'avion et qui va renvoyer le nombre de vols effectué par ce pilote sur cet avion.

Créer une fonction nP qui a un seul paramètre en entrée un nomPilote et qui va renvoyer -1 s'il existe des homonymes ou qui renvoie 0 si le pilote n'existe pas ou qui renvoie le numéro du pilote s'il est le seul à avoir ce nom.

Créer un bloc PL/SQL qui lit un nom de pilote et un numéro d'avion puis utilise les deux fonctions ci-dessus pour :
- soit supprimer de la table pilote le pilote s'il a moins de 10 vols et qu'il a pas d'homonymes
- soit afficher « attention ce pilote a des homonymes »
- soit afficher « pilote non connu »

Fonction

```
Create or replace function nbVolAvion(numP IN NUMBER, numA IN NUMBER ) return
NUMBER
As
      nb number ;
Begin
            select count(*) into nb from Vol where numPilote=numP and numAvion=numA ;
            return (nb) ;

End;
/
```

Fonction

```
Create or replace function nP(nomP IN VARCHAR2) return NUMBER
As
      cursor cx is select * from Pilote where nomPilote=nomP;
      numP number:=0 ;
Begin

  For ligne in cx Loop

      if cx%rowcount>1 then     -- si on a plus de 1 nom dans le fetch implicite
            return (-1);
      end if ;

      numP :=ligne.numPilote ;

   End Loop ;
   Return (numP) ;
End;
/
```

Notez que s'il n'y a pas de pilote avec le nom en entrée alors la boucle est ignorée et du fait de l'affectation dans le AS de numPilote number:=0 on aura 0 dans le résultat.

Fonction

Créer un bloc PL/SQL qui lit un nom de pilote et un numéro d'avion puis utilise les deux fonctions ci-dessus pour :
- soit supprimer de la table pilote le pilote s'il a moins de 10 vols et qu'il a pas d'homonymes
- soit afficher « attention ce pilote a des homonymes »
- soit afficher « pilote non connu »

```
Accept nomP prompt 'donnez le nom d'un pilote' ;
Accept numA prompt 'donnez le num d'un avion' ;
Begin
      if nP('&nomP')=0 then
            dbms_output.put_line(' pilote inconnu' ) ;
      elseif  nP('&nomP')= -1 then
            dbms_output.put_line(' attention ce pilote a des homonymes' ) ;
      elseif  nbVolAvion(nP('&nomP') , &numA)<10 then
            delete from Pilote where numPilote=nP('&nomP') ;
      end if ;
End;
/
```

Appel d'une procédure

Dans une procédure il n'y pas de valeur de retour mais une série de traitements.

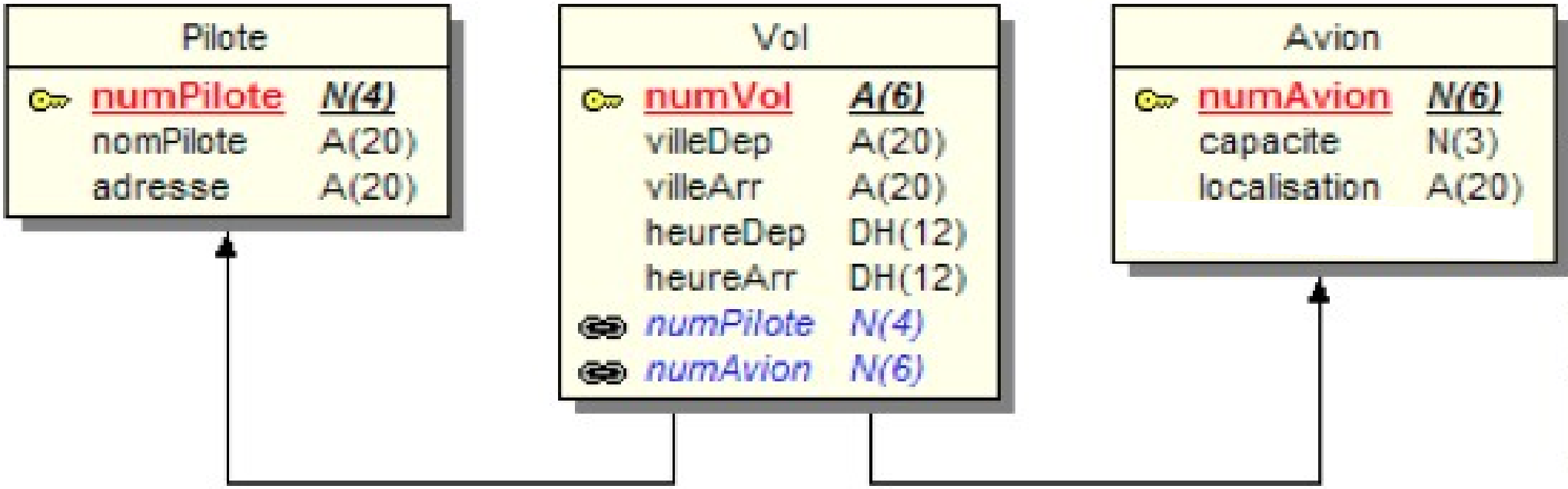
Dans un bloc PLSQL, pour déclencher l'exécution de la procédure il suffit d'écrire :

nomProcedure(paramètre_1,…,paramètre_n);

Un appel à partir de SQL*PLUS est possible via EXECUTE :

EXECUTE nomProcedure(paramètre_1,…,paramètre_n) ;

Procédure



Ecrire une procédure mouvementAvion qui en entrée prend deux localisations et un entier X puis déplace tous les avions de la première localisation vers la seconde et met dans X le nombre d'avions concernés.

