# Développement d'applications avec les Bases de données

## Partie 1 : PL/SQL

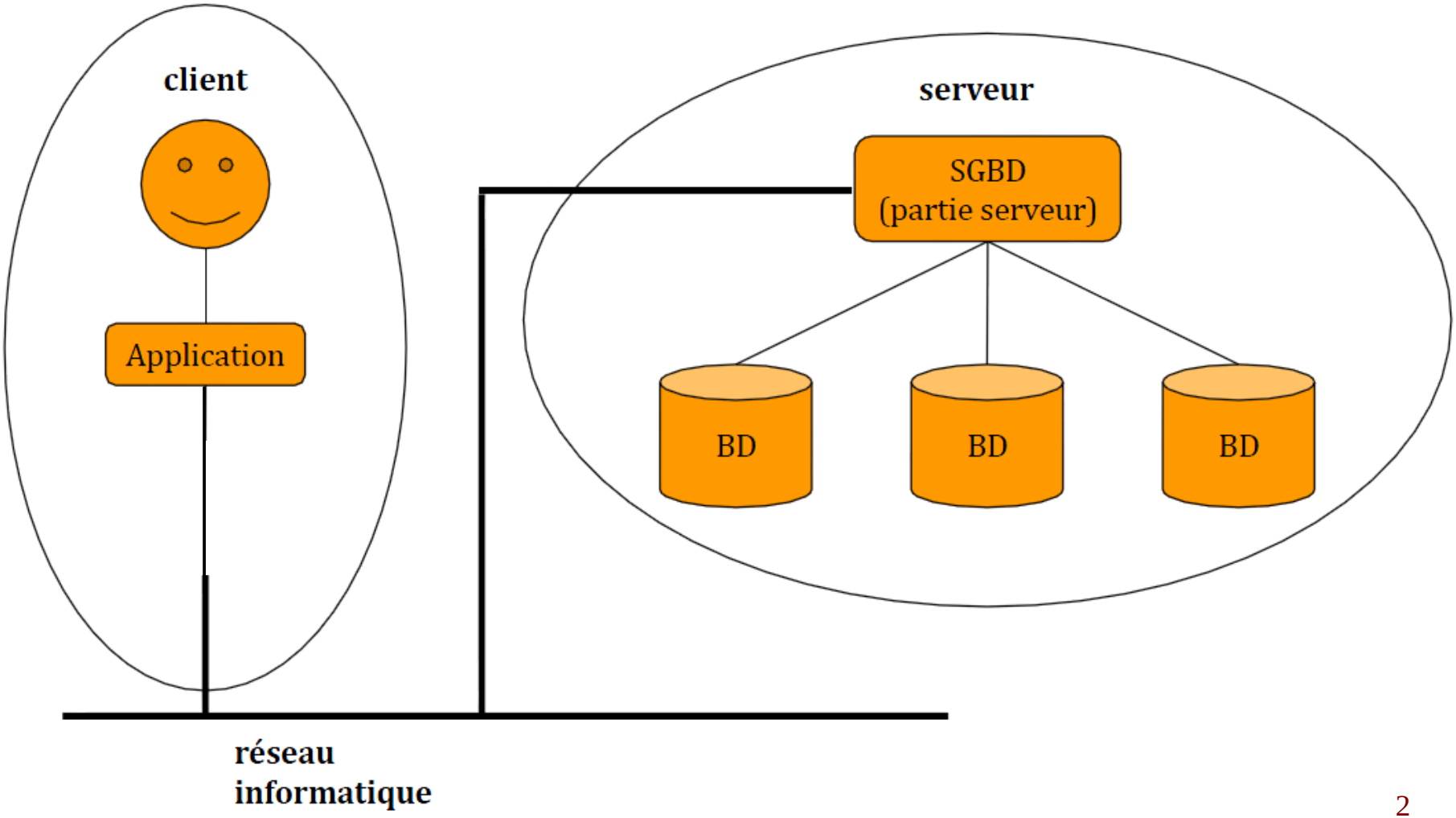
Bases de données relationnelles

Polytech Marseille

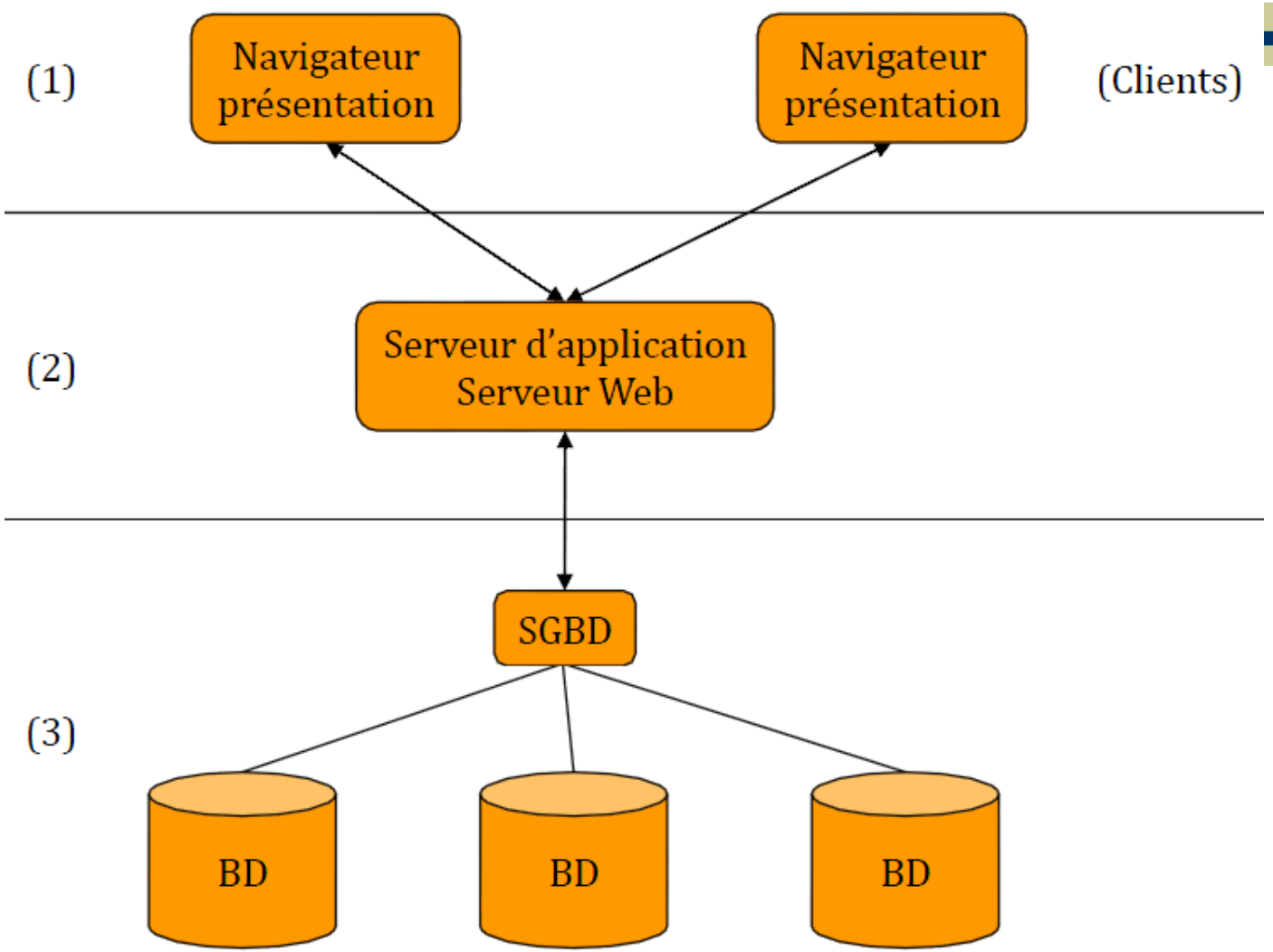
Département Informatique

3ème année

# Bases de données et architectures client-serveur



# Bases de données et architectures client-serveur



# Exemple "vols-réservations" Tables

- AVIONS(**NumAv**, NomAv, CapAv, VilleAv)
- PILOTES(**NumPil**, NomPil, NaisPil, VillePil)
- VOLS(**NumVol**,VilleD, VilleA, DateD, DateA, #NumPil, #NumAv, CoutVol)
- CLIENTS(**NumCl**, NomCl, NumRueCl, NomRueCl, CodePosteCl, VilleCl)
- DEFCLASSES(**#NumVol, Classe**, CoeffPlace, CoeffPrix)
- RESERVATIONS(**#NumCl, #NumVol, #Classe**, NbPlaces)

# Exemple "vols-réservations" Attributs

- Les numéros servant de clé sont des entiers, à l'exception du numéro de vol NumVol qui est une chaîne de caractères commençant par la lettre 'V'.
- CapAv, NaisPil, NbPlaces, NumRueCl, CodePosteCl sont des entiers.
- CoutVol, CoeffPlace et CoeffPrix sont des nombres décimaux :
  - CoeffPlace, dans l'intervalle [0,1], donne le pourcentage de places existant dans la Classe concernée, pourcentage relatif à CapAv, capacité totale de l'avion.
  - CoeffPrix, supérieur ou égal à 1, donne le coefficient multiplicatif à appliquer à CoutVol pour obtenir le prix réel d'un voyage dans la classe voulue. CoutVol est donc le prix minimal d'une place, sur le vol concerné.
- DateD et DateA sont des dates, comportant le jour et l'heure voulue.

