ECOLE SUPÉRIEURE DE LA STATISTIQUE
ET DE L'ANALYSE DE L'INFORMATION

Cours 2
Introduction à JDBC
Accès aux bases de données en Java

AÏCHA EL GOLLI
aicha.elgolli@essai.ucar.tn

Septembre 2024

Introduction

Java → monde des OBJETS

SGBD R → monde des RELATIONS

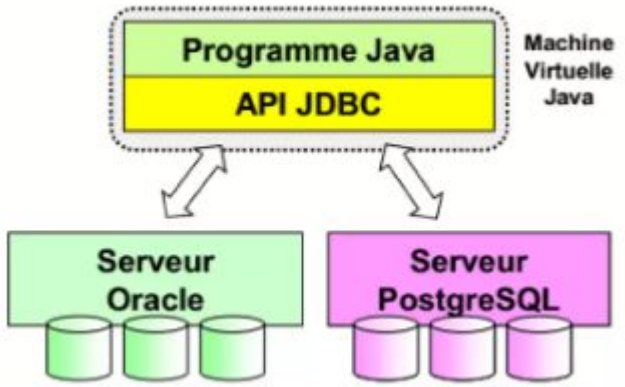
Un mapping Objet-Relationnel est nécessaire (ORM)

API JDBC — Java Database Connectivity

- Accès standardisé aux bases de données,
- Exploitation du SQL,
  - LMD,
  - LDD
- Support des protocoles réseaux

Introduction

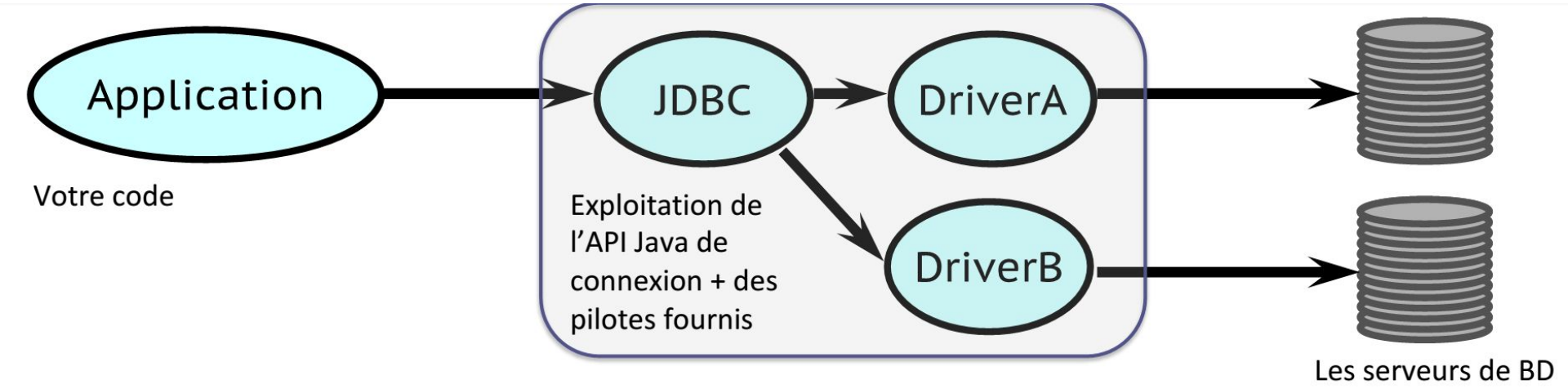
- JDBC (Java DataBase Connectivity) est une bibliothèque d'interfaces et de classes utilisées pour accéder à un SGBDR.
- Un programme JDBC envoie à un SGBDR des requêtes écrites en SQL puis exploite le résultat renvoyé en Java.
- Chaque éditeur de SGBDR fournit son driver JDBC sous forme d'un ensemble de classes rassemblées dans un fichier d'archive .jar , qu'il faut ajouter à l'option –classpath lors de l'exécution de votre programme.
- L'API JDBC est indépendante des SGBD. Un changement de SGBD ne doit pas impacter le code applicatif



