Partie 1

Langage de programmation procédural de MySQL

1. Déclaration de variables locales (dans un bloc ou une procédure)

Les variables locales sont utilisées dans les blocs de code tels que les procédures stockées, les fonctions, ou les déclencheurs. Elles sont déclarées avec la commande DECLARE et ne sont disponibles que dans le contexte où elles sont créées.

Exemple 1 d'utilisation dans une procédure stockée :

```
DELIMITER //

CREATE PROCEDURE exemple_procedure()
BEGIN
    DECLARE ma_variable INT;  -- Déclaration d'une variable locale
    SET ma_variable = 10;     -- Affectation d'une valeur à la variable

    -- Utilisation de la variable dans une requête
    SELECT ma_variable;
END //

DELIMITER ;
```

Dans cet exemple, ma_variable est une variable locale qui n'est accessible qu'à l'intérieur de la procédure exemple_procedure.

Exemple 2 d'utilisation dans une procédure stockée :

```
DELIMITER $$

CREATE PROCEDURE calcul_salaire_total()
BEGIN
    -- Déclaration de variables locales
    DECLARE salaire_base DECIMAL(10, 2);
    DECLARE bonus DECIMAL(10, 2);
    DECLARE salaire_total DECIMAL(10, 2);

    -- Initialisation des variables
    SET salaire_base = 2000.00;
    SET bonus = 500.00;

    -- Calcul du salaire total
    SET salaire_total = salaire_base + bonus;

    -- Affichage du résultat
    SELECT CONCAT('Le salaire total est : ', salaire_total) AS resultat;
END $$

DELIMITER ;
```

2. Déclaration de variables utilisateur (variables de session)

Les variables utilisateur sont des variables que vous pouvez déclarer au niveau de la session MySQL. Elles persistent tant que la session est active, et peuvent être utilisées dans plusieurs requêtes dans la même session.

Exemple d'utilisation des variables utilisateur :

```
SET @ma_variable_utilisateur = 25;  -- Déclaration et affectation de la variable
SELECT @ma_variable_utilisateur;    -- Utilisation de la variable
```

Les variables utilisateur commencent toujours par le symbole @ et peuvent être utilisées dans des requêtes ultérieures au sein de la même session.

Différences :

- **Les variables locales** ne sont valides qu'à l'intérieur du bloc de code où elles sont déclarées, comme les procédures ou les déclencheurs.
- **Les variables utilisateur** peuvent être utilisées entre différentes requêtes, mais seulement pendant la durée de vie de la session.

Ces deux types de variables permettent de gérer des données temporaires et d'améliorer la flexibilité dans les requêtes SQL.

3. IF et ELSEIF dans une procédure stockée

L'instruction IF permet de gérer des conditions, tandis que ELSEIF permet d'ajouter des conditions supplémentaires. Voici un exemple dans une procédure stockée où on utilise IF...ELSEIF...ELSE.

Exemple : Utilisation de IF...ELSEIF...ELSE

```
DELIMITER //

CREATE PROCEDURE verifier_age(IN age INT)
BEGIN
    IF age < 18 THEN
        SELECT 'Vous êtes mineur.' AS resultat;
    ELSEIF age >= 18 AND age < 65 THEN
        SELECT 'Vous êtes adulte.' AS resultat;
    ELSE
        SELECT 'Vous êtes senior.' AS resultat;
    END IF;
END //

DELIMITER ;
```

EXECUTION :

```
SET @p0='16'; CALL `verifier_age`(@p0);
SET @p0='50'; CALL `verifier_age`(@p0);
```

4. L'instruction CASE

CASE est une autre structure conditionnelle qui permet de simplifier plusieurs conditions dans une seule expression. Elle peut être utilisée soit dans une requête SQL, soit dans un bloc de code comme une procédure.

