Java : gérer une base de données avec **JDBC**

A. Larhlimi

Plan

JDBC

Pour se connecter à une base de données avec **Java**

- Il nous faut un **JDBC** (qui varie selon le **SGBD** utilisé)
- **JDBC** : **J**ava **D**ata**B**ase **C**onnectivity
- **SGBD** : Système de Gestion de Bases de données

JDBC

Pour se connecter à une base de données avec **Java**

- Il nous faut un **JDBC** (qui varie selon le **SGBD** utilisé)
- **JDBC** : **J**ava **D**ata**B**ase **C**onnectivity
- **SGBD** : Système de Gestion de Bases de données

JDBC ?

- **API** (interface d'application) créée par **Sun Microsystems**
- Permettant de communiquer avec les bases de données

JDBC

JDBC

- **API** de **JSE**
- Permettant la connexion et l'exécution de requêtes **SQL** depuis un programme **Java**
- Composé de
  - `Driver`
  - `DriverManager`
  - `Connection`
  - `Statement`
  - `ResultSet`
  - `SQLException`
  - ...

JDBC

JDBC : avantages

- Multi-base de données
- Support pour les requêtes et les procédures stockées
- Fonctionnant en synchrone et asynchrone
- Pas besoin de convertir les données

JDBC : inconvénients

- Pas de driver universel
- Trop verbeux
- Code souvent redondant
- Complexe

JDBC

JDBC

- Aller à `https://dev.mysql.com/downloads/connector/j/?os=26`
- Télécharger et Décompresser l'archive `.zip`

JDBC

Intégrer le driver dans votre projet

- Faire un clic droit sur le nom du projet et aller dans `New > Folder`
- Renommer le répertoire `lib` puis valider
- Copier le `.jar` de l'archive décompressée dans `lib`

Ajouter **JDBC** au path du projet

- Faire clic droit sur `.jar` qu'on a placé dans `lib`
- Aller dans `Build Path` et choisir `Add to Build Path`

Ajouter **JDBC** au path du projet

- Faire clic droit sur `.jar` qu'on a placé dans `lib`
- Aller dans `Build Path` et choisir `Add to Build Path`

Ou aussi

- Faire clic droit sur le projet dans `Package Explorer` et aller dans Properties `Properties`
- Dans `Java Build Path`, aller dans l'onglet `Libraries`
- Cliquer sur `Add JARs`
- Indiquer le chemin du `.jar` qui se trouve dans le répertoire `lib` du projet
- Appliquer

Ajouter **JDBC** au path du projet

- Faire clic droit sur `.jar` qu'on a placé dans `lib`
- Aller dans `Build Path` et choisir `Add to Build Path`

Ou aussi

- Faire clic droit sur le projet dans `Package Explorer` et aller dans Properties `Properties`
- Dans `Java Build Path`, aller dans l'onglet `Libraries`
- Cliquer sur `Add JARs`
- Indiquer le chemin du `.jar` qui se trouve dans le répertoire `lib` du projet
- Appliquer

Vérifier qu'une section `Referenced Libraries` a apparu.

JDBC

Avant de commencer, voici le script SQL qui permet de créer la base de données utilisée dans ce cours

```
CREATE DATABASE cours_jdbc;

USE cours_jdbc;

CREATE TABLE personne (
    num INT PRIMARY KEY AUTO_INCREMENT,
    nom VARCHAR(30),
    prenom VARCHAR(30)
)ENGINE=InnoDB;

SHOW TABLES;

INSERT INTO personne (nom, prenom) VALUES
("Wick", "John"),
("Dalton", "Jack");

SELECT * FROM personne;
```

JDBC

Trois étapes

- Charger le driver **JDBC** (pour **MySQL** dans notre cas)
- Établir la connexion avec la base de données
- Créer et exécuter des requêtes **SQL**

JDBC

Avant de commencer

Tous les imports de ce chapitre sont de `java.sql.*;`

JDBC

Chargement du driver 5

```
try {
    Class.forName("com.mysql.jdbc.Driver");
}
catch (ClassNotFoundException e) {
    System.out.println(e.getMessage());
}
```