Introduction
JDBC : Java DataBase Connectivity

- ✔Pour se connecter a une base de données:Il nous faut un JDBC
  - Une API (interface d'application) créée par Sun Microsystems
  - elle permet d'accéder aux bases de données en utilisant un driver JDBC
- ✔Framework permettant l'accès aux bases de données relationnelles dans un programme Java
- ✔Indépendamment du type de la base utilisée (mySQL, Oracle, Postgres ...)
- ✔**Seule la phase de connexion au SGBDR change**
- ✔Permet de faire tout type de requêtes
- ✔Sélection de données dans des tables
- ✔Création de tables et insertion d'éléments dans les tables
- ✔Gestion des transactions
- ✔Packages : java.sql et javax.sql

Architecture



JDBC (Java Database Connectivity) est une API permettant un accès simple et rapide à un grand nombre de bases de données.

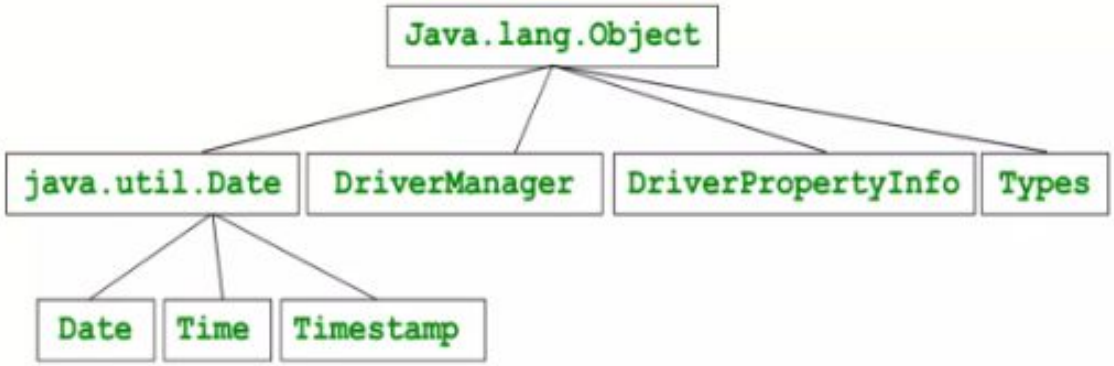
JDBC est indépendant des BD.

Il y a un pilote (Driver) par base de données.

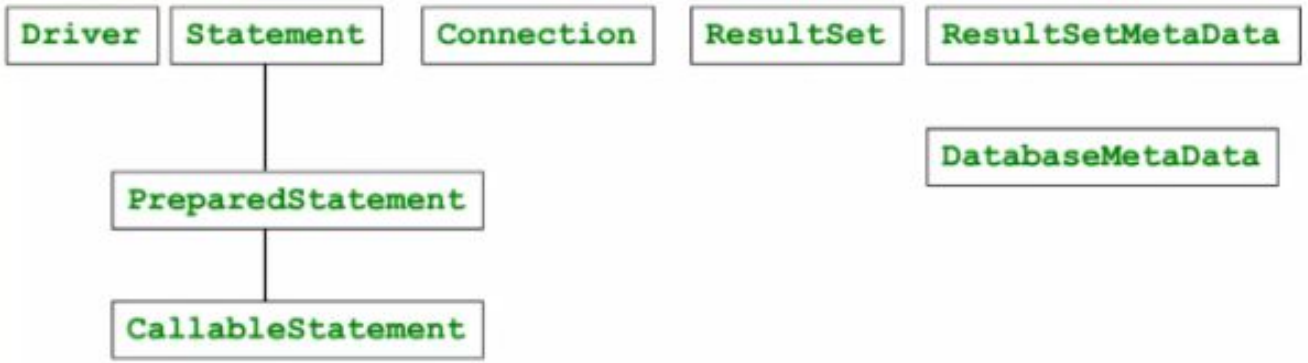
Classes et Interfaces JDBC

- *L'API JDBC définit un ensemble d'interfaces et classes (package **java.sql**) qui définissent un protocole de communication entre le programme java client et le SGBD.*
- le package java.sql est complété par le package javax.sql

