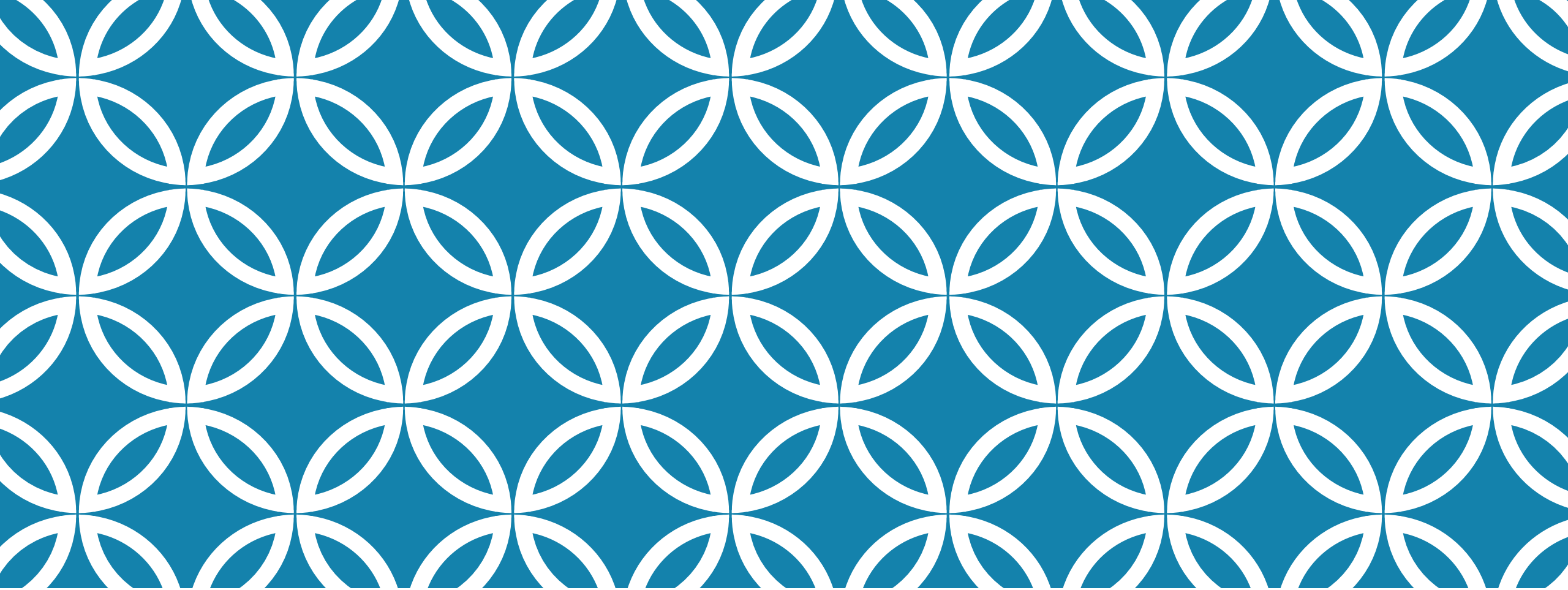


PROGRAMMATION PROCÉDURALE EN SQL

SAGIM 2024/2025

INTRODUCTION

La programmation procédurale en MySQL fait référence à l'utilisation de structures de contrôle et de routines stockées pour exécuter des opérations complexes directement dans la base de données.

INTRODUCTION

MySQL supporte **la programmation procédurale** à travers:

Procédures stockées : Des blocs de code SQL qui peuvent être appelés avec des paramètres.

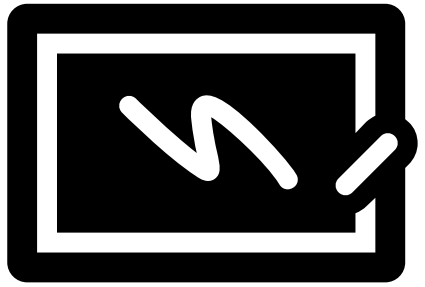
Fonctions stockées : Similaires aux procédures, mais elles retournent une valeur.

Déclencheurs (Triggers) : Des blocs de code qui s'exécutent automatiquement en réponse à certains événements (INSERT, UPDATE, DELETE).

Variables : Utilisées pour stocker des valeurs temporaires.

Structures de contrôle : IF, CASE, LOOP, WHILE, REPEAT, etc.