Ou

```
try {
    DriverManager.registerDriver(new com.mysql.jdbc.Driver());
}
catch (ClassNotFoundException e) {
    System.out.println(e.getMessage());
}
```

JDBC

Chargement du driver 8

```
try {
    Class.forName("com.mysql.cj.jdbc.Driver");
}
catch (ClassNotFoundException e) {
    System.out.println(e.getMessage());
}
```

Ou

```
try {
    DriverManager.registerDriver(new com.mysql.cj.jdbc.Driver()
       );
}
catch (ClassNotFoundException e) {
    System.out.println(e.getMessage());
}
```

JDBC

Explication

- Pour se connecter à la base de données, il faut spécifier une **URL** de la forme `jdbc:mysql://hote:port/nombd`
  - hote : adresse du serveur **MySQL** (dans notre cas `localhost` ou `127.0.0.1`)
  - port : port **TCP/IP** utilisé par **MySQL** (par défaut est `3306`)
  - nombd : le nom de la base de données **MySQL**
- Il faut aussi le nom d'utilisateur et son mot de passe (qui permettent de se connecter à la base de données **MySQL**)

JDBC

Connexion à la base

```
String url = "jdbc:mysql://localhost:3306/cours_jdbc";
String user = "root";
String password = "";
Connection connexion = null;
try {
    connexion = DriverManager.getConnection(url, user, password);
} catch (SQLException e) {
    e.printStackTrace();
}
finally {
    if (connexion != null)
        try {
            connexion.close();
        } catch (SQLException ignore) {
            ignore.printStackTrace();
        }
}
```

JDBC

Quelques paramètres à rajouter à la chaîne de connexion pour résoudre les problèmes suivants

- Problème d'incompatibilité avec l'heure de Paris : `serverTimezone=UTC`
- Problème **SSL** : `useSSL=false`
- Problème avec la demande de clé : `allowPublicKeyRetrieval=True`
- Problème de base de données inexistante : `createDatabaseIfNotExist=true`
- Problème avec les lettres accentuées : `characterEncoding=UTF-8&useUnicode=yes`

JDBC

Quelques paramètres à rajouter à la chaîne de connexion pour résoudre les problèmes suivants

- Problème d'incompatibilité avec l'heure de Paris : `serverTimezone=UTC`
- Problème **SSL** : `useSSL=false`
- Problème avec la demande de clé : `allowPublicKeyRetrieval=True`
- Problème de base de données inexistante : `createDatabaseIfNotExist=true`
- Problème avec les lettres accentuées : `characterEncoding=UTF-8&useUnicode=yes`

Exemple

```
String url =
"jdbc:mysql://localhost:3306/cours_jdbc?serverTimezone=UTC&useSSL=false
  &allowPublicKeyRetrieval=True";
```

JDBC

Préparation et exécution de la requête

```
// création de la requête (statement)
Statement statement = connexion.createStatement();

// Préparation de la requête
String selectRequest = "SELECT * FROM Personne;";

// Exécution de la requête
ResultSet result = statement.executeQuery(selectRequest);
```

JDBC

Préparation et exécution de la requête

```
// création de la requête (statement)
Statement statement = connexion.createStatement();

// Préparation de la requête
String selectRequest = "SELECT * FROM Personne;";

// Exécution de la requête
ResultSet result = statement.executeQuery(selectRequest);
```

On utilise

- `execute()` pour les requêtes de création : `CREATE`.
- `executeQuery()` pour les requêtes de lecture : `SELECT`.
- `executeUpdate()` pour les requêtes d'écriture : `INSERT`, `UPDATE` et `DELETE`.