Les classes du package *java.sql*



Les interfaces du package *java.sql*



Mise en œuvre de JDBC

- Etape 0: Importer le package java.sql et le driver
- Etape 1 Etablir la connexion au SGBD
- Etape 2 Créer la requête (ou instruction SQL)
- Etape 3 Exécuter la requête
- Etape 4 Traiter les données retournées
- Etape 5 Fermer la connexion

Pour télécharger le driver mysql pour JDBC

MySQL Connector/J 8.0.20 et plus

1. Aller sur: https://dev.mysql.com/downloads/connector/j/

2. Télécharger et Décompresser l'archive .zip

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**

Ou aussi

Faire clic droit sur le projet dans *Package Explorer* et aller dans ***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.

Étape 1 : Etablir la connexion à la BD

- **On utilise la méthode getConnection() de DriverManager avec trois arguments :**
  - ***URL** de la base de données de la forme : Protocole :<sous-protocole>:<nom-BD>?param=valeur, …*
    - **Protocole: JDBC**
    - **sous-protocole : mysql**
    - **exemples**
      - ***String url = "jdbc:odbc:maBase"***
      - ***String url = "jdbc:mysql://127.0.0.1:3306/maBase"***
      - ***String url = "jdbc:postgresql://localhost/maBase"***
  - ***User** : le nom de l'utilisateur de la base*
  - ***Password** : son mot de passe*
- **Connection connexion = DriverManager.getConnection(url ,user , pw)**
- **le DriverManager essaie tous les drivers enregistrés jusqu'à ce qu'il trouve un driver qui lui fournisse une connexion.**
- **La plupart des méthodes lèvent l'exception java.sql.SQLException.**

Principes généraux d'accès à une BDD

Au préalable

- Préciser le type de driver que l'on veut utiliser
  - Driver permet de gérer l'accès à un type particulier de SGBD

Première étape

- Récupérer un objet « **Connection** » en s'identifiant auprès du SGBD et en précisant la base utilisée

Étapes suivantes

- A partir de la connexion, **créer** un « **statement** » (état correspondant à une requête particulière
- **Exécuter** ce **statement** au niveau du SGBD
- **Fermer** le **statement**

Dernière étape

- Se déconnecter de la base en fermant la connexion

Déclaration du pilote JDBC

Méthode de chargement explicite d'un pilote :

Chargement du driver 5

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

Chargement du driver 8

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

L'appel à forName() déclenche un chargement dynamique du pilote.

- **Un programme peut utiliser plusieurs pilotes, un pour chaque base de données.**
- **Le pilote doit être accessible à partir de la variable d'environnement CLASSPATH.**
- **Le chargement explicite est inutile à partir de JDBC 4 (si vous utilisez la version 17 du JDK par exp)**

Deuxième étape
Se connecter à la base de données

Il faut spécifier l'URL :de la forme

jdbc:mysql://hote:port/nombd

- hote : le nom de l'hôte sur lequel le serveur MySQL est installé (dans notre cas localhost ou 127.0.0.1)
- port : le port TCP/IP utilisé par défaut par MySQL 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)

Connexion à la base

```
String url = "jdbc:mysql://localhost:3306/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();
        }
}
```

Connexion à la base

En cas de problème avec le chargement du driver, modifier l'URL de connexion ainsi :

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

Connexion à la base de données

- Ouverture de la connexion :

```
Connection conn = DriverManager.getConnection(url,
                                  user, password);
```

- Identification de la BD via un URL (Uniform Ressource Locator) de la forme générale

jdbc:*driver*:*base*

l'utilisation de JDBC | le driver ou le type du SGBDR | identification de la base

Méthode d'ouverture d'une nouvelle connexion :

Connection **newConnection()** throws SQLException {

final String url = "jdbc::mysql://localhost/dbessai";

Connection connec = DriverManager.getConnection(url, "root", "");

return connec;

}

Votre base de données s'appelle « dbessai »

```
/* Connexion à la base de données */
Connection c = null;
try {
    c = newConnection();
    /* Ici, nous placerons nos requêtes vers la BDD */
    /* ... */
} catch ( SQLException e ) {
    /* Gérer les éventuelles erreurs ici */
} finally {
    if ( c != null )
        try {
            /* Fermeture de la connexion */
            connexion.close();
        } catch ( SQLException ignore ) {
            /* Si une erreur survient lors de la fermeture, il suffit de
  l'ignorer. */
        }
}
```

L'établissement d'une connexion s'effectue à travers l'objet DriverManager. Il suffit d'appeler sa méthode statique getConnection() pour récupérer un objet de type **Connection**.