Exemple 1 : Utilisation de CASE dans une requête SQL

Voici un exemple d'utilisation de CASE pour afficher un message selon la note d'un étudiant.

```
1--
CREATE TABLE etudiants (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(100) NOT NULL,
    age INT NOT NULL,
    note DECIMAL(5,2) NOT NULL
);

2--
INSERT INTO etudiants (nom, age, note)
VALUES
('Alice Dupont', 20, 85.50),
('Bob Martin', 22, 78.75),
('Charlie Durand', 19, 92.00),
('Diane Bernard', 21, 45.00),
('Eve Leroy', 23, 60.25);

--3
SELECT nom,
       note,
       CASE
           WHEN note >= 90 THEN 'Excellent'
           WHEN note >= 75 THEN 'Bien'
           WHEN note >= 50 THEN 'Passable'
           ELSE 'Échec'
       END AS evaluation
FROM etudiants;
```

Exemple 2 : Utilisation de CASE dans une procédure stockée

Voici un exemple similaire dans une procédure :

```
CREATE PROCEDURE evaluer_note(IN note INT)
BEGIN
    SELECT CASE
        WHEN note >= 90 THEN 'Excellent'
        WHEN note >= 75 THEN 'Bien'
        WHEN note >= 50 THEN 'Passable'
        ELSE 'Échec'
    END AS evaluation;
END //
```

EXECUTION :

```
SET @p0='80'; CALL `evaluer_note`(@p0);
```

```
SET @p1='33'; CALL `evaluer_note`(@p1);
```

5. Les instructions itératifs (Boucle)

En MySQL, il n'y a pas de boucle FOR directement comme dans d'autres langages de programmation, mais vous pouvez utiliser les boucles **WHILE**, REPEAT, et **LOOP** dans des procédures stockées ou des fonctions pour exécuter des instructions répétées.

5.1. Boucle WHILE

La boucle **WHILE** répète les instructions tant qu'une condition est vraie. Voici un exemple de procédure utilisant une boucle WHILE pour afficher les nombres de 1 à 5.

Exemple1 : Utilisation de la boucle WHILE

```
DELIMITER //

CREATE PROCEDURE boucle_while()
BEGIN
    DECLARE i INT DEFAULT 1;  -- Déclaration et initialisation de la variable

    WHILE i <= 5 DO
        SELECT CONCAT('i = ', i);  -- Affiche la valeur de i
        SET i = i + 1;             -- Incrémentation de i
    END WHILE;
END //

DELIMITER ;
```

5.2. Boucle REPEAT

La boucle REPEAT fonctionne de manière similaire à WHILE, mais la condition est vérifiée **à la fin** de chaque itération (donc la boucle s'exécute au moins une fois).

Exemple : Utilisation de la boucle REPEAT

```
DELIMITER //

CREATE PROCEDURE boucle_repeat()
BEGIN
    DECLARE i INT DEFAULT 1;  -- Initialisation de la variable

    REPEAT
        SELECT CONCAT('i = ', i);  -- Affiche la valeur de i
        SET i = i + 1;             -- Incrémente la variable i
    UNTIL i > 5                    -- Condition d'arrêt : quand i dépasse 5
    END REPEAT;
END //

DELIMITER ;
```

5.3. Boucle LOOP

La boucle LOOP est une structure de boucle générale dans MySQL, qui continue jusqu'à ce qu'une commande explicite LEAVE soit exécutée.

Exemple : Utilisation de la boucle LOOP

```
DELIMITER //

CREATE PROCEDURE boucle_loop()
BEGIN
    DECLARE i INT DEFAULT 1;  -- Initialisation de la variable

    boucle: LOOP
        IF i > 5 THEN          -- Condition d'arrêt de la boucle
            LEAVE boucle;
        END IF;

        SELECT CONCAT('i = ', i);  -- Affiche la valeur de i
        SET i = i + 1;             -- Incrémentation de la variable i
    END LOOP;
END //

DELIMITER ;
```

6. Le types de paramètres

En MySQL, une procédure peut avoir différents types de paramètres qui déterminent comment les données sont passées à la procédure et comment elles sont manipulées. Les trois principaux types de paramètres sont :

1. **IN** : Paramètre d'entrée. Il passe une valeur à la procédure, mais cette valeur ne peut pas être modifiée à l'intérieur de la procédure.
2. **OUT** : Paramètre de sortie. Il permet à la procédure de renvoyer une valeur modifiée.
3. **INOUT** : Paramètre d'entrée et de sortie. Il passe une valeur à la procédure, mais la procédure peut également la modifier et la renvoyer.

1. Paramètre IN

Le paramètre IN est utilisé pour fournir des valeurs à une procédure. Ces valeurs ne peuvent pas être modifiées dans la procédure.