JDBC

Pour récupérer les données, on peut indiquer le nom de la colonne

```
while (result.next()) {
    int num = result.getInt("num");
    String nom = result.getString("nom");
    String prenom = result.getString("prenom");
    System.out.println(num + " " + nom + " " + prenom);
}
```

JDBC

Ou aussi son indice dans la table

```
while (result.next()) {
    int num = result.getInt(1);
    String nom = result.getString(2);
    String prenom = result.getString(3);
    System.out.println(num + " " + nom + " " + prenom);
}
```

JDBC

Pour faire une insertion

```
Statement statement = connexion.createStatement();
String insertRequest = "INSERT INTO Personne (nom,prenom)  VALUES ('Wick','John');";
int nbr = statement.executeUpdate(insertRequest);
if (nbr != 0) {
    System.out.println("insertion réussie");
}
```

JDBC

Pour faire une insertion

```
Statement statement = connexion.createStatement();
String insertRequest = "INSERT INTO Personne (nom,prenom)  VALUES ('Wick','John');";
int nbr = statement.executeUpdate(insertRequest);
if (nbr != 0) {
    System.out.println("insertion réussie");
}
```

La méthode `executeUpdate()` retourne

- 0 en cas d'échec de la requête d'insertion, et 1 en cas de succès
- le nombre de lignes respectivement mises à jour ou supprimées

JDBC

Pour récupérer la valeur de la clé primaire auto-générée

```
Statement statement = connexion.createStatement();
String insertRequest = "INSERT INTO Personne (nom, prenom) VALUES ('Wick','John');";

// on demande le renvoi des valeurs attribuées à la clé primaire
statement.executeUpdate(insertRequest, Statement.RETURN_GENERATED_KEYS);

// on parcourt les valeurs attribuées à l'ensemble de tuples ajoutés
ResultSet resultat = statement.getGeneratedKeys();

// on vérifie s'il contient au moins une valeur
if (resultat.next()) {
    System.out.println("Identifiant généré pour la personne : " +  resultat.getInt(1));
}
```

JDBC

Pour éviter les injections SQL, il faut utiliser les requêtes préparées

```
String request = "INSERT INTO Personne (nom, prenom)  VALUES (?, ?);";
PreparedStatement ps =  connexion.prepareStatement(request, PreparedStatement.
   RETURN_GENERATED_KEYS);
ps.setString(1, "Wick");
ps.setString(2, "John");
ps.executeUpdate();
ResultSet resultat = ps.getGeneratedKeys();
if (resultat.next()) {
    System.out.println("Identifiant généré pour la personne : " +  resultat.getInt(1));
}
```

JDBC

Pour éviter les injections SQL, il faut utiliser les requêtes préparées

```
String request = "INSERT INTO Personne (nom, prenom)  VALUES (?, ?);";
PreparedStatement ps =  connexion.prepareStatement(request, PreparedStatement.
   RETURN_GENERATED_KEYS);
ps.setString(1, "Wick");
ps.setString(2, "John");
ps.executeUpdate();
ResultSet resultat = ps.getGeneratedKeys();
if (resultat.next()) {
    System.out.println("Identifiant généré pour la personne : " +  resultat.getInt(1));
}
```

Attention à l'ordre des attributs

JDBC

Transactions

- Ensemble de requête **SQL**
- Appliquant le principe soit tout (toutes les requête **SQL**) soit rien
- Activées par défaut avec **MySQL**
- Pouvant être désactivées et gérées par le développeur

JDBC

Pour désactiver l'auto-commit

```
connection.setAutoCommit(false);
```

JDBC

Pour désactiver l'auto-commit

```
connection.setAutoCommit(false);
```

Pour valider une transaction

```
connection.commit();
```

JDBC

Pour désactiver l'auto-commit

```
connection.setAutoCommit(false);
```

Pour valider une transaction

```
connection.commit();
```

Pour annuler une transaction

```
connection.rollback();
```

JDBC

Exemple avec les transactions

```
// désactiver l'auto-commit
connexion.setAutoCommit(false);

String request = "INSERT INTO Personne (nom,prenom)  VALUES (?,?);";
PreparedStatement ps =  connexion.prepareStatement(request, PreparedStatement.
   RETURN_GENERATED_KEYS);
ps.setString(1, "Wick");
ps.setString(2, "John");
ps.executeUpdate();

// valider l'insertion
connexion.commit();

ResultSet resultat = ps.getGeneratedKeys();
if (resultat.next()) {
   System.out.println("Identifiant généré pour la personne : " +  resultat.getInt(1));
}
```

