

**Mohammed OUANAN**

[m.ouanan@umi.ac.ma](mailto:m.ouanan@umi.ac.ma)

**JEE (Java distribute)**

# Comment Java est-il un langage distribué ?

Java est considéré comme un langage distribué pour plusieurs raisons qui facilitent le développement d'applications capables de fonctionner sur plusieurs machines ou à travers des réseaux.

Voici les principales caractéristiques qui en font un langage distribué :

**Architecture Client-Serveur** : Java est souvent utilisé pour développer des applications qui suivent l'architecture client-serveur. Cela permet aux clients (par exemple, des applications de bureau ou web) de se connecter à un serveur pour demander des ressources ou des services.

**RMI (Remote Method Invocation)** : Java RMI permet d'invoquer des méthodes sur des objets situés sur d'autres machines comme si elles étaient locales. Cela simplifie la communication entre les applications distribuées.

**Sérialisation** : Java permet la sérialisation d'objets, c'est-à-dire la conversion d'un objet en un format qui peut être facilement transmis sur un réseau (comme des flux de données). Cela est essentiel pour la transmission d'objets entre différentes machines.

**API de Réseau** : Java fournit des bibliothèques robustes pour la création de communications réseau, telles que java.net, qui permettent aux applications de se connecter via des **sockets** TCP/IP, HTTP, etc.

**Plateforme Indépendante** : Grâce à la JVM (Java Virtual Machine), Java est un langage indépendant de la plateforme, ce qui signifie qu'un programme Java peut fonctionner sur n'importe quel système d'exploitation qui a une JVM. Cela facilite le déploiement d'applications sur différents serveurs.

**Services Web** : Java prend en charge le développement de services web, ce qui permet aux applications de communiquer entre elles via des protocoles standard comme HTTP, XML et JSON.

**Java EE (Enterprise Edition)** : Java EE fournit des fonctionnalités robustes pour le développement d'applications d'entreprise distribuées, y compris les EJB (Enterprise JavaBeans), les servlets et les JSP (JavaServer Pages), qui facilitent la création d'applications web complexes et distribuées.

**Microservices** : Avec l'avènement des architectures microservices, Java est utilisé pour développer des services indépendants qui peuvent être déployés et scalés indépendamment, favorisant ainsi une approche distribuée.

# Exemples d'applications distribuées

**Systèmes bancaires** : Les transactions bancaires sont distribuées sur des serveurs dans différentes régions pour garantir la rapidité et la disponibilité.

**Réseaux sociaux** : La distribution des données et des processus permet de gérer des millions de connexions simultanées.

**Applications de commerce en ligne** : Elles utilisent plusieurs serveurs pour gérer les stocks, les paiements et les commandes en temps réel.

# Programme du module JEE

- 1 [Chapitre1: JDBC](#) Java DataBase Connectivity
- 2 [Chapitre 2: Java et concurrence: le multithreading](#)
- 3 [Chapitre3:](#) Applications distribuées en Java - Java/RMI et IDL/CORBA
- 4 [Chapitre 4: Programmation Reseau les sockets](#)

# Java : gérer une base de données avec JDBC

**Mohammed OUANAN**

[m.ouanan@umi.ac.ma](mailto:m.ouanan@umi.ac.ma)

# Plan

- 1 [Introduction](#)
- 2 [Mise en place](#)
- 3 [Utilisation](#)
- 4 [Transactions](#)
- 5 [Restructuration du code](#)
  - [Classes connexion et DAO](#)
  - [DataSource et fichier de propriétés](#)
- 6 [Pool de connexions avec HikariCP](#)
- 7 [Cas d'une relation](#)

# Utilisation de JDBC dans des applications distribuées

Bien que JDBC lui-même ne soit pas un service d'application réparti, il peut être intégré dans une **architecture distribuée**. Par exemple :

**Serveurs d'applications** : Un serveur d'applications (comme un serveur JEE) utilise JDBC pour gérer l'accès aux bases de données, mais il offre aussi des services distribués (comme EJB ou les Web Services).

**Microservices** : Dans une architecture de microservices, chaque service peut utiliser JDBC pour accéder à sa propre base de données. Les services eux-mêmes sont répartis et communiquent entre eux via des API REST ou des messages (JMS, Kafka, etc.).