Exemple avec IN

Supposons que nous ayons une table étudiante, et nous voulons créer une procédure qui affiche les informations d'un étudiant en fonction de son id.

```
DELIMITER //

CREATE PROCEDURE afficher_etudiant(IN etudiant_id INT)
BEGIN
    SELECT nom, age, note
    FROM etudiants
    WHERE id = etudiant_id;
END //

DELIMITER ;
```

```
SET @p0='2'; CALL `afficher_etudiant`(@p0);
```

2. Paramètre OUT

Le paramètre OUT permet à une procédure de renvoyer une valeur. Contrairement à IN, vous ne passez pas de valeur au paramètre lors de l'appel, mais vous pouvez récupérer une valeur à la fin de l'exécution de la procédure.

Exemple avec OUT

Créons une procédure qui retourne la note d'un étudiant en fonction de son id.

```
DELIMITER //

CREATE PROCEDURE obtenir_note(IN etudiant_id INT, OUT etudiant_note DECIMAL(5,2))
BEGIN
    SELECT note INTO etudiant_note
    FROM etudiants
    WHERE id = etudiant_id;
END //

DELIMITER ;
```

EXECUTION:

```
SET @p0='2'; SET @p1=''; CALL `obtenir_note`(@p0, @p1);
SELECT @p1 AS `etudiant_note`;
```

3. Paramètre INOUT

Le paramètre INOUT agit à la fois comme un paramètre d'entrée et un paramètre de sortie. Vous passez une valeur à la procédure et la procédure peut modifier cette valeur et la renvoyer.

Exemple avec INOUT

Créons une procédure qui augmente la note d'un étudiant de 10%, puis retourne la nouvelle note.

```
DELIMITER //

CREATE PROCEDURE augmenter_note(INOUT etudiant_note DECIMAL(5,2))
BEGIN
    SET etudiant_note = etudiant_note * 1.10;
END //

DELIMITER ;
```

EXECUTION :

```
SET @p0='10'; CALL `augmenter_note`(@p0);
SELECT @p0 AS `etudiant_note`;
```

7. Les Fonctions

En MySQL, les fonctions sont des blocs de code réutilisables qui prennent des arguments, exécutent une série d'instructions, et retournent une valeur. Contrairement aux procédures, les fonctions doivent retourner une valeur et elles peuvent être utilisées directement dans des requêtes SQL, comme dans les clauses **SELECT, WHERE, ou ORDER BY.**

Structure d'une Fonction en MySQL

Voici la syntaxe générale pour créer une fonction en MySQL :

```
CREATE FUNCTION nom_de_la_fonction (paramètre1 TYPE, paramètre2 TYPE, ...)
RETURNS TYPE
BEGIN
    -- corps de la fonction
    RETURN valeur;
END;
```

7.1. Exemple simple : Fonction pour ajouter deux nombres

Créons une fonction qui prend deux nombres et retourne leur somme.

```
DELIMITER //

CREATE FUNCTION additionner_deux_nombres(a INT, b INT)
RETURNS INT
BEGIN
    RETURN a + b;
END //

DELIMITER ;
```

EXECUTION:

```
SELECT additionner_deux_nombres(10, 20) AS somme;
```

7.2. Exemple : Fonction pour calculer une moyenne

Cette fonction calcule la moyenne de deux nombres.

```
DELIMITER //
CREATE FUNCTION moyenne_deux_nombres(a DECIMAL(5,2), b DECIMAL(5,2))
RETURNS DECIMAL(5,2)
BEGIN
    RETURN (a + b) / 2;
END //
DELIMITER ;
```

EXECUTION :

```
SELECT moyenne_deux_nombres(75.5, 88.0) AS moyenne;
```

7.3. Exemple : Fonction pour récupérer un étudiant par son ID

Imaginons que vous ayez une table etudiants (comme dans les exemples précédents) et que vous souhaitiez récupérer le nom d'un étudiant en fonction de son ID.

Création de la table etudiants pour l'exemple :

```
CREATE TABLE etudiants (
    id INT AUTO_INCREMENT PRIMARY KEY,
    nom VARCHAR(100) NOT NULL,
    age INT NOT NULL,
    note DECIMAL(5,2) NOT NULL
);

INSERT INTO etudiants (nom, age, note)
VALUES
('Alice Dupont', 20, 85.50),
('Bob Martin', 22, 78.75),
('Charlie Durand', 19, 92.00),
('Diane Bernard', 21, 45.00);
```

```
DELIMITER //
CREATE FUNCTION obtenir_nom_etudiant(etudiant_id INT)
RETURNS VARCHAR(100)
BEGIN
    DECLARE nom_etudiant VARCHAR(100);

    SELECT nom INTO nom_etudiant
    FROM etudiants
    WHERE id = etudiant_id;

    RETURN nom_etudiant;
END //
DELIMITER ;
```