# Exemple "vols-réservations" Création de AVIONS et PILOTES

```
CREATE TABLE AVIONS(
  NumAv NUMBER(4) CONSTRAINT pk_avion PRIMARY KEY,
  NomAv VARCHAR2(20),
  CapAv NUMBER(4) CONSTRAINT dom_capav_avion CHECK (CapAv>0),
  VilleAv VARCHAR2(15)
);

CREATE TABLE PILOTES(
  NumPil NUMBER(4) CONSTRAINT pk_pilote PRIMARY KEY,
  NomPil VARCHAR2(20) CONSTRAINT nn_nom_pilote NOT NULL,
  NaisPil NUMBER(4) CONSTRAINT dom_nais_pilote CHECK
(NaisPil>1900),
  VillePil VARCHAR2(15)
);
```

# Exemple "vols-réservations" Création de CLIENTS

```
CREATE TABLE CLIENTS(
NumCl NUMBER(5) CONSTRAINT pk_client PRIMARY KEY,
NomCl VARCHAR2(20) ,
NumRueCl NUMBER(3) CONSTRAINT numrue_client CHECK
(NumRueCl > 0),
NomRueCl VARCHAR2(50),
CodePosteCl NUMBER(5) CONSTRAINT codepostal_client CHECK
(CodePosteCl > 0),
VilleCl VARCHAR2(15)
);
```

# Exemple "vols-réservations" Création de VOLS

```
CREATE TABLE VOLS(
NumVol VARCHAR2(5) CONSTRAINT pk_vol PRIMARY KEY,
VilleD VARCHAR2(15) CONSTRAINT nn_villed_vol NOT NULL,
VilleA VARCHAR2(15) CONSTRAINT nn_villea_vol NOT NULL,
DateD DATE,
DateA DATE,
NumPil NUMBER(4) CONSTRAINT pil_ref_vol
                          REFERENCES PILOTES(NumPil),
NumAv NUMBER(4) CONSTRAINT avion_ref_vol
                          REFERENCES AVIONS(NumAv),
CoutVol NUMBER(10) CONSTRAINT dom_coutvol_vol CHECK (CoutVol>0),
CONSTRAINT dom_numvol_vol CHECK(NumVol LIKE 'V%'),
CONSTRAINT dates_vol CHECK (DateD < DateA),
CONSTRAINT villes_vol CHECK (VilleD != VilleA)
);
```

# Exemple "vols-réservations" Création de DEFCLASSES

```
CREATE TABLE DEFCLASSES(
  NumVol VarChar2(5) CONSTRAINT vol_ref_defclasse references
VOLS(NumVol),
  Classe VARCHAR2(12),
  CoeffPlace NUMBER(3,2) CONSTRAINT nn_coeffplace_defclasse NOT
NULL,
  CoeffPrix NUMBER(3,2) CONSTRAINT coeffprix_defclasse CHECK
(CoeffPrix >= 1),
  CONSTRAINT pk_defclasse PRIMARY KEY (NumVol,Classe),
  CONSTRAINT quota CHECK ((CoeffPrix <= 2 AND CoeffPlace <= 0.5)
OR CoeffPrix > 2),
  CONSTRAINT coeffplace_defclasse CHECK (CoeffPlace BETWEEN 0
AND 1),
);
```

# Exemple "vols-réservations" Création de RESERVATIONS

```
CREATE TABLE RESERVATIONS(
NumCl NUMBER(5) CONSTRAINT client_ref_reserv
                                REFERENCES CLIENTS(NumCl),
NumVol VARCHAR2(5),
Classe VARCHAR2(12),
NbPlaces NUMBER(4) CONSTRAINT nn_nbplaces_reserv NOT NULL,
CONSTRAINT pk_reserv PRIMARY KEY (NumCl, NumVol, Classe),
CONSTRAINT classe_ref_reserv FOREIGN KEY (NumVol,Classe)
                        REFERENCES DEFCLASSES(NumVol,Classe),
CONSTRAINT dom_nbplaces_reserv CHECK (NbPlaces >0)
);
```

# Exemple "vols-réservations" Autres contraintes d'intégrité

- Contrainte 1 : une réservation ne peut pas être passée sur un vol dont le départ a déjà eu lieu.
- Contrainte 2 : il est impossible de supprimer une réservation relative à un vol en cours.
- Contrainte 3 : pour chaque vol, la somme des coefficients CoeffPlace de chaque classe doit être inférieure ou égale à 1.
- Contrainte 4 : à un instant donné, un pilote assure au plus un vol.
- Contrainte 5 : à un instant donné, un avion est au plus utilisé pour un vol.
- Etc.

# Exemple "vols-réservations" Applications

- 1 application client lourd (c, java, ...)
- 1 application web (php, j2ee, ...)
- 2 applications mobiles (ios, android, ...)
- ...

# Exemple "vols-réservations" Fonctionnalités

- Rechercher des informations sur un vol
- Afficher tous les vols (en cours, ou un jour donné)
- Réserver des places sur un vol
- Ajouter un nouveau vol
- Ajouter un nouveau client
- Etc.

# Exemple "vols-réservations" Utilisateurs et droits

- Utilisateur = personne ou application
- Identification
- Gestion des privilèges / Droits d'accès

# Exemple "vols-réservations" Implantation des contraintes

- Dans les applications ?
  - Non...
- Dans le SGBD ?
  - Oui, mais comment ?

    avec un langage procédural comme **PL/SQL**, PL/PgSQL, T-SQL, SQL/PSM, ...

# PL/SQL

Procedural Language / Structured Query Language

# Bibliographie


- SOUTOU, C. "SQL pour Oracle", Eyrolles, 2008 (3ème édition).
- BIZOI, R. "PL/SQL pour Oracle 10g", Eyrolles, 2006.
- DATE, C. "Introduction aux bases de données", Vuibert, 2004 (8ème édition).
- GARDARIN, G. "Bases de données", Eyrolles, 2003.

# Rappel

- SQL se divise en :
  - **DDL** (Data Definition Language)
    - Pour créer le schéma de la BD
  - **DML** (Data Manipulation Language)
    - Pour créer/supprimer/mettre à jour les données
  - **DQL** (Data Query Language)
    - Pour l'extraction de données
  - **DCL** (Data Control Language)
    - Pour gérer les droits, gérer les transactions.

# Introduction

- SQL n’est pas procédural...

→ Les SGBD fournissent une extension du langage SQL (instructions de branchement conditionnel, instructions de répétition, affectations, …)

**PL/SQL** pour ORACLE
T-SQL pour SQL Server
PL/pgSQL pour PostgreSQL
SQL/PSM (issu de la norme SQL2003) pour MySQL

# Intérêt

- Pour la facilité et l'efficacité de développement :
  - gérer le contexte et lier entre elles plusieurs requêtes,
  - créer des bibliothèques de procédures cataloguées réutilisables
- Pour l'efficacité de l'application :
  - diminuer le volume des échanges entre client et serveur (un programme PL/SQL est exécuté sur le serveur)

# PL/SQL
# Aperçu

- Structure d'un programme
- Variables, structures de contrôle
- Curseurs, interaction avec la base
- Sous-programmes (procédures, fonctions)
- Exceptions
- Déclencheurs (triggers)

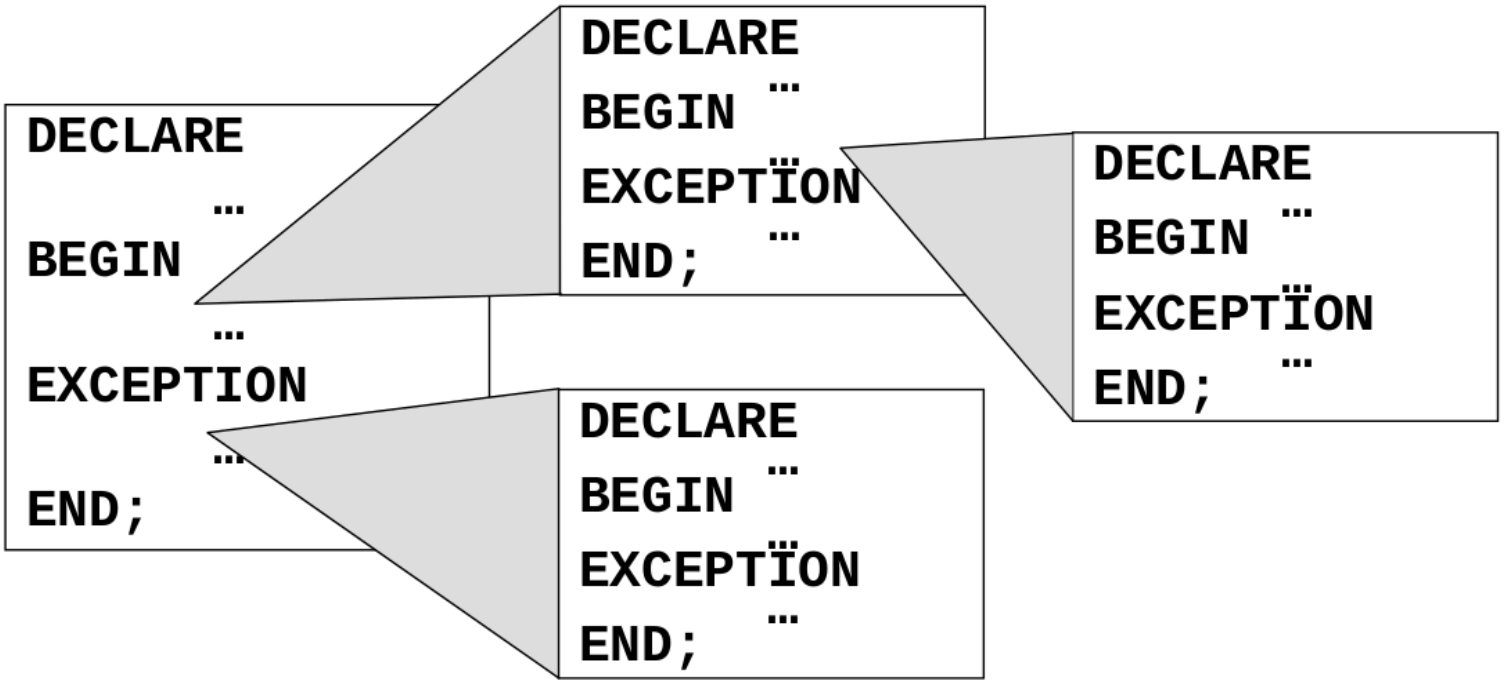
# Structure d'un programme

- Programme PL/SQL = bloc

```
DECLARE
    -- section de déclarations
    -- section optionnelle
    ...
BEGIN
    -- traitement, avec d'éventuelles directives SQL
    -- section obligatoire
    ...
EXCEPTION
    -- gestion des erreurs retournées par le SGBDR
    -- section optionnelle
    ...
END;
```

