

***www.Mcours.com***

Site N°1 des Cours et Exercices

Email: [mymcours@gmail.com](mailto:mymcours@gmail.com)

Introduction au PL/SQL Oracle

# Table des matières

<b>1</b>	<b>Notes de cours</b>	<b>3</b>
1.1	Introduction au PL/SQL . . . . .	3
1.1.1	PL/SQL . . . . .	3
1.1.2	Blocs . . . . .	3
1.1.3	Affichage . . . . .	3
1.1.4	Variables . . . . .	3
1.1.5	Traitements conditionnels . . . . .	4
1.1.6	Traitements répétitifs . . . . .	4
1.2	Tableaux et structures . . . . .	5
1.2.1	Tableaux . . . . .	5
1.2.2	Structures . . . . .	6
1.3	Utilisation du PL/SQL . . . . .	8
1.3.1	Affectation . . . . .	8
1.3.2	Tables et structures . . . . .	8
1.3.3	Transactions . . . . .	9
1.4	Exceptions . . . . .	10
1.4.1	Rattraper une exception . . . . .	10
1.4.2	Exceptions prédéfinies . . . . .	11
1.4.3	Codes d'erreur . . . . .	11
1.4.4	Déclarer et lancer ses propres exceptions . . . . .	11
1.5	Sous-programmes . . . . .	13
1.5.1	Procédures . . . . .	13
1.5.2	Fonctions . . . . .	13
1.6	Curseurs . . . . .	15
1.6.1	Introduction . . . . .	15
1.6.2	Les curseurs . . . . .	15
1.7	Curseurs paramétrés . . . . .	17
1.7.1	Introduction . . . . .	17
1.7.2	Définition . . . . .	17
1.7.3	Déclaration . . . . .	17
1.7.4	Ouverture . . . . .	17
1.7.5	Lecture d'une ligne, fermeture . . . . .	17
1.7.6	Boucle pour . . . . .	18
1.7.7	Exemple récapitulatif . . . . .	18
1.8	Triggers . . . . .	19
1.8.1	Principe . . . . .	19
1.8.2	Classification . . . . .	19
1.8.3	Création . . . . .	19
1.8.4	Accès aux lignes en cours de modification . . . . .	20
1.8.5	Contourner le problème des tables en mutation . . . . .	22
1.9	Packages . . . . .	25
1.9.1	Principe . . . . .	25
1.9.2	Spécification . . . . .	25
1.9.3	Corps . . . . .	25

<b>2 Exercices</b>	<b>27</b>
2.1 Introduction au PL/SQL . . . . .	27
2.2 Tableaux et Structures . . . . .	28
2.3 Utilisation PL/SQL . . . . .	30
2.4 Exceptions . . . . .	31
2.5 Sous-programmes . . . . .	32
2.6 Curseurs . . . . .	33
2.7 Curseurs paramétrés . . . . .	34
2.8 Triggers . . . . .	35
2.9 Packages . . . . .	36
2.10 Révisions . . . . .	37
<b>3 Corrigés</b>	<b>38</b>
3.1 Introduction au PL/SQL . . . . .	38
3.2 Tableaux et Structures . . . . .	39
3.3 Application du PL/SQL et Exceptions . . . . .	42
3.4 Sous-programmes . . . . .	46
3.5 Curseurs . . . . .	49
3.6 Curseurs paramétrés . . . . .	52
3.7 Triggers . . . . .	53
3.8 Packages . . . . .	62
3.9 Révisions . . . . .	63
<b>A Scripts de création de bases</b>	<b>67</b>
A.1 Livraisons Sans contraintes . . . . .	67
A.2 Modules et prerequis . . . . .	68
A.3 Géométrie . . . . .	69
A.4 Livraisons . . . . .	70
A.5 Arbre généalogique . . . . .	71
A.6 Comptes bancaires . . . . .	72
A.7 Comptes bancaires avec exceptions . . . . .	74
A.8 Secrétariat pédagogique . . . . .	76
A.9 Mariages . . . . .	78

# Chapitre 1

## Notes de cours

### 1.1 Introduction au PL/SQL

#### 1.1.1 PL/SQL

Le PL de PL/SQL signifie Procedural Language. Il s'agit d'une extension procédurale du SQL permettant d'effectuer des traitements complexes sur une base de données. Les possibilités offertes sont les mêmes qu'avec des langages impératifs (instructions en séquence) classiques.

Ecrivez-le dans un éditeur dont vous copierez le contenu dans SQL+. Un script écrit en PL/SQL se termine obligatoirement par un /, sinon SQL+ ne l'interprète pas. S'il contient des erreurs de compilation, il est possible d'afficher les messages d'erreur avec la commande SQL+ : SHOW ERRORS.

#### 1.1.2 Blocs

Tout code écrit dans un langage procédural est formé de blocs. Chaque bloc comprend une section de déclaration de variables, et un ensemble d'instructions dans lequel les variables déclarées sont visibles.

La syntaxe est

```
DECLARE
    /* declaration de variables */
BEGIN
    /* instructions a executer */
END;
```

#### 1.1.3 Affichage

Pour afficher le contenu d'une variable, les procédures DBMS\_OUTPUT.PUT() et DBMS\_OUTPUT.PUT\_LINE() prennent en argument une valeur à afficher ou une variable dont la valeur est à afficher. Par défaut, les fonctions d'affichage sont désactivées. Il convient, à moins que vous ne vouliez rien voir s'afficher, de les activer avec la commande SQL+ SET SERVEROUTPUT ON.

#### 1.1.4 Variables

Une variable se déclare de la sorte :

```
nom type [:= initialisation] ;
```

L'initiation est optionnelle. Nous utiliserons les mêmes types primitifs que dans les tables. Par exemple :

```
SET SERVEROUTPUT ON
DECLARE
    c varchar2(15) := 'Hello World !';
BEGIN
    DBMS_OUTPUT.PUT_LINE(c);
END;
/
```

Les affectations se font avec la syntaxe `variable := valeur ;`

### 1.1.5 Traitements conditionnels

Le IF et le CASE fonctionnent de la même façon que dans les autres langages impératifs :

```
IF /* condition 1 */ THEN
    /* instructions 1 */
ELSE
    /* instructions 2 */
END IF;
```

voire

```
IF /* condition 1 */ THEN
    /* instructions 1 */
ELSIF /* condition 2 */
    /* instructions 2 */
ELSE
    /* instructions 3 */
END IF;
```

Les conditions sont les mêmes qu'en SQL. Le switch du langage C s'implémente en PL/SQL de la façon suivante :

```
CASE /* variable */
WHEN /* valeur 1 */ THEN
    /* instructions 1 */
WHEN /* valeur 2 */ THEN
    /* instructions 2 */
...
WHEN /* valeur n */ THEN
    /* instructions n */
ELSE
    /* instructions par défaut */
END CASE;
```

### 1.1.6 Traitements répétitifs

LOOP ... END LOOP ; permet d'implémenter les boucles

```
LOOP
    /* instructions */
END LOOP;
```

L'instruction EXIT WHEN permet de quitter une boucle.

```
LOOP
    /* instructions */
    EXIT WHEN /* condition */ ;
END LOOP;
```

La boucle FOR existe aussi en PL/SQL :

```
FOR /* variable */ IN /* inf */ .. /* sup */ LOOP
    /* instructions */
END LOOP;
```

Ainsi que la boucle WHILE :

```
WHILE /* condition */ LOOP
    /* instructions */
END LOOP;
```

Est-il possible, en bidouillant, d'implémenter une boucle DO ... WHILE ?

## 1.2 Tableaux et structures

### 1.2.1 Tableaux

#### Création d'un type tableau

Les types tableau doivent être définis explicitement par une déclaration de la forme

```
TYPE /* type */ IS VARRAY (/* taille */) OF /* typeElements */;
```

- `type` est le nom du type tableau créé par cette instruction
- `taille` est le nombre maximal d'éléments qu'il est possible de placer dans le tableau.
- `typeElements` est le type des éléments qui vont être stockés dans le tableau, il peut s'agir de n'importe quel type.

Par exemple, créons un type tableau de nombres indicé de 1 à 10, que nous appellerons `numberTab`

```
TYPE numberTab IS VARRAY (10) OF NUMBER;
```

#### Déclaration d'un tableau

Dorénavant, le type d'un tableau peut être utilisé au même titre que `NUMBER` ou `VARCHAR2`. Par exemple, déclarons un tableau appelé `t` de type `numberTab`,

```
DECLARE
    TYPE numberTab IS VARRAY (10) OF NUMBER;
    t numberTab;
BEGIN
    /* instructions */
END;
```

#### Allocation d'un tableau

La création d'un type tableau met à disposition un constructeur du même nom que le type créé. Cette fonction réserve de l'espace mémoire pour ce tableau et retourne l'adresse mémoire de la zone réservée, il s'agit d'une sorte de `malloc`. Si, par exemple, un type tableau `numtab` a été créé, la fonction `numtab()` retourne un tableau vide.

```
DECLARE
    TYPE numberTab IS VARRAY (10) OF NUMBER;
    t numberTab;
BEGIN
    t := numberTab();
    /* utilisation du tableau */
END;
```

Une fois cette allocation faite, il devient presque possible d'utiliser le tableau...

#### Dimensionnement d'un tableau

Le tableau retourné par le constructeur est vide. Il convient ensuite de réserver de l'espace pour stocker les éléments qu'il va contenir. On utilise pour cela la méthode `EXTEND()`. `EXTEND` s'invoque en utilisant la notation pointée. Par exemple,

```
DECLARE
    TYPE numberTab IS VARRAY (10) OF NUMBER;
    t numberTab;
BEGIN
    t := numberTab();
    t.EXTEND(4);
    /* utilisation du tableau */
END;
```

Dans cet exemple, `t.EXTEND(4)` ; permet par la suite d'utiliser les éléments du tableau `t(1)`, `t(2)`, `t(3)` et `t(4)`. Il n'est pas possible "d'étendre" un tableau à une taille supérieure à celle spécifiée lors de la création du type tableau associé.

### Utilisation d'un tableau

On accède, en lecture et en écriture, au  $i$ -ème élément d'une variable tabulaire nommé `T` avec l'instruction `T(i)`. Les éléments sont indicés à partir de 1.

Effectuons, par exemple, une permutation circulaire vers la droite des éléments du tableau `t`.

```
DECLARE
    TYPE numberTab IS VARRAY (10) OF NUMBER;
    t numberTab;
    i number;
    k number;
BEGIN
    t := numberTab();
    t.EXTEND(10);
    FOR i IN 1..10 LOOP
        t(i) := i;
    END LOOP;
    k := t(10);
    FOR i IN REVERSE 2..10 LOOP
        t(i) := t(i - 1);
    END LOOP;
    t(1) := k;
    FOR i IN 1..10 LOOP
        DBMS_OUTPUT.PUT_LINE(t(i));
    END LOOP;
END;
/
```

### 1.2.2 Structures

Un structure est un type regroupant plusieurs types. Une variable de type structuré contient plusieurs variables, ces variables s'appellent aussi des champs.

#### Création d'un type structuré

On définit un type structuré de la sorte :

```
TYPE /* nomType */ IS RECORD
(
    /* liste des champs */
);
```

`nomType` est le nom du type structuré construit avec la syntaxe précédente. La liste suit la même syntaxe que la liste des colonnes d'une table dans un `CREATE TABLE`. Par exemple, construisons le type `point` (dans  $\mathbb{R}^2$ ),

```
TYPE point IS RECORD
(
    abscisse NUMBER,
    ordonnee NUMBER
);
```

Notez bien que les types servant à définir un type structuré peuvent être quelconques : variables scalaires, tableaux, structures, etc.

#### Déclaration d'une variable de type structuré

`point` est maintenant un type, il devient donc possible de créer des variables de type `point`, la règle est toujours la même pour déclarer des variables en PL/SQL, par exemple

```
p point;
```

permet de déclarer une variable `p` de type `point`.

### Utilisation d'une variable de type structuré

Pour accéder à un champ d'une variable de type structuré, en lecture ou en écriture, on utilise la notation pointée : `v.c` est le champ appelé `c` de la variable structuré appelée `v`. Par exemple,

```
DECLARE
    TYPE point IS RECORD
    (
        abscisse NUMBER,
        ordonnee NUMBER
    );
    p point;
BEGIN
    p.abscisse := 1;
    p.ordonnee := 3;
    DBMS_OUTPUT.PUT_LINE( 'p.abscisse = ' || p.abscisse ||
        ' and p.ordonnee = ' || p.ordonnee );
END;
```

Le script ci-dessous crée le type `point`, puis crée une variable `t` de type `point`, et enfin affecte aux champs `abscisse` et `ordonnee` du point `p` les valeurs 1 et 3.



## 1.3 Utilisation du PL/SQL

Ce cours est une introduction aux interactions possibles entre la base de données et les scripts PL/SQL.

### 1.3.1 Affectation

On place dans une variable le résultat d'une requête en utilisant le mot-clé INTO. Les instructions

```
SELECT champ_1, ..., champ_n INTO v_1, ..., v_n
FROM ...
```

affecte aux variables `v_1`, ..., `v_n` les valeurs retournées par la requête. Par exemple

```
DECLARE
    num NUMBER;
    nom VARCHAR2(30) := 'Poupée Batman' ;
BEGIN
    SELECT numprod INTO num
           FROM PRODUIT
           WHERE nomprod = nom;
    DBMS_OUTPUT.PUT_LINE('L'article ' ||
                          nom || ' a pour numéro ' || num);
END;
/
```

Prêtez attention au fait que la requête doit retourner une et une seule ligne, sinon, une erreur se produit à l'exécution.

### 1.3.2 Tables et structures

Si vous ne tenez pas à vous prendre la tête pour choisir le type de chaque variable, demandez-vous ce que vous allez mettre dedans! Si vous tenez à y mettre une valeur qui se trouve dans une colonne d'une table, il est possible de vous référer directement au type de cette colonne avec le type `nomTable.nomColonne%type`. Par exemple,

```
DECLARE
    num PRODUIT.numprod%type;
    nom PRODUIT.nomprod%type := 'Poupée Batman' ;
BEGIN
    SELECT numprod INTO num
           FROM PRODUIT
           WHERE nomprod = nom;
    DBMS_OUTPUT.PUT_LINE('L'article ' ||
                          nom || ' a pour numéro ' || num);
END;
/
```

Pour aller plus loin, il est même possible de déclarer une structure pour représenter une ligne d'une table, le type porte alors le nom suivant : `nomTable%rowtype`.

```
DECLARE
    nom PRODUIT.nomprod%type := 'Poupée Batman' ;
    ligne PRODUIT%rowtype;
BEGIN
    SELECT * INTO ligne
           FROM PRODUIT
           WHERE nomprod = nom;
    DBMS_OUTPUT.PUT_LINE('L'article ' ||
                          ligne.nomprod || ' a pour numéro ' || ligne.numprod);
END;
/
```

### 1.3.3 Transactions

Un des mécanismes les plus puissants des SGBD récents réside dans le système des transactions. Une transaction est un ensemble d'opérations "atomiques", c'est-à-dire indivisible. Nous considérerons qu'un ensemble d'opérations est indivisible si une exécution partielle de ces instructions poserait des problèmes d'intégrité dans la base de données. Par exemple, dans le cas d'une base de données de gestion de comptes en banque, un virement d'un compte à un autre se fait en deux temps : créditer un compte d'une somme  $s$ , et débiter un autre de la même somme  $s$ . Si une erreur survient pendant la deuxième opération, et que la transaction est interrompue, le virement est incomplet et le patron va vous assassiner.

Il convient donc de disposer d'un mécanisme permettant de se protéger de ce genre de désagrément. Plutôt que se casser la tête à tester les erreurs à chaque étape et à balancer des instructions permettant de "revenir en arrière", nous allons utiliser les instructions `COMMIT` et `ROLLBACK`.

Voici le squelette d'un exemple :

```
/* instructions */
IF /* erreur */ THEN
    ROLLBACK;
ELSE
    COMMIT;
END;
```

Le `ROLLBACK` annule toutes les modifications faites depuis le début de la transaction (donc depuis le précédent `COMMIT`), `COMMIT` les enregistre définitivement dans la base de données.

La variable d'environnement `AUTOCOMMIT`, qui peut être positionnée à `ON` ou à `OFF` permet d'activer la gestion des transactions. Si elle est positionnée à `ON`, chaque instruction a des répercussions immédiates dans la base, sinon, les modifications ne sont effectives qu'une fois qu'un `COMMIT` a été exécuté.

## 1.4 Exceptions

Le mécanisme des `exceptions` est implémenté dans la plupart des langages récent, notamment orientés objet. Cette façon de programmer a quelques avantages immédiats :

- **obliger les programmeurs à traiter les erreurs** : combien de fois votre prof de C a hurlé en vous suppliant de vérifier les valeurs retournées par un `malloc`, ou un `fopen`? La plupart des compilateurs des langages à `exceptions` (notamment java) ne compilent que si pour chaque erreur potentielle, vous avez préparé un bloc de code (éventuellement vide...) pour la traiter. Le but est de vous assurer que vous n’avez pas oublié d’erreur.
- **Rattraper les erreurs en cours d’exécution** : Si vous programmez un système de sécurité de centrale nucléaire ou un pilote automatique pour l’aviation civile, une erreur de mémoire qui vous afficherait l’écran bleu de windows, ou le message “Envoyer le rapport d’erreur?”, ou plus simplement le fameux “Segmentation fault” produirait un effet des plus mauvais. Certaines erreurs d’exécution sont rattrapables, autrement dit, il est possible de résoudre le problème sans interrompre le programme.
- **Ecrire le traitement des erreurs à part** : Pour des raisons fiabilité, de lisibilité, il a été considéré que mélanger le code “normal” et le traitement des erreurs était un style de programmation perfectible... Dans les langages à exception, les erreurs sont traitées à part.

