Cours 5 - Accès aux données via JDBC

6 juin 2019

Note globale : DS Promo (2/3) + DS de groupe (1/3)

Isabelle Gonçalves - isabelle.goncalves@univ-lyon1.fr

* contient des extraits des cours de Véronique Deslandres, Ikbel Guidara, Alain Pillot, Mooneswar Ramburrun + de la documentation Java

Présentation

JDBC (Java DataBase Connectivity)

JDBC est une interface de programmation (API) qui permet l'accès à des bases de données avec le langage SQL depuis un programme Java. L'API JDBC est 'presque' totalement indépendante des SGBD. Elle fournit uniquement des interfaces qui doivent être implémentées par les pilotes correspondants.

API (Application Programming Interface) : Ensemble de classes et de méthodes, dédiées à la manipulation d'une certaine catégories d'objets.

Les classes de JDBC sont regroupées dans le package java.sql

Présentation

Pilotes

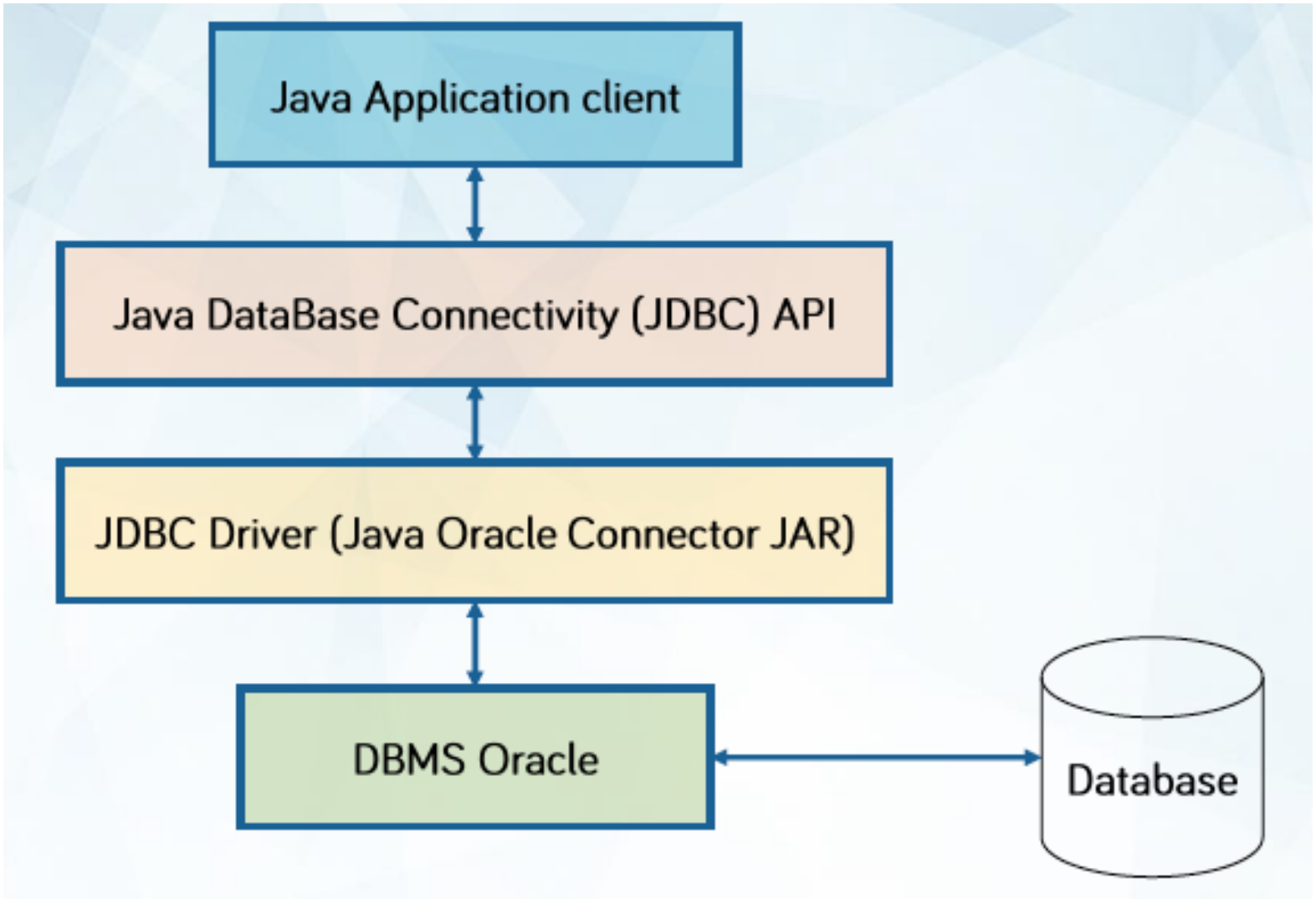
Avant de pouvoir interagir avec une base de données, il est nécessaire de télécharger le pilote et de l'inclure dans le projet.
Un pilote est une bibliothèque qui implémente les méthodes fournies par JDBC pour pouvoir interagir avec un SGBD particulier (dépend donc du SGBD).