# Structure d'un programme

- Blocs imbriqués




- Portée d'un identificateur : un descendant peut accéder aux identificateurs déclarés par un parent, pas l'inverse

# Identificateurs et commentaires

- Identificateur (variable, curseur, exception, etc.) :
  - Commence par une lettre
  - Peut contenir : lettres, chiffres, $, #, _
  - Interdits : &, -, /, espace
  - Jusqu'à 30 caractères
  - Insensible à la casse ! (nompilote = NomPILOTE)
- Commentaires :

  -- Commentaire sur une seule ligne

  /* Commentaire sur plusieurs lignes */

# Variables

- Types de variables PL/SQL :
  - Scalaires : par exemple NUMBER(5,2), VARCHAR2, DATE, BOOLEAN, ...
  - Composites : %ROWTYPE, RECORD, TABLE
  - Référence : REF
  - LOB (Large Object jusqu'à 4 Go ; pointeur si externe)
- Un programme PL/SQL peut également manipuler des variables non PL/SQL :
  - Variables de substitution ou globales définies dans SQL*Plus
  - Variables hôtes (définies dans d'autres programmes et utilisées par le programme PL/SQL)
- Toute variable PL/SQL doit obligatoirement être déclarée dans une section DECLARE avant utilisation

# Variables scalaires

- Syntaxe de déclaration :

  ```
  Identificateur [CONSTANT] type [[NOT NULL]
  {:= | DEFAULT} expression];
  ```

  - CONSTANT : c'est une constante (sa valeur ne peut pas changer, doit être initialisée lors de la déclaration)
  - NOT NULL : on ne peut pas lui affecter une valeur nulle (en cas de tentative, une exception VALUE_ERROR est levée)
  - Initialisation : affectation := ou DEFAULT
- Pas de déclaration multiple dans PL/SQL !

  number1, number2 NUMBER; ← déclaration incorrecte !
- Autre possibilité pour affecter une valeur à une variable

  SELECT ... INTO identificateur FROM ... ;

# Nouveaux types PL/SQL

- Nouveaux types prédéfinis :
  - BINARY_INTEGER : entiers signés entre –231 et 231
  - PLS_INTEGER : entiers signés entre –231 et 231 ; plus performant que NUMBER et BINARY_INTEGER au niveau des opérations arithmétiques ; en cas de dépassement, exception levée
- Sous-types PL/SQL : restriction d'un type de base
  - Prédéfinis : CHARACTER, INTEGER, NATURAL, POSITIVE, FLOAT, SMALLINT, SIGNTYPE, etc.
  - Utilisateur (restriction : précision ou taille maximale) :

```
SUBTYPE nomSousType IS typeBase [(contrainte)]
[NOT NULL];
```

  - Exemple de sous-type utilisateur :

```
SUBTYPE numInsee IS NUMBER(13) NOT NULL;
```

# Base de données utilisée pour les exemples

**avion :**

| Numav | Capacite | Type | Entrepot |
|---|---|---|---|
| 14 | 25 | A400 | Garches |
| 345 | 75 | B200 | Lille |

**pilote :**

| Matricule | Nom | Ville | Age | Salaire |
|---|---|---|---|---|
| 1 | Durand | Cannes | 45 | 28004 |
| 2 | Dupont | Touquet | 24 | 11758 |

**vol :**

| Numvol | Heure_depart | Heure_arrivee | Ville_depart | Ville_arrivee |
|---|---|---|---|---|
| AL12 | 08-18 | 09-12 | Paris | Lille |
| AF8 | 11-20 | 23-54 | Paris | Rio |

# Variables scalaires : exemple

```
DECLARE
   villeDepart CHAR(10) := 'Paris';
   villeArrivee CHAR(10);
   numVolAller CONSTANT CHAR := 'AF8';
   numVolRetour CHAR(10);
BEGIN
   SELECT Ville_arrivee INTO villeArrivee FROM vol
   WHERE Numvol = numVolAller; -- vol aller
   numVolRetour := 'AF9';
   INSERT INTO vol VALUES (numVolRetour, NULL, NULL,
   villeArrivee, villeDepart); -- vol retour
END;
```

# Conversions

- Conversions implicites :
  - Lors du calcul d'une expression ou d'une affectation
  - Si la conversion n'est pas autorisée, une exception est levée
- Conversions explicites :

| De A | CHAR | NUMBER | DATE | RAW | ROWID |
|---|---|---|---|---|---|
| **CHAR** | | TO_NUMBER | TO_DATE | HEXTORAW | CHARTOROWID |
| **NUMBER** | TO_CHAR | | TO_DATE | | |
| **DATE** | TO_CHAR | | | | |
| **RAW** | RAWTOHEX | | | | |
| **ROWID** | ROWIDTOHEX | | | | |

# Quelques conversions implicites

| De \ A | CHAR | VARCHAR2 | BINARY_ INTEGER | NUMBER | LONG | DATE | RAW | ROWID |
|---|---|---|---|---|---|---|---|---|
| CHAR | | OUI | OUI | OUI | OUI | OUI | OUI | OUI |
| VARCHAR2 | OUI | | OUI | OUI | OUI | OUI | OUI | OUI |
| BINARY_ INTEGER | OUI | OUI | | OUI | OUI | | | |
| NUMBER | OUI | OUI | OUI | | OUI | | | |
| LONG | OUI | OUI | | | | | OUI | |
| DATE | OUI | OUI | | | OUI | | | |
| RAW | OUI | OUI | | | OUI | | | |
| ROWID | OUI | OUI | | | | | | |

# Déclaration %TYPE

- Déclarer une variable de même type que
  - Une colonne (attribut) d'une table existante :

    ```
    nomNouvelleVariable NomTable.NomColonne
    %TYPE [{:= | DEFAULT} expression];
    ```

  - Une autre variable, déclarée précédemment :

    ```
    nomNouvelleVariable nomAutreVariable%TYPE
    [{:= | DEFAULT} expression];
    ```

- Cette propagation du type permet de réduire le nombre de changements à apporter au code PL/SQL en cas de modification des types de certaines colonnes

# Déclaration %TYPE : exemple

```
DECLARE
    nomPilote pilote.Nom%TYPE; /* table pilote, colonne nom */
    nomCoPilote nomPilote%TYPE;
BEGIN
    ...
    SELECT Nom INTO nomPilote FROM pilote WHERE
    matricule = 1;
    SELECT Nom INTO nomCoPilote FROM pilote WHERE
    matricule = 2;
    ...
END;
```

# Variables %ROWTYPE

- Déclarer une variable composite du même type que les n-uplets d'une table :

```
nomNouvelleVariable NomTable%ROWTYPE;
```

- Les composantes de la variables composite, identifiées par nomNouvelleVariable.nomColonne, sont du même type que les colonnes correspondantes de la table
- Les contraintes NOT NULL déclarées au niveau des colonnes de la table ne sont pas transmises aux composantes correspondantes de la variable !
- Attention, on peut affecter un seul n-uplet à une variable définie avec %ROWTYPE !

# Variables %ROWTYPE : exemple

```
DECLARE
    piloteRecord pilote%ROWTYPE;
    agePilote NUMBER(2) NOT NULL := 35;
BEGIN
    piloteRecord.Age := agePilote;
    piloteRecord.Nom := ‘Pierre’;
    piloteRecord.Ville := ‘Bordeaux’;
    piloteRecord.Salaire := 45000;
    INSERT INTO pilote VALUES piloteRecord;
END;
```

# Variables RECORD

- Déclarer une variable composite personnalisée (similaire à struct en C) :

```
TYPE nomTypeRecord IS RECORD
(nomChamp typeDonnee [[NOT NULL] {:= |
DEFAULT} expression]
[, nomChamp typeDonnee ...]...);
...
nomVariableRecord nomTypeRecord;
```

- Les composantes de la variables composite peuvent être des variables PL/SQL de tout type et sont identifiées par nomVariableRecord.nomChamp

# Variables RECORD : exemple

```
DECLARE
    TYPE aeroport IS RECORD
        (ville CHAR(10), distCentreVille NUMBER(3),
        capacite NUMBER(5));
    ancienAeroport aeroport;
    nouvelAeroport aeroport;
BEGIN
      ancienAeroport.Ville := ‘Paris’;
      ancienAeroport.DistCentreVille := 20;
      ancienAeroport.Capacite := 2000;
      nouvelAeroport := ancienAeroport;
END;
```

# Variables TABLE

- Définir des tableaux dynamiques (dimension initiale non précisée) composés d'une clé primaire et d'une colonne de type scalaire, %TYPE, %ROWTYPE ou RECORD

```
TYPE nomTypeTableau IS TABLE OF
   {typeScalaire | variable%TYPE |
   table.colonne%TYPE | table%ROWTYPE |
   nomTypeRecord} [NOT NULL]
   [INDEX BY BINARY_INTEGER];
nomVariableTableau nomTypeTableau;
```