EXECUTION:

```
SELECT obtenir_nom_etudiant(1) AS nom;
```

7.4. Exemple : Fonction pour augmenter la note d'un étudiant

Voici une fonction qui prend la note actuelle d'un étudiant et l'augmente de 10%.

```
DELIMITER //

CREATE FUNCTION augmenter_note(note_actuelle DECIMAL(5,2))
RETURNS DECIMAL(5,2)
BEGIN
    RETURN note_actuelle * 1.10;
END //

DELIMITER ;
```

EXECUTION:

```
SELECT augmenter_note(75.50) AS nouvelle_note;
```

7.5. Exemple : Fonction pour vérifier si une note est valide

Cette fonction vérifie si une note est valide (entre 0 et 100).

```
DELIMITER //

CREATE FUNCTION note_valide(note DECIMAL(5,2))
RETURNS BOOLEAN
BEGIN
    IF note >= 0 AND note <= 100 THEN
        RETURN TRUE;
    ELSE
        RETURN FALSE;
    END IF;
END //

DELIMITER ;
```

EXECUTION :

```
SELECT note_valide(85.50) AS est_valide;
```

7.6. La déclaration informative

Elle peut prendre l'une des valeurs :

1. **DETERMINISTIC / NOT DETERMINISTIC** :
    - **DETERMINISTIC** : Indique que la fonction renverra toujours le même résultat lorsqu'elle est appelée avec les mêmes arguments.
    - **NOT DETERMINISTIC** : Indique que la fonction pourrait retourner des résultats différents pour les mêmes arguments, par exemple si elle utilise une fonction non déterministe comme RAND() ou NOW().
2. **READS SQL DATA** :
    - Indique que la fonction lit des données SQL (via des requêtes SELECT par exemple) mais ne modifie pas la base de données.

3. **MODIFIES SQL DATA** :
    - Indique que la fonction peut modifier la base de données (par exemple en utilisant INSERT, UPDATE, DELETE, etc.).
4. **CONTAINS SQL** :
    - Indique que la fonction contient des instructions SQL, mais ne lit ni ne modifie les données de la base (cela peut inclure des requêtes qui n'interagissent pas directement avec les données, comme SET).

Exemples pour chaque cas :

1. Fonction DETERMINISTIC

Une fonction déterministe retourne toujours le même résultat pour les mêmes arguments.

Exemple :

```
DELIMITER //

CREATE FUNCTION ajouter_cinq(valeur INT)
RETURNS INT
DETERMINISTIC
BEGIN
    RETURN valeur + 5;
END //

DELIMITER ;
```

Cette fonction est **DETERMINISTIC** parce que pour un même argument, elle renverra toujours le même résultat. Par exemple, si vous passez 10 à cette fonction, elle retournera toujours 15.

2. Fonction NOT DETERMINISTIC

Si la fonction utilise une fonction comme RAND() ou NOW(), elle sera considérée comme **NOT DETERMINISTIC**.

Exemple :

```
DELIMITER //

CREATE FUNCTION obtenir_nombre_aleatoire()
RETURNS DECIMAL(5,2)
NOT DETERMINISTIC
BEGIN
    RETURN RAND();
END //

DELIMITER ;
```

```
SELECT obtenir_nombre_aleatoire()
```

Cette fonction est **NOT DETERMINISTIC** parce que la fonction RAND() retourne un nombre aléatoire à chaque appel, même si les arguments sont les mêmes.

3. Fonction READS SQL DATA

Une fonction qui lit les données d'une table sans les modifier doit être marquée comme **READS SQL DATA**.

Exemple :

```
DELIMITER //
CREATE FUNCTION obtenir_nom_etudiant1(etudiant_id INT)
RETURNS VARCHAR(100)
READS SQL DATA
BEGIN
    DECLARE nom_etudiant VARCHAR(100);

    SELECT nom INTO nom_etudiant
    FROM etudiants
    WHERE id = etudiant_id;

    RETURN nom_etudiant;
END //
DELIMITER ;
```

```
select obtenir_nom_etudiant1(2) nom_etutiant;
```

Cette fonction **lit** les données de la table **etudiants** (via SELECT), mais ne modifie pas la base de données. Elle est donc marquée comme **READS SQL DATA**.