### 1.4.1 Rattraper une exception

Je vous ai menti dans le premier cours, un bloc en PL/SQL a la forme suivante :

```
DECLARE
    /* declarations */
BEGIN
    /* instructions */
EXCEPTION
    /* traitement des erreurs */
END;
```

Une exception est une “erreur type”, elle porte un nom, au même titre qu’une variable a une identificateur, par exemple `GLUBARF`. Lorsque dans les instructions, l’erreur `GLUBARF` se produit, le code du `BEGIN` s’interrompt et le code de la section `EXCEPTION` est lancé. On dit aussi que quand une exception est **levée** (raised) (on dit aussi **jetée** (thrown)), on la **rattrape** (catch) dans le bloc `EXCEPTION`. La section `EXCEPTION` a la forme suivante :

```
EXCEPTION
    WHEN E1 THEN
        /* traitement */
    WHEN E2 THEN
        /* traitement */
    WHEN E3 THEN
        /* traitement */
    WHEN OTHERS THEN
        /* traitement */
END;
```

On énumère les erreurs les plus pertinentes en utilisant leur nom et en consacrant à chacune d’elle un traitement particulier pour rattraper (ou propager) l’erreur. Quand un bloc est traité, les `WHEN` suivants ne sont pas évalués. `OTHERS` est l’exception par défaut, `OTHERS` est toujours vérifié, sauf si un cas précédent a été vérifié. Dans l’exemple suivant :

```
DECLARE
    /* declarations */
BEGIN
    /* instructions */
    COMMIT;
EXCEPTION
    WHEN GLUBARF THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE( 'GLUBARF exception raised!' );
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE( 'SQLCODE = ' || SQLCODE );
        DBMS_OUTPUT.PUT_LINE( 'SQLERRM = ' || SQLERRM );
```

END;

Les deux variables globales `SQLCODE` et `SQLERRM` contiennent respectivement le code d'erreur Oracle et un message d'erreur correspondant à la dernière exception levée. Chaque exception a donc, en plus d'un nom, un code et un message.

### 1.4.2 Exceptions prédéfinies

Bon nombre d'exceptions sont prédéfinies par Oracle, par exemple

- `NO_DATA_FOUND` est levée quand la requête d'une instruction de la forme `SELECT ... INTO ...` ne retourne aucune ligne
  - `TOO_MANY_ROWS` est levée quand la requête d'une instruction de la forme `SELECT ... INTO ...` retourne plusieurs lignes
  - `DUP_VAL_ON_INDEX` est levée si une insertion (ou une modification) est refusée à cause d'une contrainte d'unicité.
- On peut enrichir notre exemple de la sorte :

```
DECLARE
    num NUMBER;
    nom VARCHAR2(30) := 'Poupée Batman' ;
BEGIN
    SELECT numprod INTO num
           FROM PRODUIT
           WHERE nomprod = nom;
    DBMS_OUTPUT.PUT_LINE('L'article ' ||
                          nom || ' a pour numéro ' || num);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Aucun article ne porte le nom '
                              || nom);
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Plusieurs articles portent le nom '
                              || nom);
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Il y a un gros problème...');
END;
/
```

`SELECT numprod INTO num...` lève une exception si la requête renvoie un nombre de lignes différent de 1.

### 1.4.3 Codes d'erreur

Je vous en ai menti, certaines exceptions n'ont pas de nom. Elles ont seulement un code d'erreur, il est conseillé de se reporter à la documentation pour les obtenir. On les traite de la façon suivante

```
EXCEPTION
    WHEN OTHERS THEN
        IF SQLCODE = CODE1 THEN
            /* traitement */
        ELSIF SQLCODE = CODE2 THEN
            /* traitement */
        ELSE
            DBMS_OUTPUT.PUT_LINE('J''vois pas c''que ca
                                  peut etre...');
END;
```

C'est souvent le cas lors de violation de contraintes.

### 1.4.4 Déclarer et lancer ses propres exceptions

Exception est un type, on déclare donc les exceptions dans une section `DECLARE`. Une exception se lance avec l'instruction `RAISE`. Par exemple,

```
DECLARE
    GLUBARF EXCEPTION;
BEGIN
    RAISE GLUBARF;
EXCEPTION
    WHEN GLUBARF THEN
        DBMS_OUTPUT.PUT_LINE('glubarf raised. ');
END;
/
```

## 1.5 Sous-programmes

### 1.5.1 Procédures

#### Syntaxe

On définit une procédure de la sorte

```
CREATE OR REPLACE PROCEDURE /* nom */ (/* parametres */) IS
    /* declaration des variables locales */
BEGIN
    /* instructions */
END;
```

les paramètres sont une simple liste de couples `nom type`. Par exemple, la procédure suivante affiche un compte à rebours.

```
CREATE OR REPLACE PROCEDURE compteAREbours (n NUMBER) IS
BEGIN
    IF n >= 0 THEN
        DBMS_OUTPUT.PUT_LINE(n);
        compteAREbours(n - 1);
    END IF;
END;
```

#### Invocation

En PL/SQL, une procédure s'invoque tout simplement avec son nom. Mais sous SQL+, on doit utiliser le mot-clé CALL. Par exemple, on invoque le compte à rebours sous SQL+ avec la commande `CALL compteAREbours(20)`.

#### Passage de paramètres

Oracle permet le passage de paramètres par référence. Il existe trois types de passage de paramètres :

- IN : passage par valeur
- OUT : aucune valeur passée, sert de valeur de retour
- IN OUT : passage de paramètre par référence

Par défaut, le passage de paramètre se fait de type IN.

```
CREATE OR REPLACE PROCEDURE incr (val IN OUT NUMBER) IS
BEGIN
    val := val + 1;
END;
```

### 1.5.2 Fonctions

#### Syntaxe

On crée une nouvelle fonction de la façon suivante :

```
CREATE OR REPLACE FUNCTION /* nom */ (/* parametres */) RETURN /* type */ IS
    /* declaration des variables locales */
BEGIN
    /* instructions */
END;
```

L'instruction RETURN sert à retourner une valeur. Par exemple,

```
CREATE OR REPLACE FUNCTION module (a NUMBER, b NUMBER) RETURN NUMBER IS
BEGIN
    IF a < b THEN
        RETURN a;
    ELSE
```

```
                RETURN module(a - b, b);  
    END IF;  
END;
```

### Invocation

Tout comme les procédures, l'invocation des fonctions ne pose aucun problème en PL/SQL, par contre, sous SQL+, c'est quelque peu particulier. On passe par une pseudo-table nommée DUAL de la façon suivante :

```
SELECT module(21, 12) FROM DUAL;
```

### Passage de paramètres

Les paramètres sont toujours passés avec le type IN.

## 1.6 Curseurs

### 1.6.1 Introduction

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

### 1.6.2 Les curseurs

Un curseur est un objet contenant le résultat d'une requête (0, 1 ou plusieurs lignes).

#### déclaration

Un curseur se déclare dans une section `DECLARE` :

```
CURSOR /* nomcurseur */ IS /* requête */;
```

Par exemple, si on tient à récupérer tous les employés de la table `EMP`, on déclare le curseur suivant.

```
CURSOR emp_cur IS  
    SELECT * FROM EMP ;
```

#### Ouverture

Lors de l'ouverture d'un curseur, la requête du curseur est évaluée, et le curseur contient toutes les données retournées par la requête. On ouvre un curseur dans une section `BEGIN` :

```
OPEN /* nomcurseur */;
```

Par exemple,

```
DECLARE  
    CURSOR emp_cur IS  
        SELECT * FROM EMP ;  
BEGIN  
    OPEN emp_cur ;  
    /* Utilisation du curseur */  
END;
```

#### Lecture d'une ligne

Une fois ouvert, le curseur contient toutes les lignes du résultat de la requête. On les récupère une par une en utilisant le mot-clé `FETCH` :

```
FETCH /* nom_curseur */ INTO /* liste_variables */;
```

La liste de variables peut être remplacée par une structure de type `nom_curseur%ROWTYPE`. Si la lecture de la ligne échoue, parce qu'il n'y a plus de ligne à lire, l'attribut `%NOTFOUND` prend la valeur vrai.

```
DECLARE  
    CURSOR emp_cur IS  
        SELECT * FROM EMP ;  
    ligne emp_cur%rowtype  
BEGIN  
    OPEN emp_cur ;  
    LOOP  
        FETCH emp_cur INTO ligne ;  
        EXIT WHEN emp_cur%NOTFOUND ;  
        DBMS_OUTPUT.PUT_LINE(ligne.ename) ;  
    END LOOP ;  
    /* ... */  
END;
```



## Fermeture

Après utilisation, il convient de fermer le curseur.

```
CLOSE /* nomcurseur */;
```

Complétons notre exemple,

```
DECLARE
    CURSOR emp_cur IS
        SELECT * FROM EMP;
    ligne emp_cur%rowtype;
BEGIN
    OPEN emp_cur;
    LOOP
        FETCH emp_cur INTO ligne;
        EXIT WHEN emp_cur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(ligne.ename);
    END LOOP;
    CLOSE emp_cur;
END;
/
```

Le programme ci-dessus peut aussi s'écrire

```
DECLARE
    CURSOR emp_cur IS
        SELECT * FROM EMP;
    ligne emp_cur%rowtype;
BEGIN
    OPEN emp_cur;
    FETCH emp_cur INTO ligne;
    WHILE emp_cur%FOUND LOOP
        DBMS_OUTPUT.PUT_LINE(ligne.ename);
        FETCH emp_cur INTO ligne;
    END LOOP;
    CLOSE emp_cur;
END;
```

## Boucle FOR

Il existe une boucle FOR se chargeant de l'ouverture, de la lecture des lignes du curseur et de sa fermeture,

```
FOR ligne IN emp_cur LOOP
    /* Traitement */
END LOOP;
```

Par exemple,

```
DECLARE
    CURSOR emp_cur IS
        SELECT * FROM EMP;
    ligne emp_cur%rowtype;
BEGIN
    FOR ligne IN emp_cur LOOP
        DBMS_OUTPUT.PUT_LINE(ligne.ename);
    END LOOP;
END;
/
```

## 1.7 Curseurs paramétrés

### 1.7.1 Introduction

A votre avis, le code suivant est-il valide ?

```
DECLARE
    NUMBER n := 14;
BEGIN
    DECLARE
        CURSOR C IS
            SELECT *
            FROM PERSONNE
            WHERE numpers >= n;
        ROW C%rowType;
    BEGIN
        FOR ROW IN C LOOP
            DBMS_OUTPUT.PUT_LINE(ROW.numpers);
        END LOOP;
    END;
END;
```

Réponse : non. La requête d'un curseur ne peut pas contenir de variables dont les valeurs ne sont pas fixées. Pourquoi ? Parce que les valeurs de ces sont susceptibles de changer entre la déclaration du curseur et son ouverture. Le remède est un curseur paramétré.

### 1.7.2 Définition

Un curseur paramétré est un curseur dont la requête contient des variables dont les valeurs ne seront fixées qu'à l'ouverture.

### 1.7.3 Déclaration

On précise la liste des noms et des type des paramètres entre parenthèses après le nom du curseur :

```
CURSOR /* nom */ ( /* liste des paramètres */ ) IS
    /* requête */
```

Par exemple, créons une requête qui, pour une personne donnée, nous donne la liste des noms et prénoms de ses enfants :

```
CURSOR enfants (numparent NUMBER) IS
    SELECT *
    FROM PERSONNE
    WHERE pere = numparent
    OR mere = numparent;
```

### 1.7.4 Ouverture

On ouvre un curseur paramétré en passant en paramètre les valeurs des variables :

```
OPEN /* nom */ ( /* liste des paramètres */ )
```

Par exemple,

```
OPEN enfants (1);
```

### 1.7.5 Lecture d'une ligne, fermeture

la lecture d'une ligne suit les mêmes règles qu'avec un curseur non paramétré.

### 1.7.6 Boucle pour

La boucle pour se charge de l'ouverture, il convient donc de placer les paramètre dans l'entête de la boucle,

```
FOR /* variable */ IN /* nom */ (/* liste paramètres */) LOOP
    /* instructions */
END LOOP;
```

Par exemple,

```
FOR e IN enfants(1) LOOP
    DBMS_OUTPUT.PUT_LINE(e.nompers || ' ' || e.prenompers);
END LOOP;
```

### 1.7.7 Exemple récapitulatif

```
DECLARE
    CURSOR parent IS
        SELECT *
        FROM PERSONNE;
    p parent%rowtype;
    CURSOR enfants (numparent NUMBER) IS
        SELECT *
        FROM PERSONNE
        WHERE pere = numparent
        OR mere = numparent;
    e enfants%rowtype;
BEGIN
    FOR p IN parent LOOP
        DBMS_OUTPUT.PUT_LINE('Les enfants de ' || p.prenom ||
            ' ' || p.nom || ' sont : ');
        FOR e IN enfants(p.numbers) LOOP
            DBMS_OUTPUT.PUT_LINE(' * ' || e.prenom
                || ' ' || e.nom );
        END LOOP;
    END LOOP;
END;
```

[www.Mcours.com](http://www.Mcours.com)

Site N°1 des Cours et Exercices

Email: [mymcours@gmail.com](mailto:mymcours@gmail.com)

## 1.8 Triggers

### 1.8.1 Principe

Un trigger est une procédure stockée qui se lance automatiquement lorsqu'un événement se produit. Par événement, on entend dans ce cours toute modification des données se trouvant dans les tables. On s'en sert pour contrôler ou appliquer des contraintes qu'il est impossible de formuler de façon déclarative.

### 1.8.2 Classification

#### Type d'événement

Lors de la création d'un trigger, il convient de préciser quel est le type d'événement qui le déclenche. Nous réaliserons dans ce cours des triggers pour les événements suivants :

- INSERT
- DELETE
- UPDATE

#### Moment de l'exécution

On précise aussi si le trigger doit être exécuté avant (BEFORE) ou après (AFTER) l'événement.

#### Événements non atomiques

Lors que l'on fait un DELETE ..., il y a une seule instruction, mais plusieurs lignes sont affectées. Le trigger doit-il être exécuté pour chaque ligne affectée (FOR EACH ROW), ou seulement une fois pour toute l'instruction (STATEMENT) ?

- un FOR EACH ROW TRIGGER est exécuté à chaque fois qu'une ligne est affectée.
- un STATEMENT TRIGGER est exécutée à chaque fois qu'une instruction est lancée.

### 1.8.3 Création

#### Syntaxe

On déclare un trigger avec l'instruction suivante :

```
CREATE OR REPLACE TRIGGER nomtrigger
[BEFORE | AFTER] [INSERT | DELETE | UPDATE] ON nomtable
[FOR EACH ROW | ]
DECLARE
    /* declarations */
BEGIN
    /* instructions */
END;
```

Par exemple,

```
SQL> CREATE OR REPLACE TRIGGER pasDeDeleteDansClient
2 BEFORE DELETE ON CLIENT
3 BEGIN
4 RAISE_APPLICATION_ERROR(-20555, 'Va te faire ...');
5 END;
6 /
```

Déclencheur créé.

```
SQL> SELECT COUNT(*)
2 FROM CLIENT;
```

```
COUNT(*)
```

```
-----
21
```

```
SQL> DELETE FROM CLIENT;
```

```

DELETE FROM CLIENT
      *
ERREUR à la ligne 1 :
ORA-20555: Va te faire ...
ORA-06512: à "SCOTT.PASDELETEDANSCLIENT", ligne 2
ORA-04088: erreur lors d exécution du déclencheur 'SCOTT.PASDELETEDANSCLIENT'

SQL> SELECT COUNT(*)
      2 FROM CLIENT;

COUNT(*)
-----
         21

```

L'instruction `RAISE_APPLICATION_ERROR(code, message)` lève une exception sans nom portant un code `code` et un message d'erreur `message`. Vous remarquez que comme l'erreur a été levée avant la suppression, les données sont toujours présentes dans la table `CLIENT`. Le trigger a contrôlé une règle, et comme elle n'était pas respectée, il a lancé une erreur.