- Si INDEX BY BINARY_INTEGER est présente, l'index peut être entre –231 et 231 (type BINARY_INTEGER) !
- Fonctions PL/SQL dédiées aux tableaux : EXISTS(x), PRIOR(x), NEXT(x), DELETE(x,...), COUNT, FIRST, LAST, DELETE

# Variables TABLE : exemple

```
DECLARE
    TYPE pilotesProspectes IS TABLE OF pilote%ROWTYPE INDEX
    BY BINARY_INTEGER;
    tabPilotes pilotesProspectes;
    tmpIndex BINARY_INTEGER;
BEGIN
    ...
    tmpIndex := tabPilotes.FIRST;
    tabPilotes(4).Age := 37;
    tabPilotes(4).Salaire := 42000;
    tabPilotes.DELETE(5);
    ...
END;
```

# Résolution des noms

- Règles de résolution des noms :
  - Le nom d'une variable du bloc l'emporte sur le nom d'une variable externe au bloc (et visible)
  - Le nom d'une variable l'emporte sur le nom d'une table
  - Le nom d'une colonne d'une table l'emporte sur le nom d'une variable
- Étiquettes de blocs :

```
<<blocExterne>>
DECLARE
    dateUtilisee DATE;
BEGIN
    DECLARE
        dateUtilisee DATE;
    BEGIN
        dateUtilisee := blocExterne.dateUtilisee;
    END;
END blocExterne;
```

# Entrées et sorties

- Paquetage DBMS_OUTPUT :
  - Mise d'une valeur dans le buffer :

    ```
    PUT(valeur IN {VARCHAR2 | DATE | NUMBER});
    ```

  - Mise du caractère fin de ligne dans le buffer :

    ```
    NEW_LINE(ligne OUT VARCHAR2, statut OUT
    INTEGER);
    ```

  - Mise d'une valeur suivie de fin de ligne dans le buffer :

    ```
    PUT_LINE(valeur IN {VARCHAR2 | DATE |
    NUMBER});
    ```

# Entrées et sorties (suite)

Rendre les sorties possibles avec SQL*Plus :

`SET SERVEROUT ON` (ou `SET SERVEROUTPUT ON`)

- Lecture d'une ligne :

  GET_LINE(ligne OUT VARCHAR2(255), statut OUT INTEGER);

- Lecture de plusieurs lignes :

  GET_LINES(tableau OUT DBMS_OUTPUT.CHARARR, nombreLignes IN OUT INTEGER);

# Entrées et sorties : exemple

- Exemple :

```
DECLARE
   lgnLues DBMS_OUTPUT.CHARARR;
   nombreLignes INTEGER := 2;
BEGIN
   DBMS_OUTPUT.PUT_LINE('Nombre de
   lignes : ' || nombreLignes);
   DBMS_OUTPUT.GET_LINES(lgnLues,
   nombreLignes);
END;
```

# Entrées et sorties (suite)

- Autres API pour des E/S spécifiques :
  - DBMS_PIPE : échanges avec les commandes du système d'exploitation
  - UTL_FILE : échanges avec des fichiers
  - UTL_HTTP : échanges avec un serveur HTTP (Web)
  - UTL_SMTP : échanges avec un serveur SMTP (courriel)
  - HTP : affichage des résultats sur une page HTML

# Structures de contrôle

- Structures conditionnelles :
  - IF ... THEN ... ELSIF... THEN ... ELSE ... END IF;
  - CASE ... WHEN ... THEN ... ELSE ... END CASE;
- Structures répétitives :
  - WHILE ... LOOP ... END LOOP;
  - LOOP ... EXIT WHEN ... END LOOP;
  - FOR ... IN ... LOOP ... END LOOP;

# Structures conditionnelles : IF

- Syntaxe :

```
IF condition1 THEN instructions1;
   [ELSIF condition2 THEN
   instructions3;]
   [ELSE instructions2;]
END IF;
```

- Conditions : expressions dans lesquelles peuvent intervenir noms de colonnes, variables PL/SQL, valeurs

# IF : exemple

```
SQL> ACCEPT s_agePilote PROMPT ‘Age pilote : ’
DECLARE
    salairePilote pilote.Salaire%TYPE;
BEGIN
    IF &s_agePilote = 45 THEN
        salairePilote := 28004;
    ELSIF &s_agePilote = 24 THEN
        salairePilote := 11758;
    END IF;
    DBMS_OUTPUT.PUT_LINE(‘Salaire de base : ’ || salairePilote);
END;
/
```

# Structures conditionnelles : CASE

- Syntaxe avec expressions :

```
[<<etiquette>>]
CASE variable
   WHEN expression1 THEN
   instructions1;
   WHEN expression2 THEN
   instructions2; ...
   [ELSE instructionsj;]
END CASE [etiquette];
```

# Structures conditionnelles : CASE

- Syntaxe avec conditions (searched case) :

```
[<<etiquette>>]
CASE
   WHEN condition1 THEN instructions1;
   WHEN condition2 THEN
   instructions2; ...
   [ELSE instructionsj;]
END CASE [etiquette];
```

- Le premier cas valide est traité, les autres sont ignorés
- Aucun cas valide : exception CASE_NOT_FOUND levée !

# CASE : exemple

```
SQL> ACCEPT s_agePilote PROMPT ‘Age pilote : ’
DECLARE
    salairePilote pilote.Salaire%TYPE;
BEGIN
    CASE &s_agePilote
        WHEN 45 THEN salairePilote := 28004;
        WHEN 24 THEN salairePilote := 11758;
    END CASE;
    DBMS_OUTPUT.PUT_LINE(‘Salaire de base : ’ || salairePilote);
EXCEPTION
    WHEN CASE_NOT_FOUND THEN
        DBMS_OUTPUT.PUT_LINE(‘Évaluer expérience’);
END;
```

# Structures répétitives : WHILE

- Syntaxe :

```
[<<etiquette>>]
WHILE condition LOOP
   instructions;
END LOOP [etiquette];
```

- La condition est vérifiée au début de chaque itération ;

  tant que la condition est vérifiée, les instructions sont exécutées

# WHILE : exemple

```
DECLARE
    valeurEntiere INTEGER := 1;
    produitSerie INTEGER := 1;
BEGIN
    WHILE (valeurEntiere <= 10) LOOP
        produitSerie := produitSerie * valeurEntiere;
        valeurEntiere := valeurEntiere + 1;
    END LOOP;
END;
```

# Structures répétitives : répéter (LOOP)

- Syntaxe :

```
[<<etiquette>>]
LOOP
   instructions1;
   EXIT [WHEN condition];
   [instructions2;]
END LOOP [etiquette];
```

- La condition est vérifiée à chaque itération ; si elle est vraie, on quitte la boucle
- Si la condition est absente, les instructions sont exécutées une seule fois

# LOOP : exemple

```
DECLARE
    valeurEntiere INTEGER := 1;
    produitSerie INTEGER := 1;
BEGIN
    LOOP
        produitSerie := produitSerie * valeurEntiere;
        valeurEntiere := valeurEntiere + 1;
    EXIT WHEN valeurEntiere >= 10;
    END LOOP;
END;
```

# LOOP : boucles imbriquées

```
DECLARE
    ...
BEGIN
    <<boucleExterne>>
    LOOP
        ...
        LOOP
            ...
            EXIT boucleExterne WHEN ...; /* quitter boucle externe */
            ...
            EXIT WHEN ...; -- quitter boucle interne
        END LOOP;
    END LOOP;
END;
```

# Structures répétitives : FOR

- Syntaxe :

```
[<<etiquette>>]
FOR compteur IN [REVERSE] valInf..valSup LOOP
   instructions;
END LOOP;
```

- Les instructions sont exécutées une fois pour chaque valeur prévue du compteur
- Les bornes valInf,valSup sont évaluées une seule fois
- Après chaque itération, le compteur est incrémenté de 1 (ou décrémenté si REVERSE) ; la boucle est terminée quand le compteur sort de l'intervalle spécifié

# FOR : exemple

```
DECLARE
    valeurEntiere INTEGER := 1;
    produitSerie INTEGER := 1;
BEGIN
    FOR valeurEntiere IN 1..10 LOOP
        produitSerie := produitSerie * valeurEntiere;
    END LOOP;
END;
```

# Interaction avec la base

- Interrogation directe des données : doit retourner 1 enregistrement (sinon TOO_MANY_ROWS ou NO_DATA_FOUND)

```
SELECT listeColonnes INTO var1PLSQL [, var2PLSQL ...] FROM nomTable [WHERE condition];
```

- Manipulation des données :
  - Insertions :

```
INSERT INTO nomTable (liste colonnes) VALUES (liste expressions);
```

  - Modifications :

```
UPDATE nomTable SET nomColonne = expression [WHERE condition];
```

  - Suppressions :

```
DELETE FROM nomTable [WHERE condition];
```

# Curseurs

- Les échanges entre l'application et la base sont réalisés grâce à des ***curseurs*** : zones de travail capables de stocker un ou plusieurs enregistrements et de gérer l'accès à ces enregistrements
- Types de curseurs :
  - Curseurs implicites :
    - Déclarés implicitement et manipulés par SQL
    - Pour toute requête SQL du DML et pour les interrogations qui retournent un seul enregistrement
  - Curseurs explicites :
    - Déclarés et manipulés par l'utilisateur
    - Pour les interrogations qui retournent plus d'un enregistrement