JDBC

Organisation du code

- Il faut mettre toutes les données (url, nomUtilisateur, motDePasse...) relatives à notre connexion dans une classe connexion
- Pour chaque table de la base de données, on crée une classe java ayant comme attributs les colonnes de cette table
- Il faut mettre tout le code correspondant à l'accès aux données (de la base de données) dans des nouvelles classes et interfaces (qui constitueront la couche **DAO** : Data Access Object)

JDBC

La classe `MySqlConnection`

```
package org.eclipse.config;

import java.sql.Connection;
import java.sql.DriverManager;

public class MySqlConnection {

    private static Connection connexion = null;

    static {
        try {
            String url = "jdbc:mysql://localhost:3306/cours_jdbc";
            String utilisateur = "root";
            String motDePasse = "";

            Class.forName("com.mysql.cj.jdbc.Driver");
            connexion = DriverManager.getConnection(url, utilisateur, motDePasse);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private MySqlConnection() { }

    public static Connection getConnection() {
        return connexion;
    }
}
```

JDBC

La classe `Personne`

```
package org.eclipse.model;

public class Personne {

    private int num;
    private String nom;
    private String prenom;

    // + getters + setters + constructeur sans param
       ètre + constructeur avec 2 paramètres nom et
       prénom  + constructeur avec  3 paramètres

}
```

JDBC

L'interface `PersonneDao`

```
package org.eclipse.dao;

import java.util.List;

import org.eclipse.model.Personne;

public interface PersonneDao {
        Personne save(Personne personne);
        boolean remove(Personne personne);
        Personne update(Personne personne);
        Personne findById(int id);
        List<Personne> getAll();
}
```

JDBC

Déclarons une classe `PersonneDaoImpl` **dans** `org.eclipse.dao`

```
public class PersonneDaoImpl implements PersonneDao {

}
```

JDBC

Implémentons la méthode `save`

```
@Override
public Personne save(Personne personne) {
        Connection c = MySqlConnection.getConnection();
        try {
                PreparedStatement ps = c.prepareStatement("INSERT INTO personne (nom,
                   prenom) VALUES (?,?); ", PreparedStatement.RETURN_GENERATED_KEYS);
                ps.setString(1, personne.getNom());
                ps.setString(2, personne.getPrenom());
                ps.executeUpdate();
                ResultSet resultat = ps.getGeneratedKeys();
                if (resultat.next()) {
                        personne.setNum(resultat.getInt(1));
                        return personne;
                }
        } catch (SQLException e) {
                e.printStackTrace();
        }
        finally {
                try {
                        c.close();
                } catch (SQLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
        return null;
}
```

Exemple de la méthode `save` en utilisant les transactions

```
@Override
public Personne save(Personne personne) {
        Connection c = MySqlConnection.getConnection();
        try {
                c.setAutoCommit(false);
                PreparedStatement ps = c.prepareStatement("INSERT INTO personne (nom,
                  prenom) VALUES (?,?); ", PreparedStatement.RETURN_GENERATED_KEYS);
                ps.setString(1, personne.getNom());
                ps.setString(2, personne.getPrenom());
                ps.executeUpdate();
                ResultSet resultat = ps.getGeneratedKeys();
                if (resultat.next()) {
                        c.commit();
                        personne.setNum(resultat.getInt(1));
                        return personne;
                }
        } catch (SQLException e) {
                e.printStackTrace();
        }
        finally {
                try {
                        c.close();
                } catch (SQLException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
        return null;
}
```

JDBC

La méthode `findById`

```
@Override
public Personne findById(int id) {
    Personne personne = null;
    Connection c = MySqlConnection.getConnection();
    if (c != null) {
        try {
            String request = "SELECT * FROM personne WHERE num = ?;";
            PreparedStatement ps = c.prepareStatement(request);
            ps.setInt(1, id);
            ResultSet r = ps.executeQuery();
            if (r.next())
                personne = new Personne(r.getInt("num"), r.getString("nom"), r.
                   getString("prenom"));
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            try {
                c.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
    return personne;
}
```