### Combinaisons d'événements

Il est possible, en séparant les types d'événement par le mot-clé `OR`, de définir un trigger déclenché par plusieurs événements. Les variables booléennes `INSERTING`, `UPDATING` et `DELETING` permettent d'identifier l'événement qui a déclenché le trigger.

```

CREATE OR REPLACE TRIGGER afficheEvenement
BEFORE INSERT OR UPDATE OR DELETE ON CLIENT
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('Insertion dans CLIENT');
    ELSIF UPDATING THEN
        DBMS_OUTPUT.PUT_LINE('Mise a jour dans CLIENT');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Suppression dans CLIENT');
    END IF;
END;

```

### 1.8.4 Accès aux lignes en cours de modification

Dans les `FOR EACH ROW` triggers, il est possible avant la modification de chaque ligne, de lire l'ancienne ligne et la nouvelle ligne par l'intermédiaire des deux variables structurées `:old` et `:new`. Par exemple le trigger suivant empêche de diminuer un salaire :

```

CREATE OR REPLACE TRIGGER pasDeBaisseDeSalaire
BEFORE UPDATE ON EMP
FOR EACH ROW
BEGIN
    IF (:old.sal > :new.sal) THEN
        RAISE_APPLICATION_ERROR(-20567,
            'Pas de baisse de salaire !');
    END IF;
END;

```

### Tables en mutation

Il est impossible, dans un trigger de type `FOR EACH ROW` de faire un `SELECT` sur la table en cours de modification.

```
SQL> CREATE OR REPLACE TRIGGER beforeStatement
  2 BEFORE UPDATE ON CLIENT
  3 DECLARE
  4   NB NUMBER;
  5 BEGIN
  6   SELECT COUNT(*) INTO NB
  7   FROM CLIENT;
  8 END;
  9 /
```

Déclencheur créé.

```
SQL>
SQL> CREATE OR REPLACE TRIGGER afterStatement
  2 AFTER UPDATE ON CLIENT
  3 DECLARE
  4   NB NUMBER;
  5 BEGIN
  6   SELECT COUNT(*) INTO NB
  7   FROM CLIENT;
  8 END;
  9 /
```

Déclencheur créé.

```
SQL>
SQL> UPDATE CLIENT SET nomcli = nomcli;
```

21 ligne(s) mise(s) à jour.

```
SQL>
SQL> CREATE OR REPLACE TRIGGER beforeForEachRow
  2 BEFORE UPDATE ON CLIENT
  3 FOR EACH ROW
  4 DECLARE
  5   NB NUMBER;
  6 BEGIN
  7   SELECT COUNT(*) INTO NB
  8   FROM CLIENT;
  9 END;
 10 /
```

Déclencheur créé.

```
SQL>
SQL> UPDATE CLIENT SET nomcli = nomcli;
UPDATE CLIENT SET nomcli = nomcli
```

```
*
ERREUR à la ligne 1 :
ORA-04091: la table SCOTT.CLIENT est en mutation ; le déclencheur ou la
fonction ne peut la voir
ORA-06512: à "SCOTT.BEFOREFOREACHROW", ligne 4
ORA-04088: erreur lors d exécution du déclencheur 'SCOTT.BEFOREFOREACHROW'
```

```
SQL> DROP TRIGGER beforeForEachRow;
```

Déclencheur supprimé.

```

SQL>
SQL>
SQL> CREATE OR REPLACE TRIGGER afterForEachRow
  2 AFTER UPDATE ON CLIENT
  3 FOR EACH ROW
  4 DECLARE
  5   NB NUMBER;
  6 BEGIN
  7   SELECT COUNT(*) INTO NB
  8   FROM CLIENT;
  9 END;
10 /

```

Déclencheur créé.

```

SQL>
SQL> UPDATE CLIENT SET nomcli = nomcli;
UPDATE CLIENT SET nomcli = nomcli
      *
ERREUR à la ligne 1 :
ORA-04091: la table SCOTT.CLIENT est en mutation ; le déclencheur ou la
fonction ne peut la voir
ORA-06512: à "SCOTT.AFTERFOREACHROW", ligne 4
ORA-04088: erreur lors d exécution du déclencheur 'SCOTT.AFTERFOREACHROW'

```

### 1.8.5 Contourner le problème des tables en mutation

Il existe plusieurs façons de contourner ce problème :

- Utiliser un STATEMENT trigger. Comme on ne sait pas quelles lignes ont été modifiées, on est obligé de toutes les traiter. Cette approche présente donc un inconvénient majeur : elle nous amène à effectuer de nombreux traitements inutiles.
- En ayant des données redondantes. Il suffit que les données servant à la vérification se trouvent dans une autre table que celle en mutation. Cette méthode a pour inconvénient la mémoire occupée et la quantité de code à écrire pour maintenir la cohérence des données. Dans la plupart des cas, cette solution est malgré tout la meilleure.

#### Colonnes supplémentaires

Par exemple, si l'on souhaite empêcher un client d'avoir plus de 10 comptes en banque, une solution est de placer dans la table client une colonne contenant le nombre de comptes.

```

ALTER TABLE CLIENT ADD nbComptes number;
UPDATE CLIENT SET nbComptes = 0;

```

Une fois cette table créée, il convient de s'assurer que les données de la colonne nbComptes contient toujours les bonnes valeurs. On le fait avec plusieurs sous-programmes :

```

CREATE OR REPLACE TRIGGER metAJourNbComptes
AFTER INSERT OR UPDATE OR DELETE ON COMPTECLIENT
BEGIN
    UPDATE CLIENT SET nbComptes =
        (
            SELECT COUNT(*)
            FROM COMPTECLIENT CC
            WHERE CC.numCli = numCli
        );
END;
/

CREATE OR REPLACE TRIGGER verifieNbComptes

```

```

BEFORE INSERT ON COMPTECLIENT
FOR EACH ROW
DECLARE
    nbComptes NUMBER;
BEGIN
    SELECT nbComptes INTO nbComptes
    FROM CLIENT
    WHERE numCli = :new.numcli;
    IF (nbComptes >= 10) THEN
        RAISE_APPLICATION_ERROR(-20556,
            'Ce client a deja trop de comptes');
    END IF;
END;
/

```

On peut affiner en remplaçant `metaAJourNbComptes` par plusieurs sous-programmes :

```

CREATE OR REPLACE TRIGGER initialiseNbComptes
BEFORE INSERT ON CLIENT
FOR EACH ROW
BEGIN
    :new.nbComptes := 0;
END;
/

CREATE OR REPLACE TRIGGER metaAJourNbComptes
AFTER INSERT OR UPDATE OR DELETE ON COMPTECLIENT
FOR EACH ROW
BEGIN
    IF DELETING OR UPDATING THEN
        UPDATE CLIENT SET nbComptes = nbComptes - 1
        WHERE numcli = :old.numcli;
    END IF;
    IF INSERTING OR UPDATING THEN
        UPDATE CLIENT SET nbComptes = nbComptes + 1
        WHERE numcli = :new.numcli;
    END IF;
END;
/

```

## Tables supplémentaires

Si l'on souhaite par exemple empêcher les circuits dans la table `PERSONNE`, il est nécessaire de faire un parcours de graphe. Ce qui nécessite des `SELECT` dans la table en cours de mutation. La seule solution est dans ce cas d'avoir une table miroir qui contient les colonnes clés primaire et étrangères de cette table, et de s'en servir pour détecter les circuits.

```

CREATE TABLE MIRRORPERSONNE
(
    numpers NUMBER PRIMARY KEY,
    pere NUMBER,
    mere NUMBER
);

```

Nous allons ensuite procéder de même, en répercutant chaque opération de `PERSONNE` sur `MIRRORPERSONNE`.

```

CREATE OR REPLACE TRIGGER miseAJourMirrorPersonne
BEFORE UPDATE OR INSERT OR DELETE ON PERSONNE
FOR EACH ROW
BEGIN
    IF DELETING OR UPDATING THEN

```



```

        DELETE FROM MIRRORPERSONNE
        WHERE numpers = :old.numpers;
    END IF;
    IF INSERTING OR UPDATING THEN
        INSERT INTO MIRRORPERSONNE VALUES
            (:new.numpers, :new.pere, :new.mere);
    END IF;
END;
/

```

Une fois cela fait, il suffit de rechercher si une personne insérée est une descendante d'elle-même dans MIRRORPERSONNE.

```

CREATE OR REPLACE FUNCTION trouveCircuit(current NUMBER, toFind NUMBER)
RETURN BOOLEAN IS
    numPere NUMBER;
    numMere NUMBER;
BEGIN
    IF (current IS NULL) THEN
        RETURN FALSE;
    END IF;
    SELECT pere, mere INTO numPere, numMere
        FROM MIRRORPERSONNE
        WHERE numPers = current;
    RETURN (numPere = toFind OR numMere = toFind OR
        trouveCircuit(numPere, toFind) OR
        trouveCircuit(numMere, toFind));
END;
/

CREATE OR REPLACE TRIGGER verifieCircuit
AFTER UPDATE OR INSERT ON PERSONNE
FOR EACH ROW
BEGIN
    IF (trouveCircuit(:new.numPers, :new.numPers)) THEN
        RAISE_APPLICATION_ERROR(-20557,
            'Circuit dans l''arbre généalogique. ');
    END IF;
END;
/

```

## 1.9 Packages

### 1.9.1 Principe

Un package est un ensemble de sous-programmes et de variables formé par

- Une spécification : déclaration de variables et de sous-programmes
- Un corps : implémentation des sous-programmes

Tout ce qui se trouve dans la spécification doit se trouver dans le corps, mais la réciproque est fausse. Un package satisfait les points suivants :

- **encapsulation** : certains traitements sont masqués, seule la spécification du package est visible. Cela a pour avantage de simplifier la tâche de celui qui va utiliser le package.
- **modularité** : il est possible de développer séparément les diverses parties de l'application. le développement devient ainsi un assemblage de package.

Ces deux aspects fournissent une souplesse certaine au niveau du développement : il est possible de modifier le corps d'un package sans changer sa spécification, donc sans modifier le fonctionnement de l'application.

### 1.9.2 Spécification

La syntaxe permettant de créer l'entête est la suivante :

```
CREATE OR REPLACE PACKAGE nompackage IS
/*
    declarations
*/
END nomPackage;
/
```

Par exemple,

```
CREATE OR REPLACE PACKAGE compteur IS
    procedure reset;
    function nextValue return number;
END compteur;
/
```

### 1.9.3 Corps

La syntaxe permettant de créer le corps est la suivante :

```
CREATE OR REPLACE PACKAGE BODY nompackage IS
/*
    implementation
*/
END nomPackage;
/
```

Par exemple,

```
CREATE OR REPLACE PACKAGE BODY compteur IS
    cpt NUMBER := 0;

    PROCEDURE reset IS
    BEGIN
        cpt := 0;
    END;

    FUNCTION nextValue RETURN NUMBER IS
    BEGIN
        cpt := cpt + 1;
        RETURN cpt - 1;
    END;
END compteur;
```

/

On peut utiliser un package depuis n'importe quel script PL/SQL :

```
DECLARE
  nb NUMBER;
BEGIN
  FOR nb IN 4..20 LOOP
    DBMS_OUTPUT.PUT_LINE (COMPTEUR.nextValue ());
  END LOOP;
  COMPTEUR.RESET ();
  FOR nb IN REVERSE 0..10 LOOP
    DBMS_OUTPUT.PUT_LINE (COMPTEUR.nextValue ());
  END LOOP;
END;
```

/

# Chapitre 2

## Exercices

### 2.1 Introduction au PL/SQL

#### Exercice 1

Ecrivez un programme affectant les valeurs 1 et 2 à deux variables  $a$  et  $b$ , puis permutant les valeurs de ces deux variables.

#### Exercice 2

Ecrivez un programme plaçant la valeur 10 dans une variable  $a$ , puis affichant la factorielle de  $a$ .

#### Exercice 3

Ecrivez un programme plaçant les valeurs 48 et 84 dans deux variables  $a$  et  $b$  puis affichant le *pgcd* de  $a$  et  $b$ .

## 2.2 Tableaux et Structures

### Exercice 1

1. Créez un type tableau pouvant contenir jusqu'à 50 entiers.
2. Créez une variable de ce type , faites une allocation dynamique et dimensionnez ce tableau à 20 emplacements.
3. Placez dans ce tableau la liste des 20 premiers carrés parfaits : 1, 4, 9, 16, 25, ...
4. Inversez l'ordre des éléments du tableau
5. Affichez le tableau.

### Exercice 2

Triez le tableau précédent avec la méthode du tri à bulle.

### Exercice 3

Recherchez, par dichotomie, si l'élément 225 se trouve dans le tableau.

### Exercice 4

On implémente des listes chaînées avec des tableaux de la sorte,

```
SET SERVEROUTPUT ON
DECLARE
-- Maillon d'une liste chaînée
TYPE CELL IS RECORD
(
-- Donnée de chaque maillon
data INTEGER,
-- Indice du maillon précédent de la liste,
-- -1 s'il n'y en a pas
previous INTEGER,
-- Indice du maillon suivant de la liste,
-- -1 s'il n'y en a pas
next INTEGER
);
-- Type tableau contenant les maillons de la liste
TYPE TREE IS VARRAY (19) OF CELL;
-- Tableau contenant les maillons de la liste
t TREE;
-- indice du premier élément de la liste
first integer;
-- indice du dernier élément de la liste
last integer;
BEGIN
t := TREE();
t.extend(19);

-- Initialisation
FOR i IN 1..19 LOOP
t(i).data := power(i, 5) mod 19 ;
t(i).previous := i-1;
t(i).next := i+1;
END LOOP;
first := 1;
last := 19;
t(first).previous := -1;
t(last).next := -1;
```

```

-- Affichage
DECLARE
p integer := first;
BEGIN
WHILE p <> -1 LOOP
DBMS_OUTPUT.PUT_LINE(' ' || p || ' ', ' ' ||
                    t(p).data || ' ', ' ' ||
t(p).previous || ' ', ' ' || t(p).next || ' ');
p := t(p).next;
END LOOP;
END;
        /* Ecrivez la suite vous-même... */
END;
/

```

Inversez l'ordre des éléments de la liste, sans changer les indices des maillons (seulement en modifiant le chaînage).

### Exercice 5

Utilisez le tri à bulle pour remettre les éléments dans l'ordre. Les indications sont les mêmes : ne déplacez pas les maillons, vous n'avez le droit de toucher qu'au chaînage. Bon courage, l'aspirine n'est pas fournie.



## 2.3 Utilisation PL/SQL

Nous travaillerons sur les données A.6 et A.5.

Vous n'oubliez pas de placer des `commit` en des lieux bien choisis.

### Exercice 1

Vous remarquerez que les valeurs des `numpers` de la table `PERSONNE` forment une séquence de nombres de 1 à 21. Utilisez une boucle dans laquelle vous placerez une requête pour recopier les couples nom/prénom de la table `PERSONNE` dans la table `CLIENT`.

### Exercice 2

Ecrivez un script récupérant le client de clé primaire la plus élevée, et injectant ce client dans la table `PERSONNEL`.

### Exercice 3

Ouvrez un compte courant pour chaque personne, effectuez un dépôt en espèce égal à  $numpers * 100$  euros.

### Exercice 4

Ouvrez un livret pour chaque personne ayant un `numpers` pair, faites un virement de leur compte courant vers ce livret de sorte qu'il ne reste plus que 500 sur leur compte.

## 2.4 Exceptions

Nous utiliserons les données de A.7 et A.5

Vous êtes invités à modifier le code de la séance précédente. Chaque fois qu'un `SELECT ... INTO ...` sera effectué, vous rattraperez les exceptions `NO_DATA_FOUND` et `TOO_MANY_ROWS`. A chaque insertion, vous rattraperez l'exception `DUP_VAL_ON_INDEX`.

### Exercice 1

Faites de sorte que les scripts important les données des tables `CLIENT` ne puissent être exécutés qu'une seule fois.

### Exercice 2

Les scripts remplissant la table `Operation` ne fonctionneront pas aujourd'hui... Même s'il fonctionnaient la dernière fois. Trouvez les codes d'erreurs des exceptions levées par ces scripts, rattrapez-les de la façon la plus appropriée qui soit.



## 2.5 Sous-programmes

### Exercice 1

Ecrire une fonction récursive retournant  $b^n$ , avec  $n$  entier positif ou nul.

### Exercice 2

Améliorer la fonction précédente en utilisant le fait que

$$b^n = (b^2)^{\frac{n}{2}}$$

si  $n$  est pair.

Pour les questions suivantes, utilisez les données de A.5.

### Exercice 3

Ecrire une fonction `demi-freres` prenant deux numéros de personnes en paramètre et retournant vrai si et seulement si ces deux personnes ont un parent en commun.

### Exercice 4

Ecrire une fonction `cousins germain` prenant deux numéros de personnes en paramètre et retournant vrai si et seulement si ces deux individus sont cousins germain.

### Exercice 5

Ecrire une procédure récursive affichant le nom de la personne dont le numéro est passé en paramètre et se rappelant récursivement sur le père de cette personne. Faites de sorte à ne pas utiliser d'exceptions.

### Exercice 6

Ecrire une procédure récursive affichant les noms des ascendants de sexe masculin de la personne dont le numéro est passé en paramètre.

### Exercice 7

Ecrire une fonction récursive prenant deux numéros de personne  $A$  et  $B$  et retournant vrai si  $A$  est un ascendant de  $B$ .

### Exercice 8

Ecrire une fonction prenant en paramètre deux numéros de personne  $A$  et  $B$  et retournant, si l'un est un ascendant de l'autre, le nombre de générations les séparant,  $-1$  si l'un n'est pas un ascendant de l'autre.

### Exercice 9

Préparez un verre d'aspirine et écrivez une requête retournant le(s) couples(s) personnes séparées par le plus de générations.

### Exercice 10

Reprendre le code du tp précédent, le découper en sous-programmes de la façon la moins inintelligente possible. Bon courage.

## 2.6 Curseurs

### Exercice 1

Refaites les exercices de 2.3 en utilisant les curseurs.

### Exercice 2

En utilisant les données A.5, écrivez une fonction affichant toute la descendance d'une personne.

## 2.7 Curseurs paramétrés

L'intérêt de ces exercices étant de vous familiariser avec les curseurs paramétrés, vous ferez en sorte de ne pas contourner leur usage. Nous utiliserons les données de A.6

### Exercice 1

Ecrire une procédure qui affiche tous les clients, et pour chaque client, la liste des comptes.

### Exercice 2

Ecrire une procédure qui affiche tous les clients, et pour chaque client, la liste des comptes, et pour chacun de ces comptes, l'historique des opérations.

## 2.8 Triggers

Implémentez les contraintes suivantes dans les données de les données de A.8. Vous ferez des sous-programmes tenant sur une page, et ne contenant pas plus de trois niveaux d'imbrication. Vous répertorierez les numéros d'erreurs que vous affecterez à chaque levée d'exception.