Ecrire un bloc PL/SQL qui saisie deux localisations puis appelle la procédure ci-dessous en affichant le nombre d'avions concernés.

Procédure

Ecrire une procédure mouvementAvion qui en entrée prend deux localisations et un entier X puis déplace tous les avions de la première localisation vers la seconde et met dans X le nombre d'avions concernés.

Ecrire un bloc PL/SQL qui saisie deux localisations puis appelle la procédure ci-dessous

```
Create or replace procedure mouvementAvion
        (loc1 IN VARCHAR2, loc2 IN VARCHAR2, x OUT NUMBER)
As


Begin

    update Avion set localisation=loc2 where localisation=loc1 ;
    x:=SQL%ROWCOUNT;

End;
/
```

Procédure

Ecrire un bloc PL/SQL qui saisie deux localisations puis appelle la procédure ci-dessous

```
Accept loc1 prompt ‘donnez la localisation 1’ ;
Accept loc2 prompt ‘donnez la localisation 2 ’ ;

Declare

      volume number ;

Begin
      mouvementAvion(‘&loc1’, ‘&loc2’, volume) ;
      dbms_output.put_line ( ‘il y a :’ || volume ||  ‘avions concernes’) ;

End ;
/
```

Fonctions et procédures : gestion des erreurs

Si on utilise un outil comme SQL Developer la gestion des erreurs est assez simple sinon il faut passer par des vues spécifiques.

La vue USER_ERRORS contient les erreurs détectées selon le schéma suivant :

```
desc user_errors
Nom                                       NULL ?   Type
----------------------------------------- -------- --------------
NAME                                      NOT NULL VARCHAR2(30)
TYPE                                               VARCHAR2(12)
SEQUENCE                                  NOT NULL NUMBER
LINE                                      NOT NULL NUMBER
POSITION                                  NOT NULL NUMBER
TEXT                                      NOT NULL VARCHAR2(4000)
```

```
select line, position, text
from user_errors ;
```

Fonctions et procédures : gestion des erreurs

Le code source de la procédure ou de la fonction est visible dans la vue USER_SOURCE via la requête :

```
select text from user_source where name = 'nom_fonction' ;
```

On peut utiliser la commande de SQL*Plus DESC pour retrouver les caractéristiques d'une fonction ou procédure:

```
desc moy_salaire
FUNCTION moy_salaire RETURNS NUMBER
 Argument Name                  Type            In/Out Default?
 ------------------------------ --------------- ------ --------
 <return value>                 NUMBER          OUT
 N_PREM                         NUMBER          IN
```

TRIGGERS

Les triggers

Ils servent à gérer l'intégrité d'une base de données et à définir des contraintes d'intégrité complexes.

Les triggers ou "déclencheurs" permettent de programmer en "événementiel" sur une base de données.

Un trigger est une procédure stockée qui s'exécute en fonction d'un ou plusieurs événements, qui se produisent sur la table.

Chaque trigger est défini sur UNE seule table.

Leurs déclenchements sont provoqués par des INSERT ou des UPDATE ou des DELETE effectués sur la table.

Les triggers

On commence par préciser le contexte de déclenchement du trigger : si le trigger doit être déclenché avant ou après cet événement, puis le type d'événement concerné (insertion et/ou, mise à jour et/ou suppression) puis la table concernée.

```
CREATE [OR REPLACE] TRIGGER [nomSchéma.] nomTrigger
  {BEFORE
  {AFTER }
  {DELETE                                                    }
  {INSERT                                                    }
  {UPDATE [ OF nomColonne [ , nomColonne]... ]               }
  [     {DELETE                                              }     ]
  [ OR  {INSERT                                              }     ] ...
  [     {UPDATE [ OF nomColonne [ , nomColonne]... ]         }     ]
   ON [nomSchéma.]nomTable
  [{FOR EACH ROW      }]
  [{FOR EACH STATEMENT}]
  [WHEN < condition de restriction >]
< bloc PL/SQL >
```

Les triggers

La clause FOR EACH précise que le trigger est de niveau ligne → il doit être exécuté pour chaque tuple de la table concerné par l'ordre d'insertion, de mise à jour ou de suppression.

La clause WHEN permet, pour les triggers de niveau ligne, de restreindre les circonstances d'exécution du trigger : par exemple, une condition à vérifier par des attributs de la table.