# Curseurs implicites

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

# Curseurs implicites : exemple

- Exemple :

```
DELETE FROM pilote WHERE Age >= 55;

DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT
|| ‘ pilotes partent à la retraite
’);
```

# Curseurs explicites : instructions

- Définition = association à une requête ; dans la section des déclarations :

```
CURSOR nomCurseur IS SELECT ... FROM ... WHERE ...;
```

- Ouverture du curseur (avec exécution de la requête) :

```
OPEN nomCurseur;
```

- Chargement de l'enregistrement courant et positionnement sur l'enregistrement suivant :

```
FETCH nomCurseur INTO listeVariables;
```

- Fermeture du curseur (avec libération zone mémoire) :

```
CLOSE nomCurseur;
```

# Curseurs explicites : attributs

- nomCurseur%ISOPEN : TRUE si le curseur est ouvert

```
IF nomCurseur%ISOPEN THEN ... END IF;
```

- nomCurseur%FOUND : TRUE si le dernier accès (FETCH) a retourné une ligne

```
IF nomCurseur%FOUND THEN ... END IF;
```

- nomCurseur%NOTFOUND : TRUE si le dernier accès (FETCH) n'a pas retourné de ligne (fin curseur)

```
IF nomCurseur%NOTFOUND THEN ... END IF;
```

- nomCurseur%ROWCOUNT : nombre total de lignes traitées jusqu'à présent (par ex. nombre de FETCH)

```
DBMS_OUTPUT.PUT_LINE('Lignes traitées : ' ||
nomCurseur%ROWCOUNT);
```

# Curseurs explicites : exemple

```
DECLARE
    CURSOR curseur1 IS SELECT Salaire FROM pilote
        WHERE (Age >= 30 AND Age <= 40);
    salairePilote pilote.Salaire%TYPE;
    sommeSalaires NUMBER(11,2) := 0;
    moyenneSalaires NUMBER(11,2);
BEGIN
    OPEN curseur1;
    LOOP
        FETCH curseur1 INTO salairePilote;
        EXIT WHEN curseur1%NOTFOUND;
        sommeSalaires := sommeSalaires + salairePilote;
    END LOOP;
    moyenneSalaires := sommeSalaires / curseur1%ROWCOUNT;
    CLOSE curseur1;
    DBMS_OUTPUT.PUT_LINE('Moyenne salaires 30 à 40 ans : ' || moyenneSalaires);
END;
```

# Curseurs explicites paramétrés

- Objectif : paramétrer la requête associée à un curseur (procédures, éviter de multiplier les curseurs similaires)
- Syntaxe de définition :

```
CURSOR nomCurseur(param1[, param2, ...])
IS ...;
```

avec, pour chaque paramètre,

```
nomPar [IN] type [{:= | DEFAULT} valeur];
```

(nomPar est inconnu en dehors de la définition !)

- Les valeurs des paramètres sont transmises à l'ouverture du curseur :

OPEN nomCurseur(valeurPar1[, valeurPar2, ...]);

- Il faut évidemment fermer le curseur avant de l'appeler avec d'autres valeurs pour les paramètres

# Curseurs paramétrés : exemple

```
DECLARE
    CURSOR curseur1(ageInf NUMBER, ageSup NUMBER) IS
            SELECT Salaire FROM pilote WHERE (Age >= ageInf AND Age <= ageSup);
    salairePilote pilote.Salaire%TYPE;
    sommeSalaires NUMBER(11,2) := 0;
    moyenneSalaires NUMBER(11,2);
BEGIN
    OPEN curseur1(30,40);
    LOOP
        FETCH curseur1 INTO salairePilote;
        EXIT WHEN curseur1%NOTFOUND;
        sommeSalaires := sommeSalaires + salairePilote;
    END LOOP;
    moyenneSalaires := sommeSalaires / curseur1%ROWCOUNT;
    CLOSE curseur1;
    DBMS_OUTPUT.PUT_LINE(...);
END;
```

# Boucle FOR avec curseur

- Exécuter des instructions pour tout enregistrement retourné par la requête associée à un curseur :

```
DECLARE
   CURSOR nomCurseur(...) IS ...;
BEGIN
   /* un seul OPEN nomCurseur(...), implicite, puis un
      FETCH à chaque itération ; enregistrement
   déclaré implicitement de type nomCurseur%ROWTYPE */
   FOR enregistrement IN nomCurseur(...) LOOP
   ...
   END LOOP; -- implicitement CLOSE nomCurseur
   ...
END;
```

# Boucle FOR avec curseur : exemple

```
DECLARE
    CURSOR curseur1(ageInf NUMBER, ageSup NUMBER) IS SELECT Salaire FROM
    pilote WHERE (Age >= ageInf AND Age <= ageSup);
    sommeSalaires NUMBER(11,2) := 0;
    moyenneSalaires NUMBER(11,2) := 0;
    nbPilotes NUMBER(11,0) := 0;
BEGIN
    FOR salairePilote IN curseur1(30,40) LOOP
        sommeSalaires := sommeSalaires + salairePilote;
        nbPilotes := curseur1%ROWCOUNT;
    END LOOP;
    /* curseur1 fermé, plus possible de lire %ROWCOUNT */
    IF nbPilotes > 0 THEN
        moyenneSalaires := sommeSalaires / nbPilotes;
    END IF;
    DBMS_OUTPUT.PUT_LINE(...);
END;
```

# Curseurs et verrouillage

- Objectif : lorsqu'un curseur est ouvert, verrouiller l'accès aux colonnes référencées des lignes retournées par la requête afin de pouvoir les modifier
- Syntaxe de déclaration :

```
CURSOR nomCurseur[(parametres)] IS
   SELECT listeColonnes1 FROM nomTable
   WHERE condition
   FOR UPDATE [OF listeColonnes2]
      [NOWAIT | WAIT intervalle]
```

- Absence de OF : toutes les colonnes sont verrouillées
- Absence de NOWAIT | WAIT intervalle : on attend (indéfiniment) que les lignes visées soient disponibles

# Modification des lignes verrouillées

- Restrictions : DISTINCT, GROUP BY, opérateurs ensemblistes et fonctions de groupe ne sont pas utilisables dans les curseurs FOR UPDATE
- Modification de la ligne courante d'un curseur :

```
UPDATE nomTable SET
modificationsColonnes
   WHERE CURRENT OF nomCurseur;
```

- Suppression de la ligne courante d'un curseur :

```
DELETE FROM nomTable
   WHERE CURRENT OF nomCurseur;
```

# Modification des lignes : exemple

```
DECLARE
    CURSOR curseur1(villePrime pilote.Ville%TYPE) IS
        SELECT Salaire FROM pilote WHERE (Ville = villePrime) FOR
        UPDATE;
    prime pilote.Salaire%TYPE := 5000;
BEGIN
    -- salaireActuel : implicitement curseur1%ROWTYPE
    FOR salaireActuel IN curseur1('Paris') LOOP
        UPDATE pilote SET Salaire = salaireActuel.Salaire + prime
        WHERE CURRENT OF curseur1;
    END LOOP;
END;
```

# Variables curseurs

- Éviter de manipuler de nombreux curseurs associés à des requêtes différentes : définir des variables curseurs, associées dynamiquement aux requêtes
- Syntaxe de déclaration :

```
TYPE nomTypeCurseur IS REF CURSOR [RETURN type];
nomVariableCurseur nomTypeCurseur;
```

- Le curseur est typé si RETURN type est présente (en général, type est nomDeTable%ROWTYPE) ; ne peut être associé qu'aux requêtes ayant le même type de retour
- Association à une requête et ouverture :

```
OPEN nomVariableCurseur FOR
SELECT ... FROM ... WHERE ...;
```

# Variables curseurs : exemple

```
DECLARE
    TYPE curseurNonType IS REF CURSOR;
    curseur1 curseurNonType;
    salaireInf pilote.Salaire%TYPE := 25000;
    salairePilote pilote.Salaire%TYPE;
    ageInf pilote.Age%TYPE := 27;
BEGIN
    OPEN curseur1 FOR SELECT Salaire FROM pilote
    WHERE Salaire <= salaireInf;
    LOOP
        FETCH curseur1 INTO salairePilote;
        EXIT WHEN curseur1%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Salaire : '|| salairePilote);
    END LOOP;
    CLOSE curseur1;
    OPEN curseur1 FOR SELECT Age FROM pilote WHERE Age <= ageInf;
END;
```

# Sous-programmes

- Blocs PL/SQL nommés et paramétrés
  - Procédures : réalisent des traitements ; peuvent retourner ou non un ou plusieurs résultats
  - Fonctions : retournent un résultat unique ; peuvent être appelées dans des requêtes SQL
- Sont stockés avec la base
- Intérêt de l'utilisation de sous-programmes :
  - Productivité de la programmation : modularité (avantage pour la conception et la maintenance), réutilisation
  - Intégrité : regroupement des traitements dépendants
  - Sécurité : gestion des droits sur les programmes qui traitent les données
- La récursivité est permise (à utiliser avec précaution) !