LES PROCÉDURES STOCKÉES



LES PROCÉDURES STOCKÉES

- Une procédure stockée est un ensemble d'instructions SQL précompilées et stockées directement dans une base de données.
- Elle permet d'encapsuler une série d'opérations complexes, comme des mises à jour, des insertions ou des sélections de données, en une seule unité réutilisable.

POURQUOI UTILISER DES PROCÉDURES STOCKÉES ?

Réutilisabilité: Une fois créée, une procédure peut être appelée plusieurs fois depuis différents endroits de votre application, ce qui évite la redondance de code.

Performance: Les procédures sont compilées une seule fois et stockées dans un format optimisé, ce qui les rend plus rapides à exécuter que des requêtes SQL individuelles.

Sécurité: Les procédures peuvent être accordées à des utilisateurs spécifiques, limitant ainsi les accès aux données et les opérations possibles.

POURQUOI UTILISER DES PROCÉDURES STOCKÉES ?

Modularité: Elles permettent de découper une logique complexe en blocs plus petits et plus faciles à gérer.

Complexité: Les procédures peuvent contenir des structures de contrôle (boucles, conditions) et des appels à d'autres procédures, ce qui permet de réaliser des traitements plus élaborés.

COMMENT FONCTIONNE UNE PROCÉDURE STOCKÉE ?

Création: La procédure est définie avec un langage spécifique à la base de données (par exemple, T-SQL pour SQL Server, PL/SQL pour Oracle).

Compilation: Le système de gestion de base de données (SGBD) compile la procédure et la stocke dans un format optimisé.

Appel: La procédure est appelée par son nom, avec éventuellement des paramètres.

Exécution: Les instructions de la procédure sont exécutées séquentiellement.

Retour: La procédure peut retourner une valeur ou un ensemble de résultats.

EXEMPLE D'UNE PROCÉDURE STOCKÉE

```
CREATE PROCEDURE MaProcedure
    AS
     BEGIN
          SELECT * FROM MaTable;
     END
```

EXEMPLE DE CRÉATION ET D'UTILISATION

Champ (Field)	Type	Null	Key	Default	Extra
emp_id	int	NO	PRI	NULL	
emp_name	varchar(30)	YES		NULL	
emp_firstName	varchar(30)	YES		NULL	
email	varchar(30)	YES		NULL	
tel	int	YES		NULL	

EXEMPLE DE CRÉATION ET D'UTILISATION

- Ajouter un employé

```
DELIMITER $$

CREATE PROCEDURE AddEmployee( IN emp_id INT,  IN emp_name VARCHAR(30),
                              IN emp_firstName VARCHAR(30),
                              IN emailVARCHAR(30),IN tel INT )

    BEGIN
       INSERT INTO emplyees (emp_id, emp_name, emp_firstName, email, tel)
       VALUES (emp_id, emp_name, emp_firstName, email, tel);
    END

$$DELIMITER ;
```

EXEMPLE DE CRÉATION ET D'UTILISATION

- **Ajouter un employé :**

Pour ajouter un nouvel employé avec l'ID 101, voici l'appel :

CALL AddEmployee(101, 'fati', 'el Alaoui', 'Fatima@gmail.com', 06457823);

EXEMPLE DE CRÉATION ET D'UTILISATION

➢ **Modifier un employé:**

```
DELIMITER $$
        CREATE PROCEDURE UpdateEmployee( IN p_emp_id INT, IN new_emp_name VARCHAR(30),
                                         IN new_emp_firstName VARCHAR(30),
                                         IN new_email VARCHAR(30), IN new_tel INT )
   BEGIN
        UPDATE emplyees007
        SET     emp_name = new_emp_name  ,  emp_firstName = new_emp_firstName,
                email = new_email,  tel = new_tel
        WHERE   emp_id =  p_emp_id;
    END $$
DELIMITER ;
```

EXEMPLE DE CRÉATION ET D'UTILISATION

➢ **Modifier un employé :**

Pour mettre à jour les informations de l'employé avec l'ID 30 :

CALL UpdateEmployee(101, 'fatiha', 'el Alaoui', 'FatihaElAlaoui@gmail.com', 06550328);

EXEMPLE DE CRÉATION ET D'UTILISATION

Supprimer un employé:

```
DELIMITER $$
    CREATE PROCEDURE DeleteEmployee(IN p_emp_id INT)
        BEGIN
            DELETE FROM emplyees    WHERE emp_id = p_emp_id;
        END $$
DELIMITER ;
```