Plusieurs évènements (INSERT, DELETE ou UPDATE) peuvent être spécifiés pour un même trigger en les séparant par OR. Dans ce cas, pour déterminer dans le corps du trigger (bloc PL/SQL) quel événement est déclenché, on peut utiliser les prédicats de déclencheurs :

IF INSERTING THEN …

IF UPDATING THEN …

IF DELETING THEN …

Les triggers

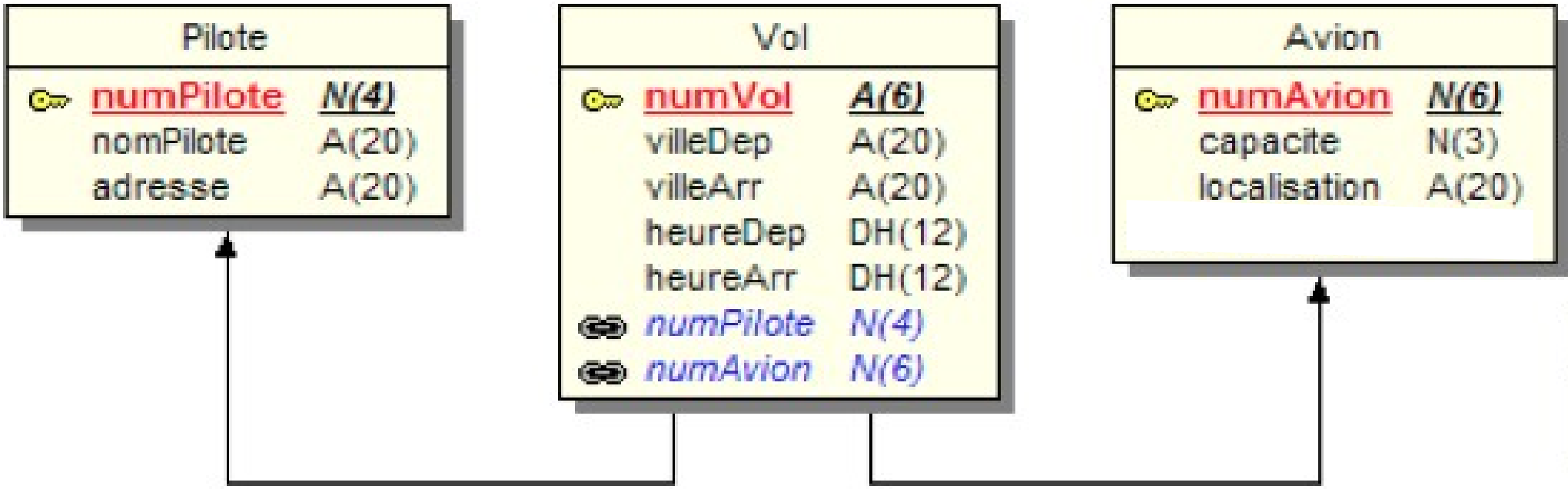
Exemple de base : on aimerait garder trace de toute action sur la table Drh. Pour cela, on crée une table historique_Drh(nom_modif,date_modif).

Ainsi, si deux lignes sont supprimées le 01/06/2015 puis une ligne a été mise à jour alors on retrouvera dans la table historique_Drh :

2 lignes contenant : suppression 01/06/2015
1 ligne contenant : mise à jour 01/06/2015

```
Create Or Replace TRIGGER Drh_histo
AFTER INSERT OR UPDATE OR DELETE ON Drh
FOR EACH ROW
Begin

    If DELETING Then Insert into historique_Drh values (' Suppression ', SYSDATE) ;

    ELSIF UPDATING Then Insert into historique_Drh values (' MAJ ', SYSDATE) ;

    ELSE Insert into historique_Drh values (' Insertion ', SYSDATE) ;

    END If;

End;
/
```

Trigger

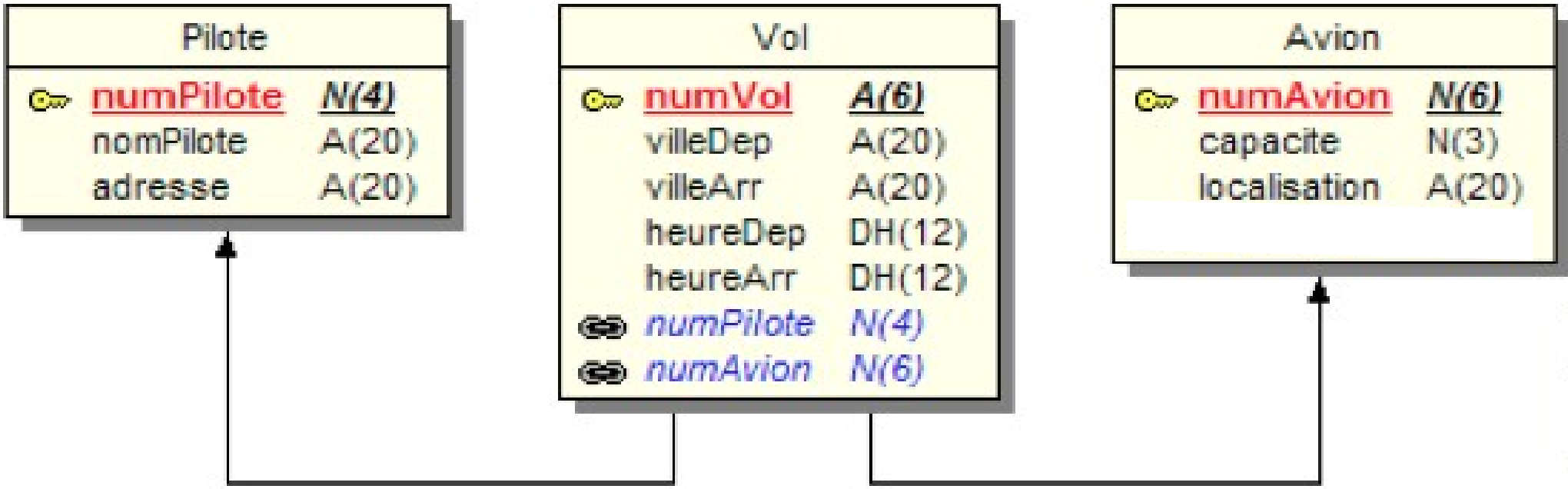


Ecrire un trigger qui après chaque ordre « delete from Pilote ... » va écrire :
« une suppression vient d'être faite : il reste 5 pilotes »