# Procédures

- Syntaxe :

```
PROCEDURE nomProcedure
   [(par1 [IN | OUT | IN OUT] [NOCOPY] type1
      [{:= | DEFAULT} expression]
   [, par2 [IN | OUT | IN OUT] [NOCOPY] type2
      [{:= | DEFAULT} expression ... )]
   {IS | AS} [declarations;]
BEGIN
   instructions;
[EXCEPTION
   traitementExceptions;]
END[nomProcedure];
```

- Se termine à la fin du bloc ou par une instruction RETURN

# Fonctions

- Syntaxe :

```
FUNCTION nomFonction
   [(par1 [IN | OUT | IN OUT] [NOCOPY] type1
      [{:= | DEFAULT} expression]
   [, par2 [IN | OUT | IN OUT] [NOCOPY] type2
      [{:= | DEFAULT} expression ... )]
   RETURN typeRetour
   {IS | AS} [declarations;]
BEGIN
   instructions;
[EXCEPTION
   traitementExceptions;]
END[nomFonction];
```

- Se termine obligatoirement par RETURN qui doit renvoyer une valeur de type typeRetour

# Paramètres de sous-programme

- Types de paramètres :
  - Entrée (IN) : on ne peut pas lui affecter une valeur dans le sous-programme ; le paramètre effectif associé peut être une constante, une variable ou une expression ; toujours passé par référence !
  - Sortie (OUT) : on ne peut pas l'affecter (à une variable ou à lui-même) dans le sous-programme ; le paramètre effectif associé doit être une variable ; par défaut (sans NOCOPY) passé par valeur !
  - Entrée et sortie (IN OUT) : le paramètre effectif associé doit être une variable ; par défaut (sans NOCOPY) passé par valeur !

# Paramètres de sous-programme

- NOCOPY : passage par référence de paramètre OUT | IN OUT, utile pour paramètres volumineux ; attention aux effets de bord !
- Paramètres OUT | IN OUT pour FUNCTION : mauvaise pratique qui produit des effets de bord. Lorsqu'une fonction doit être appelée depuis SQL, seuls des paramètres IN sont autorisés !

# Définition de sous-programme

- Définition de procédure ou fonction locale dans un bloc

  PL/SQL : à la fin de la section de déclarations

```
DECLARE ...
   PROCEDURE nomProcedure ...; END nomProcedure;
   FUNCTION nomFonction ...; END nomFonction;
BEGIN ... END;
```

- Définition de procédure ou fonction stockée (isolée ou dans un paquetage) :

```
CREATE [OR REPLACE] PROCEDURE nomProcedure ...;
END nomProcedure;
CREATE [OR REPLACE] FUNCTION nomFonction ...;
END nomFonction;
```

# Manipulation de sous-programme

- Création ou modification de sous-programme :

```
CREATE [OR REPLACE] {PROCEDURE | FUNCTION}
nom ...
```

- Oracle recompile automatiquement un sous-programme quand la structure d'un objet dont il dépend a été modifiée
  - Pour une compilation manuelle :

```
ALTER {PROCEDURE | FUNCTION} nom COMPILE
```

  - Affichage des erreurs de compilation sous SQL*Plus :

```
SHOW ERRORS
```

- Suppression de sous-programme :

```
DROP {PROCEDURE | FUNCTION} nom
```

# Appel de sous-programme

- Appel de procédure depuis un bloc PL/SQL :

```
nomProcedure(listeParEffectifs);
```

- Appel de procédure stockée depuis SQL*Plus :

```
SQL> EXECUTE
nomProcedure(listeParEffectifs);
```

- Appel de fonction depuis un bloc PL/SQL : introduction dans une instruction PL/SQL ou SQL de `nomFonction(listeParEffectifs)`
- Appel de fonction stockée depuis SQL*Plus : introduction dans une instruction SQL de `nomFonction(listeParEffectifs)`

# Procédure locale : exemple

```
DECLARE
    nbPilotesPrimes INTEGER := 0;
    PROCEDURE accordPrime(villePrime IN pilote.Ville%TYPE,
            valPrime IN NUMBER, nbPilotes OUT INTEGER) IS
    BEGIN
        UPDATE pilote SET Salaire = Salaire + valPrime
            WHERE (Ville = villePrime);
        nbPilotes := SQL%ROWCOUNT;
    END accordPrime;
BEGIN
    accordPrime('Toulouse', 1000, nbPilotesPrimes);
    DBMS_OUTPUT.PUT_LINE('Nombre pilotes primés : ' || nbPilotesPrimes);
END;
```

# Procédure stockée : exemple

```
CREATE OR REPLACE PROCEDURE
   accordPrime(villePrime IN pilote.Ville%TYPE,
      valPrime IN NUMBER, nbPilotes OUT INTEGER) IS
BEGIN
   UPDATE pilote SET Salaire = Salaire + valPrime
      WHERE (Ville = villePrime);
   nbPilotes := SQL%ROWCOUNT;
END accordPrime;
```

- Appel depuis SQL*Plus :

```
SQL> EXECUTE accordPrime('Gap',1000,nbPilotesPrimes);
```

- Appel depuis un bloc PL/SQL :

```
BEGIN ...
   accordPrime('Gap', 1000, nbPilotesPrimes);
END;
```

# Fonction locale : exemple

```
DECLARE
    salaireMoyInt pilote.Salaire%TYPE;
    FUNCTION moyInt(ageInf IN pilote.Age%TYPE,
        ageSup IN pilote.Age%TYPE)
        RETURN pilote.Salaire%TYPE IS
    BEGIN
        -- la variable du parent est visible ici !
        SELECT AVG(Salaire) INTO salaireMoyInt FROM pilote
                WHERE (Age >= ageInf AND Age <= ageSup);
        RETURN salaireMoyInt;
    END moyInt;
BEGIN
    salaireMoyInt := moyInt(32,49);
    DBMS_OUTPUT.PUT_LINE('Salaire moyen pour âge 32-49 ' || salaireMoyInt);
END;
```

# Fonction stockée : exemple

```
CREATE OR REPLACE FUNCTION
   moyInt(ageInf IN pilote.Age%TYPE,
      ageSup IN pilote.Age%TYPE)
   RETURN pilote.Salaire%TYPE IS
   salaireMoyInt pilote.Salaire%TYPE;
BEGIN
   SELECT AVG(Salaire) INTO salaireMoyInt FROM pilote
      WHERE (Age >= ageInf AND Age <= ageSup);
   RETURN salaireMoyInt;
END moyInt;
```

- Appel depuis SQL*Plus :

```
SQL> SELECT ... FROM pilote WHERE (Salaire > moyInt(32,49));
```

- Appel depuis un bloc PL/SQL :

```
salaireMoyenIntervalle := moyInt(32,49);
```

# Paquetages

- Paquetage = regroupement de variables, curseurs, fonctions, procédures, etc. PL/SQL qui fournissent un ensemble cohérent de services
- Distinction entre ce qui est accessible depuis l'extérieur et ce qui n'est accessible qu'à l'intérieur du paquetage

  → encapsulation
- Structure :
  - Section de spécification : déclarations des variables, curseurs, sous-programmes accessibles depuis l'extérieur
  - Section d'implémentation : code des sous-programmes accessibles depuis l'extérieur + sous-programmes accessibles en interne (privés)

# Section de spécification

- Syntaxe :

```
CREATE [OR REPLACE] PACKAGE nomPaquetage {IS | AS}
   [declarationTypeRECORDpublique ...; ]
   [declarationTypeTABLEpublique ...; ]
   [declarationSUBTYPEpublique ...; ]
   [declarationRECORDpublique ...; ]
   [declarationTABLEpublique ...; ]
   [declarationEXCEPTIONpublique ...; ]
   [declarationCURSORpublique ...; ]
   [declarationVariablePublique ...; ]
   [declarationFonctionPublique ...; ]
   [declarationProcedurePublique ...; ]
END [nomPaquetage];
```

# Spécification : exemple

```
CREATE PACKAGE gestionPilotes AS
    ...
    CURSOR accesPilotes(agePilote pilote.Age%TYPE)
        RETURN pilote%ROWTYPE;
    FUNCTION moyInt(ageInf IN pilote.Age%TYPE,
                    ageSup IN pilote.Age%TYPE)
        RETURN pilote.Salaire%TYPE;
    PROCEDURE accordPrime(villePrime IN pilote.Ville%TYPE,
            valPrime IN NUMBER,nbPilotes OUT INTEGER);
    ...
END gestionPilotes;
```

# Section d'implémentation

- Syntaxe :

```
CREATE [OR REPLACE] PACKAGE BODY
nomPaquetage {IS | AS}
   [declarationTypePrive ...; ]
   [declarationObjetPrive ...; ]
   [definitionFonctionPrivee ...; ]
   [definitionProcedurePrivee ...; ]
   [instructionsFonctionPublique ...; ]
   [instructionsProcedurePublique ...; ]
END [nomPaquetage];
```

# Implémentation : exemple

```
CREATE PACKAGE BODY gestionPilotes AS
    CURSOR accesPilotes(agePilote pilote.Age%TYPE)
        RETURN pilote%ROWTYPE
        IS SELECT * FROM pilote WHERE Age = agePilote;
    FUNCTION moyInt(ageInf IN pilote.Age%TYPE,
                    ageSup IN pilote.Age%TYPE)
        RETURN pilote.Salaire%TYPE IS
    BEGIN
        SELECT AVG(Salaire) INTO salaireMoyInt FROM pilote
            WHERE (Age >= ageInf AND Age <= ageSup);
        RETURN salaireMoyInt;
    END moyInt;
```

# Implémentation : exemple (suite)

```
PROCEDURE accordPrime(villePrime IN pilote.Ville%TYPE,
        valPrime IN NUMBER, nbPilotes OUT INTEGER) IS
BEGIN
    UPDATE pilote SET Salaire = Salaire + valPrime WHERE (Ville =
    villePrime);
    nbPilotes := SQL%ROWCOUNT;
END accordPrime;
END gestionPilotes;
```

# Référence au contenu d'un paquetage

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

```
nomPaquetage.nomObjet

nomPaquetage.nomSousProgramme(...)
```

- Les paquetages autorisent la surcharge des noms de fonctions ou de procédures
  - Toutes les versions (qui diffèrent par le nombre et/ou le type des paramètres) doivent être déclarées dans la section de spécification
  - Les références seront les mêmes pour les différentes versions, le choix est fait en fonction des paramètres effectifs