1. Il ne doit pas être possible de modifier la note min dans la table **prerequis**.
2. Dans un module, il ne doit pas y avoir plus de **effecMax** élèves inscrits.
3. On ne peut créer un examen pour un module que s'il y a des élèves inscrits dans ce module.
4. Un élève ne peut passer un examen que si sa date d'inscription est antérieure à la date de l'examen.
5. Il ne doit pas y avoir de circuit dans la table **prerequis** (il existe une façon de la vérifier en PL/SQL, mais comme vous ne la connaissez pas, faites un parcours en profondeur du graphe des pré-requis)
6. Un élève s'inscrivant à un module doit avoir eu au moins la note min à tous les modules pré-requis.
7. Ajouter dans étudiant un champ moyenne, celui-ci contiendra la moyenne de chaque étudiant s'il a passé les examens de tous les modules dans lesquels il est inscrit.
8. Revenez sur la première contrainte : il ne doit être possible de modifier une note min dans la table **prerequis** que s'il n'existe pas d'élève dont une inscription serait invalidée.
9. Il ne doit être possible de modifier **effecMax** que si des étudiants ne se retrouvent pas avec une inscription invalidée.

Libre à vous par la suite de trouver d'autres contraintes et de les implémenter.

## 2.9 Packages

### Exercice 1

Lancez deux sessions simultanément sur le même serveur et invoquez les sous-programmes du package `compteur` depuis chacune des sessions. Que remarquez-vous ?

### Exercice 2

Implémenter le corps du package suivant (utilisez les données de A.5).

```
CREATE OR REPLACE PACKAGE gestion_arbre IS
    circuit exception;

    cursor feuilles return personne%rowtype;

    procedure ajoutePersonne(nom personne.nom%type ,
                             prenom personne.prenom%type , pere personne.pere%type ,
                             mere personne.mere%type);

    procedure modifieParents(pers personne.numbers%type ,
                             numPere personne.pere%type , numMere personne.mere%type);

END gestion_arbre;
/
```

## 2.10 Révisions

Implémentez les contraintes suivantes dans les données de A.9.

1. Les parents d'une même personne sont des personnes différentes.
2. L'arbre généalogique ne contient pas de circuit.
3. Les dates de divorce sont ultérieures aux dates de mariage.
4. Une même personne ne peut pas être mariée à plusieurs personnes simultanément.
5. Personne ne peut être père d'une personne et mère d'une autre.
6. Un mari ne peut pas être mère et une femme ne peut pas être père.
7. Deux personnes ayant du sang en commun ne peuvent se marier.

# Chapitre 3

# Corrigés

## 3.1 Introduction au PL/SQL

```
-- Exercice 1
DECLARE
  a NUMBER;
  b NUMBER;
  t NUMBER;
BEGIN
  a := 1;
  b := 2;
  DBMS_OUTPUT.PUT_LINE('a = '||a);
  DBMS_OUTPUT.PUT_LINE('b = '||b);
  DBMS_OUTPUT.PUT_LINE('Let''s swap a and b... The result is:');
  t := a;
  a := b;
  b := t;
  DBMS_OUTPUT.PUT_LINE('a = '||a);
  DBMS_OUTPUT.PUT_LINE('b = '||b);
END;
/

-- Exercice 2
DECLARE
  a NUMBER;
  res NUMBER;
  counter NUMBER;
BEGIN
  a := 10;
  res := 1;
  counter := a;
  WHILE counter > 0 LOOP
    res := res * counter;
    counter := counter - 1;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(a || '!= '||res);
END;
/

-- Exercice 3
DECLARE
  a NUMBER := 48;
  b NUMBER := 84;
  amodb NUMBER;
BEGIN
  DBMS_OUTPUT.PUT('PGCD('||a||', '||b||') = ');
  WHILE b > 0 LOOP
    amodb := a;
    WHILE amodb >= b LOOP
      amodb := amodb - b;
    END LOOP;
    a := b;
    b := amodb;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE(a);
END;
/
```

## 3.2 Tableaux et Structures

```
SET SERVEROUTPUT ON

-- Tableaux

DECLARE
    TYPE montab IS VARRAY (50) OF INTEGER;
    t montab;
BEGIN
    t := montab();
    t.extend(20);

    -- Initialisation
    FOR i IN 1..20 LOOP
        t(i) := i*i;
    END LOOP;

    -- Inversion de l'ordre des éléments
    DECLARE
        temp integer;
    BEGIN
        FOR i IN 1..10 LOOP
            temp := t(i);
            t(i) := t(20-i+1);
            t(20-i+1) := temp;
        END LOOP;
    END;

    -- Affichage
    FOR i IN 1..20 LOOP
        DBMS_OUTPUT.PUT_LINE('t(' ||
            i || ') = ' || t(i));
    END LOOP;

    -- Tri à bulle
    DECLARE
        temp integer;
    BEGIN
        FOR i IN REVERSE 2..20 LOOP
            FOR j IN 2..i LOOP
                IF t(j-1) > t(j) THEN
                    temp := t(j);
                    t(j) := t(j-1);
                    t(j-1) := temp;
                END IF;
            END LOOP;
        END LOOP;
    END;

    -- Affichage
    FOR i IN 1..20 LOOP
        DBMS_OUTPUT.PUT_LINE('t(' ||
            i || ') = ' || t(i));
    END LOOP;

    -- Recherche par dichotomie de l'élément 225
    DECLARE
        inf INTEGER := 1;
        sup INTEGER := 20;
        m INTEGER;
        X INTEGER := 400;
    BEGIN
        LOOP
            DBMS_OUTPUT.PUT_LINE('inf = ' || inf ||
                ', sup = ' || sup);
            m := (inf + sup)/2;
            EXIT WHEN
                t(m) = X OR inf = sup;
            IF t(m) > X THEN
                sup := m-1;
            ELSE
                inf := m+1;
            END IF;
        END LOOP;
        IF t(m) = X THEN
            DBMS_OUTPUT.PUT_LINE(X ||
                ' est dans le tableau');
        ELSE
            DBMS_OUTPUT.PUT_LINE(X ||
                ' n''est pas dans le tableau');
        END IF;
    END;

END;
/
```



```

-- Structures
DECLARE
  -- Maillon d'une liste chaînée
  TYPE CELL IS RECORD
  (
    -- Donnée de chaque maillon
    data INTEGER,
    -- Indice du maillon précédent de la liste,
    -- -1 s'il n'y en a pas
    previous INTEGER,
    -- Indice du maillon suivant de la liste,
    -- -1 s'il n'y en a pas
    next INTEGER
  );
  -- Type tableau contenant les maillons de la liste
  TYPE TREE IS VARRAY (19) OF CELL;
  -- Tableau contenant les maillons de la liste
  t TREE;
  -- indice du premier élément de la liste
  first integer;
  -- indice du dernier élément de la liste
  last integer;
BEGIN
  t := TREE ();
  t.extend(19);

  -- Initialisation
  FOR i IN 1..19 LOOP
    t(i).data := power(i, 5) mod 19 ;
    t(i).previous := i-1;
    t(i).next := i+1;
  END LOOP;
  first := 1;
  last := 19;
  t(first).previous := -1;
  t(last).next := -1;

  -- Affichage
  DECLARE
    p integer := first;
  BEGIN
    WHILE p <> -1 LOOP
      DBMS_OUTPUT.PUT_LINE('(' || p || ', ' ||
        t(p).data || ', ' ||
        t(p).previous || ', ' ||
        t(p).next || ')');
      p := t(p).next;
    END LOOP;
  END;

  -- Inversion de l'ordre des éléments
  DECLARE
    temp INTEGER;
  BEGIN
    FOR i IN 1..19 LOOP
      temp := t(i).previous;
      t(i).previous := t(i).next;
      t(i).next := temp;
    END LOOP;
    first := 19;
    last := 1;
  END;

  -- Affichage
  DECLARE
    p integer := first;
  BEGIN
    WHILE p <> -1 LOOP
      DBMS_OUTPUT.PUT_LINE('(' ||
        p || ', ' ||
        t(p).data || ', ' ||
        t(p).previous || ', ' ||
        t(p).next || ')');
      p := t(p).next;
    END LOOP;
  END;

  -- Tri à bulle
  DECLARE
    i integer := last;
    j integer;
  BEGIN
    WHILE t(t(i).previous).previous <> -1 LOOP
      j := first;
      WHILE i <> j LOOP
        IF(t(j).data > t(t(j).next).data) THEN

```

```

-- Echange de j et t(j).next
-- par modification du chaînage
DECLARE
    afterJ INTEGER := t(j).next;
    beforeJ INTEGER := t(j).previous;
BEGIN
    t(j).next := t(afterJ).next;
    t(afterJ).next := j;
    t(afterJ).previous := beforeJ;
    t(j).previous := afterJ;
    IF t(j).next <> -1 THEN
        t(t(j).next).previous := j;
    ELSE
        last := j;
    END IF;
    IF t(afterJ).previous <> -1 THEN
        t(t(afterJ).previous).next := afterJ;
    ELSE
        first := afterJ;
    END IF;
    IF afterJ = i THEN
        i := j;
    END IF;
END;

ELSE
    j := t(j).next;
END IF;
END LOOP;
i := t(i).previous;
END LOOP;
END;

-- Affichage
DECLARE
    p integer := first;
BEGIN
    WHILE p <> -1 LOOP
        DBMS_OUTPUT.PUT_LINE('(' || p || ', ' ||
            t(p).data || ', ' ||
            t(p).previous || ', ' ||
            t(p).next || ')');
        p := t(p).next;
    END LOOP;
END;

END;
/

```

### 3.3 Application du PL/SQL et Exceptions

```

SET SERVEROUTPUT ON
SET AUTOCOMMIT OFF

-- Exercice 1

DECLARE
  unClient PERSONNE%ROWTYPE;
  numClient PERSONNE.numbers%TYPE;
  Y_A_EU_UNE_MERDE EXCEPTION;
BEGIN
  FOR numClient IN 1..21 LOOP
    BEGIN
      SELECT * INTO unClient
      FROM PERSONNE
      WHERE numbers = numClient;

      INSERT INTO CLIENT (numcli, nomcli, prenomcli)
      VALUES
      (unClient.numbers,
      unClient.nom,
      unClient.prenom);

    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE(
          'Personne n" a l"identifiant ' ||
          numClient);
      WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE(
          'Cette message ne devrait jamais apparaître !');
      WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE(
          'Contrainte de clé violée ! Message SQL : ' ||
          SQLERRM);
      WHEN OTHERS THEN
        RAISE Y_A_EU_UNE_MERDE;
    END;
  END LOOP;
  COMMIT;
EXCEPTION
  WHEN Y_A_EU_UNE_MERDE THEN
    DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
    DBMS_OUTPUT.PUT_LINE('Il y a eu une Merde !');
    ROLLBACK;
END;
/

-- Exercice 2

DECLARE
  unClient CLIENT%rowtype;
BEGIN
  SELECT * INTO unClient
  FROM CLIENT WHERE numCli =
  (
    SELECT MAX(numcli)
    FROM CLIENT
  );
  INSERT INTO PERSONNEL VALUES
  (
    1,
    unClient.nomcli,
    unClient.prenomcli,
    NULL,
    1254.28
  );
  COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Aucun client');
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE(
      'Il y a un gros problème... J"comprends pas c"qui s"passe');
END;
/

-- Exercice 3

DECLARE
  numClient CLIENT.numcli%TYPE;
  tCCL TYPECCL.numtypeccl%TYPE;
  nto TYPEOPERATION.numtypeoper%TYPE;
  Y_A_UN_GRO_BLEME EXCEPTION;
BEGIN
  SELECT numtypeoper INTO nto

```

```

FROM TYPEOPERATION
WHERE nomtypeoper = 'dépôt espèces';
SELECT numtypeccl INTO tCCL
FROM TYPECCL
WHERE nomtypeCCL = 'Compte courant';
FOR numClient IN 1..21 LOOP
BEGIN
    INSERT INTO COMPTECLIENT VALUES
    (
        numClient,
        1,
        tCCL,
        SYSDATE,
        1
    );
    INSERT INTO OPERATION VALUES
    (
        numClient,
        1,
        1,
        nto,
        SYSDATE,
        numClient * 100,
        'inauguration du compte'
    );

    COMMIT;
EXCEPTION
WHEN OTHERS THEN
    -- Adaptez le numéro du code,
    -- chez moi ça donne -2290
    IF SQLCODE = -2290 THEN
        DECLARE
            total OPERATION.montantoper%TYPE := numClient * 100;
            toInsert OPERATION.montantoper%TYPE;
            cpt NUMBER := 1;
        BEGIN
            WHILE total > 0 LOOP
                IF total > 1000 THEN
                    toInsert := 1000;
                ELSE
                    toInsert := total;
                END IF;
                INSERT INTO OPERATION VALUES
                (
                    numClient,
                    1,
                    (SELECT nvl(MAX(numoper), 0) + 1
                     FROM OPERATION
                     WHERE numcli = numClient
                     AND numccl = 1
                    ),
                    nto,
                    SYSDATE,
                    toInsert,
                    'Inauguration du compte ' || cpt
                );
                total := total - toInsert;
                cpt := cpt + 1;
            END LOOP;
        EXCEPTION
        WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('MOD(total, 1000) = ' || MOD(total, 1000));
            DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
            DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
            RAISE Y_A_UN_GRO_BLEME;
        END;
    ELSE
        DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
    ROLLBACK;
    END IF;
    END;
END LOOP;
EXCEPTION
WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Pas de données !');
WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('Trop de données !');
WHEN Y_A_UN_GRO_BLEME THEN
    DBMS_OUTPUT.PUT_LINE('Il y a un gros problème !');
    DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
    DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
    DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
END;
/
-- Exercice 4

```

```

DECLARE
    numClient CLIENT.numcli%TYPE := 2;
    numCompteLivret TYPECCL.numtypeCCL%TYPE;
    nto TYPEOPERATION.numtypeoper%TYPE;
    montant OPERATION.montantoper%TYPE;
    Y_A_UN_GRO_BLEME EXCEPTION;
BEGIN
    SELECT numtypeoper INTO nto
        FROM TYPEOPERATION
        WHERE nomtypeoper = 'virement';
    SELECT numtypeccl INTO numCompteLivret
        FROM TYPECCL
        WHERE nomtypeCcl = 'livret';
    WHILE numClient <= 21 LOOP
        BEGIN
            montant := 100 * numClient - 500;
            INSERT INTO COMPTECLIENT VALUES
                (
                    numClient,
                    2,
                    numCompteLivret,
                    SYSDATE,
                    1
                );
            INSERT INTO OPERATION VALUES
                (
                    numClient,
                    1,
                    (SELECT nvl(MAX(numoper), 0) + 1
                     FROM OPERATION
                     WHERE numcli = numClient
                     AND numccl = 1),
                    nto,
                    SYSDATE,
                    -montant,
                    'versement livret'
                );
            INSERT INTO OPERATION VALUES
                (
                    numClient,
                    2,
                    (SELECT nvl(MAX(numoper), 0) + 1
                     FROM OPERATION
                     WHERE numcli = numClient
                     AND numccl = 2),
                    nto,
                    SYSDATE,
                    montant,
                    'versement livret'
                );
            COMMIT;
        EXCEPTION
        WHEN OTHERS THEN
            -- idem
            IF SQLCODE = -2290 THEN
                DECLARE
                    total OPERATION.montantoper%TYPE := montant;
                    toMove OPERATION.montantoper%TYPE;
                    cpt NUMBER := 1;
                BEGIN
                    WHILE total > 1000 LOOP
                        IF total > 1000 THEN
                            toMove := 1000;
                        ELSE
                            tomove := total;
                        END IF;
                        INSERT INTO OPERATION VALUES
                            (
                                numClient,
                                1,
                                (SELECT nvl(MAX(numoper), 0) + 1
                                 FROM OPERATION
                                 WHERE numcli = numClient
                                 AND numccl = 1),
                                nto,
                                SYSDATE,
                                -toMove,
                                'versement livret ' || cpt
                            );
                        INSERT INTO OPERATION VALUES
                            (
                                numClient,
                                2,
                                (SELECT nvl(MAX(numoper), 0) + 1
                                 FROM OPERATION
                                 WHERE numcli = numClient
                                 AND numccl = 2),
                                nto,

```

```

                                SYSDATE ,
                                toMove ,
                                'versement livret ' || cpt
                                );
                                total := total - toMove;
                                cpt := cpt + 1;
                                END LOOP;
                                COMMIT;
                                EXCEPTION
                                WHEN OTHERS THEN
                                    RAISE Y_A_UN_GRO_BLEME;
                                END;
                                ELSE
                                    DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
                                    DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
                                    ROLLBACK;
                                END IF;
                                END;
                                COMMIT;
                                numClient := numClient + 2;
                                END LOOP;
                                EXCEPTION
                                WHEN NO_DATA_FOUND THEN
                                    DBMS_OUTPUT.PUT_LINE('Pas de données !');
                                WHEN TOO_MANY_ROWS THEN
                                    DBMS_OUTPUT.PUT_LINE('Trop de données !');
                                WHEN Y_A_UN_GRO_BLEME THEN
                                    DBMS_OUTPUT.PUT_LINE('Il y a un gros problème !');
                                    DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
                                    DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
                                WHEN OTHERS THEN
                                    DBMS_OUTPUT.PUT_LINE('SQLCODE = ' || SQLCODE);
                                    DBMS_OUTPUT.PUT_LINE('SQLERRM = ' || SQLERRM);
                                END;
                                /

```

## 3.4 Sous-programmes

*-- Exercice 1*

```
CREATE OR REPLACE FUNCTION
    bad_puissance (b NUMBER, n NUMBER)
    RETURN NUMBER IS
BEGIN
    IF (n = 0) THEN
        RETURN 1;
    ELSE
        RETURN b * bad_puissance (b, n - 1);
    END IF;
END;
```