Par exemple : delete * from Pilote where adresse='nice' va effacer tous les pilotes d'un coup qui sont à nice et votre trigger doit écrire UNE SEULE FOIS : nombre de pilotes restants 5

Est ce un trigger ligne ou pas ?

Trigger

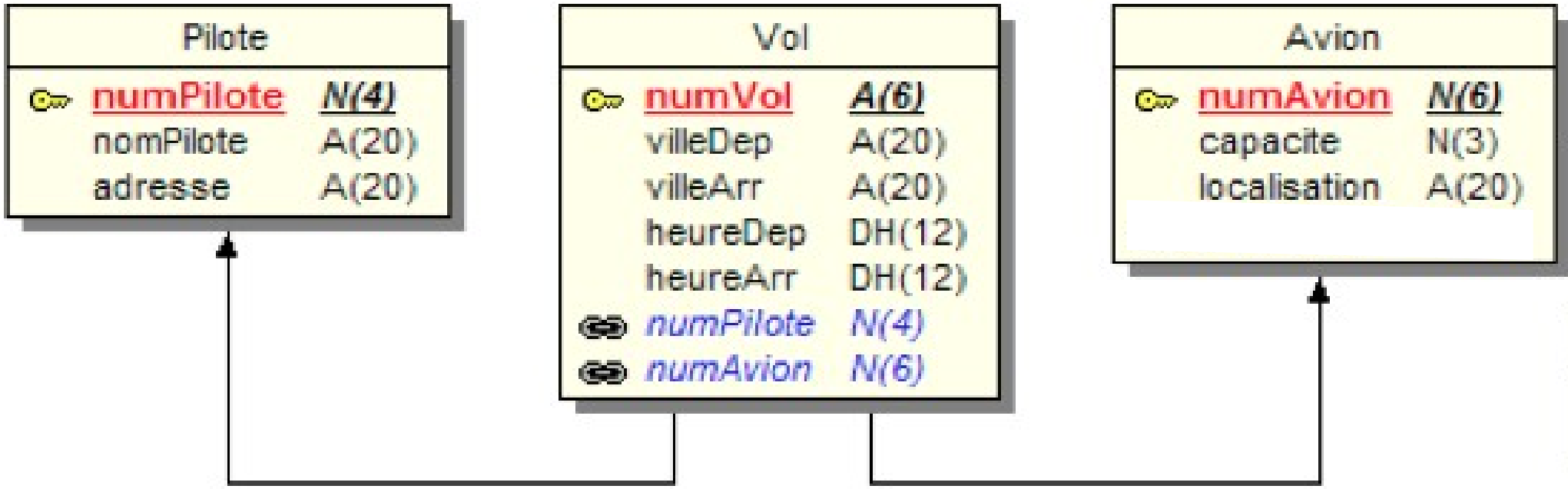


C'est un trigger global et non pas ligne car on veut une seule exécution après toute la série d'effacement.

Si après chaque pilote supprimé habitant nice on voulait afficher le message alors ça serait un trigger ligne.

On utilisera donc pas l'option : for each row

Trigger




```
Create Or Replace TRIGGER pilote_supp
AFTER DELETE ON Pilote
Delare
      nb number ;
Begin
      select count(*) into nb from Pilote ;
      dbms_output.put_line ( 'Une suppression vient d etre faite. Il reste :' || nb ||  'pilotes') ;

End;
/
```

Les triggers de niveau ligne

Les actions prévues dans un trigger de niveau LIGNE (FOR EACH ROW) sont exécutées autant de fois que de lignes concernées par un événement de déclenchement du trigger.

Il est possible d'accéder, dans un trigger ligne, aux anciennes et aux nouvelles valeurs des colonnes de la ligne concernée par la manipulation de données avec :NEW.nomColonne et :OLD.nomColonne.

Pour un trigger d'insertion, la nouvelle valeur de la colonne est dans :NEW.nomColonne.

Pour un trigger de suppression, l'ancienne valeur de la colonne est dans :OLD.nomColonne.