# 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

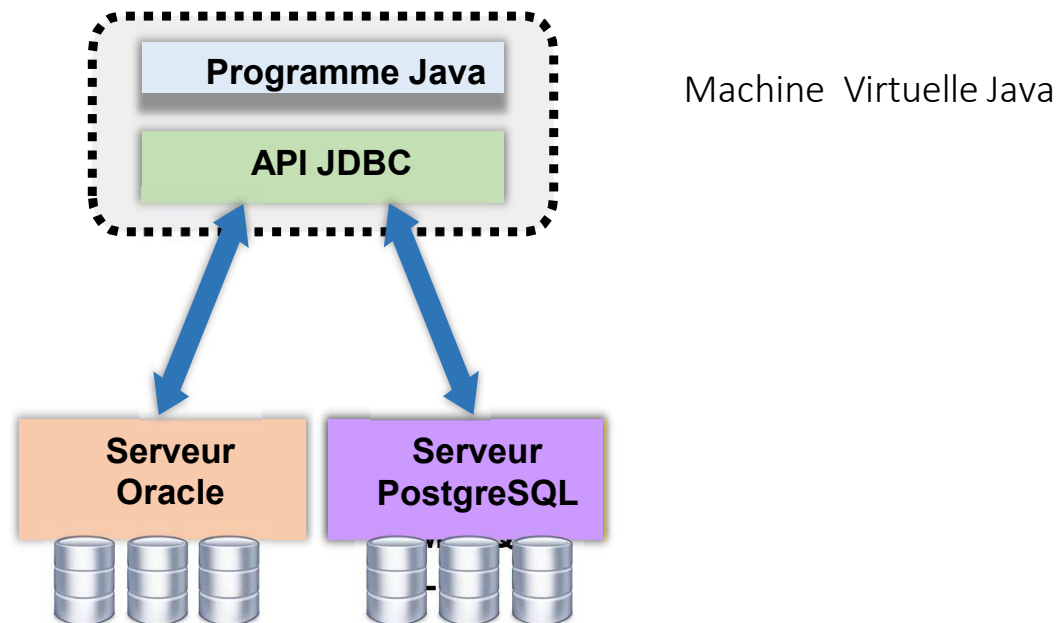
- Il nous faut un **JDBC** (qui varie selon le **SGBD** utilisé)
- **JDBC** : Java **DataBase** **Connectivity**
- **SGBD** : Système de Gestion de Bases de données

## JDBC ?

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

- JDBC Java Data Base Connectivity

- API java standard (fait partie de Java SE) qui permet un accès homogène à des bases de données depuis un programme Java au travers du langage SQL.



- L'API JDBC est indépendante des SGBD.
  - Un changement de SGBD ne doit pas impacter le code applicatif.