# Manipulation d'un paquetage

- Re-compilation d'un paquetage :
  - Utiliser CREATE OR REPLACE PACKAGE et terminer par / une des sections (sous SQL*Plus)
  - La modification d'une des sections entraîne la re-compilation automatique de l'autre section
  - Affichage des erreurs de compilation avec SQL*Plus :

    ```
    SHOW ERRORS
    ```
- Suppression d'un paquetage :

  ```
  DROP BODY nomPaquetage;
  DROP nomPaquetage;
  ```

# Exceptions

- Les exceptions correspondent à des conditions d'erreur constatées lors de l'exécution d'un programme PL/SQL ou de requêtes SQL
- PL/SQL propose un mécanisme de traitement pour les exceptions déclenchées, permettant d'éviter l'arrêt systématique du programme
- Les programmeurs peuvent se servir de ce mécanisme non seulement pour les erreurs Oracle, mais pour toute condition qu'ils peuvent définir (→ communication plus riche entre appelants et appelés ou blocs imbriqués)

# Traitement des exceptions

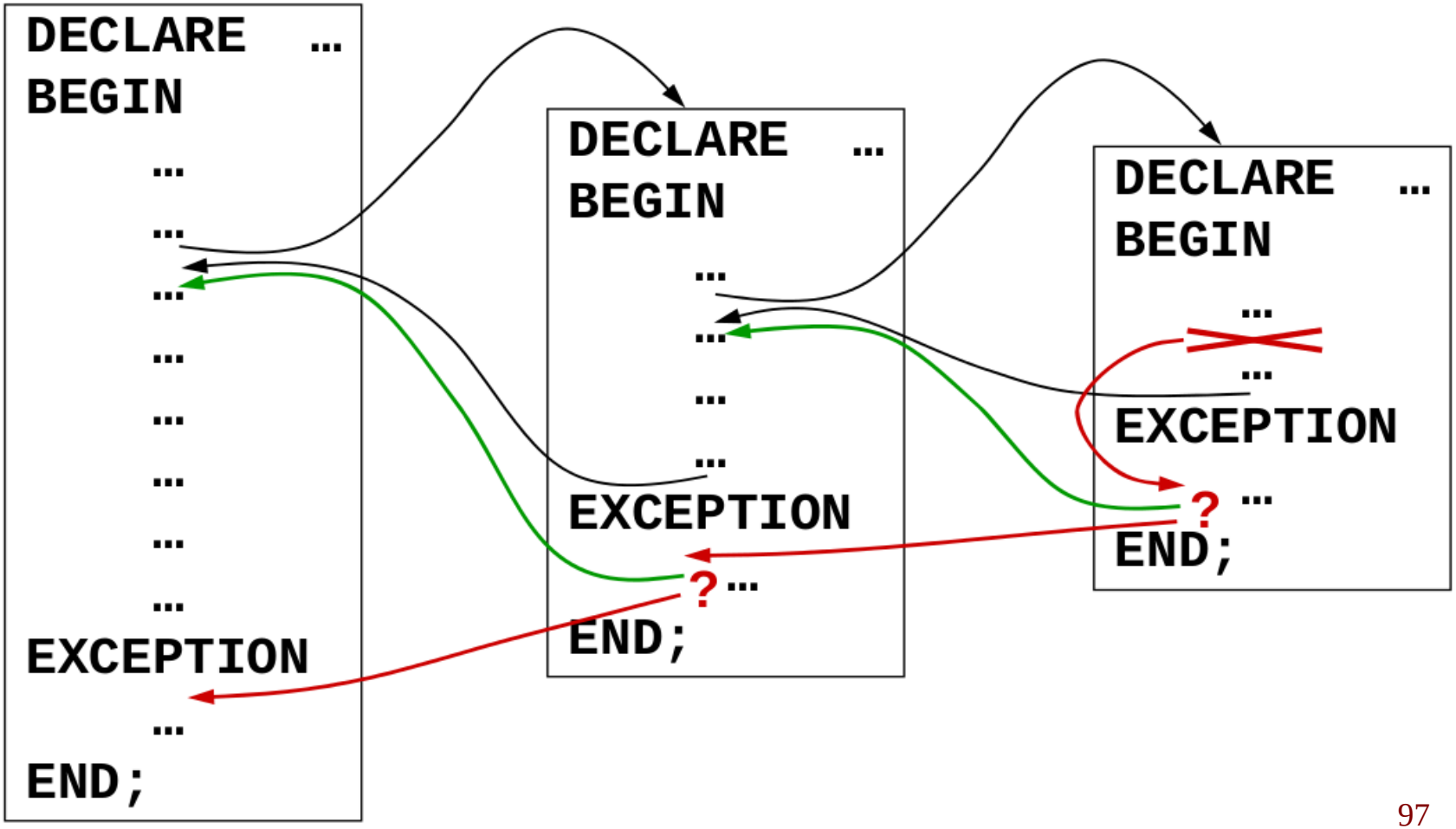
- Syntaxe :

```
...
EXCEPTION
   WHEN nomException1 [OR nomException2 ...]
   THEN
      instructions1;
   WHEN nomException3 [OR nomException4 ...]
   THEN
      instructions3;
   WHEN OTHERS THEN
      instructionsAttrapeTout;
END;
```

# Suites de l'apparition d'une exception

- Aucun traitement n'est prévu : le programme s'arrête
- Un traitement est prévu :
  - L'exécution du bloc PL/SQL courant est abandonnée
  - Le traitement de l'exception est recherché dans la section EXCEPTION associée au bloc courant, sinon dans les blocs parents (englobant le bloc courant) ou le programme appelant
  - L'exception est traitée suivant les instructions trouvées (spécifiques ou attrape-tout)
  - L'exécution se poursuit normalement dans le bloc parent (ou le programme appelant) de celui qui a traité l'exception

# Suites de l'apparition d'une exception



# Mécanismes de déclenchement

- Déclenchement automatique suite à l'apparition d'une des erreurs prédéfinies Oracle : `VALUE_ERROR, ZERO_DIVIDE, TOO_MANY_ROWS`, `NO_DATA_FOUND`, etc. ou non nommées
- Déclenchement programmé (permet au programmeur d'exploiter le mécanisme de traitement des erreurs) :
  - Déclaration (dans DECLARE) :

    ```
    nomException EXCEPTION;
    ```

  - Déclenchement (dans BEGIN) : `RAISE nomException;`

  Il est également possible de déclencher avec RAISE une exception prédéfinie Oracle !

  ```
  RAISE nomExceptionPreDefinie;
  ```

# Exceptions non nommées

- Traitement non spécifique grâce à l'attrape-tout :

```
WHEN OTHERS THEN instructions;
```

Exemple :

```
WHEN OTHERS THEN
 DBMS_OUTPUT.PUT_LINE(SQLERRM || '(' || SQLCODE ||
')');
```

- Identification et traitement spécifique :
  - Identification dans la section DECLARE :

```
nomAPrendre EXCEPTION;
PRAGMA EXCEPTION_INIT(nomAPrendre, numErrOracle);
```

  - Traitement spécifique :

```
WHEN nomAPrendre THEN instructions;
```

# Mécanismes de déclenchement (2)

- Propagation explicite au parent ou à l'appelant après traitement local :

```
WHEN nomException1 THEN
   ...;
   RAISE; -- la même exception nomException1
WHEN autreException THEN ...
```

- Déclenchement avec message et code d'erreur personnalisé :

```
RAISE_APPLICATION_ERROR(numErr,messageErr,[TRUE|FALSE]);
```

  - Utilise le mécanisme des exceptions non nommées, avec SQLERRM et SQLCODE
  - TRUE : mise dans une pile d'erreurs à propager (par défaut) ;
    FALSE : remplace les erreurs précédentes dans la pile

# Déclencheurs (Triggers)

# Déclencheurs
# Définition

Un déclencheur est une règle, dite *active*, de la forme :

"événement-condition-action"

**Déclencheur** = procédure stockée qui est déclenchée automatiquement par des événements *spécifiés par le programmeur* et ne s'exécutant que lorsqu'une condition est satisfaite

# Déclencheurs
# Utilité

Les déclencheurs permettent :

- La possibilité d'**éviter les risques d'incohérence** dus à la présence de redondance
- L'**enregistrement automatique** de certains événements
- La **spécification de contraintes** liées à l'évolution de données
  - Exemple : un salaire ne peut qu'augmenter
- De **définir toutes règles complexes** liées à l'environnement d'exécution (restrictions sur des horaires, des utilisateurs, …)

# Déclencheurs
# Principes

Séquence Événement-Condition-Action :

Trigger déclenché par un **événement**, spécifié par le programmeur

- Insertion, destruction, modification sur une table

  (mais aussi CREATE, ALTER, DROP, Démarrage ou arrêt de la base, Connexion ou déconnexion d'utilisateur, Erreur d'exécution)

Le trigger teste une **condition** : si cette dernière n'est pas vérifiée, alors l'exécution s'arrête

L'**action** est réalisée (toutes opérations sur la base de données)

# Déclencheurs
# Caractéristiques

Il ne peut y avoir qu'**un seul trigger par événement sur une table**

Les triggers permettent de rendre une base de données **dynamique**

- Une opération peut en déclencher d'autres, qui elles-mêmes peuvent entraîner en cascade d'autres triggers

***Ce mécanisme n'est pas sans danger !***

- À cause de risque de boucle infinie

# Déclencheurs
# Caractéristiques

Point important concernant cette procédure (action) :

- **Manipulation simultanée** de l'*ancienne* et de la *nouvelle* valeur d'un attribut (→ permet tests sur l'évolution)

Un trigger peut être exécuté :