*-- Exercice 2*

```
CREATE OR REPLACE FUNCTION
    good_puissance (b NUMBER, n NUMBER)
    RETURN NUMBER IS
BEGIN
    IF (n = 0) THEN
        RETURN 1;
    END IF;
    IF (MOD(n, 2) = 0) THEN
        RETURN good_puissance (b * b, n / 2);
    END IF;
    RETURN b * good_puissance (b, n - 1);
END;
```

*-- Exercice 3*

```
CREATE OR REPLACE FUNCTION
    demiFreres (A PERSONNE.numbers%type, B PERSONNE.numbers%type)
    RETURN BOOLEAN IS
    rowA PERSONNE%rowtype;
    rowB PERSONNE%rowtype;
BEGIN
    SELECT * INTO rowA FROM PERSONNE WHERE numbers = A;
    SELECT * INTO rowB FROM PERSONNE WHERE numbers = B;
    RETURN rowA.pere = rowB.pere OR rowA.mere = rowB.mere;
END;
```

*-- Exercice 4*

```
CREATE OR REPLACE FUNCTION
    freres (A PERSONNE.numbers%type, B PERSONNE.numbers%type)
    RETURN BOOLEAN IS
    rowA PERSONNE%rowtype;
    rowB PERSONNE%rowtype;
BEGIN
    SELECT * INTO rowA FROM PERSONNE WHERE numbers = A;
    SELECT * INTO rowB FROM PERSONNE WHERE numbers = B;
    RETURN rowA.pere = rowB.pere AND rowA.mere = rowB.mere;
END;
```

```
CREATE OR REPLACE FUNCTION
    cousinsGermain (A PERSONNE.numbers%type, B PERSONNE.numbers%type)
    RETURN BOOLEAN IS
    rowA PERSONNE%rowtype;
    rowB PERSONNE%rowtype;
BEGIN
    SELECT * INTO rowA FROM PERSONNE WHERE numbers = A;
    SELECT * INTO rowB FROM PERSONNE WHERE numbers = B;
    RETURN
        freres(rowA.pere, rowB.pere)
        OR
        freres(rowA.pere, rowB.mere)
        OR
        freres(rowA.mere, rowB.pere)
        OR
        freres(rowA.mere, rowB.mere);
END;
```

*-- Exercice 5*

```
CREATE OR REPLACE PROCEDURE
    aieul (P PERSONNE.numbers%type) IS
    row PERSONNE%rowtype;
    nb NUMBER;
BEGIN
    SELECT count(*) INTO nb
    FROM PERSONNE
```

```

        WHERE numpers = P;
    IF (NB = 1) THEN
        SELECT * INTO row
            FROM PERSONNE
            WHERE numpers = P;
        DBMS_OUTPUT.PUT_LINE(row.nom);
        aieul(row.pere);
    END IF;
END;
/

```

*--- Exercice 6*

```

CREATE OR REPLACE PROCEDURE
    mecs (P PERSONNE.numpers%type) IS
    row PERSONNE%rowtype;
    nb NUMBER;
BEGIN
    SELECT count(*) INTO NB
        FROM PERSONNE
        WHERE numpers = P;
    IF (NB = 1) THEN
        SELECT * INTO row
            FROM PERSONNE
            WHERE numpers = P;
        SELECT count(*) INTO NB
            FROM PERSONNE
            WHERE pere = P;
        IF (NB > 0) THEN
            DBMS_OUTPUT.PUT_LINE(row.nom);
        END IF;
        mecs(row.pere);
        mecs(row.mere);
    END IF;
END;
/

```

*--- Exercice 7*

```

CREATE OR REPLACE FUNCTION
    ascendant (A PERSONNE.numpers%type, B PERSONNE.numpers%type)
    RETURN BOOLEAN IS
    row PERSONNE%rowtype;
BEGIN
    SELECT * INTO row FROM PERSONNE WHERE numpers = B;
    IF (row.pere = A OR row.mere = A) THEN
        RETURN TRUE;
    END IF;
    RETURN (row.pere IS NOT NULL AND ascendant(A, row.pere))
        OR
        (row.mere IS NOT NULL AND ascendant(A, row.mere));
END;
/
BEGIN
    IF (ascendant(1, 8)) THEN
        DBMS_OUTPUT.PUT_LINE('OK');
    ELSE
        DBMS_OUTPUT.PUT_LINE('pas OK');
    END IF;
END;
/

```

*--- Exercice 8*

```

CREATE OR REPLACE FUNCTION
    fmax (A NUMBER, B NUMBER)
    RETURN NUMBER IS
BEGIN
    IF (A > B) THEN
        RETURN A;
    ELSE
        RETURN B;
    END IF;
END;
/
CREATE OR REPLACE FUNCTION
    ecartAscendant (A PERSONNE.numpers%type, B PERSONNE.numpers%type)
    RETURN NUMBER IS
    row PERSONNE%rowtype;
    NB NUMBER;
BEGIN
    SELECT * INTO row FROM PERSONNE WHERE numpers = B;
    IF (row.pere = A OR row.mere = A) THEN
        RETURN 1;
    END IF;
    IF (row.pere IS NULL) THEN

```



```

        NB := -1;
    ELSE
        NB := ecartAscendant(A, row.pere);
    END IF;
    IF (row.mere IS NULL) THEN
        NB := fmax(-1, NB);
    ELSE
        NB := fmax(ecartAscendant(A, row.pere), NB);
    END IF;
    IF (NB <> -1) THEN
        NB := NB + 1;
    END IF;
    RETURN NB;
END;
/

CREATE OR REPLACE FUNCTION
    ecart (A PERSONNE.numpers%type, B PERSONNE.numpers%type)
    RETURN NUMBER IS
    row PERSONNE%rowtype;
    NB NUMBER;
BEGIN
    RETURN fmax(ecartAscendant(A, B), ecartAscendant(B, A));
END;
/

```

— *Exercice 9*

```

SELECT A.nom, A.prenom, B.nom, B.prenom
FROM PERSONNE A, PERSONNE B
WHERE ecartAscendant(A.numpers, B.numpers) =
    (
        SELECT MAX(ec)
        FROM
            (
                SELECT ecart(A.numpers, B.numpers) AS ec
                FROM PERSONNE A, PERSONNE B
            )
    );

```

— *Exercice 10*

## 3.5 Curseurs

```
CREATE OR REPLACE PROCEDURE copyFromPersonneToClient IS
  CURSOR C IS
    SELECT *
    FROM PERSONNE;
  ROW C%rowtype;
BEGIN
  FOR ROW IN C LOOP
    INSERT INTO CLIENT
      (numcli, nomcli, prenomcli)
    VALUES
      (ROW.numpers, ROW.nom, ROW.prenom);
  END LOOP;
  COMMIT;
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE('Copy can be done only once.');
```

```
END;
/

CALL copyFromPersonneToClient();

CREATE OR REPLACE PROCEDURE takeClientToPersonnel IS
  Row client%rowtype;
BEGIN
  SELECT * INTO Row
    FROM CLIENT
    WHERE numcli =
      (
        SELECT MAX(numcli)
        FROM CLIENT);
  INSERT INTO PERSONNEL
    (numpers, nompers, prenompers)
  VALUES
    (Row.numcli, Row.nomcli, Row.prenomcli);
  COMMIT;
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.PUT_LINE('This row has already been imported.');
```

```
ROLLBACK;
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Table CLIENT is empty.');
```

```
ROLLBACK;
END;
/

CALL takeClientToPersonnel();

CREATE OR REPLACE PROCEDURE
  creditAccount(numclient CLIENT.numcli%type, value NUMBER) IS
BEGIN
  IF (value > 100) THEN
    creditAccount(numclient, 100);
    creditAccount(numclient, value - 100);
  ELSE
    INSERT INTO OPERATION VALUES
      (numclient,
       1,
       (SELECT nvl(MAX(numoper), 0) + 1
        FROM OPERATION
        WHERE numcli = numclient
        AND numccl = 1
       ),
       (SELECT numtypeoper
        FROM TYPEOPERATION
        WHERE nomtypeoper = 'virement'
       ),
       sysdate,
       value,
       'cadeau !'
      );
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    IF (SQLCODE = -22900) THEN
      DBMS_OUTPUT.PUT_LINE('Too much money at once.');
```

```
END IF;
END;
/

CREATE OR REPLACE PROCEDURE
  createVirement(numclient CLIENT.numcli%type, value NUMBER) IS
BEGIN
  INSERT INTO OPERATION VALUES
    (numclient,
     1,
     (SELECT nvl(MAX(numoper), 0) + 1
```

```

                FROM OPERATION
                WHERE numcli = numclient
                AND numccl = 1
            ),
            (SELECT numtypeoper
             FROM TYPEOPERATION
             WHERE nomtypeoper = 'virement'
            ),
            sysdate ,
            -value ,
            'cadeau !'
        );
INSERT INTO OPERATION VALUES
    (numclient ,
     2,
     (SELECT nvl(MAX(numoper), 0) + 1
      FROM OPERATION
      WHERE numcli = numclient
      AND numccl = 1
     ),
     (SELECT numtypeoper
      FROM TYPEOPERATION
      WHERE nomtypeoper = 'virement'
     ),
     sysdate ,
     value ,
     'cadeau !'
    );
EXCEPTION
    WHEN OTHERS THEN
        IF (SQLCODE = -22900) THEN
            DBMS_OUTPUT.PUT_LINE('Too much money at once. ');
        END IF;
END;
/

CREATE OR REPLACE PROCEDURE
    moveToLivret(numclient CLIENT.numcli%type, value NUMBER) IS
BEGIN
    IF (value >= 0) THEN
        IF (value > 100) THEN
            moveToLivret(numclient, 100);
            moveToLivret(numclient, value - 100);
        ELSE
            createVirement(numclient, value);
        END IF;
    END IF;
EXCEPTION
    WHEN OTHERS THEN
        IF (SQLCODE = -22900) THEN
            DBMS_OUTPUT.PUT_LINE('Too much money at once. ');
        END IF;
END;
/

CREATE OR REPLACE PROCEDURE
    openAccount(numclient CLIENT.numcli%type) IS
BEGIN
    INSERT INTO COMPTECLIENT VALUES
        (numclient ,
         1,
         (SELECT numtypeccl
          FROM TYPECCL
          WHERE nomtypeccl = 'Compte courant'
         ),
         sysdate ,
         (SELECT numpers
          FROM PERSONNEL
          WHERE numpers =
            (SELECT MAX(numcli)
             FROM CLIENT
            )
         )
        );
    INSERT INTO COMPTECLIENT VALUES
        (numclient ,
         2,
         (SELECT numtypeccl
          FROM TYPECCL
          WHERE nomtypeccl = 'virement'
         ),
         sysdate ,
         (SELECT numpers
          FROM PERSONNEL
          WHERE numpers =
            (SELECT MAX(numcli)
             FROM CLIENT
            )
         )
        );

```

```

        )
    );
    creditAccount(numclient, numclient * 100);
    moveToLivret(numclient, numclient * 100 - 500);
EXCEPTION
    WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE('This account has already been opened. ');
END;
/

CREATE OR REPLACE PROCEDURE openAccounts IS
    CURSOR C IS
        SELECT numcli FROM CLIENT;
    ROW C%rowtype;
BEGIN
    FOR ROW IN C LOOP
        openAccount(ROW.numcli);
    END LOOP;
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('An error has occurred. ');
        ROLLBACK;
END;
/

CALL openAccounts ();

CREATE OR REPLACE PROCEDURE
    afficheDescendance(numpersonne NUMBER) IS
    CURSOR C IS
        SELECT *
        FROM PERSONNE
        WHERE pere = numpersonne
        OR mere = numpersonne;
    ROW C%rowType;
BEGIN
    FOR ROW IN C LOOP
        DBMS_OUTPUT.PUT_LINE(row.nom || ' ' || row.prenom);
        afficheDescendance(ROW.numbers);
    END LOOP;
END;
/

```

## 3.6 Curseurs paramétrés

*— Exercice 1*

```
CREATE OR REPLACE PROCEDURE afficheClient(unClient CLIENT%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Client ' || unClient.prenomcli || ' ' || unClient.nomcli);
END;
/
```

```
CREATE OR REPLACE PROCEDURE afficheCompte(unCompte COMPTECLIENT%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('* Compte ' || unCompte.numcli || ' ' || unCompte.numccl);
END;
/
```

```
CREATE OR REPLACE PROCEDURE afficheComptesClients IS
    CURSOR clients IS
        SELECT *
        FROM CLIENT;
    unClient clients%rowtype;
    CURSOR comptes (numclient CLIENT.numcli%type) IS
        SELECT *
        FROM COMPTECLIENT
        WHERE numcli = numclient;
    unCompte clients%rowtype;
BEGIN
    FOR unClient IN clients LOOP
        afficheClient(unClient);
        FOR unCompte IN comptes(unClient.numcli) LOOP
            afficheCompte(unCompte);
        END LOOP;
    END LOOP;
END;
/
```

```
SET SERVEROUTPUT ON SIZE 1000000
```

```
call afficheComptesClients();
```

*— Exercice 2*

```
CREATE OR REPLACE PROCEDURE afficheOperation(uneOperation OPERATION%rowtype) IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('* * Operation ' || uneOperation.numOper || ' , montant : ' || uneOperation.montantOper);
END;
/
```

```
CREATE OR REPLACE PROCEDURE afficheOperComptesClients IS
    CURSOR clients IS
        SELECT *
        FROM CLIENT;
    unClient clients%rowtype;
    CURSOR comptes (numclient CLIENT.numcli%type) IS
        SELECT *
        FROM COMPTECLIENT
        WHERE numcli = numclient;
    unCompte clients%rowtype;
    CURSOR operations
        (numclient CLIENT.numcli%type,
         numcompte COMPTECLIENT.numccl%type) IS
        SELECT *
        FROM OPERATION
        WHERE numcli = numclient
        AND numccl = numcompte;
    uneOperation operations%rowtype;
BEGIN
    FOR unClient IN clients LOOP
        afficheClient(unClient);
        FOR unCompte IN comptes(unClient.numcli) LOOP
            afficheCompte(unCompte);
            FOR uneOperation IN operations(unClient.numcli, unCompte.numccl) LOOP
                afficheOperation(uneOperation);
            END LOOP;
        END LOOP;
    END LOOP;
END;
/
```

```
call afficheOperComptesClients();
```

## 3.7 Triggers

```
-- Il convient d'abord de modifier quelque peu l'organisation des
-- donnees, on ajoute par exemple dans la table MODULE le nombre
-- d'etudiants inscrits

DROP TABLE RESULTAT;
DROP TABLE EXAMEN;
DROP TABLE PREREQUIS;
DROP TABLE INSCRIPTION;
DROP TABLE MODULE;
DROP TABLE ETUDIANT;

CREATE TABLE ETUDIANT
  (numEtud number,
  nom varchar2(40),
  prenom varchar2(40),
  datenaiss date,
  civilite varchar2(4),
  patronyme varchar2(40),
  numsecu varchar2(15) NOT NULL,
  moyenne NUMBER DEFAULT NULL);

CREATE TABLE MODULE
  (codMod number,
  nomMod varchar2(15),
  effecMax number DEFAULT 30,
  effec number default 0);

CREATE TABLE EXAMEN
  (codMod number,
  codExam number,
  dateExam date);

CREATE TABLE INSCRIPTION
  (numEtud number,
  codMod number,
  dateInsc date default sysdate);

CREATE TABLE PREREQUIS
  (codMod number,
  codModPrereq number,
  noteMin number(4, 2) NOT NULL);

CREATE TABLE RESULTAT
  (codMod number,
  codExam number,
  numEtud number,
  note number(4, 2));

ALTER TABLE ETUDIANT ADD
  CONSTRAINT pk_etudiant
  PRIMARY KEY (numEtud);
ALTER TABLE MODULE ADD
  CONSTRAINT pk_module
  PRIMARY KEY (codMod);
ALTER TABLE EXAMEN ADD
  CONSTRAINT pk_examen
  PRIMARY KEY (codMod, codExam);
ALTER TABLE PREREQUIS ADD
  CONSTRAINT pk_prerequis
  PRIMARY KEY (codMod, codModPrereq);
ALTER TABLE INSCRIPTION ADD
  CONSTRAINT pk_inscription
  PRIMARY KEY (codMod, numEtud);
ALTER TABLE RESULTAT ADD
  CONSTRAINT pk_resultat
  PRIMARY KEY (codMod, numEtud, codExam);

ALTER TABLE INSCRIPTION ADD
  (CONSTRAINT fk_inscription_etudiant
  FOREIGN KEY (numEtud)
  REFERENCES ETUDIANT(numEtud),
  CONSTRAINT fk_inscription_module
  FOREIGN KEY (codMod)
  REFERENCES MODULE(codMod));
ALTER TABLE PREREQUIS ADD
  (CONSTRAINT fk_prerequis_codmod
  FOREIGN KEY (codMod)
  REFERENCES MODULE(codMod),
  CONSTRAINT fk_prerequis_codmodprereq
  FOREIGN KEY (codModPrereq)
  REFERENCES MODULE(codMod));
ALTER TABLE EXAMEN ADD
  CONSTRAINT fk_examen
  FOREIGN KEY (codMod)
  REFERENCES MODULE(codMod);
ALTER TABLE RESULTAT ADD
```

```

(CONSTRAINT fk_resultat_examen
FOREIGN KEY (codMod, codExam)
REFERENCES EXAMEN(codMod, codExam),
CONSTRAINT fk_resultat_inscription
FOREIGN KEY (codMod, numEtud)
REFERENCES INSCRIPTION(codMod, numEtud));