# JDBC

## JDBC

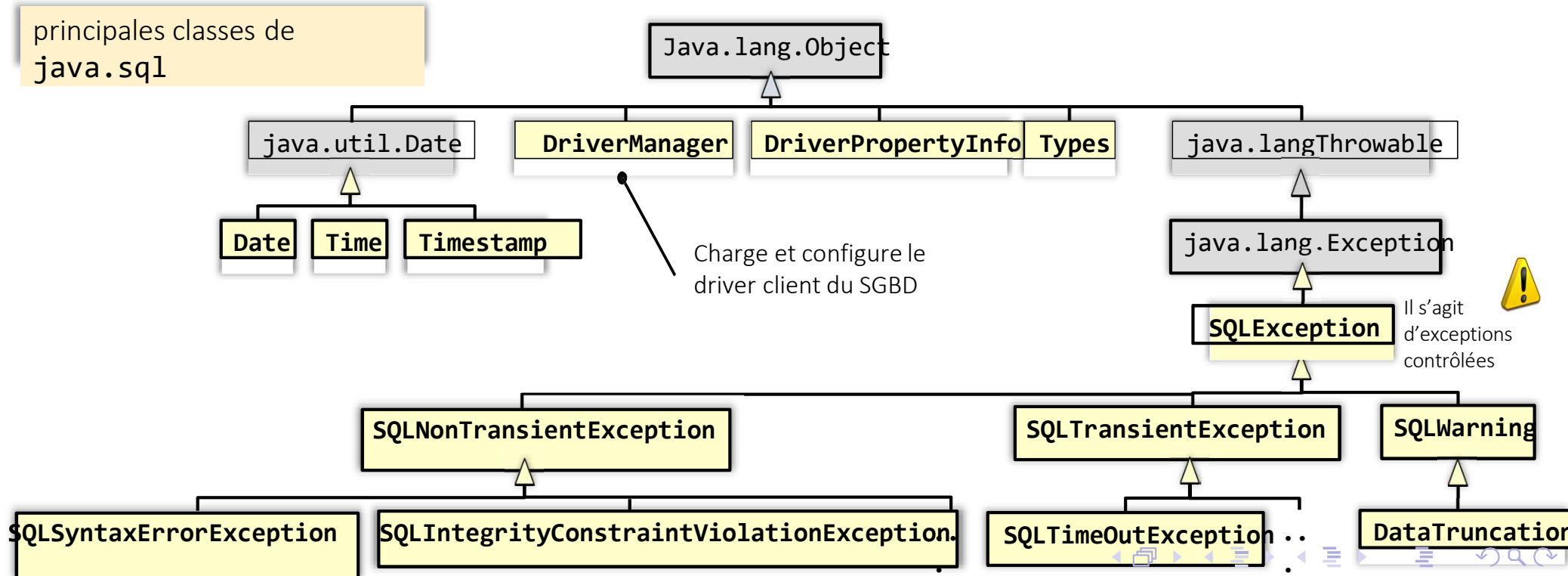
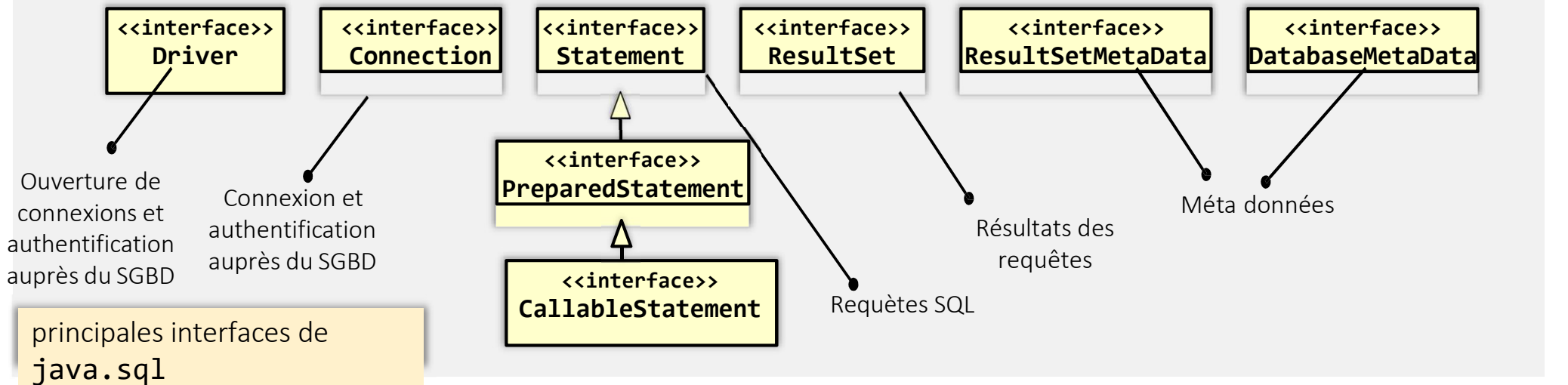
- **API de JSE**

- Permettant la connexion et l'exécution de requêtes **SQL** depuis un programme **Java**

- Composé de

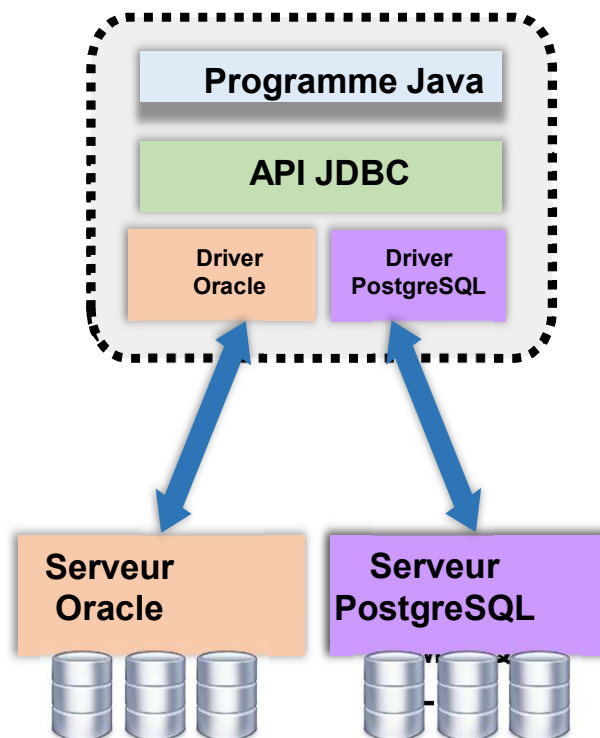
- Driver
  - DriverManager
  - Connection
  - Statement
  - ResultSet
  - SQLException
  - ...