4. Fonction MODIFIES SQL DATA

Si la fonction effectue une modification sur la base de données, elle doit être marquée comme :
MODIFIES SQL DATA.

Exemple :

```
DELIMITER //
CREATE FUNCTION augmenter_note_etudiant(etudiant_id INT, pourcentage
DECIMAL(5,2))
RETURNS BOOLEAN
MODIFIES SQL DATA
BEGIN
    UPDATE etudiants
    SET note = note * (1 + pourcentage / 100)
    WHERE id = etudiant_id;

    RETURN TRUE;
END //
DELIMITER ;
```

```
SELECT augmenter_note_etudiant(1,20);
```

Cette fonction modifie la base de données en mettant à jour la colonne note de la table etudiants. Elle est donc marquée comme **MODIFIES SQL DATA**.

5. Fonction CONTAINS SQL

Si une fonction contient des instructions SQL mais n'interagit pas avec les données de la base (elle ne lit ni ne modifie des données), elle est marquée comme **CONTAINS SQL**.

Exemple :

```
DELIMITER //
CREATE FUNCTION multiplier_par_deux(valeur INT)
RETURNS INT
CONTAINS SQL
BEGIN
    RETURN valeur * 2;
END //
DELIMITER ;
```

```
select multiplier_par_deux(15);
```

Cette fonction contient du SQL (dans ce cas, elle utilise une simple opération arithmétique), mais elle ne fait ni SELECT, ni INSERT, ni UPDATE. Elle est donc marquée comme **CONTAINS SQL**.

Synthèse des déclarations informatives :

Déclaration	Explication
DETERMINISTIC	Retourne toujours le même résultat pour les mêmes arguments.
NOT DETERMINISTIC	Peut retourner un résultat différent pour les mêmes arguments (utilise des fonctions non déterministes comme RAND(), NOW(), etc.).
READS SQL DATA	Lit des données de la base sans les modifier.
MODIFIES SQL DATA	Modifie les données de la base de données.
CONTAINS SQL	Contient des instructions SQL mais n'interagit pas directement avec les données (ne fait ni SELECT, ni INSERT, ni UPDATE, etc.).

Ces déclarations aident MySQL à optimiser l'exécution des fonctions, à garantir l'intégrité des transactions, et à améliorer la gestion des requêtes.

Série d'exercices / Procédures Stockées

EXERCICE 1

1) Ecrire une procédure **Somme_P (N1 Entier, N2 Entier)** a deux Paramètres qui calcule la somme de de 2 nombres.

2) Ecrire une Fonction **Somme_F (N1 Entier, N2 Entier)** a deux Paramètres qui retourne la somme de de 2 nombres.

3) Ecrire une procédure Factoriel **Fact1 ()** de n=4 sans Paramètre.

4) Ecrire une procédure Factoriel avec un Paramètre **Fact2(N entier)**.

5) Ecrire une Fonction Factoriel **FactF1 ()** de n=4 sans Paramètre.

6) Ecrire une Fonction Factoriel avec un Paramètre **FactF1 (N entier)**.

7) Ecrire une procédure : **Somme (x Entier, n Entier)** a deux Paramètres x et n
S = $\mathbf{x^1/1! + x^2 / 2! + \ldots + x^n/n!}$ pour x et n

8) Ecrire une procédure : **PGDC**(**a Entier, b Entier)** a deux Paramètres
(ex a=18 b=45 pgdc 9)

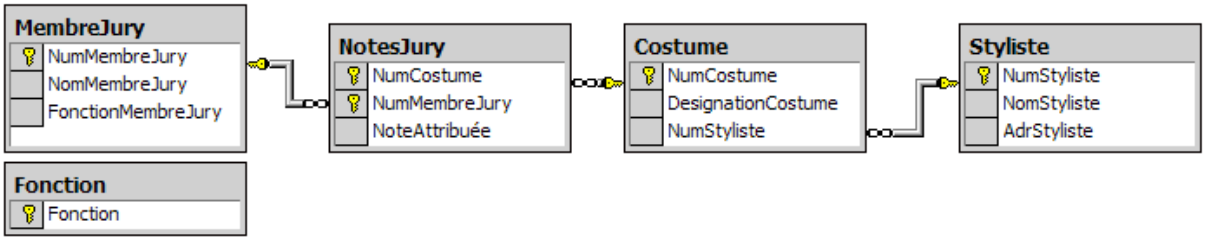
9) Ecrire une procédure **TableMult** (**n Entier**) qui affiche un Tableau de multiplication a un Paramètres.