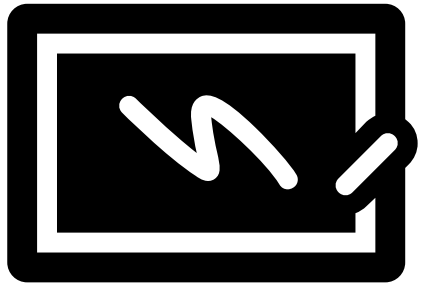
EXEMPLE DE CRÉATION ET D'UTILISATION

➢ **Supprimer un employé :**

Pour supprimer un employé avec l'ID 100 :

```
CALL DeleteEmployee(100);
```

LES **TRIGGERS** (OU DÉCLENCHEURS)



LES TRIGGERS

Les **triggers** (ou déclencheurs) en MySQL sont des objets de base de données qui exécutent automatiquement un bloc de code (souvent une instruction SQL) en réponse à un événement spécifique.

Ces événements sont liés à des opérations sur les tables, comme les commandes **INSERT**, **UPDATE**, ou **DELETE**.

QU'EST-CE QU'UN TRIGGER ?

Un trigger est activé automatiquement lorsqu'une action particulière est effectuée sur une table. Il peut être configuré pour se déclencher :

- **Avant** l'exécution de l'action (**BEFORE**).
- **Après** l'exécution de l'action (**AFTER**).

POURQUOI UTILISER DES TRIGGERS ?

- **Vérifications de contraintes:** Assurer que les données respectent des règles spécifiques (par exemple, que les valeurs d'un champ soient positives, que les dates soient cohérentes).
- **Contrôles d'audit:** Enregistrer les modifications apportées aux données pour des raisons de sécurité ou de traçabilité.
- **Mise à jour automatique de données liées:** Par exemple, mettre à jour un champ de total lorsque des lignes sont ajoutées ou modifiées dans une table de détails.
- **Notifications:** Envoyer des alertes par e-mail ou SMS lors de certains événements.
- **Limiter les modifications:** Empêcher certaines modifications de données en fonction de conditions spécifiques.

COMMENT FONCTIONNENT-ILS ?

- **Définition:** Vous créez **un trigger** en spécifiant :
- **L'événement déclencheur: INSERT**, **UPDATE** ou **DELETE**.
- **La table cible:** La table sur laquelle le trigger sera activé.
- **Le moment d'exécution:** Avant ou après l'événement.
- **Le code à exécuter:** Les instructions **SQL** qui seront exécutées lorsque le trigger est déclenché.
- **Exécution:** Lorsque l'événement se produit sur la table cible, le système de gestion de base de données (**SGBD**) détecte l'événement et exécute automatiquement le code du trigger.

EXEMPLE (SIMPLIFIÉ):

Supposons que vous ayez une table "**Client**" avec un champ "**Solde**".

Vous pouvez créer un **trigger** qui vérifie que le solde ne peut pas devenir **négatif** lors d'une mise à jour :

EXEMPLE (SIMPLIFIÉ):

```
CREATE database VendeurIPhone;
use VendeurIPhone;

        CREATE TABLE client (
                id INT PRIMARY KEY,
                nom VARCHAR(50),
                solde DECIMAL(10, 2)
                        );
```

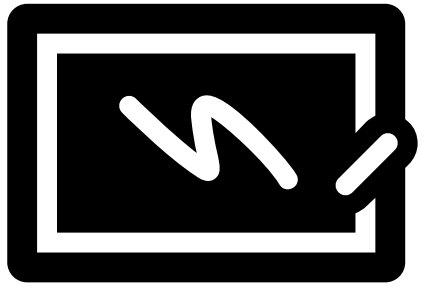
EXEMPLE (SIMPLIFIÉ):

```
INSERT INTO client (id, nom, solde)
                    VALUES (1, 'Alice Dupont', 1000);
INSERT INTO client (id, nom, solde)
                    VALUES (2, 'Bob Martin', 500);
INSERT INTO client (id, nom, solde)
                    VALUES (3, 'Charlie Brown', 200);

UPDATE client SET solde = 1500 WHERE id = 1;
```

EXEMPLE (SIMPLIFIÉ):

```
DELIMITER $$
CREATE TRIGGER CheckSoldeBEFORE UPDATE ON client
    FOR EACH ROW
      BEGIN
         IF NEW.solde < 0 THEN
            SIGNAL SQLSTATE '45000'
            SET MESSAGE_TEXT = 'Le solde ne peut pas être négatif.';
         END IF;
      END
$$ DELIMITER ;

UPDATE client SET solde = -100 WHERE id = 2;
```

EXEMPLE DE SYTHÈSE :