# Classes et Interfaces JDBC

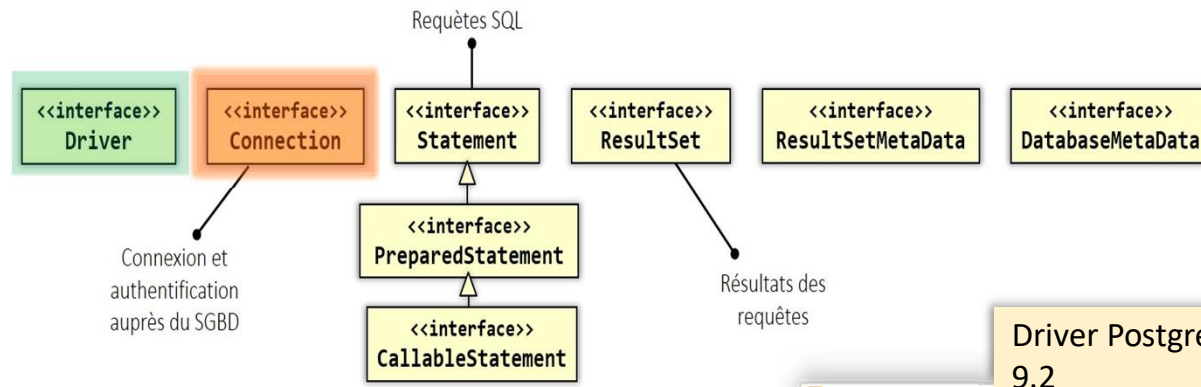


# JDBC

- Le code applicatif est basé sur les interfaces du JDBC
- Pour accéder à un SGBD il est nécessaire de disposer de classes implémentant ces interfaces.
  - Elles dépendent du SGBD adressé.
  - L'ensemble de ces classes pour un SGBD donné est appelé pilote (driver) JDBC



- Il existe des *pilotes* pour tous les SGBD importants du marché (Oracle, MySQL, PostgreSQL, DB2, SQLite...)
- JDBC spécifie uniquement l'API que ces *pilotes* doivent respecter. Ils sont réalisés par une tierce partie (fournisseur du SGBD, «éditeur de logiciel...)
- l'implémentation des drivers est totalement libre

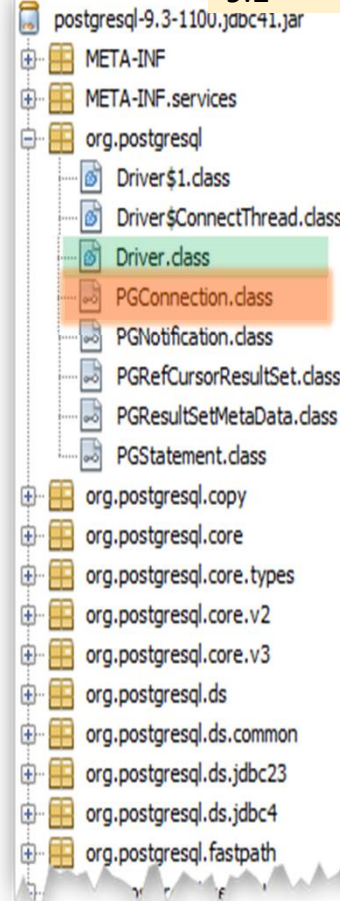


Les interfaces définissent une abstraction du pilote (driver) de la base de données. Chaque fournisseur propose sa propre implémentation de ces interfaces.