```

```

ALTER TABLE ETUDIANT ADD
(CONSTRAINT ck_civilite
CHECK
(
civilite IN ('Mr', 'Mme', 'Mlle')
),
CONSTRAINT ck_civilite_numsecu
CHECK
(
SUBSTR(numsecu, 1, 1) = '2' OR patronyme IS NULL
),
CONSTRAINT ck_length_numsecu
CHECK
(
length(numsecu) = 15
),
CONSTRAINT ck_annee_numsecu CHECK
(
to_char(datenaiss, 'yy') = substr(numsecu, 2, 2)
)
);

```

---

*--- Conainte 1 ---*

---

```

CREATE OR REPLACE TRIGGER beforeUpdateFERPrerequis
BEFORE UPDATE ON PREREQUIS
FOR EACH ROW
BEGIN
    IF (:new.noteMin < :old.noteMin) THEN
        :new.noteMin := :old.noteMin;
    END IF;
END;
/

```

---

*--- Conainte 2 ---*

---

```

CREATE OR REPLACE PROCEDURE incrEffec (module NUMBER) IS
BEGIN
    UPDATE MODULE SET effec = effec + 1 WHERE codmod = module;
END;
/

```

```

CREATE OR REPLACE PROCEDURE decrEffec (module NUMBER) IS
BEGIN
    UPDATE MODULE SET effec = effec - 1 WHERE codmod = module;
END;
/

```

```

CREATE OR REPLACE TRIGGER BeforeInsertFERModule
BEFORE INSERT ON MODULE
FOR EACH ROW
BEGIN
    :new.effec := 0;
END;
/

```

```

CREATE OR REPLACE TRIGGER afterInsertFERInsc
AFTER INSERT ON INSCRIPTION
FOR EACH ROW
BEGIN
    incrEffec (:new.codmod);
END;
/

```

```

CREATE OR REPLACE TRIGGER afterDeleteFERInsc
AFTER DELETE ON INSCRIPTION
FOR EACH ROW
BEGIN
    decrEffec(:old.codmod);
END;
/

```

```

CREATE OR REPLACE TRIGGER afterUpdateFERInsc
AFTER UPDATE ON INSCRIPTION
FOR EACH ROW
BEGIN
    decrEffec(:old.codmod);
    incrEffec(:new.codmod);
END;
/

```

```

DROP VIEW modulesDisponibles;

CREATE VIEW modulesDisponibles AS
SELECT codmod
FROM MODULE
WHERE effec < effecMax;

```

```

CREATE OR REPLACE TRIGGER beforeInsertUpdateFERInsc
BEFORE INSERT OR UPDATE ON INSCRIPTION
FOR EACH ROW
DECLARE
    nbLignes NUMBER;
BEGIN
    SELECT count(*) INTO nbLignes
    FROM modulesDisponibles
    WHERE codmod = :new.codmod;
    IF (nbLignes = 0) THEN
        RAISE_APPLICATION_ERROR(-20001, 'Plus de places disponibles. ');
    END IF;
END;
/

```

--- *Contrainte 3* ---

```

DROP VIEW examensPossibles;

CREATE VIEW examensPossibles AS
SELECT codMod
FROM MODULE M
WHERE
    (
        SELECT COUNT(*)
        FROM INSCRIPTION I
        WHERE I.codmod = M.codmod
    ) > 0 ;

```

```

CREATE OR REPLACE TRIGGER beforeInsertUpdateFERExam
BEFORE INSERT OR UPDATE ON EXAMEN
FOR EACH ROW
DECLARE
    nbLignes NUMBER;
BEGIN
    SELECT count(*) INTO nbLignes
    FROM examensPossibles
    WHERE codMod = :new.codmod;
    IF (nbLignes = 0) THEN
        RAISE_APPLICATION_ERROR(-20002, 'Pas d"élève dans ce module. ');
    END IF;
END;
/

```



---

---

-- Conainte 4 --

---

---

DROP VIEW etudiantsExamens;

CREATE VIEW etudiantsExamens AS  
SELECT I.numetud, E.codmod, E.codexam  
FROM INSCRIPTION I, EXAMEN E  
WHERE I.codmod = E.codmod  
AND I.dateInsc < E.dateExam;

---

---

CREATE OR REPLACE TRIGGER beforeInsertUpdateFERResult  
BEFORE INSERT OR UPDATE ON RESULTAT  
FOR EACH ROW  
DECLARE  
    nbLignes NUMBER;  
BEGIN  
    SELECT count(\*) INTO nbLignes  
    FROM etudiantsExamens  
    WHERE numetud = :new.numetud  
    AND codmod = :new.codmod  
    AND codexam = :new.codexam;  
    IF (nbLignes = 0) THEN  
        RAISE\_APPLICATION\_ERROR(-20002, 'Examen antérieur à l"inscription dans le module.');

END IF;  
END;

---

---

-- Conainte 5 --

---

---

-- On crée une table temporaire contenant les mêmes valeurs que prerequis,  
-- On la met à jour AVANT la table prerequis pour vérifier que l'insertion  
-- ne construit pas de circuit.

DROP TABLE MIRRORPREREQ;

CREATE TABLE MIRRORPREREQ  
(codmod NUMBER,  
codmodprereq NUMBER,  
noteMin NUMBER);

---

---

CREATE OR REPLACE FUNCTION  
findModule(root number, moduleToFind number)  
RETURN BOOLEAN  
IS  
    CURSOR C IS  
        SELECT codmod  
        FROM MIRRORPREREQ  
        WHERE codmodprereq = root;  
    SON C%rowtype;  
BEGIN  
    FOR SON IN C LOOP  
        IF  
            (son.codmod = moduleToFind OR  
            findModule(son.codmod, moduleToFind))  
        THEN  
            RETURN TRUE;  
        END IF;  
    END LOOP;  
    RETURN FALSE;  
END;

---

---

CREATE OR REPLACE PROCEDURE  
insertMirrorPrereq(codmodValue NUMBER, codmodprereqValue NUMBER, note NUMBER) IS  
BEGIN  
    INSERT INTO MIRRORPREREQ  
    (codmod, codmodprereq, noteMin)  
    VALUES  
    (codmodValue, codmodprereqValue, note);  
END;

---

---

```

CREATE OR REPLACE PROCEDURE
  deleteMirrorPrereq(codmodValue NUMBER, codmodprereqValue NUMBER) IS
BEGIN
  DELETE FROM MIRRORPREREQ
    WHERE codmod = codmodValue
    AND codmodprereq = codmodprereqValue;
END;
/

```

```

CREATE OR REPLACE PROCEDURE
  updateMirrorPrereq
  (codmodValue NUMBER,
  codmodNewValue NUMBER,
  codmodprereqValue NUMBER,
  codmodprereqNewValue NUMBER,
  newNote NUMBER) IS
BEGIN
  UPDATE MIRRORPREREQ SET
    codmod = codmodNewValue ,
    codmodprereq = codmodprereqNewValue ,
    noteMin = newNote
  WHERE codmod = codmodValue
    AND codmodprereq = codmodprereqValue;
END;
/

```

```

CREATE OR REPLACE TRIGGER afterDeleteFERPrereq
AFTER DELETE ON PREREQUIS
FOR EACH ROW
BEGIN
  deleteMirrorPrereq(:old.codmod, :old.codmodprereq);
END;
/

```

```

CREATE OR REPLACE TRIGGER beforeInsertUpdateFERPrereq
BEFORE INSERT OR UPDATE ON PREREQUIS
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    insertMirrorPrereq(:new.codmod, :new.codmodprereq, :new.noteMin);
  END IF;
  IF UPDATING THEN
    updateMirrorPrereq(:old.codmod, :new.codmod,
      :old.codmodprereq, :new.codmodprereq, :new.noteMin);
  END IF;
  IF (findModule(:new.codmod, :new.codmod)) THEN
    IF INSERTING THEN
      deleteMirrorPrereq(:new.codmod, :new.codmodprereq);
    END IF;
    IF UPDATING THEN
      updateMirrorPrereq(:new.codmod, :old.codmod,
        :new.codmodprereq, :old.codmodprereq, :old.noteMin);
    END IF;
    RAISE_APPLICATION_ERROR(-20003, 'Circuit dans prerequis. ');
  END IF;
END;
/

```

--- Contrainte 6 ---

```

CREATE OR REPLACE FUNCTION
  checkInscription(etud NUMBER, mod NUMBER)
RETURN BOOLEAN
IS
  CURSOR prereq IS
    SELECT noteMin, codmodprereq
    FROM MIRRORPREREQ
    WHERE codmod = mod;
  p prereq%rowtype;
  nbLignes NUMBER;
BEGIN

```

```

FOR p IN prereq LOOP
    SELECT count(*) INTO nbLignes
    FROM RESULTAT
    WHERE codmod = p.codmodprereq
    AND numetud = etud
    AND note < p.noteMin;
    IF (nbLignes = 0) THEN
        RETURN FALSE;
    END IF;
END LOOP;
RETURN TRUE;
END;
/

-----

CREATE OR REPLACE TRIGGER beforeInsertUpdateFERInsc
BEFORE INSERT OR UPDATE ON INSCRIPTION
FOR EACH ROW
DECLARE
    nbLignes NUMBER;
BEGIN
    SELECT count(*) INTO nbLignes
    FROM modulesDisponibles
    WHERE codmod = :new.codmod;
    IF (nbLignes = 0) THEN
        RAISE_APPLICATION_ERROR(-20001, 'Plus de places disponibles. ');
    END IF;
    IF (NOT(checkInscription(:new.numetud, :new.codmod))) THEN
        RAISE_APPLICATION_ERROR(-20004, 'Pré requis non satisfait. ');
    END IF;
END;
/

-----

-- Contrainte 7 -----

-----

-- La aussi un probleme se pose, on ne peut pas faire de requete sur
-- la table resultat, comme de plus, on tient à prendre pour chaque etudiant
-- le meilleure note dans chaque module, on cree une table temporaire contenant
-- les notes obetnues par les eleves.

DROP TABLE MIRRORRESULT;

CREATE TABLE MIRRORRESULT
(
    numetud NUMBER,
    codmod NUMBER,
    codexam NUMBER,
    note NUMBER,
    PRIMARY KEY(numetud, codmod, codexam)
);

-----

DROP VIEW MEILLEURENOTE;

CREATE VIEW MEILLEURENOTE AS
SELECT numetud, codmod, MAX(note) AS noteMax
FROM MIRRORRESULT
GROUP BY numetud, codmod;

-----

DROP VIEW NOMBREINSCRIPTIONS;

CREATE VIEW NOMBREINSCRIPTIONS AS
SELECT numetud,
(
    SELECT COUNT(*)
    FROM INSCRIPTION I
    WHERE I.numetud = E.numetud
) AS nbInscriptions
FROM ETUDIANT E;

-----

DROP VIEW NOMBRENOTES;

CREATE VIEW NOMBRENOTES AS
SELECT numetud,

```

```

        (SELECT COUNT(*) AS nbNotes
         FROM MEILLEURENOTE M
         WHERE M.numetud = E.numetud
         ) AS nbNotes
FROM ETUDIANT E;

```

```

CREATE OR REPLACE PROCEDURE
  updateMoyenne(etud NUMBER)
IS
  nbNotes NUMBER;
  nbInscriptions NUMBER;
BEGIN
  SELECT nbNotes INTO nbNotes
  FROM NOMBRENOTES
  WHERE numetud = etud;
  SELECT nbInscriptions INTO nbInscriptions
  FROM NOMBREINSCRIPTIONS
  WHERE numetud = etud;
  IF (nbNotes = nbInscriptions) THEN
    UPDATE ETUDIANT SET moyenne =
      (SELECT AVG(noteMax)
       FROM MEILLEURENOTE
       WHERE numetud = etud
      )
    WHERE numetud = etud;
  ELSE
    UPDATE ETUDIANT SET
      moyenne = NULL
    WHERE numetud = etud;
  END IF;
END;
/

```

```

CREATE OR REPLACE TRIGGER afterInsertFERResult
AFTER INSERT ON RESULTAT
FOR EACH ROW
BEGIN
  INSERT INTO MIRRORRESULT VALUES
    (:new.numetud, :new.codmod, :new.codexam, :new.note);
  updateMoyenne(:new.numetud);
END;
/

```

```

CREATE OR REPLACE TRIGGER afterUpdateFERResult
AFTER UPDATE ON RESULTAT
FOR EACH ROW
BEGIN
  UPDATE MIRRORRESULT SET
    numetud = :new.numetud,
    codmod = :new.codmod,
    codexam = :new.codexam,
    note = :new.note
  WHERE numetud = :old.numetud
    AND codmod = :old.codmod
    AND codexam = :old.codexam;
  updateMoyenne(:new.numetud);
END;
/

```

```

CREATE OR REPLACE TRIGGER afterDeleteFERResult
AFTER DELETE ON RESULTAT
FOR EACH ROW
BEGIN
  DELETE FROM MIRRORRESULT
  WHERE numetud = :new.numetud
    AND codmod = :new.codmod
    AND codexam = :new.codexam;
  updateMoyenne(:new.numetud);
END;
/

```

--- Contrainte 9 ---

```

CREATE OR REPLACE FUNCTION
  checkAllStudents
  RETURN BOOLEAN
IS
  CURSOR C IS
    SELECT numetud, codmod
    FROM INSCRIPTION;
  e C%rowtype;
BEGIN
  FOR e IN C LOOP
    IF (NOT(checkInscription(e.numetud, e.codmod))) THEN
      RETURN FALSE;
    END IF;
  END LOOP;
  RETURN TRUE;
END;
/

```

```

CREATE OR REPLACE TRIGGER BeforeUpdateFERModule
BEFORE UPDATE ON MODULE
FOR EACH ROW
BEGIN
  IF (:new.effecmax < :new.effec) THEN
    RAISE_APPLICATION_ERROR(-20005,
      'L effectif ne peut être en dessous de ' || :new.effec);
  END IF;
END;
/

```

--- *Contrainte 8* ---

```

CREATE OR REPLACE TRIGGER beforeInsertUpdateFERPrereq
BEFORE INSERT OR UPDATE ON PREREQUIS
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    insertMirrorPrereq(:new.codmod, :new.codmodprereq, :new.noteMin);
  END IF;
  IF UPDATING THEN
    updateMirrorPrereq(:old.codmod, :new.codmod,
      :old.codmodprereq, :new.codmodprereq, :new.noteMin);
  END IF;
  IF (findModule(:new.codmod, :new.codmod)) THEN
    IF INSERTING THEN
      deleteMirrorPrereq(:new.codmod, :new.codmodprereq);
    END IF;
    IF UPDATING THEN
      updateMirrorPrereq(:new.codmod, :old.codmod,
        :new.codmodprereq, :old.codmodprereq, :old.noteMin);
    END IF;
    RAISE_APPLICATION_ERROR(-20003, 'Circuit dans prerequis. ');
  END IF;
  IF (NOT(checkAllStudents())) THEN
    IF INSERTING THEN
      deleteMirrorPrereq(:new.codmod, :new.codmodprereq);
    END IF;
    IF UPDATING THEN
      updateMirrorPrereq(:new.codmod, :old.codmod,
        :new.codmodprereq, :old.codmodprereq, :old.noteMin);
    END IF;
    RAISE_APPLICATION_ERROR(-20006, 'Impossible de diminuer cette note. ');
  END IF;
END;
/

```

--- *Quelques insertions pour tester* ---

```

INSERT INTO ETUDIANT VALUES
(
  (SELECT nvl(MAX(numEtud), 0) + 1 FROM ETUDIANT),
  'Fourier',
  'Joseph',
  to_date('21031768', 'ddmmyyyy'),
  'Mr',
  NULL,
  '168031234567890',
  NULL
);

```

```

INSERT INTO MODULE
(codMod, nomMod)
VALUES
(
(SELECT nvl(MAX(codMod), 0) + 1 FROM MODULE),
'Maths'
);

INSERT INTO INSCRIPTION
(codMod, numEtud)
VALUES
(
(SELECT numEtud FROM ETUDIANT WHERE nom = 'Fourier'),
(SELECT codMod FROM MODULE WHERE nomMod = 'Maths')
);

INSERT INTO EXAMEN VALUES
(
(SELECT codMod FROM MODULE WHERE nomMod = 'Maths'),
1,
to_date('02012008', 'ddmmyyyy')
);

INSERT INTO RESULTAT VALUES
(
(SELECT codMod FROM MODULE WHERE nomMod = 'Maths'),
1,
(SELECT numEtud FROM ETUDIANT WHERE nom = 'Fourier'),
19
);

UPDATE RESULTAT SET
note = 20
WHERE
numEtud = (SELECT numEtud FROM ETUDIANT WHERE nom = 'Fourier')
AND codMod = (SELECT codMod FROM MODULE WHERE nomMod = 'Maths')
AND codExam = 1;

INSERT INTO MODULE VALUES
(2, 'Algo', 30, 22);

INSERT INTO PREREQUIS VALUES
(1, 2, 10);

INSERT INTO PREREQUIS VALUES
(2, 1, 10);

UPDATE PREREQUIS SET noteMin = 2;

INSERT INTO EXAMEN VALUES (2, 1, sysdate);