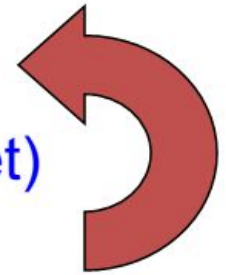
celle-ci prend en argument l'adresse de la base de données, le nom d'utilisateur et le mot de passe associé.

L'appel à cette méthode peut retourner des erreurs de type SQLException :

- si une erreur **SQLException: No suitable driver** est envoyée, alors cela signifie que le driver JDBC n'a pas été chargé ou que l'URL n'a été reconnue par aucun des drivers chargés par votre application ;
- si une erreur **SQLException: Connection refused** ou **Connection timed out** ou encore **CommunicationsException: Communications link failure** est envoyée, alors cela signifie que la base de données n'est pas joignable.

Enfin, peu importe que la connexion ait réussi ou non, retenez bien que sa fermeture dans un bloc finally est extrêmement importante. Si vous ne fermez pas les connexions que vous ouvrez, et en gardez un grand nombre ouvertes sur une courte période, le serveur risque d'être saturé et de ne plus accepter aucune nouvelle connexion ; votre application risque alors de planter.

- Ouvrir la connexion
- Créer un objet Statement
- Exécuter une requête sur ce Statement
- Si SELECT, traiter le résultat obtenu (ResultSet)
- Fermer le Statement
- Fermer la connexion



Nota : un seul ResultSet par Statement

Gestion des connexions

Interface java.sql.Connection

Préparation de l'exécution d'instructions sur la base, 2 types

- Instruction simple : **classe Statement**
  - On exécute directement et une fois l'action sur la base
- Instruction paramétrée : **classe PreparedStatement**
  - L'instruction est générique, des champs sont non remplis
  - Permet une pré-compilation de l'instruction optimisant les performances
  - Pour chaque exécution, on précise les champs manquants
- Pour ces 2 instructions, **2 types d'ordres possibles**
  - Update : mise à jour du contenu de la base
  - Query : consulatation (avec un select) des données de la base

Gestion des connexions

Méthodes principales de Connection

Statement **createStatement()**

- Retourne un état permettant de réaliser une instruction simple

PreparedStatement **prepareStatement(String ordre)**

- Retourne un état permettant de réaliser une instruction paramétrée et pré-compilée pour un ordre « ordre »
- Dans l'ordre, les champs libres (au nombre quelconque) sont précisés par des « ? »
  - Ex : ''select nom from clients where ville=?''
  - Lors de l'exécution de l'ordre, on précisera la valeur du champ

void **close()**

- Ferme la connexion avec le SGBD

Instruction simple

Classe Statement

ResultSet **executeQuery(String ordre)**

- Exécute un ordre de type SELECT sur la base
- Retourne un objet de type ResultSet contenant tous les résultats de la requête

int **executeUpdate(String ordre)**

- Exécute un ordre de type INSERT, UPDATE, ou DELETE
- 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

void **close()**

- Ferme l'état

Instruction paramétrée

Classe **PreparedStatement**

Avant d'exécuter l'ordre, on remplit les champs avec

- void set[Type](int index, [Type] val)
  - Remplit le champ en i^{ème} position définie par index avec la valeur val de type [Type]
  - [Type] peut être : String, int, float, long ...
  - Ex : void setString(int index, String val)

ResultSet **executeQuery()**

- Exécute un ordre de type SELECT sur la base
- Retourne un objet de type ResultSet contenant tous les résultats de la requête

int **executeUpdate()**

- Exécute un ordre de type INSERT, UPDATE, ou DELETE
- 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