Le `Main` pour tester toutes ces classes

```
package org.eclipse.classes;

import org.eclipse.dao.PersonneDaoImpl;
import org.eclipse.model.Personne;

public class Main {

  public static void main(String args []) {

    PersonneDaoImpl personneDaoImpl = new PersonneDaoImpl();
    Personne personne = new Personne ("Wick", "John");
    Personne insertedPersonne = personneDaoImpl.save(personne);

    if (insertedPersonne != null) {
      System.out.println("personne numéro " + insertedPersonne.getNum()
          + " a été insérée");
    } else {
      System.out.println("problème d'insertion");
    }
  }
}
```

JDBC

Remarque

N'oublions pas d'implémenter les trois autres méthodes de l'interface `PersonneDao`.

JDBC

Utilisation de la généricité avec les **DAO**

- Nous devons créer autant d'interfaces **DAO** que tables de la bases de données
- Pour éviter cela, on peut utiliser une seule interface `GenericDao` avec un type générique que toutes les classes d'accès aux données doivent l'implémenter.

JDBC

L'interface générique `GenericDao`

```
package org.eclipse.dao;

import java.util.List;

public interface GenericDao<Entity, PK> {

    List<Entity> findAll();

    Entity findById(PK id);

    Entity save(Entity entity);

    Entity update(Entity entity);

    boolean remove(PK id);

}
```

JDBC

La classe `PersonneDaoImpl`

```
package org.eclipse.dao;

public class PersonneDaoImpl implements GenericDao<Personne, Integer> {

    ...

}
```

JDBC

La classe `PersonneDaoImpl`

```
package org.eclipse.dao;

public class PersonneDaoImpl implements GenericDao<Personne, Integer> {

    ...

}
```

Le reste du code est le même.

JDBC

Encore de la restructuration du code

- Mettre les données (url, nomUtilisateur, motDePasse...) relatives à notre connexion dans un fichier de propriétés que nous appelons `db.properties` (utilisé par certain framework comme Spring)
- Créer une nouvelle classe (`DataSourceFactory`) qui va lire et construire les différentes propriétés de la connexion
- Utiliser `DataSourceFactory` dans `MySqlConnection`

JDBC

Le fichier `db.properties` situé à la racine du projet (ayant la forme `clé = valeur`, le nom de la clé est à choisir par l'utilisateur)

```
url=jdbc:mysql://localhost:3306/cours_jdbc?serverTimezone=UTC
username=root
password=root
```

JDBC

Créons la classe `MyDataSourceFactory` **dans** `org.eclipse.config`

```
package org.eclipse.config;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

import javax.sql.DataSource;

import com.mysql.cj.jdbc.MysqlDataSource;

public class MyDataSourceFactory {

    public static DataSource getMySQLDataSource() {
        Properties props = new Properties();
        FileInputStream fis = null;
        MysqlDataSource mysqlDataSource = null;
        try {
            fis = new FileInputStream("db.properties");
            props.load(fis);
            mysqlDataSource = new MysqlDataSource();
            mysqlDataSource.setURL(props.getProperty("url"));
            mysqlDataSource.setUser(props.getProperty("username"));
            mysqlDataSource.setPassword(props.getProperty("password"));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return mysqlDataSource;
    }
}
```

JDBC

Remarque

Dans `MyDataSourceFactory`, on ne précise pas le driver `com.mysql.jdbc.Driver` car on utilise un objet de la classe `MysqlDataSource` qui charge lui même le driver.