```

## 3.8 Packages

```
CREATE OR REPLACE PACKAGE BODY gestion_arbre IS

    cursor feuilles return personne%rowtype IS
        SELECT * FROM PERSONNE;

    PROCEDURE ajoutePersonne(nom personne.nom%type ,
        prenom personne.prenom%type , pere personne.pere%type ,
        mere personne.mere%type) IS
    BEGIN
        INSERT INTO PERSONNE VALUES (
            (SELECT nvl(MAX(numbers), 0) + 1 FROM PERSONNE),
            nom , prenom , pere , mere);
    END;

    FUNCTION descendDe(numbers personne.numbers%type ,
        descendant personne.numbers%type) RETURN BOOLEAN IS
    BEGIN
        IF (descendant IS NULL) THEN
            RETURN FALSE;
        ELSIF (descendant = numbers) THEN
            RETURN TRUE;
        ELSE
            DECLARE
                pers PERSONNE%rowtype;
            BEGIN
                SELECT * INTO pers
                FROM PERSONNE
                WHERE numbers = descendant;
                RETURN descendDe(numbers , pers.pere)
                OR descendDe(numbers , pers.mere);
            END;
        END IF;
    END;

    procedure modifieParents(pers personne.numbers%type ,
        numPere personne.pere%type , numMere personne.mere%type) IS
    BEGIN
        IF (descendDe(pers , numPere) OR descendDe(pers , numMere)) THEN
            RAISE CIRCUIT;
        END IF;
        UPDATE PERSONNE SET pere = numPere , mere = numMere
        WHERE numPers = pers;
    END;

END;
/
CALL gestion_Arbre.modifieParents(20, 14, 15);
```

**[www.Mcours.com](http://www.Mcours.com)**

Site N°1 des Cours et Exercices

Email: [mymcours@gmail.com](mailto:mymcours@gmail.com)

## 3.9 Révisions

```
-- Preparatifs ...

DROP TABLE MIRRORPERSONNE;

CREATE TABLE MIRRORPERSONNE
(
  numpers NUMBER PRIMARY KEY,
  pere NUMBER,
  mere NUMBER
);

CREATE OR REPLACE TRIGGER miseAJourMirrorPersonne
BEFORE UPDATE OR INSERT OR DELETE ON PERSONNE
FOR EACH ROW
BEGIN
  IF DELETING OR UPDATING THEN
    DELETE FROM MIRRORPERSONNE
    WHERE numpers = :old.numpers;
  END IF;
  IF INSERTING OR UPDATING THEN
    INSERT INTO MIRRORPERSONNE VALUES
      (:new.numpers, :new.pere, :new.mere);
  END IF;
END;
/

DROP TABLE MIRRORMARIAGE;

CREATE TABLE MIRRORMARIAGE
(
  nummari NUMBER,
  numfemme NUMBER,
  datemariage DATE,
  datedivorce DATE
);

CREATE OR REPLACE TRIGGER miseAJourMirrorMariage
BEFORE UPDATE OR INSERT OR DELETE ON MARIAGE
FOR EACH ROW
BEGIN
  IF DELETING OR UPDATING THEN
    DELETE FROM MIRRORMARIAGE
    WHERE nummari = :old.nummari
    AND numfemme = :old.numfemme
    AND datemariage = :old.datemariage;
  END IF;
  IF INSERTING OR UPDATING THEN
    INSERT INTO MIRRORMARIAGE VALUES
      (:new.nummari, :new.numfemme, :new.datemariage, :new.datedivorce);
  END IF;
END;
/

-- Contrainte 1

ALTER TABLE PERSONNE ADD CONSTRAINT ck_parents_differents CHECK(pere <> mere);

-- Contrainte 2

CREATE OR REPLACE PACKAGE contrainteCircuit IS
  CIRCUIT EXCEPTION;
  PROCEDURE verifieCircuit(pers personne.numbers%type);
  FUNCTION descendDe(numbers personne.numbers%type,
    descendant personne.numbers%type) RETURN BOOLEAN;
END;
/

CREATE OR REPLACE TRIGGER verifieContrainteCircuit
AFTER UPDATE OR INSERT ON PERSONNE
FOR EACH ROW
BEGIN
  contrainteCircuit.verifieCircuit(:new.numPers);
END;
/

CREATE OR REPLACE PACKAGE BODY contrainteCircuit IS
  FUNCTION descendDe(numbers personne.numbers%type,
    descendant personne.numbers%type) RETURN BOOLEAN IS
  BEGIN
    IF (descendant IS NULL) THEN
      RETURN FALSE;
    ELSIF (descendant = numbers) THEN
      RETURN TRUE;
    ELSE
      DECLARE
```



```

        pers MIRRORPERSONNE%rowtype;
    BEGIN
        SELECT * INTO pers
            FROM MIRRORPERSONNE
            WHERE numpers = descendant;
        RETURN descendDe(numpers, pers.pere)
            OR descendDe(numpers, pers.mere);
    END;
END IF;
END;

PROCEDURE verifieCircuit(pers personne.numpers%type) IS
    ligne mirrorpersonne%rowtype;
BEGIN
    SELECT * INTO LIGNE
        FROM mirrorpersonne
        WHERE numpers = pers;
    IF (descendDe(pers, ligne.pere) OR descendDe(pers, ligne.mere)) THEN
        RAISE CIRCUIT;
    END IF;
END;

END;
/

-- Contrainte 3
ALTER TABLE MARIAGE ADD CONSTRAINT ck_dates_mariage CHECK(dateDivorce IS NULL OR dateMariage <= dateDivorce);

-- Contrainte 4
CREATE OR REPLACE PACKAGE contraintesMariages IS
    mariagesSuperposes EXCEPTION;
    PROCEDURE verifieMariagesSuperposes(nouveauMariage mariage%rowtype);
END contraintesMariages;
/

CREATE OR REPLACE TRIGGER verifieContraintesMariages
BEFORE UPDATE OR INSERT ON MARIAGE
FOR EACH ROW
DECLARE
    nouveauMariage MARIAGE%rowtype;
BEGIN
    nouveauMariage.numMari := :new.numMari;
    nouveauMariage.numFemme := :new.numFemme;
    nouveauMariage.dateMariage := :new.dateMariage;
    nouveauMariage.dateDivorce := :new.dateDivorce;
    contraintesMariages.verifieMariagesSuperposes(nouveauMariage);
END;
/

CREATE OR REPLACE PACKAGE BODY contraintesMariages IS

    FUNCTION seSuperposent(m1 mirrorMariage%rowtype, m2 mirrorMariage%rowtype) RETURN BOOLEAN IS
    BEGIN
        IF (m1.nummari <> m2.nummari OR m1.numfemme <> m2.numfemme) THEN
            RETURN FALSE;
        END IF;
        RETURN NOT(
            (m2.datedivorce IS NOT NULL AND m1.dateMariage <= m2.dateDivorce)
            OR (m1.datedivorce IS NOT NULL AND m2.dateMariage <= m1.dateDivorce)
        );
    END;

    PROCEDURE verifieMariagesSuperposes(nouveauMariage mariage%rowtype) IS
        CURSOR autresMariages IS
            SELECT * FROM MIRRORMARIAGE
            WHERE numMari = nouveauMariage.numMari
            OR numFemme = nouveauMariage.numFemme;
        autreMariage autresMariages%ROWTYPE;
    BEGIN
        FOR autreMariage IN autresMariages LOOP
            IF (seSuperposent(nouveauMariage, autreMariage)) THEN
                RAISE mariagesSuperposes;
            END IF;
        END LOOP;
    END;
END contraintesMariages;
/

-- Contraintes 5 et 6
CREATE OR REPLACE package contraintesTrans IS
    trans EXCEPTION;

    PROCEDURE verifiePereMere(nouvellePersonne MIRRORPERSONNE%rowtype);
    PROCEDURE verifieMariFemme(nouveauMariage MARIAGE%rowtype);

```

```

end contraintesTrans;
/

CREATE OR REPLACE TRIGGER pereMere
AFTER UPDATE OR INSERT ON PERSONNE
FOR EACH ROW
DECLARE
    nouvellePersonne MIRRORPERSONNE%rowtype;
BEGIN
    nouvellePersonne.numpers := :new.numpers ;
    nouvellePersonne.pere := :new.pere ;
    nouvellePersonne.mere := :new.mere ;
    contraintesTrans.verifiePereMere(nouvellePersonne);
END;
/

CREATE OR REPLACE TRIGGER mariFemme
AFTER UPDATE OR INSERT ON MARIAGE
FOR EACH ROW
DECLARE
    nouveauMariage MARIAGE%rowtype;
BEGIN
    nouveauMariage.numMari := :new.numMari;
    nouveauMariage.numFemme := :new.numFemme;
    nouveauMariage.dateMariage := :new.dateMariage;
    nouveauMariage.dateDivorce := :new.dateDivorce;
    contraintesTrans.verifieMariFemme(nouveauMariage);
END;
/

CREATE OR REPLACE package BODY contraintesTrans IS

    PROCEDURE verifiePereMere(nouvellePersonne MIRRORPERSONNE%rowtype) IS
        nb INT;
    BEGIN
        SELECT COUNT(*) INTO nb
        FROM MIRRORPERSONNE
        WHERE pere = nouvellePersonne.mere
        OR mere = nouvellePersonne.pere;
        IF (nb <> 0) THEN
            RAISE TRANS;
        END IF;
        SELECT COUNT(*) INTO nb
        FROM MIRRORMARIAGE
        WHERE numMari = nouvellePersonne.mere
        OR numFemme = nouvellePersonne.pere;
        IF (nb <> 0) THEN
            RAISE TRANS;
        END IF;
    END;

    PROCEDURE verifieMariFemme(nouveauMariage MARIAGE%rowtype) IS
        nb INT;
    BEGIN
        SELECT COUNT(*) INTO nb
        FROM MIRRORMARIAGE
        WHERE numMari = nouveauMariage.numFemme
        OR numFemme = nouveauMariage.numMari;
        IF (nb <> 0) THEN
            RAISE TRANS;
        END IF;
        SELECT COUNT(*) INTO nb
        FROM MIRRORPERSONNE
        WHERE pere = nouveauMariage.numFemme
        OR mere = nouveauMariage.numMari;
        IF (nb <> 0) THEN
            RAISE TRANS;
        END IF;
    END;

END contraintesTrans;
/

-- Contrainte 7

CREATE OR REPLACE PACKAGE contrainteMariageConsanguin IS
    MariageConsanguin EXCEPTION;

    PROCEDURE verifieMariageConsanguin(nouveauMariage MARIAGE%rowtype);
END contrainteMariageConsanguin;
/

CREATE OR REPLACE TRIGGER mariageConsanguin
AFTER UPDATE OR INSERT ON MARIAGE
FOR EACH ROW
DECLARE
    nouveauMariage MARIAGE%rowtype;
BEGIN

```

```

nouveauMariage.numMari := :new.numMari;
nouveauMariage.numFemme := :new.numFemme;
nouveauMariage.dateMariage := :new.dateMariage;
nouveauMariage.dateDivorce := :new.dateDivorce;
contrainteMariageConsanguin.verifieMariageConsanguin(nouveauMariage);
END;
/
CREATE OR REPLACE PACKAGE BODY contrainteMariageConsanguin IS
FUNCTION pere(p PERSONNE.numbers%type) RETURN PERSONNE.numbers%type IS
numPere PERSONNE.numbers%type;
BEGIN
SELECT pere INTO numPere
FROM MIRRORPERSONNE
WHERE numbers = p;
RETURN numPere;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RETURN NULL;
END;

FUNCTION mere(p PERSONNE.numbers%type) RETURN PERSONNE.numbers%type IS
numMere PERSONNE.numbers%type;
BEGIN
SELECT mere INTO numMere
FROM MIRRORPERSONNE
WHERE numbers = p;
RETURN numMere;
EXCEPTION
WHEN NO_DATA_FOUND THEN
RETURN NULL;
END;

FUNCTION rechercheAncetreCommun(a PERSONNE.numbers%type,
b PERSONNE.numbers%type) RETURN BOOLEAN IS
BEGIN
IF (a IS NULL) THEN
RETURN FALSE;
ELSE
RETURN (contrainteCircuit.descendDe(a, b)) OR rechercheAncetreCommun(pere(a), b) OR rechercheAncetreCommun(mere(a), b);
END IF;
END;

PROCEDURE verifieMariageConsanguin(nouveauMariage MARIAGE%rowtype) IS
BEGIN
IF (rechercheAncetreCommun(nouveauMariage.numMari, nouveauMariage.numFemme)) THEN
RAISE MariageConsanguin;
END IF;
END;
END contrainteMariageConsanguin;
/

```

# Annexe A

## Scripts de création de bases

### A.1 Livraisons Sans contraintes

Attention : Le numéro de livraison est une clé secondaire, c'est-à-dire un numéro unique étant donné un fournisseur.

```
CREATE TABLE PRODUIT
(numprod number,
nomprod varchar2(30));

CREATE TABLE FOURNISSEUR
(numfou number,
nomfou varchar2(30));

CREATE TABLE PROPOSER
(numfou number,
numprod number,
prix number);

CREATE TABLE LIVRAISON
(numfou number,
numli number,
dateli date default sysdate
);

CREATE TABLE DETAILLIVRAISON
(numfou number,
numli number,
numprod number,
qte number);
```

## A.2 Modules et prerequis

les modules sont répertoriés dans une table, et les modules pré-requis pour s'y inscrire (avec la note minimale) se trouvent dans la table prerequis. Une ligne de la table PREREQUIS nous indique que pour s'inscrire dans le module numéro numMod, il faut avoir eu au moins noteMin au module numModPrereq.

```
CREATE TABLE MODULE
(numMod number primary key,
nomMod varchar2(30)
);

CREATE TABLE PREREQUIS
(
numMod number references MODULE(numMod),
numModPrereq number references MODULE(numMod),
noteMin number(2) DEFAULT 10 NOT NULL ,
PRIMARY KEY(numMod, numModPrereq)
);

INSERT INTO MODULE VALUES (1, 'Oracle');
INSERT INTO MODULE VALUES (2, 'C++');
INSERT INTO MODULE VALUES (3, 'C');
INSERT INTO MODULE VALUES (4, 'Algo');
INSERT INTO MODULE VALUES (5, 'Merise');
INSERT INTO MODULE VALUES (6, 'PL/SQL Oracle');
INSERT INTO MODULE VALUES (7, 'mySQL');
INSERT INTO MODULE VALUES (8, 'Algo avanc e');

INSERT INTO PREREQUIS (numMod, numModPrereq) VALUES (1, 5);
INSERT INTO PREREQUIS (numMod, numModPrereq) VALUES (2, 3);
INSERT INTO PREREQUIS VALUES (6, 1, 12);
INSERT INTO PREREQUIS (numMod, numModPrereq) VALUES (6, 5);
INSERT INTO PREREQUIS (numMod, numModPrereq) VALUES (8, 5);
INSERT INTO PREREQUIS (numMod, numModPrereq) VALUES (7, 5);
```

## A.3 Géométrie

La table INTERVALLE contient des intervalles spécifiés par leurs bornes inférieure et supérieure. Supprimer de la table intervalle tous les intervalles qui n'en sont pas avec une seule instruction.

```
CREATE TABLE INTERVALLE
(borneInf NUMBER,
 borneSup NUMBER,
 PRIMARY KEY (borneInf , borneSup));

CREATE TABLE RECTANGLE
(xHautGauche NUMBER,
 yHautGauche NUMBER,
 xBasDroit NUMBER,
 yBasDroit NUMBER,
 PRIMARY KEY (xHautGauche , yHautGauche , xBasDroit , yBasDroit));

INSERT INTO INTERVALLE VALUES (2, 56);
INSERT INTO INTERVALLE VALUES (12, 30);
INSERT INTO INTERVALLE VALUES (2, 3);
INSERT INTO INTERVALLE VALUES (12, 3);
INSERT INTO INTERVALLE VALUES (8, 27);
INSERT INTO INTERVALLE VALUES (34, 26);
INSERT INTO INTERVALLE VALUES (5, 10);
INSERT INTO INTERVALLE VALUES (7, 32);
INSERT INTO INTERVALLE VALUES (0, 30);
INSERT INTO INTERVALLE VALUES (21, 8);

INSERT INTO RECTANGLE VALUES (2, 12, 5, 7);
INSERT INTO RECTANGLE VALUES (2, 12, 1, 13);
INSERT INTO RECTANGLE VALUES (10, 13, 1, 11);
INSERT INTO RECTANGLE VALUES (10, 13, 10, 11);
INSERT INTO RECTANGLE VALUES (2, 7, 5, 13);
INSERT INTO RECTANGLE VALUES (21, 73, 15, 22);
INSERT INTO RECTANGLE VALUES (1, 2, 3, 4);
INSERT INTO RECTANGLE VALUES (1, 5, 3, 2);
INSERT INTO RECTANGLE VALUES (1, 6, 3, 6);
INSERT INTO RECTANGLE VALUES (4, 2, 1, 4);
INSERT INTO RECTANGLE VALUES (2, 3, 4, 0);
INSERT INTO RECTANGLE VALUES (5, 4, 2, 1);
```