Lecture des résultats

Classe ResultSet

Contient les résultats d'une requête SELECT

- Plusieurs lignes contenant plusieurs colonnes
- On y accède ligne par ligne puis valeur par valeur dans la ligne

Changements de ligne

- boolean next()
  - Se place à la ligne suivante s'il y en a une
  - Retourne true si le déplacement a été fait, false s'il n'y avait pas d'autre ligne
- boolean previous()
  - Se place à la ligne précédente s'il y en a une
  - Retourne true si le déplacement a été fait, false s'il n‘y avait pas de ligne précédente
- boolean absolute(int index)
  - Se place à la ligne numérotée index
  - Retourne true si le déplacement a été fait, false sinon

Lecture des résultats

Classe **ResultSet**

Accès aux colonnes/données dans une ligne

[type] get[Type](int col)

- Retourne le contenu de la colonne col dont l'élément est de type [type] avec [type] pouvant être String, int, float, boolean ...
- Ex : String getString(int col)

[type] get[Type](**String** nom_col)

Fermeture du ResultSet

- void close()

Remarques :

- l'indice commence à **1**,
- Il fait référence au numéro de colonne du ResultSet (celui défini dans l'ordre SELECT) et **non au numéro de colonne** de la table.
- Si le type de la colonne est différent, il faut transtyper.

Correspondance des types

Type SQL	Type Java
CHAR, VARCHAR2,	String
NUMERIC, DECIMAL	java.math.BigDecimal
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
BINARY, VARBINARY, LONGVARBINARY	byte [ ]
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

Exception SQLException

Toutes les méthodes présentées précédemment peuvent lever l'exception ***SQLException***

Exception générique lors d'un problème d'accès à la base lors de la connexion, d'une requête ...

- Plusieurs spécialisations sont définies (voir API)

Opérations possibles sur cette exception

- int getErrorCode() : le code de l'erreur renvoyé par le SGBD (et dépendant du type du SGBD)
- SQLException getNextException() : si plusieurs exceptions sont chaînées entre elles, retourne la suivante ou null s'il n'y en a pas
- String getSQLState() : retourne « l'état SQL » associé à l'exception

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

Exemple

Accès à une base « Mabase » contenant 2 tables

categorie (codecat (int), libellecat(varchar))

codecat	libellecat
1	Legumes
2	Fruits

produit (codeprod(int), nomprod(varchar), codecat*)

Exécution d'une instruction simple de type SELECT

Lister tous les produits d'une catégorie

La classe MyConnection

```
package monappli.config;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class MyConnection {
  private static String url = "jdbc:mysql://localhost:3306/Mabase";
  private static String utilisateur = "root";
  private static String motDePasse = "";
  private static Connection connexion = null;
  private MyConnection() {
  try {
  Class.forName("com.mysql.cj.jdbc.Driver");
  connexion = DriverManager.getConnection( url, utilisateur, motDePasse );
  } catch ( Exception e ) {
  e.printStackTrace();}
  }
```

La suite de la classe

```
public static Connection getConnection() {
 if (connexion == null) {
 new MyConnection();}
 return connexion; }


public static void stop() {
 if (connexion != null) {
 try {
 connexion.close();
 } catch (SQLException e) {
 e.printStackTrace();}
 }
 }

}
```

Les classes Categorie & Produit

```
package monappli.model;
 public class Catégorie{
 private int codecat;
 private String libellecat;

 // + getters + setters + constructeur
 sans paramètre + constructeur avec 2
 paramètres + constructeur avec
 catégorie
 }
```

```
package monappli.model;
 public class Produit{
 private int codeprod, codecat;
 private String nomprod;

 // + getters + setters + constructeur
 paramètres + constructeur avec
 produit
 }
```

```
public void listCategorie()throws SQLException{
Connection c = MyConnection.getConnection();
if (c !=null) {
   try {
   Statement st = c.createStatement();
   ResultSet res = st.executeQuery("select codecat, libellecat from categorie");
        while(res.next())
   System.out.println("Catégorie : "+res.getInt(1)+", " + res.getString(2));

   } catch (SQLException e) {
   e.printStackTrace();
      }
}
}
```