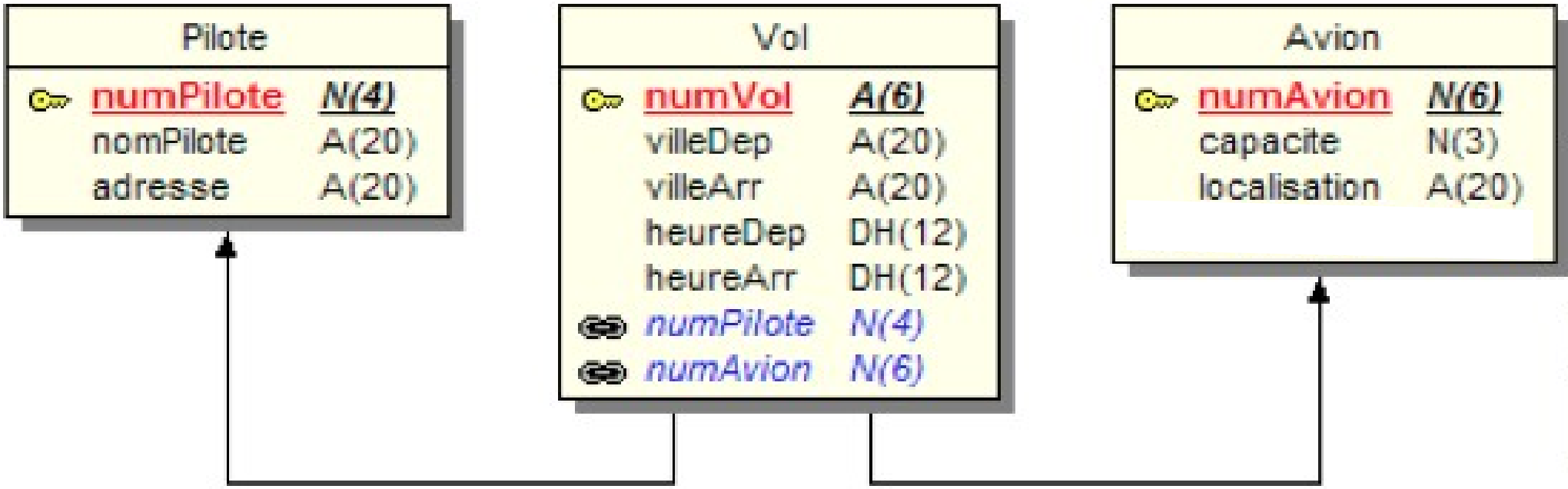
Pour un trigger de modification, les valeurs des colonnes sont dans :NEW.nomColonne et :OLD.nomColonne.

Les triggers de niveau ligne

On reprend l'exemple précédent et on va enregistrer dans la table historique_Drh, en plus de type de la modification et de sa date, l'anciennes valeur d'ID de chaque ligne supprimée ou mise-à-jour. En cas d'insertion d'un nouvel employé on prend son nouvel ID.

```
Create Or Replace TRIGGER Drh_histo
AFTER INSERT OR UPDATE OR DELETE ON Drh
FOR EACH ROW
Begin

    If DELETING Then
        Insert into historique_Drh values (' Suppression ', SYSDATE,  :Old.id) ;

    ELSIF UPDATING Then
        Insert into historique_Drh values (' MAJ ', SYSDATE,   :Old.id) ;

    ELSE
        Insert into historique_Drh values (' Insertion ', SYSDATE, :NEW.id) ;

    END If;

End;
/
```

Trigger

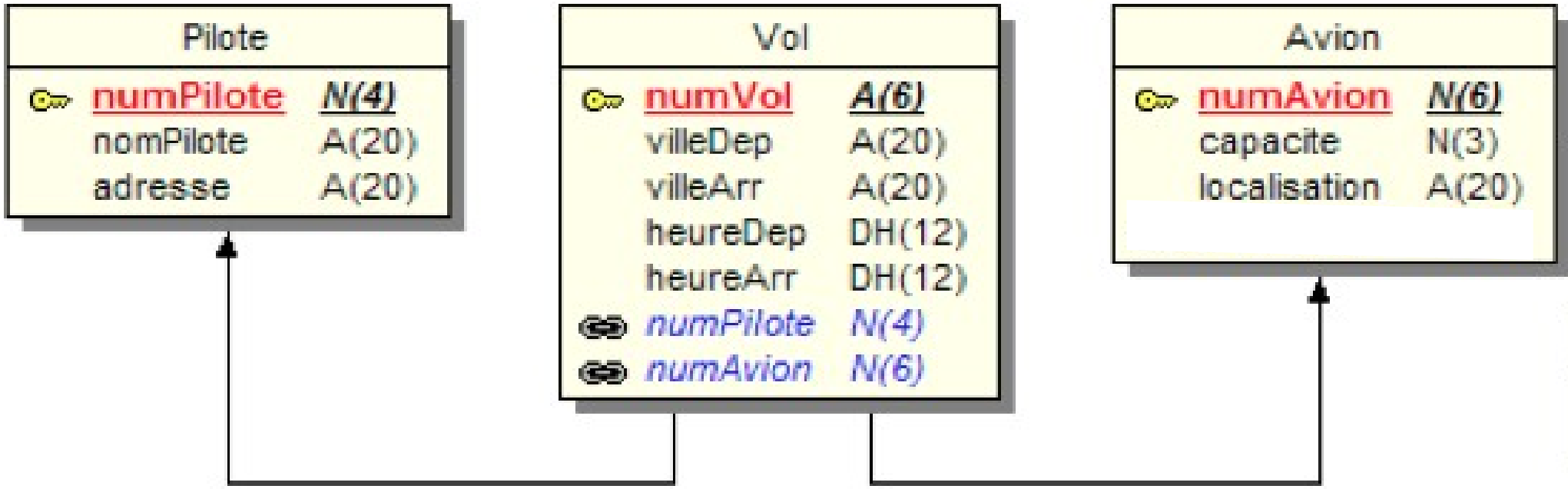


Reprendre le trigger précédent et le modifier pour que :
- en cas de suppression il écrit pour chaque pilote supprimé le nombre de ses vols. Par exemple « le Pilote X a fait Y vols » où X est le numPilote supprimé et Y le nombre de vols de X.
- en cas d'insertion il envoie un message « un nouveau pilote inséré avec numPilote X » où X représente le numPilote inséré.