La classe `MySqlConnection` du package `org.eclipse.config`

```
import java.sql.Connection;
import java.sql.SQLException;
import javax.sql.DataSource;

public class MySqlConnection {

  private static Connection connexion = null;

  private MySqlConnection() {

    DataSource dataSource = MyDataSourceFactory.getMySQLDataSource();
    try {
      connexion = dataSource.getConnection();
    } catch (SQLException e) {
      e.printStackTrace();
    }
  }
  public static Connection getConnection() {
    if (connexion == null) {
      new MySqlConnection();
    }
    return connexion;
  }
```

Relançons le `Main` et vérifier que tout fonctionne correctement

```
package org.eclipse.test;

import org.eclipse.dao.PersonneDaoImpl;
import org.eclipse.model.Personne;

public class Main {

  public static void main(String args []) {

    PersonneDaoImpl personneDaoImpl = new PersonneDaoImpl();
    Personne personne = new Personne ("Wick", "John");
    Personne insertedPersonne = personneDaoImpl.save(personne);

    if (insertedPersonne != null)
      System.out.println("personne numéro " + insertedPersonne.
        getNum() + " a été insérée");
    else
      System.out.println("problème d'insertion");
  }
}
```

JDBC

Rappel

Notre approche est idéale pour une application de petite taille ou mono-utilisateur.

JDBC

Problématique

- Chaque méthode d'une classe **DAO** ouvre la connexion, exécute une requête puis ferme la connexion.
- La connexion à une base de données
  - a un coût non négligeable (l'opération la plus coûteuse dans une application **Web**),
  - ne peut être partagée par des threads.

JDBC

Problématique

- Chaque méthode d'une classe **DAO** ouvre la connexion, exécute une requête puis ferme la connexion.
- La connexion à une base de données
  - a un coût non négligeable (l'opération la plus coûteuse dans une application **Web**),
  - ne peut être partagée par des threads.

Quelle solution alors ?

JDBC

Solution : utiliser un pool de connexions déjà ouvertes (**connection pooling**)

- Le nombre de connexions ouvertes est paramétrable : au démarrage de l'application, un nombre de connexions sera créé en fonction d'un nombre donné.
- Les connexions resteront toujours ouvertes.
- Le pool de connexions se charge de retourner un objet `Connection` aux méthodes de l'application qui la demandent.
- Le client qui appelle la méthode `connection.close` perd la connexion sans la fermer réellement. La connexion sera libérée et pourra être redistribuée de nouveau.

JDBC

Techniquement, comment faire ?

- Utiliser `DataSource` à la place de `DriverManager`.
- Utiliser une implémentation **Java** pour le **Connection pool**.

JDBC

Techniquement, comment faire ?

- Utiliser `DataSource` à la place de `DriverManager`.
- Utiliser une implémentation **Java** pour le **Connection pool**.

Exemples d'implémentation de **Connection pool** pour **Java**

- **HikariCP**
- BoneCP
- DBPool
- Apache DBCP
- c3p0
- ...

JDBC

HikariCP, pourquoi ?

- Plus performant
- Plus utilisé
- Écrit en **Java**
- ...

JDBC

HikariCP, pourquoi ?

- Plus performant
- Plus utilisé
- Écrit en **Java**
- ...

Dépôt **GitHub**

`https://github.com/brettwooldridge/HikariCP`

JDBC

Intégrer **HikariCP** dans le projet

- Aller à `https://jar-download.com/artifacts/com.zaxxer/HikariCP/5.0.1`
- Télécharger et Décompresser l'archive
- Déplacer les deux fichiers `.jar` (**HikariCP** et **slf4j**) dans le dossier `lib` du projet
- Ajouter les deux fichiers au `build path` du projet

JDBC

Remplaçons la clé `url` de `db.properties`

```
url=jdbc:mysql://localhost:3306/cours_jdbc?serverTimezone=UTC
username=root
password=root
```

Par `jdbcUrl`

```
jdbcUrl=jdbc:mysql://localhost:3306/cours_jdbc?serverTimezone=UTC
username=root
password=root
```

JDBC

Créons une classe `DataSource` avec un constructeur privé

```
package org.eclipse.config;

public class DataSource {

        private DataSource() {

        }

}
```

JDBC

Déclarons les deux attributs suivants

```
package org.eclipse.config;

import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

public class DataSource {

  private static HikariDataSource ds;
  private static HikariConfig conf = new HikariConfig("db.properties");

  private DataSource() {

  }

}
```