## A.4 Livraisons

```
CREATE TABLE PRODUIT
(numprod number,
nomprod varchar2(30));

CREATE TABLE FOURNISSEUR
(numfou number,
nomfou varchar2(30));

CREATE TABLE PROPOSER
(numfou number,
numprod number,
prix number NOT NULL);

CREATE TABLE LIVRAISON
(numfou number,
numli number,
dateli date default sysdate
);

CREATE TABLE DETAILLIVRAISON
(numfou number,
numli number,
numprod number,
qte number NOT NULL);

alter table produit add constraint pk_produit
PRIMARY KEY (numprod);
alter table fournisseur add constraint pk_fournisseur
PRIMARY KEY (numfou);
alter table proposer add constraint pk_proposer
PRIMARY KEY (numfou, numprod);
alter table livraison add constraint pk_livraison
PRIMARY KEY (numfou, numli);
alter table detaillivraison add constraint pk_detail_livraison
PRIMARY KEY (numfou, numli, numprod);
alter table proposer add constraint fk_proposer_fournisseur
FOREIGN KEY (numfou) REFERENCES fournisseur (numfou);
alter table proposer add constraint fk_proposer_produit
FOREIGN KEY (numprod) REFERENCES produit (numprod);
alter table livraison add constraint fk_livraison
FOREIGN KEY (numfou) REFERENCES fournisseur (numfou);
alter table detaillivraison add constraint fk_detail_livraison
FOREIGN KEY (numfou, numli) REFERENCES livraison (numfou, numli);
alter table detaillivraison add constraint fk_detail_livraison_proposer
FOREIGN KEY (numfou, numprod) REFERENCES proposer (numfou, numprod);

INSERT INTO PRODUIT values (1, 'Roue de secours');
INSERT INTO PRODUIT values (2, 'Poup e Batman');
INSERT INTO PRODUIT values (3, 'Cotons tiges');
INSERT INTO PRODUIT values (4, 'Cornichons');

INSERT INTO FOURNISSEUR values (1, 'f1');
INSERT INTO FOURNISSEUR values (2, 'f2');
INSERT INTO FOURNISSEUR values (3, 'f3');
INSERT INTO FOURNISSEUR values (4, 'f4');

INSERT INTO PROPOSER values (1, 1, 200);
INSERT INTO PROPOSER values (1, 2, 15);
INSERT INTO PROPOSER values (2, 2, 1);
INSERT INTO PROPOSER values (3, 3, 2);

INSERT INTO LIVRAISON (numfou, numli) values (1, 1);
INSERT INTO LIVRAISON (numfou, numli) values (1, 2);
INSERT INTO LIVRAISON (numfou, numli) values (3, 1);

INSERT INTO DETAILLIVRAISON values (3, 1, 3, 10);
INSERT INTO DETAILLIVRAISON values (1, 1, 1, 25);
INSERT INTO DETAILLIVRAISON values (1, 1, 2, 20);
INSERT INTO DETAILLIVRAISON values (1, 2, 1, 15);
INSERT INTO DETAILLIVRAISON values (1, 2, 2, 17);
```

## A.5 Arbre généalogique

La table PERSONNE, le champ `pere` contient le numéro du père de la personne, le champ `mere` contient le numéro de la mère de la personne.

```
CREATE TABLE PERSONNE
(numbers number PRIMARY KEY,
 nom varchar2(30) NOT NULL,
 prenom varchar2(30),
 pere REFERENCES PERSONNE(numbers),
 mere REFERENCES PERSONNE(numbers)
);

INSERT INTO PERSONNE VALUES (1, 'de Montmirail, dit le Hardi', 'Godefroy', NULL, NULL);
INSERT INTO PERSONNE VALUES (16, 'ET', NULL, NULL, NULL);
INSERT INTO PERSONNE VALUES (2, 'Le Croquant', 'Jacou', 1, 16);
INSERT INTO PERSONNE VALUES (3, 'La Fripouille', 'Jacquille', 1, 16);
INSERT INTO PERSONNE VALUES (4, 'Bush', 'Kate', NULL, NULL);
INSERT INTO PERSONNE VALUES (13, 'Granger', 'Hermione', NULL, NULL);
INSERT INTO PERSONNE VALUES (5, 'Du FÃ©mur', 'MÃ©dor', 3,4 );
INSERT INTO PERSONNE VALUES (12, 'KobalevskaÃ´a', 'Sofia', NULL, NULL);
INSERT INTO PERSONNE VALUES (6, 'Rieu', 'AndrÃ©', NULL, NULL);
INSERT INTO PERSONNE VALUES (7, 'Bontoutou', 'Rex', 6, 4);
INSERT INTO PERSONNE VALUES (8, 'Dijkstra', 'Edvard', 2, 13);
INSERT INTO PERSONNE VALUES (9, 'Leibniz', 'Gottfried Wilhelm', 8, 12);
INSERT INTO PERSONNE VALUES (10, 'Bach', 'Johann Sebastien', 5, 12);
INSERT INTO PERSONNE VALUES (17, 'Mathieu', 'Mireille', NULL, NULL);
INSERT INTO PERSONNE VALUES (11, 'Lemarchal', 'Gregory', 10, 17);
INSERT INTO PERSONNE VALUES (15, 'Socrate', NULL, 3, 13);
INSERT INTO PERSONNE VALUES (19, 'Leroy', 'Nolwen', NULL, NULL);
INSERT INTO PERSONNE VALUES (20, 'Bartoli', 'Jennifer', 9, 19);
INSERT INTO PERSONNE VALUES (21, 'Fabian', 'Lara', 10, 17);
INSERT INTO PERSONNE VALUES (14, 'Stone', 'Sharon', 15, 20);
INSERT INTO PERSONNE VALUES (18, 'Frege', 'Elodie', 7, 13);
```



## A.6 Comptes bancaires

```
DROP TABLE OPERATION;
DROP TABLE TYPEOPERATION;
DROP TABLE COMPTECLIENT;
DROP TABLE TYPECCL;
DROP TABLE PERSONNEL;
DROP TABLE CLIENT;

CREATE TABLE CLIENT
(numcli number,
 nomcli varchar2(30),
 prenomcli varchar2(30),
 adresse varchar2(60),
 tel varchar(10)
);

CREATE TABLE PERSONNEL
(numpers number,
 nompers varchar2(30),
 prenompers varchar2(30),
 manager number,
 salaire number
);

CREATE TABLE TYPECCL
(numtypeccl number,
 nontypeccl varchar2(30)
);

CREATE TABLE COMPTECLIENT
(numcli number,
 numccl number,
 numtypeccl number,
 dateccl date default sysdate not null,
 numpers number
);

CREATE TABLE TYPEOPERATION
(numtypeoper number,
 nontypeoper varchar2(30)
);

CREATE TABLE OPERATION
(numcli number,
 numccl number,
 numoper number,
 numtypeoper number,
 dateoper date default sysdate not null,
 montantoper number not null,
 libeloper varchar2(30)
);

ALTER TABLE CLIENT ADD
(
 CONSTRAINT pk_client PRIMARY KEY (numcli),
 CONSTRAINT ck_telephone CHECK(LENGTH(tel)=10)
);

ALTER TABLE PERSONNEL ADD
(
 CONSTRAINT pk_personnel PRIMARY KEY (numpers),
 CONSTRAINT ck_salaire CHECK(SALAIRE >= 1254.28)
);

ALTER TABLE TYPECCL ADD
 CONSTRAINT pk_typeccl PRIMARY KEY (numtypeccl);

ALTER TABLE TYPEOPERATION ADD
 CONSTRAINT pk_typeoperation PRIMARY KEY (numtypeoper);

ALTER TABLE COMPTECLIENT ADD
(
 CONSTRAINT pk_compteclient
 PRIMARY KEY (numcli, numccl),
 CONSTRAINT fk_ccl_typeccl
 FOREIGN KEY (numtypeccl)
 REFERENCES TYPECCL (numtypeccl),
 CONSTRAINT fk_ccl_client
 FOREIGN KEY (numcli)
 REFERENCES CLIENT (numcli),
 CONSTRAINT fk_ccl_personnel
 FOREIGN KEY (numpers)
 REFERENCES PERSONNEL (numpers)
);

ALTER TABLE OPERATION ADD
(
```

```

CONSTRAINT pk_operation
PRIMARY KEY (numcli, numccl, numoper),
CONSTRAINT fk_oper_ccl
FOREIGN KEY (numcli, numoper)
REFERENCES COMPTECLIENT (numcli, numccl),
CONSTRAINT fk_oper_codeoper
FOREIGN KEY (numtypeoper)
REFERENCES typeoperation (numtypeoper),
CONSTRAINT montant_operation
CHECK(montantoper <> 0)
);

INSERT INTO TYPECCL VALUES (
(SELECT nv1(MAX(numtypeccl), 0) + 1
FROM TYPECCL
),
'Compte courant');

INSERT INTO TYPECCL VALUES (
(SELECT nv1(MAX(numtypeccl), 0) + 1
FROM TYPECCL
),
'livret');

INSERT INTO TYPECCL VALUES (
(SELECT nv1(MAX(numtypeccl), 0) + 1
FROM TYPECCL
),
'PEL');

INSERT INTO TYPEOPERATION VALUES (
(SELECT nv1(MAX(numtypeoper), 0) + 1
FROM TYPEOPERATION
),
'dÃ©pÃ´t espÃ©ces');

INSERT INTO TYPEOPERATION VALUES (
(SELECT nv1(MAX(numtypeoper), 0) + 1
FROM TYPEOPERATION
),
'prÃ©lÃ©vement');

INSERT INTO TYPEOPERATION VALUES (
(SELECT nv1(MAX(numtypeoper), 0) + 1
FROM TYPEOPERATION
),
'virement');

INSERT INTO TYPEOPERATION VALUES (
(SELECT nv1(MAX(numtypeoper), 0) + 1
FROM TYPEOPERATION
),
'retrait');

```

## A.7 Comptes bancaires avec exceptions

```
DROP TABLE OPERATION;
DROP TABLE COMPTECLIENT;
DROP TABLE TYPECCL;
DROP TABLE TYPEOPERATION;
DROP TABLE PERSONNEL;
DROP TABLE CLIENT;

CREATE TABLE CLIENT
(numcli number,
 nomcli varchar2(30),
 prenomcli varchar2(30),
 adresse varchar2(60),
 tel varchar(10)
);

CREATE TABLE PERSONNEL
(numpers number,
 nompers varchar2(30),
 prenompers varchar2(30),
 manager number,
 salaire number
);

CREATE TABLE TYPECCL
(numtypeccl number,
 nontypeccl varchar2(30)
);

CREATE TABLE COMPTECLIENT
(numcli number,
 numccl number,
 numtypeccl number,
 dateccl date default sysdate not null,
 numpers number
);

CREATE TABLE TYPEOPERATION
(numtypeoper number,
 nontypeoper varchar2(30)
);

CREATE TABLE OPERATION
(numcli number,
 numccl number,
 numoper number,
 numtypeoper number,
 dateoper date default sysdate not null,
 montantoper number not null,
 libeloper varchar2(30)
);

ALTER TABLE CLIENT ADD
(
 CONSTRAINT pk_client PRIMARY KEY (numcli),
 CONSTRAINT ck_telephone CHECK(LENGTH(tel)=10)
);

ALTER TABLE PERSONNEL ADD
(
 CONSTRAINT pk_personnel PRIMARY KEY (numpers),
 CONSTRAINT ck_salaire CHECK(SALAIRE >= 1254.28)
);

ALTER TABLE TYPECCL ADD
 CONSTRAINT pk_typeccl PRIMARY KEY (numtypeccl);

ALTER TABLE TYPEOPERATION ADD
 CONSTRAINT pk_typeoperation PRIMARY KEY (numtypeoper);

ALTER TABLE COMPTECLIENT ADD
(
 CONSTRAINT pk_compteclient
 PRIMARY KEY (numcli, numccl),
 CONSTRAINT fk_ccl_typeccl
 FOREIGN KEY (numtypeccl)
 REFERENCES TYPECCL (numtypeccl),
 CONSTRAINT fk_ccl_client
 FOREIGN KEY (numcli)
 REFERENCES CLIENT (numcli),
 CONSTRAINT fk_ccl_personnel
 FOREIGN KEY (numpers)
 REFERENCES PERSONNEL (numpers)
);

ALTER TABLE OPERATION ADD
(
```

```

CONSTRAINT pk_operation
PRIMARY KEY (numcli, numccl, numoper),
CONSTRAINT fk_oper_ccl
FOREIGN KEY (numcli, numoper)
REFERENCES COMPTECLIENT (numcli, numccl),
CONSTRAINT fk_oper_codeoper
FOREIGN KEY (numtypeoper)
REFERENCES typeoperation (numtypeoper),
CONSTRAINT montant_operation
CHECK(montantoper <> 0 AND montantoper >= -1000 AND montantoper <= 1000)
);

INSERT INTO TYPECCL VALUES (
(SELECT nv1(MAX(numtypeccl), 0) + 1
FROM TYPECCL
),
'Compte courant');

INSERT INTO TYPECCL VALUES (
(SELECT nv1(MAX(numtypeccl), 0) + 1
FROM TYPECCL
),
'livret');

INSERT INTO TYPECCL VALUES (
(SELECT nv1(MAX(numtypeccl), 0) + 1
FROM TYPECCL
),
'PEL');

INSERT INTO TYPEOPERATION VALUES (
(SELECT nv1(MAX(numtypeoper), 0) + 1
FROM TYPEOPERATION
),
'd  p  t esp  ces');

INSERT INTO TYPEOPERATION VALUES (
(SELECT nv1(MAX(numtypeoper), 0) + 1
FROM TYPEOPERATION
),
'pr  l  vement');

INSERT INTO TYPEOPERATION VALUES (
(SELECT nv1(MAX(numtypeoper), 0) + 1
FROM TYPEOPERATION
),
'virement');

INSERT INTO TYPEOPERATION VALUES (
(SELECT nv1(MAX(numtypeoper), 0) + 1
FROM TYPEOPERATION
),
'retrait');

```

## A.8 Secrétariat pédagogique

```
DROP TABLE RESULTAT;
DROP TABLE EXAMEN;
DROP TABLE PREREQUIS;
DROP TABLE INSCRIPTION;
DROP TABLE MODULE;
DROP TABLE ETUDIANT;

CREATE TABLE ETUDIANT
  (numEtud number,
  nom varchar2(40),
  prenom varchar2(40),
  datenaiss date,
  civilite varchar2(4),
  patronyme varchar2(40),
  numsecu varchar2(15) NOT NULL);

CREATE TABLE MODULE
  (codMod number,
  nomMod varchar2(15),
  effecMax number DEFAULT 30);

CREATE TABLE EXAMEN
  (codMod number,
  codExam number,
  dateExam date);

CREATE TABLE INSCRIPTION
  (numEtud number,
  codMod number,
  dateInsc date default sysdate);

CREATE TABLE PREREQUIS
  (codMod number,
  codModPrereq number,
  noteMin number(4, 2) NOT NULL);

CREATE TABLE RESULTAT
  (codMod number,
  codExam number,
  numEtud number,
  note number(4, 2));

ALTER TABLE ETUDIANT ADD
  CONSTRAINT pk_etudiant
  PRIMARY KEY (numEtud);
ALTER TABLE MODULE ADD
  CONSTRAINT pk_module
  PRIMARY KEY (codMod);
ALTER TABLE EXAMEN ADD
  CONSTRAINT pk_examen
  PRIMARY KEY (codMod, codExam);
ALTER TABLE PREREQUIS ADD
  CONSTRAINT pk_prerequis
  PRIMARY KEY (codMod, codModPrereq);
ALTER TABLE INSCRIPTION ADD
  CONSTRAINT pk_inscription
  PRIMARY KEY (codMod, numEtud);
ALTER TABLE RESULTAT ADD
  CONSTRAINT pk_resultat
  PRIMARY KEY (codMod, numEtud, codExam);

ALTER TABLE INSCRIPTION ADD
  (CONSTRAINT fk_inscription_etudiant
  FOREIGN KEY (numEtud)
  REFERENCES ETUDIANT(numEtud),
  CONSTRAINT fk_inscription_module
  FOREIGN KEY (codMod)
  REFERENCES MODULE(codMod));
ALTER TABLE PREREQUIS ADD
  (CONSTRAINT fk_prerequis_codmod
  FOREIGN KEY (codMod)
  REFERENCES MODULE(codMod),
  CONSTRAINT fk_prerequis_codmodprereq
  FOREIGN KEY (codModPrereq)
  REFERENCES MODULE(codMod));
ALTER TABLE EXAMEN ADD
  CONSTRAINT fk_examen
  FOREIGN KEY (codMod)
  REFERENCES MODULE(codMod);
ALTER TABLE RESULTAT ADD
  (CONSTRAINT fk_resultat_examen
  FOREIGN KEY (codMod, codExam)
  REFERENCES EXAMEN(codMod, codExam),
  CONSTRAINT fk_resultat_inscription
  FOREIGN KEY (codMod, numEtud)
  REFERENCES INSCRIPTION(codMod, numEtud));
```

```

ALTER TABLE ETUDIANT ADD
  (CONSTRAINT ck_civilite
  CHECK
    (
      civilite IN ( 'Mr', 'Mme', 'Mlle' )
    ),
  CONSTRAINT ck_civilite_numsecu
  CHECK
    (
      SUBSTR(numsecu, 1, 1) = '2' OR patronyme IS NULL
    ),
  CONSTRAINT ck_length_numsecu
  CHECK
    (
      length(numsecu) = 15
    ),
  CONSTRAINT ck_annee_numsecu CHECK
    (
      to_char(datenaiss, 'yy') = substr(numsecu, 2, 2)
    )
);

```

**[www.Mcours.com](http://www.Mcours.com)**

Site N°1 des Cours et Exercices

Email: [mymcours@gmail.com](mailto:mymcours@gmail.com)

## A.9 Mariages

```
CREATE TABLE PERSONNE
(numpers number PRIMARY KEY,
 nom varchar2(30) NOT NULL,
 prenom varchar2(30),
 pere REFERENCES PERSONNE(numpers),
 mere REFERENCES PERSONNE(numpers)
);

CREATE TABLE MARIAGE
(
 nummari NUMBER REFERENCES PERSONNE(numpers),
 numfemme NUMBER REFERENCES PERSONNE(numpers),
 datemariage DATE DEFAULT SYSDATE,
 datedivorce DATE DEFAULT NULL,
 PRIMARY KEY(nummari, numfemme, dateMariage)
);
```