Par exemple : delete * from Pilote where adresse='nice' va afficher autant de messages qu'il y a de pilotes effacés

Est ce un trigger ligne ?

Trigger



Reprendre le trigger précédent et le modifier pour que :

- en cas de suppression il écrit pour chaque pilote supprimé le nombre de ses vols. Par exemple « le Pilote X a fait Y vols » où X est le numPilote supprimé et Y le nombre de vols de X.
- en cas d'insertion il envoie un message « un nouveau pilote inséré avec numPilote X » où X représente le numPilote inséré.

Oui c'est un trigger ligne car cette fois ci il va afficher autant de messages qu'il y a de pilotes effacés alors que dans l'exemple précédent pour toute une série de suppression de pilote habitant nice il n'y avait qu'un seul message affiché contenant le nombre de pilotes restants.

Trigger

```
Create Or Replace TRIGGER Pilote_maj_supp
AFTER INSERT OR DELETE ON Pilote
FOR EACH ROW
Declare
      nb number:=0  ;
Begin

  If DELETING Then

      select count(*) into nb from vol where numPilote=:OLD.numPilote ;

      dbms_output.put_line ( 'Le pilote' ||:OLD.numPilote || 'a fait' || nb || 'vols') ;

  ELSIF INSERTING Then

dbms_output.put_line ( 'Nouveau pilote insere avec numPilote' || :NEW.numPilote) ;

  END If;

End;
/
```

Trigger : REFERENCING

Une option [REFERENCING] permet de donner un alias à :NEW et :OLD

Par exemple : REFERENCING old AS ancien, new AS nouveau → signifie que pour une variable X on pourra écrire ancien.X et nouveau.X le tout sans « : »

```
Create Or Replace TRIGGER Pilote_maj_supp
AFTER INSERT OR DELETE ON Pilote
REFERENCING old AS ancien, new AS nouveau
FOR EACH ROW
Declare
      nb number:=0  ;
Begin
  If DELETING Then
      select count(*) into nb from vol where numPilote=ancien.numPilote ;
      dbms_output.put_line ( 'Le pilote' ||ancien.numPilote || 'a fait' || nb || 'vols') ;

  ELSIF INSERTING Then
      dbms_output.put_line ( 'Nouveau pilote insere avec numPilote' || nouveau.numPilote) ;
  END If;
End;
/
```

Trigger : WHEN

Il est possible d'ajouter une clause WHEN pour restreindre les cas où le trigger est exécuté.

WHEN est suivi de la condition nécessaire à l'exécution du trigger.

Cette condition peut référencer la nouvelle et l'ancienne valeur d'une colonne de la table mais dans ce cas on utilisera NEW au lieu de:NEW et OLD au lieu de:OLD

Exercice : écrire un triggers qui renvoie un message d'erreur si l'on tente de mettre à jour le salaire d'un employé avec une valeur inférieur à son ancien salaire.

Employe (id,nom,prénom,département,salaire)

Pour l'affichage du message d'erreur, on utlisera raise_application_error(num,message) qui déclencher une erreur en lui associant un message et un numéro (compris entre -20000 et -20999).

Trigger : WHEN

```
CREATE OR REPLACE TRIGGER erreur_salaire

BEFORE UPDATE OF salaire ON Employe  -- notez bien le BEFORE

FOR EACH ROW

WHEN  (NEW.sal < OLD.sal) -- pas de :OLD ni :NEW dans un WHEN

BEGIN

raise_application_error(-20001, 'Interdit de baisser le salaire ' || :old.salaire  );

END ;
```

Notons l'absence des « : » de NEW.sal et OLD.sal dans la clause WHEN mais leur présence dans le corp du trigger entre le BEGIN et le END (:old.salaire)

Restriction sur les triggers de niveau ligne

Si un trigger LIGNE, qui doit être déclenché après/avant un événement de manipulation de données, tente de lire (faire un Selec) ou de modifier la table de base du trigger, l'erreur table ... is mutating, survient car Oracle verrouille toute table en mutation.

Seul un Insert d'une seule ligne avant/après ne génère pas de message d'erreur

Exemple :

```
Create Or Replace TRIGGER Drh_histo
AFTER UPDATE OR DELETE ON Drh
FOR EACH ROW

DECLARE
      le_total number ;
BEGIN
      Select count(id) as "total" into le_total from Drh ;    → erreur car Drh est vérrouillée
```

Triggers de type INSTEAD OF

Dans Oracle, un tel trigger ne peut porter que sur une vue (pas sur une table).