JDBC

Définissons une méthode `getConnection()` qui retournera un objet `Connection`

```
package org.eclipse.config;

import java.sql.Connection;
import java.sql.SQLException;

import com.zaxxer.hikari.HikariConfig;
import com.zaxxer.hikari.HikariDataSource;

public class DataSource {

  private static HikariDataSource ds;
  private static HikariConfig conf = new HikariConfig("db.properties");

  private DataSource() { }

  public static Connection getConnection() throws SQLException {
    ds = new HikariDataSource(conf);
    return ds.getConnection();
  }
}
```

Relançons le `Main` et vérifier que tout fonctionne correctement

```
package org.eclipse.test;

import org.eclipse.dao.PersonneDaoImpl;
import org.eclipse.model.Personne;

public class Main {

  public static void main(String args []) {

    PersonneDaoImpl personneDaoImpl = new PersonneDaoImpl();
    Personne personne = new Personne ("Wick", "John");
    Personne insertedPersonne = personneDaoImpl.save(personne);

    if (insertedPersonne != null)
      System.out.println("personne numéro " + insertedPersonne.
        getNum() + " a été insérée");
    else
      System.out.println("problème d'insertion");
  }
}
```

JDBC

Pour fixer le nombre de connexion de la pool, on ajoute la clé `maximumPoolSize` avec la valeur souhaitée

```
jdbcUrl=jdbc:mysql://localhost:3306/cours_jdbc?serverTimezone=UTC
username=root
password=root
maximumPoolSize=10
```

JDBC

Considérons la classe `Adresse` suivante

```
package com.example.demo.model;

public class Adresse {
  private Integer id;
  private String rue;
  private String codePostal;
  private String ville;

  public Adresse() {
  }

  public Adresse(Integer id, String rue, String codePostal, String
     ville) {
    this.id = id;
    this.rue = rue;
    this.codePostal = codePostal;
    this.ville = ville;
  }

  // + getters / setters / toString
```

JDBC

Dans `Personne`, définissons un nouvel attribut `adresses`

```
public class Personne {

  private Integer num;
  private String nom;
  private String prenom;

  private List<Adresse> adresses;

  // + getter / setter / toString

}
```

Exécutons le script suivant pour mettre à jour la base de données avec les nouvelles tables

```
DROP DATABASE cours_jdbc;
CREATE DATABASE cours_jdbc;
USE cours_jdbc;

CREATE TABLE personne(
num INT PRIMARY KEY AUTO_INCREMENT,
nom VARCHAR(30),
prenom VARCHAR(30)
)ENGINE=InnoDB;

INSERT INTO personne (nom, prenom) VALUES ("Wick", "John"), ("Dalton", "Jack");

CREATE TABLE adresse(
id INT PRIMARY KEY AUTO_INCREMENT,
rue VARCHAR(30),
code_postal VARCHAR(30),
ville VARCHAR(30)
)ENGINE=InnoDB;

INSERT INTO adresse (rue, code_postal, ville) VALUES
("paradis", "13006", "Marseille"),
("plantes", "75014", "Paris");

CREATE TABLE personne_adresse(
id INT PRIMARY KEY AUTO_INCREMENT,
num_personne INT,
id_adresse INT,
FOREIGN  KEY (num_personne) REFERENCES personne (num),
FOREIGN  KEY (id_adresse) REFERENCES adresse (id)
)ENGINE=InnoDB;

INSERT INTO personne_adresse (num_personne, id_adresse) VALUES (1, 1), (1, 2), (2, 2);
```

JDBC

Exercice 1

- Créez une classe `AdresseDao` qui implémente `Dao`
- Implémentez les méthodes de `Dao`
- Dans `main`, testez toutes les méthodes implémentées dans `AdresseDao`.

JDBC

Dans `AdresseDao`, implémenter les méthodes suivantes

```
public List<Adresse> findAdressesByPersonneId(int id) {
}

public Adresse findAdresseById(int idPers, int idAdr) {
}
```

JDBC

Implémenter les méthodes suivantes dans une classe DAO

```
public int mapPersonneAdresse(Integer idPers, Integer idAdr) {
}

public int unmapPersonneAdresse(Integer idPers, Integer idAdr) {
}
```