EXEMPLE : CRÉATION DES TABLES

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(30) NOT NULL,
    emp_firstName VARCHAR(30) NOT NULL,
    email VARCHAR(100) NOT NULL,
    tel VARCHAR(15),
    CONSTRAINT unique_email UNIQUE (email)
);
```

EXEMPLE : CRÉATION DES TABLES

```
CREATE TABLE audit_employees (
    audit_id INT AUTO_INCREMENT PRIMARY KEY,
    emp_id INT NOT NULL,
    old_tel VARCHAR(15),
    new_tel VARCHAR(15),
    changed_at DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (emp_id) REFERENCES employees(emp_id)
);
```

TRIGGER 1 : AFTER UPDATE

Création du trigger after_tel_update pour auditer les changements de numéro de téléphone:

```
CREATE TRIGGER after_tel_update
    AFTER UPDATE ON employees
        FOR EACH ROW
            BEGIN
                IF OLD.tel != NEW.tel THEN
                    INSERT INTO audit_employees (emp_id, old_tel, new_tel)
                    VALUES (OLD.emp_id, OLD.tel, NEW.tel);
                END IF;
            END
```

TRIGGER 2 : BEFORE INSERT

Création du trigger before_employee_insert pour vérifier l'unicité de l'email avant insertion :

```
CREATE TRIGGER before_employee_insert
    BEFORE INSERT ON employees
        FOR EACH ROW
        BEGIN
          DECLARE email_count INT;
            SELECT COUNT(*)  INTO email_count
            FROM employees
            WHERE email = NEW.email;
                IF email_count > 0 THEN
                    SIGNAL SQLSTATE '45000'
                    SET MESSAGE_TEXT = 'Email address must be unique.';
                END IF;
        END
```

TESTES :

Insertion d'un employé :

```
INSERT INTO employees (emp_id, emp_name, emp_firstName, email, tel)
VALUES (1, 'Mohammed', 'Radi', 'Radi@gmail.com', '0623567801');
```

Tentative d'insertion d'un employé avec un e-mail déjà existant

```
INSERT INTO employees (emp_id, emp_name, emp_firstName, email, tel)
VALUES (2, 'Fatiha', 'Erradi', 'Radi@gmail.com', '0612121230');
```

TESTES :

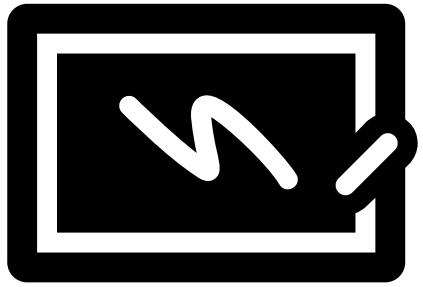
Mise à jour du numéro de téléphone de l'employé:

```
UPDATE employees
SET tel = ‘0625465078'
WHERE emp_id = 1;
```

Affichage des tables pour vérifier les résultats:

```
SELECT * FROM employees;
SELECT * FROM audit_employees;
```

LES FONCTIONS EN MYSQL



SAGIM 2024/2025

QU'EST-CE QU'UNE FONCTION EN MYSQL ?

Une fonction en MySQL est un bloc de code précompilé qui effectue une tâche spécifique sur les données.

Elle prend en entrée des valeurs (appelées arguments) et retourne un résultat.

Les fonctions sont réutilisables, ce qui facilite la création de requêtes complexes et améliore la lisibilité du code SQL.

EXEMPLE :
CALCULER L'ÂGE À PARTIR D'UNE DATE DE NAISSANCE

Exemple : une table personnes avec une colonne date_naissance :

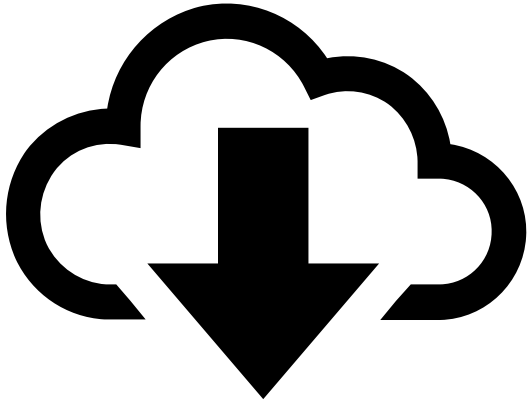
```
CREATE TABLE personnes (
    id INT PRIMARY KEY AUTO_INCREMENT,
    nom VARCHAR(50),
    date_naissance DATE
);
INSERT INTO personnes (nom, date_naissance)
VALUES  ('Ahmed', '1985-08-20'),('Ali', '1995-03-10'),('Fatima', '2000-11-30');
```

EXEMPLE :
CALCULER L'ÂGE À PARTIR D'UNE DATE DE NAISSANCE

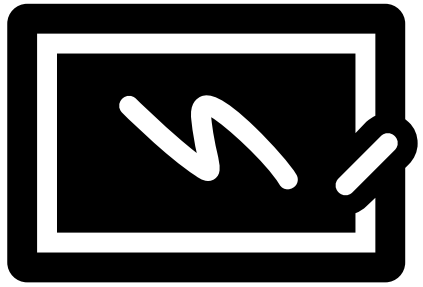
```
DELIMITER $$
CREATE FUNCTION CalculerAge(date_naissance DATE) RETURNS INT DETERMINISTIC
BEGIN
   DECLARE age INT;  SET age = YEAR(CURDATE()) - YEAR(date_naissance);
          IF        MONTH(date_naissance) > MONTH(CURDATE())
                    OR (MONTH(date_naissance) = MONTH(CURDATE())
                    AND DAY(date_naissance) > DAY(CURDATE()))
                        THEN SET age = age - 1;
         END IF;
                     RETURN age;
END$$  DELIMITER ;
```

TESTER LA FONCTION :