Insertion/suppression/mise à jour par l'intermédiaire de vues

Comme son nom l'indique, l'action d'un trigger INSTEAD OF s'exécute à la place de l'instruction SQL qui l'a déclenchée.

CREATE [OR REPLACE] TRIGGER nom-trigger

INSTEAD OF {INSERT | DELETE | UPDATE}

ON nom-vue

[REFERENCING …]

[FOR EACH ROW]

BLOC PL/SQL

Utilisation des triggers de type INSTEAD OF

A partir d'une la table Employé(id,nom,salaire) nous mettons à la disposition de certains utilisateurs une vue permettant de sélectionner les employés qui sont payé au SMIC.

CREATE VIEW VueSmic AS SELECT id, nom FROM Employe WHERE salaire = 1000

A travers cette vue, ces utilisateurs peuvent insérer des lignes qui seront évidement insérées dans la table d'origine c'est-à-dire dans Employe.

INSERT INTO VueSmic VALUES (9994, 'Durand')

Cette requête a pour conséquence l'insertion du tuple (9994,Durand,NULL) dans la table d'origine de la vue : Employe.

Cependant, les utilisateurs de la vue ne peuvent pas voir leurs tuples insérés car la colonne « salaire » ne fait pas partie de la vue ! Il n'ont donc pas le droit de lui affecter une valeur dans la requête d'insertion.

Donc SELECT * FROM VueSmic n'affichera pas Durand !

Utilisation des triggers de type INSTEAD OF

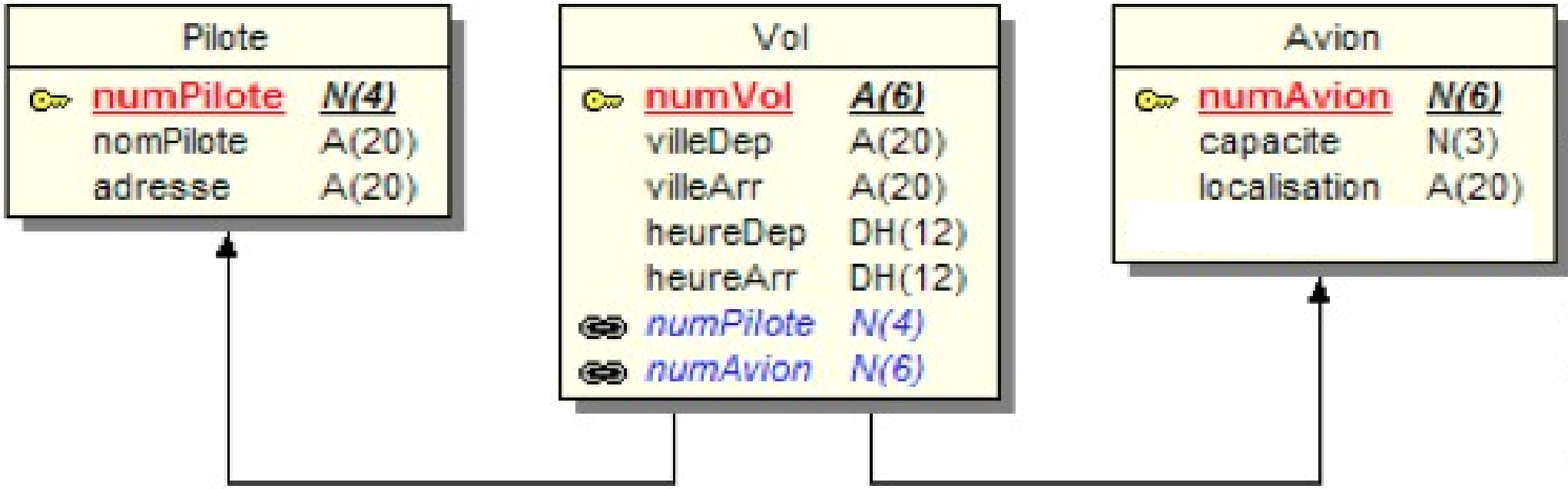
Remédions à ce problème on créant un trigger sur vue par insertion.

```
CREATE OR REPLACE TRIGGER insertion_vue

INSTEAD OF INSERT

ON VueSmic

FOR EACH ROW

Begin
 Insert into Employe ( id,nom,salaire)  Values (:NEW.id, :NEW.nom, 1000 )
End ;
```

Désormais l'ordre INSERT INTO VueSmic VALUES (9994, 'Durand') déclenche le trigger ce qui aura pour conséquence d'insérer le tuple (9994,Durand,1000) dans la table Employe et par conséquent dans la vue VueSmic vu que le salaire = 1000.

Donc SELECT * FROM VueSmic affichera bien Durand !

Trigger



Soit la vue VolAvion composé de la jointure entre Vol et Avion. Elle contient donc tous les champs de Vol et Avion : pour chaque vol on aura toutes les informations de l'avion utilisé.

Si on insère une ligne dans VolAvion on aura un message d'erreur car il est pas capable de deviner qu'il faut aller insérer dans chacune des tables Vol et Avion !

Proposez une solution !