10) Ecrire une procédure **NB_Premier** (**N Entier**) qui affiche les Nombre premier < N

11) Ecrire une procédure : **PPMC(a Entier, b Entier)** a deux Paramètres PPMC a et b
(ex a=6 b=8 PPMC 24;)

EXERCICE 2

"Inter Défilés" est une société d'organisation de défilés de modes. Une de ces activités les plus réputées : Grand Défilé "Tradition Marocaine". Dans ce défilé, des costumes défilent devant un jury professionnel composé de plusieurs membres. Chaque membre va attribuer une note à chaque costume. La base de données a la structure suivante :



Créer la base de données et les procédures stockées suivantes :

PS 1. Qui affiche la liste des costumes avec pour chaque costume le numéro, la désignation, le nom et l'adresse du styliste qui l'a réalisé

PS 2. Qui reçoit un numéro de costume et qui affiche la désignation, le nom et l'adresse du styliste concerné

PS 3. Qui reçoit un numéro de costume et qui affiche la liste des notes attribuées avec pour chaque note le numéro du membre de jury qui l'a attribué, son nom, sa fonction et la note.

PS 4. Qui **retourne** le nombre total de costumes

PS 5. Qui reçoit un numéro de costume et un numéro de membre de jury et qui **retourne** la note que ce membre a attribuée à ce costume

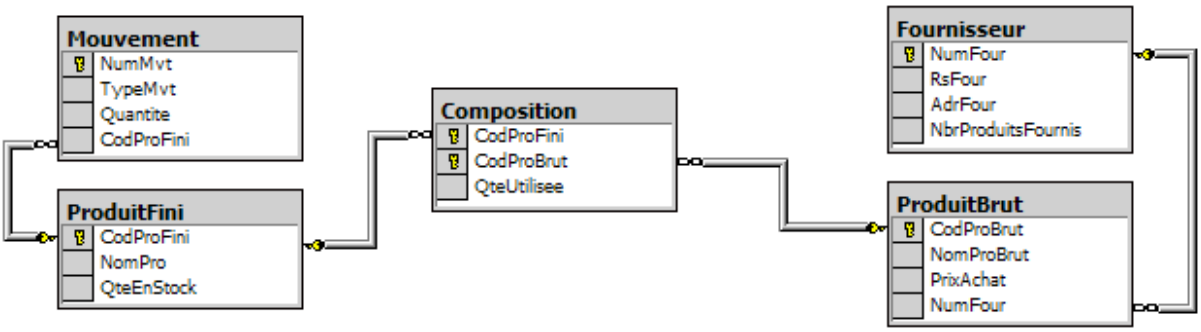
PS 6. Qui reçoit un numéro de costume et qui **retourne** sa moyenne.

EXERCICE 3

Une société achète à ses fournisseurs des produits bruts qu'elle utilise dans la fabrication de produits finis. On souhaite gérer la composition et les mouvements de stock de chaque produit fini.

Les Mouvements de stock sont les opérations d'entrée ou de sortie (type=S ou type=E) de produits finis vers ou depuis le magasin.

La base de données a la structure suivante :



On suppose que les tables 'Mouvement', 'Produit Fini' et 'Fournisseur' sont créées. Créer les procédures suivantes :

PS 1. Qui crée les tables ProduitBrut et Composition
PS 2. Qui affiche le nombre de produits bruts par produit Fini
PS 3. Qui **retourne** en sortie le prix d'achat le plus élevé
PS 4. Qui affiche la liste des produits finis utilisant plus de deux produits bruts
PS 5. Qui reçoit le nom d'un produit brut et **retourne** en sortie la raison sociale de son fournisseur
PS 6. Qui reçoit le code d'un produit fini et qui affiche la liste des mouvements de sortie pour ce produit
PS 7. Qui reçoit le code d'un produit fini et le type de mouvement et qui affiche la liste des mouvements de ce type pour ce produit fini
PS 8. Qui pour chaque produit fini affiche :
- La quantité en stock pour ce produit
- La liste des mouvements concernant ce produit
- La quantité totale en sortie et la quantité totale en entrée
- La différence sera comparée à la quantité en stock. Si elle correspond afficher 'Stock Ok' sinon afficher 'Problème de Stock'

PS 9. Qui reçoit un code produit fini et **retourne** en sortie son prix de reviens