- **Une fois** pour un seul ordre SQL
- Ou **à chaque ligne concernée** par cet ordre

L'action peut être réalisée **avant** ou **après** l'événement

# Déclencheurs
# Création

Pour définir un déclencheur, il faut :

- Spécifier l'**événement** qui déclenche l'action en indiquant le type de la mise à jour (INSERT, UPDATE, DELETE), le nom de la table et éventuellement le nom des attributs mis à jour
- Indiquer si l'action est réalisée **avant** ou **après**
- Éventuellement, donner un nom à l'**ancien** et au **nouveau n-uplet** (uniquement le nouveau en cas d'insertion et uniquement l'ancien en cas de suppression)
- Décrire la **condition** sous laquelle se déclenche l'événement sous la forme d'une expression SQL booléenne, c.-à-d. une expression pouvant être placée dans une clause WHERE
- Décrire l'**action à réaliser** sous la forme d'une procédure
- Indiquer si l'action est **réalisée pour chaque** n-uplet mis à jour ou **une seule fois** pour la requête

# Définition d'un déclencheur

- Syntaxe pour déclenchement sur instruction LMD :

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
{BEFORE | AFTER | INSTEAD OF}
{DELETE | INSERT | UPDATE [OF colonne 1, ...]
[OR ...]}
ON {nomTable | nomVue}
[REFERENCING {OLD [AS] nomAncien | NEW [AS]
nomNouveau | PARENT [AS] nomParent } ...]
[FOR EACH ROW]
[WHEN conditionSupplementaire]
{[DECLARE ...] BEGIN ... [EXCEPTION ...] END;
| CALL nomSousProgramme(listeParametres)}
```

# Déclencheurs sur instruction DML

- Quand le déclenchement a lieu (si concevable) :
  - Avant l'événement : BEFORE
  - Après l'événement : AFTER
  - À la place de l'événement : INSTEAD OF (uniquement pour vues multi-tables)
- Description de l'événement (pour instructions DML) :
  - La ou les (OR) instructions,
  - Si l'événement concerne des colonnes spécifiques ([OF colonne 1, ...]) ou non,
  - Le nom de la table (ou vue) (ON {nomTable | nomVue})

# Déclencheurs sur instruction DML

- Changement des noms par défaut : REFERENCING
  - :OLD désigne un enregistrement à effacer (déclencheur sur DELETE, UPDATE) : REFERENCING OLD AS nomAncien
  - :NEW désigne un enregistrement à insérer (déclencheur sur INSERT, UPDATE) : REFERENCING NEW AS nomNouveau
  - :PARENT pour des nested tables : REFERENCING PARENT AS nomParent
- FOR EACH ROW :
  - Avec FOR EACH ROW, 1 exécution par ligne concernée par l'instruction LMD (row trigger)
  - Sans FOR EACH ROW, 1 exécution par instruction DML (statement trigger)

# Base de données « exemple »

- Tables de la base :

```
immeuble (Adr, NbEtg, DateConstr, NomGerant)

appart (Adr, Num, Type, Superficie, Etg,
NbOccup)

personne (Nom, Age, CodeProf)

occupant (#Adr, #NumApp, #NomOccup,
DateArrivee, DateDepart)

propriete (#Adr, #NomProprietaire, QuotePart)
```

# Base de données « exemple »

- Exemples de contraintes à satisfaire :
  - Intégrité référentielle (clé étrangère) : lors de la création de la table propriete :

```
CONSTRAINT prop_pers FOREIGN KEY
(NomProprietaire) REFERENCES personne (Nom)
```

- Condition entre colonnes : lors de la création de la table occupant :

```
CONSTRAINT dates CHECK (DateArrivee <
DateDepart)
```

- Règles de gestion : somme quotes-parts pour une propriété = 100 ; date construction immeuble < dates arrivée de tous ses occupants...

→ **déclencheurs**

# Déclencheur sur INSERT

- Pour un nouvel occupant, vérifie si occupant.DateArrivee > immeuble.DateConstr (FOR EACH ROW est nécessaire pour avoir accès à :NEW, l'enregistrement ajouté) :

```
CREATE TRIGGER TriggerVerificationDates
    BEFORE INSERT ON occupant FOR EACH ROW
```

- 
```
DECLARE
    Imm immeuble%ROWTYPE;
BEGIN
    SELECT * INTO Imm FROM immeuble WHERE immeuble.Adr = :NEW.Adr;
    IF :NEW.DateArrivee < Imm.DateConstr THEN
        RAISE_APPLICATION_ERROR(-20100, :NEW.Nom || ' arrivé avant
        construction immeuble ' || Imm.Adr);
    END IF;
END;
```

- Événement déclencheur : INSERT INTO occupant … VALUES ...;

# Déclencheur sur INSERT (2)

- Si chaque nouvel immeuble doit avoir au moins un appartement, insérer un appartement après la création de l'immeuble (FOR EACH ROW est nécessaire pour avoir accès à :NEW, l'enregistrement ajouté) :

```
CREATE TRIGGER TriggerAppartInitial
   AFTER INSERT ON immeuble FOR EACH ROW
BEGIN
   INSERT INTO appart (Adr, Num, NbOccup)
   VALUES (:NEW.Adr, 1, 0);
END;
```

- Événement déclencheur : INSERT INTO immeuble ... VALUES ...;

# Déclencheur sur DELETE

- Au départ d'un occupant, décrémente appart.NbOccup après effacement de l'occupant (FOR EACH ROW est nécessaire car la suppression peut concerner plusieurs occupants, ainsi que pour avoir accès à :OLD, l'enregistrement éliminé) :

```
CREATE TRIGGER TriggerDiminutionNombreOccupants
   AFTER DELETE ON occupant FOR EACH ROW
BEGIN
   UPDATE appart SET NbOccup = NbOccup – 1
      WHERE appart.Adr = :OLD.Adr
         AND appart.Num = :OLD.NumApp;
END;
```

- Événement déclencheur : DELETE FROM occupant WHERE ... ;

# Déclencheur sur UPDATE

- En cas de modification d'un occupant, met à jour les valeurs de appart.NbOccup pour 2 les appartements éventuellement concernés (utilise à la fois :OLD et :NEW) :

```
CREATE TRIGGER TriggerMAJNombreOccupants
   AFTER UPDATE ON occupant FOR EACH ROW
BEGIN
   IF :OLD.Adr<>:NEW.Adr OR :OLD.NumApp<>:NEW.NumApp THEN
      UPDATE appart SET NbOccup = NbOccup – 1 WHERE
      appart.Adr = :OLD.Adr AND appart.Num = :OLD.NumApp;
      UPDATE appart SET NbOccup = NbOccup + 1 WHERE
      appart.Adr = :NEW.Adr AND appart.Num = :NEW.NumApp;
   END IF;
END;
```

- Événement déclencheur : UPDATE occupant SET ... WHERE ...;

# Déclencheur sur conditions multiples

- Un seul déclencheur pour INSERT, DELETE, UPDATE qui met à jour les valeurs de appart.NbOccup pour le(s) appartement(s) concerné(s) :

```
CREATE TRIGGER TriggerCompletMAJNombreOccupants
    AFTER INSERT OR DELETE OR UPDATE ON occupant FOR
    EACH ROW
```

- ```
  BEGIN
      IF (INSERTING) THEN ...
      ELSIF (DELETING) THEN ...
      ELSIF (UPDATING) THEN ...
      END IF;
  END;
  ```
- Exemple d'événement déclencheur : INSERT INTO occupant VALUES ...;
  ```

# Déclencheurs sur instruction DDL

- Syntaxe pour déclenchement sur instruction DDL :

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
BEFORE | AFTER action [OR action ...]
ON {[nomSchema.]SCHEMA | DATABASE}
{[DECLARE ...] BEGIN ... [EXCEPTION ...] END;
| CALL nomSousProgramme(listeParametres)}
```

- `SCHEMA` : déclencheur valable pour schéma courant
- Quelques actions :
  - CREATE, RENAME, ALTER, DROP sur un objet du dictionnaire
  - GRANT, REVOKE privilège(s) à un utilisateur

# Déclencheurs d'instance

- Syntaxe :

```
CREATE [OR REPLACE] TRIGGER nomDeclencheur
BEFORE | AFTER evenement [OR evenement ...]
ON {[nomSchema.]SCHEMA | DATABASE}
{[DECLARE ...] BEGIN ... [EXCEPTION ...] END;
| CALL nomSousProgramme(listeParametres)}
```

- Événements déclencheurs concernés :
  - Démarrage ou arrêt de la base : STARTUP ou SHUTDOWN
  - Connexion ou déconnexion d'utilisateur : LOGON ou LOGOFF
  - Erreurs : SERVERERROR, NO_DATA_FOUND, ...

# Manipulation d'un déclencheur

- Tout déclencheur est actif dès sa compilation !
- Re-compilation d'un déclencheur après modification :

```
ALTER TRIGGER nomDeclencheur COMPILE;
```

- Désactivation de déclencheurs :

```
ALTER TRIGGER nomDeclencheur DISABLE;
ALTER TABLE nomTable DISABLE ALL TRIGGERS;
```

- Réactivation de déclencheurs :

```
ALTER TRIGGER nomDeclencheur ENABLE;
ALTER TABLE nomTable ENABLE ALL TRIGGERS;
```

- Suppression d'un déclencheur :

```
DROP TRIGGER nomDeclencheur;
```

# Droits de création et manipulation

- Déclencheurs d'instance : privilège `ADMINISTER DATABASE TRIGGER`
- Autres déclencheurs :
  - Dans tout schéma : privilège `CREATE ANY TRIGGER`
  - Dans votre schéma : privilège `CREATE TRIGGER` (rôle `RESOURCE`)