Trigger

```
CREATE OR REPLACE TRIGGER insertion

INSTEAD OF INSERT

ON VolAvion

FOR EACH ROW

Begin

Si l'avion n'existe pas dans la table AVION (a faire chez vous) alors
 Insert into Avion(numAvion,capacité,localisation) values
(:NEW.numAvion,:NEW.capacité,:NEW.localisation)

 Insert into Vol(numVol,villeDep,villeArr,heureDep,heureArr,numPilote,numAvion) values
(:NEW.numVol,:NEW.villeDep,:NEW.villeArr,:NEW.heureDep,:NEW.heureArr,:NEW.numP
ilote,:NEW.numAvion )

Sinon (à faire chez vous)

…
```

EXCEPTION

Les exceptions

La section EXCEPTION permet d'affecter un traitement approprié aux erreurs survenues lors de l'exécution du bloc PL/SQL.

Il n'est pas possible de revenir dans la section BEGIN pendant ou après le traitement d'exception.

On distingue deux types d'exception :

(1) Les exceptions implicitement : déclenchées automatiquement par les erreurs internes à Oracle. Elles sont, soient prédéfinies (NO_DATA_FOUND, etc.), soient non-prédéfinies (transgression d'une F.K., …)

(2) Les exceptions explicitement déclenchées, définies et gérées par l'utilisateur.

Les exceptions

```
DECLARE
      nomErreur  EXCEPTION ;
BEGIN
      …
      IF (anomalie)  THEN
            RAISE nomErreur ;
      END IF ;
      …
EXCEPTION
      {WHEN nomErreur THEN
            {<instruction>}…}…
      [WHEN OTHERS THEN {<instruction>}…]
END;
```

Les exceptions

Reprenons la table Drh(id, nom, prenom, salaire, grade). Nous allons calculer le total des salaires et générer une exception si un salaire est à NULL.

```
DECLARE

      cursor C is Select salaire From drh where grade=grd ;
      total Drh.salaire%TYPE  :=0 ;
      salaire_vide Exception ;

Begin

   For Ligne in C Loop
      IF Ligne.salaire is NULL THEN
            raise salaire_vide ;
      ELSE total := total + Ligne.salaire ;
      END IF ;
   End loop;
   Dbms_output.put_line(total) ;

EXCEPTION
      WHEN salaire_vide THEN Dbms_output.put_line(' il y a un salaire vide ') ;
END ;
/
```

Les exceptions

Certaines erreurs Oracle ont un nom d'exception et un code d'erreur prédéfinis.

CURSORALREADY_OPEN	-6511
DUPVAL_ON_INDEX	-1
INVALIDCURSOR	-1001
INVALIDNUMBER	-1722
LOGIN_DENIED	-1017
NO_DATA_FOUND	+100
NO_LOGGED_ON	-1012
PROGRAM_ERROR	-6501
STORAGE_ERROR	-6500
TIMEOUT_ONRESOURCE	-51
TOO_MANY_ROWS	-1422
TRANSACTIONBACKED_OUT	-61
VALUE_ERROR	-6502
ZERO_DIVIDE	-1476
OTHERS	toutes les autres

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN ...
```

Les exceptions : SQLCODE et SQLERRM(codeErreur)

SQLCODE : renvoie le code de l'erreur courante (c'est une valeur numérique).

0 : aucune erreur rencontrée

1 : exception utilisateur

+100 : NO_DATA_FOUND (ORA-01403)

<0 : (nombre négatif) erreur oracle ORA-0nnnn

SQLERRM (codeErreur) : renvoie le libellé correspondant au code de l'erreur.

Exemple :

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        Dbms_output.put_line(SQLCODE)   -- affichera 100
```

Les erreurs fatales : raise_application_error

raise_application_error(numéro_erreur, message_erreur) ;

Génère une erreur dont le numéro est le premier paramètre et affiche le message donné en second paramètre. Le numéro_erreur est obligatoirement compris entre -20000 et -20999.

Reprenons la table Drh(id, nom, prenom, salaire, grade). Nous allons créer une procédure qui supprime un employé dont l'id est donné en paramètre. Si l'id ne correspond à aucun employé alors on génère une erreur fatale.

```
Create or replace procedure Drh ( Id_emp in Drh.id%TYPE) is

Begin

    delete From Drh where id = Id_emp ;
    if sql%notfound then -- si l instruction précédente n'a traité aucune ligne
        raise_application_error(-20200, 'Cet employe n''existe pas');
    end if;

End ;
/
```

Les erreurs fatales : raise_application_error

Reprenons la table Drh(id, nom, prenom, salaire, grade). Nous allons calculer la moyenne des salaires et générer une erreur fatale si un salaire est à NULL ou qu'il n'y personne dans la table.

```
DECLARE

      cursor C is Select salaire From drh ;
      total Drh.salaire%TYPE  :=0 ;
      nbLigne number  ;

Begin

   For Ligne in C Loop
      IF Ligne.salaire is NULL THEN
            raise_application_error(-20200, 'Il y a un employe sans salaire !!!') ;
      ELSE total := total + Ligne.salaire ;
      END IF ;
      nbLigne:=C%ROWCOUNT ;
   End loop;
   IF nbLigne=0 then raise_application_error(-20200, 'il y a 0 employe dans la table') ;
   ELSE Dbms_output.put_line(total/nbLigne) ;
   END IF ;
END ;
/
```