# Introduction à JDBC avec MySQL

## Java Database Connectivity

# Plan du cours

1. Introduction à JDBC
2. Installation et configuration de MySQL
3. Connexion à MySQL avec JDBC
4. Première requête SQL
5. Exemples DAO/ORM
6. Gestion des transactions
7. Bonnes pratiques et erreurs courantes

# 1. Introduction à JDBC

- JDBC (Java Database Connectivity) est une API pour interagir avec des bases de données relationnelles en Java.
- Fournit une interface standardisée pour l'accès aux bases de données.
- Permet l'exécution de requêtes SQL via Java.

# 2. Installation et configuration de MySQL

1. Télécharger et installer MySQL depuis mysql.com (choisir *Platform independant*)
2. Installer le **MySQL Connector/J** (driver JDBC pour MySQL)
3. Ajouter le driver JDBC au classpath du projet Java

# 3. Connexion à MySQL avec JDBC

## Création de la connexion

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseConnection {
    public static Connection creeConnexion() {
        String url = "jdbc:mysql://devbdd.iutmetz.univ-lorraine.fr:3306/laroche5_r304";
        String user = "laroche5_appli";
        String password = "********";
        Connection co = null;
        try {
            co = DriverManager.getConnection(url, user, password);
        } catch (SQLException e) {
            System.out.println(e.getMessage());
        }
        return co;
    }
}
```

# 4. Exécution de requêtes SQL

## Création d'une table

```
import java.sql.Connection;
import java.sql.Statement;

public class CreateTable {
    public static void createTable() {
        try (Connection conn = DatabaseConnection.creeConnexion();
             Statement stmt = conn.createStatement()) {
            String sql = "CREATE TABLE etudiant (id INT PRIMARY KEY AUTO_INCREMENT, nom VARCHAR(50), prenom VARCHAR(50), id_promotion INT)";
            stmt.executeUpdate(sql);
            System.out.println("Table créée avec succès !");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

- A noter : *try with resources*

# 5. Exemples DAO/ORM

## ORM (*Object-Relational Mapping*) - Classe `Etudiant`

```
public class Etudiant {
    private int id;
    private String nom;
    private String prenom;
    private int idPromotion;

    public Etudiant(int id, String nom, String prenom, int idPromotion) {
        this.setId(id);
        this.setNom(nom);
        this.setPrenom(prenom);
        this.setIdPromotion(idPromotion);
    }
}
```

# Requête *select* avec `Statement`

```
import java.sql.*;

public class EtudiantDAO {

    public static Etudiant getEtudiantById(int id) {
        String sql = "SELECT * FROM etudiant WHERE id = " + id;
        // try-with-resources
        try (Connection conn = DatabaseConnection.creeConnexion();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql)) {

            if (rs.next()) {
                return new Etudiant(rs.getInt("id"), rs.getString("nom"), rs.getString("prenom"), rs.getInt("id_promotion"));
            }
        } catch (SQLException e) {
        e.printStackTrace();
    }
    return null;
}
```

- Risque d'injection SQL dû à la création de la requête par concaténation de variables

# Requête *select* avec `PreparedStatement`

```java
import java.sql.*;

public class EtudiantDAO {
    public static Etudiant getEtudiantById(int id) {
        String sql = "SELECT * FROM etudiant WHERE id = ?";
        // try-with-resources
        try (Connection conn = DatabaseConnection.creeConnexion();
             PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setInt(1, id);
            ResultSet rs = pstmt.executeQuery();
            if (rs.next()) {
                return new Etudiant(rs.getInt("id"), rs.getString("nom"), rs.getString("prenom"), rs.getInt("id_promotion"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

- Pas d'injection SQL grâce aux `?` remplacés par les variables, sans avoir à gérer les guillemets
- Pré-compilation de la requête : plus rapide

# *Select* multiple avec `PreparedStatement`

```
import java.util.*;

public class EtudiantDAO {
    public static List<Etudiant> getAllEtudiants() {
        List<Etudiant> etudiants = new ArrayList<>();
        String sql = "SELECT * FROM etudiant";
        try (Connection conn = DatabaseConnection.creeConnexion();
             PreparedStatement pstmt = conn.prepareStatement(sql);
             ResultSet rs = pstmt.executeQuery()) {
            while (rs.next()) {
                etudiants.add(new Etudiant(rs.getInt("id"), rs.getString("nom"), rs.getString("prenom"), rs.getInt("id_promotion")));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return etudiants;
    }
}
```

# Insertion d'un étudiant et récupération de l'ID généré

```
public class EtudiantDAO {
    public static boolean insertEtudiant(Etudiant etudiant) {
        boolean ok = false;
        String sql = "INSERT INTO etudiant (nom, prenom, id_promotion) VALUES (?, ?, ?)";
        try (Connection conn = DatabaseConnection.creeConnexion();
             PreparedStatement pstmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS)) {
            pstmt.setString(1, etudiant.getNom());
            pstmt.setString(2, etudiant.getPrenom());
            pstmt.setInt(3, etudiant.getIdPromotion());
            int nbResults = pstmt.executeUpdate();
            ok = (nbResults==1);
            if (ok) {
                ResultSet rs = pstmt.getGeneratedKeys();
                if (rs.next()) {
                    etudiant.setId(rs.getInt(1));
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return ok;
    }
}
```

# 6. Gestion des transactions

- Les transactions permettent d'assurer l'intégrité de la base
- Utilisation des méthodes `commit()` et `rollback()`.

```
public class TransactionExample {
    public static void executeTransaction() {
        try (Connection conn = DatabaseConnection.creeConnexion()) {
            conn.setAutoCommit(false);
            try (Statement stmt = conn.createStatement()) {
                stmt.executeUpdate("INSERT INTO promotion VALUES (1, 'BUT2')");
                stmt.executeUpdate("INSERT INTO etudiant VALUES (53, 'Durand', 'Jacques', 1)");
                conn.commit();
            } catch (SQLException e) {
                conn.rollback();
                e.printStackTrace();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

# 7. Bonnes pratiques et erreurs courantes

✅ Utiliser des `PreparedStatement` pour éviter l'injection SQL.
✅ Fermer les ressources (`Connection`, `Statement`, `ResultSet`).
✅ Gérer les exceptions proprement.
✅ Configurer le pooling de connexions pour améliorer les performances.

# Conclusion

- JDBC est un outil puissant pour interagir avec MySQL en Java.
- Importance des bonnes pratiques pour éviter les failles et améliorer les performances.
- Prolongement possible : Explorer JPA et Hibernate pour une gestion avancée des bases de données.

# Application

Utiliser une base de données sur *devbdd* pour stocker et récupérer un portefeuille de l'exercice 9

Pour intégrer Connector/J à un projet IntellijIDEA :

- Dans votre projet, créer un dossier *lib* et y mettre le fichier `mysql-connector-...jar`
- Menu File, Project Structure
- clic sur *Librairies*, '+', 'Java' et choisir le fichier précédent