```
SELECT nom, date_naissance, CalculerAge(date_naissance) AS age
FROM personnes;
```



LA PROGRAMMATION PROCÉDURALE



FONCTIONS

Définition :

Une fonction en MySQL est un bloc de code réutilisable qui retourne une valeur. Contrairement aux procédures, les fonctions sont conçues pour être utilisées dans des expressions SQL (par exemple, dans une clause SELECT, WHERE, etc.).

Caractéristiques :

Elles doivent obligatoirement **retourner une valeur** avec l'instruction **RETURN**.

Elles peuvent accepter des paramètres d'entrée (**IN**), mais pas de paramètres de sortie (**OUT** ou **INOUT**).

Elles sont souvent utilisées pour encapsuler des calculs ou des transformations de données.

FONCTIONS

```
DELIMITER $$
CREATE FUNCTION calculer_taxe(price DECIMAL(10,2))
        RETURNS DECIMAL(10,2)
DETERMINISTIC
                BEGIN
                        RETURN price * 0.20;
                END $$
DELIMITER ;
-- Utilisation dans une requête SQL
SELECT name, price, calculer_taxe(price) AS tax FROM products;
```

VARIABLES

Définition :

- Une **variable** est un conteneur qui stocke une valeur temporaire pendant l'exécution d'une procédure ou d'une fonction.
- Les variables sont déclarées avec l'instruction **DECLARE** et ont une portée limitée au bloc de code dans lequel elles sont définies.

Caractéristiques :

- Elles doivent être déclarées au début du bloc **BEGIN...END**.
- Elles peuvent stocker des valeurs de différents types (entiers, chaînes de caractères, décimaux, etc.).
- Elles sont utilisées pour manipuler des données temporaires ou pour stocker des résultats intermédiaires.

VARIABLES

```
DELIMITER $$

CREATE PROCEDURE calculate_total(IN a INT, IN b INT)
        BEGIN

                DECLARE total INT;

                        SET total = a + b;

                        SELECT total AS result;

        END $$

DELIMITER ;

CALL calculate_total(5, 10);
```

STRUCTURES DE CONTRÔLE

Les structures de contrôle permettent de gérer le flux d'exécution du code en fonction de conditions ou de boucles.

IF...THEN...ELSE

Permet d'exécuter un bloc de code si une condition est **vraie**, et éventuellement un autre bloc si elle est **fausse**.

IF...THEN...ELSE

```
DELIMITER $$   CREATE PROCEDURE check_age(IN age INT)
  BEGIN
      IF age >= 18 THEN
         SELECT 'Majeur' AS status;   ELSE  SELECT 'Mineur' AS status;
      END IF;
  END $$  DELIMITER ;
-- Appel de la procédure
CALL check_age(20);
CALL check_age(15);
```

LOOP, WHILE, REPEAT

Ces structures permettent de répéter un bloc de code plusieurs fois.

LOOP :

Répète un bloc de code indéfiniment jusqu'à ce qu'une instruction LEAVE soit exécutée.

LOOP

```
DELIMITER $$   CREATE PROCEDURE print_numbers_loop()
BEGIN
    DECLARE counter INT DEFAULT 1;
        my_loop: LOOP
                SELECT counter;  SET counter = counter + 1;
                IF counter > 10 THEN  LEAVE my_loop;   END IF;
        END LOOP;
END $$
DELIMITER ;
CALL print_numbers_loop();
```