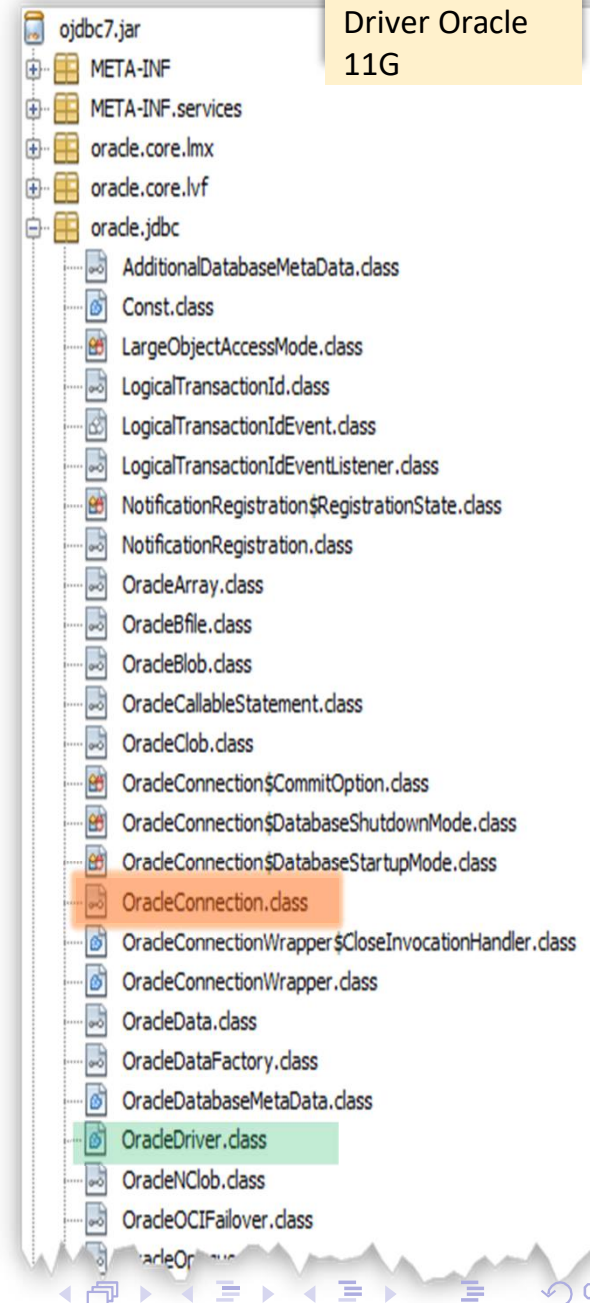
Les classes d'implémentation du driver jdbc sont dans une archive (fichier **.jar**) qu'il faut intégrer (sans la décompresser) au niveau du **classpath** de l'application au moment de l'exécution.

Au niveau du programme d'application on ne travaille qu'avec les abstractions (interfaces) sans se soucier des classes effectives d'implémentation.

Driver Postgres  
9.2



Driver Oracle  
11G



# Fonctionnement général de JDBC

## 1) Etablir une connexion à la base de données

**DriverManager** permet de créer des objets **Connection**  
(objets instance d'une classe qui implémente l'interface **Connection**)

```
Connection c = DriverManager.getConnection(...);
```

## 2) Créer un objet de transport qui permettra d'envoyer des requêtes SQL à la BD

l'objet **Connection** permet de créer des objets **Statement**  
(objets instance d'une classe qui implémente l'interface **Statement**)

```
Statement stmt = c.createStatement(...);
```

## 3) Exécuter la requête sur la BD et récupérer ses résultats

l'objet **Statement** permet d'envoyer requêtes SQL et de créer des objets

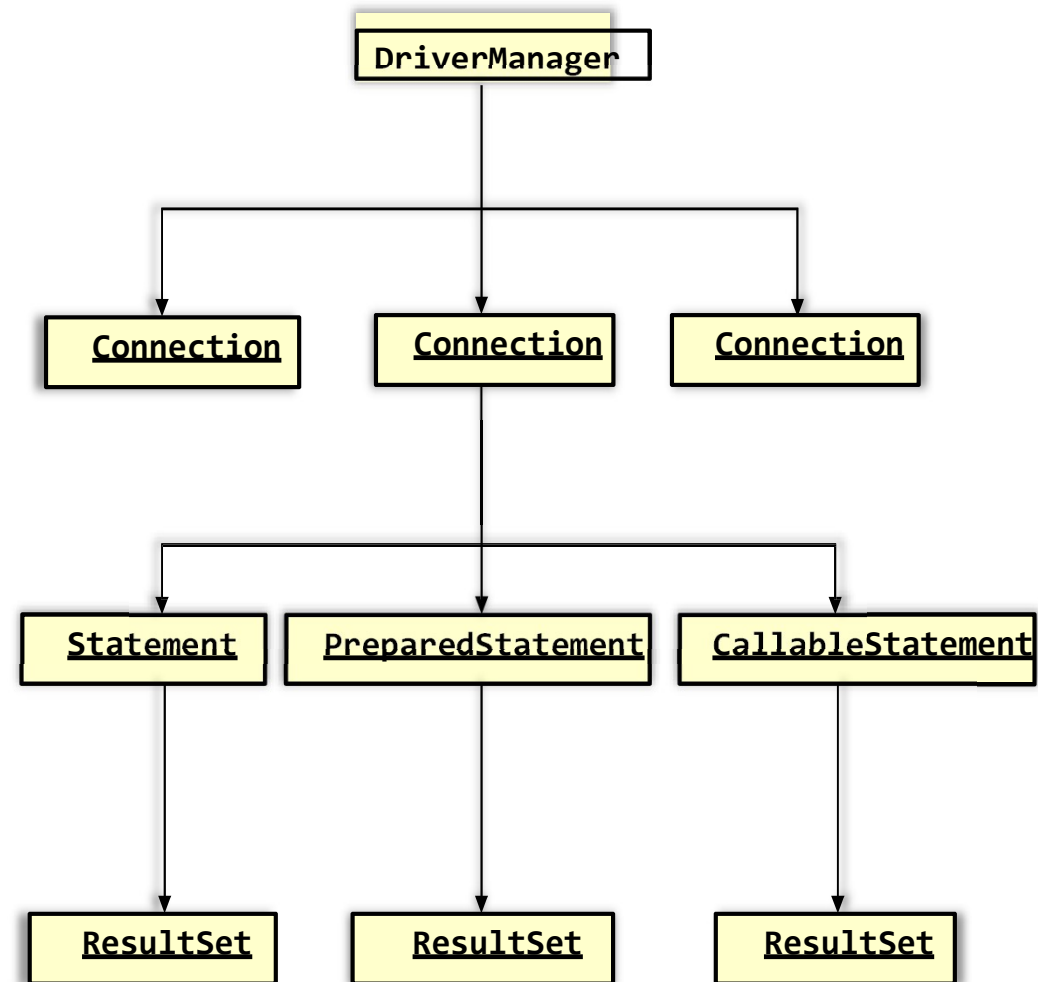
**ResultSet** encapsulant le résultat des requêtes

```
ResultSet rs = stmt.executeQuery(...);
```

## 4) Exploiter les résultats de la requête

l'objet **ResultSet** permet de parcourir le résultat de la requête et de récupérer les données

```
while (rs.next()) {  
    String nom =  
        rs.getString("NOM");  
    ...  
}
```



- A la compilation on n'utilise que les types définis dans l'API JDBC (`java.sql`)
- A l'exécution les objets sont instanciés à des classes d'implémentation fournies par le pilote (driver) JDBC



# 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 apparue.

# 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

Avant de pouvoir être utilisé, le driver doit être enregistré auprès du **DriverManager** de 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/nomdb`
  - `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`)
  - `nomdb` : 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 votre ville : `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 votre ville : `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 r'eussie");
}
```

# 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 r'eussie");
}
```

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 générée 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 générée pour la personne : " + resultat.getInt(1));
}
```

# DAO

Le **DAO** (Data Access Object) est un concept utilisé pour organiser le code qui interagit avec une base de données dans une application. Il s'agit d'un modèle de conception (design pattern) qui permet de rendre le code plus propre et plus facile à maintenir en séparant la logique d'accès aux données du reste de l'application.

## **Pourquoi utiliser un DAO ?**

Imagine que tu crées une application qui doit récupérer, ajouter, modifier, ou supprimer des informations dans une base de données. Sans DAO, tu pourrais avoir du code SQL ou des appels directs à la base de données dispersés dans plusieurs parties de ton application. Cela peut vite devenir compliqué à gérer et rend le code difficile à lire et à maintenir.

## Structure d'un DAO

Un DAO se compose généralement de deux parties principales :

**Une interface** qui définit les opérations possibles sur les données, comme créer, lire, mettre à jour, et supprimer (CRUD).

**Une classe qui implémente cette interface** et contient le code spécifique pour accéder à la base de données

# 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 num´ero " + insertedPersonne.getNum()
                + " a ´et´e ins´er´ee");
        } else {
            System.out.println("probl`eme 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 a 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) {  
}
```