Pilote Oracle

Dans ce cours, nous allons commencer par travailler avec le pilote correspondant au SGBD Oracle :
http://www.oracle.com/technetwork/database/features/jdbc/default-2280470.html

Télécharger la bibliothèque ojdbc7.jar

Architecture Générale



Utilisation du JDBC

JDBC se compose :

- du package java.sql
- du package javax.sql

Exceptions

Les méthodes des classes de ces packages sont susceptibles de lever des exceptions qui sont de type **SQLException**.

Connexions avec un DataSource (1/4)

- La classe **DataSource** est un interface, une fabrique de connexions à une source physique de données.

DataSource

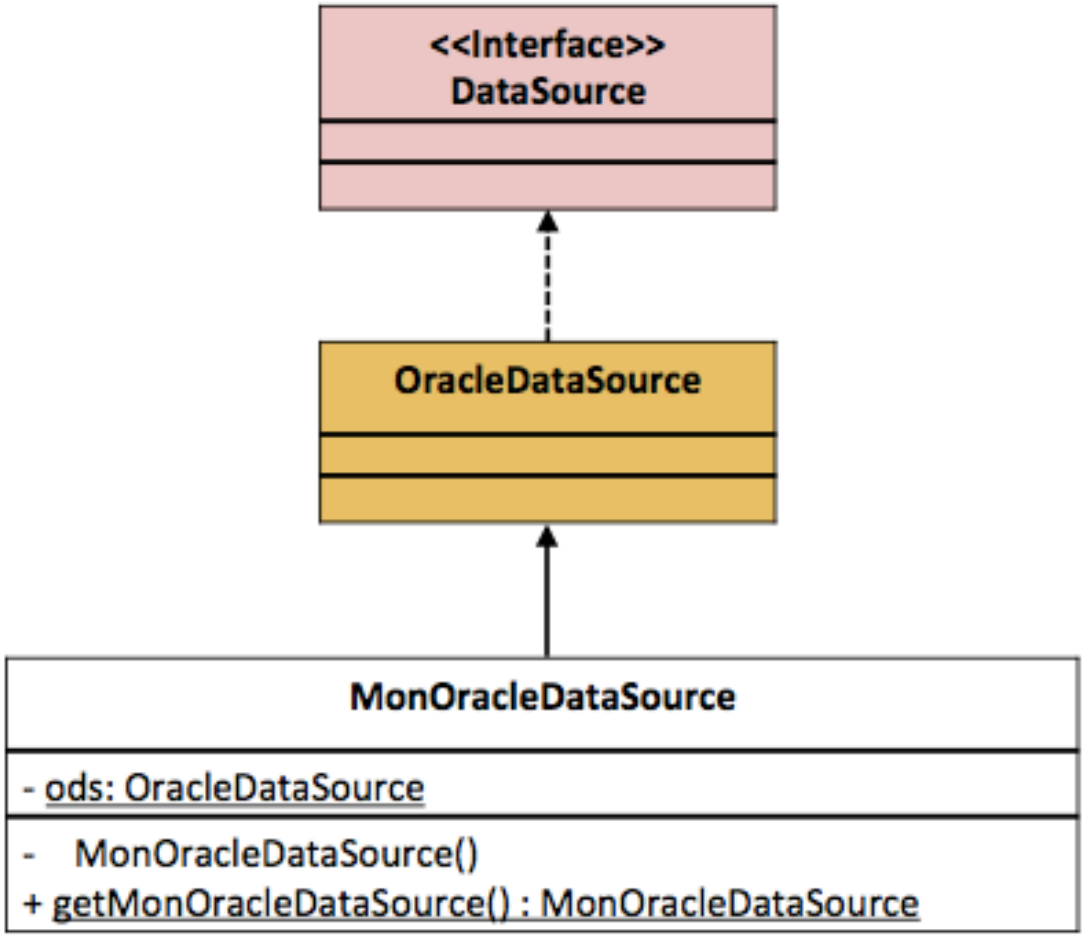
Package javax.sql

- La classe **OracleDataSource** est l'implémentation de cette interface pour le SGBD Oracle.

OracleDataSource

Package oracle.jdbc.pool (dans ojdbc7.jar)

Connexions avec un DataSource (2/4)



Connexions avec un DataSource (3/4)

Un DataSource permet un pool de connexions.
Une seule instance sera nécessaire
=> utilisation du patron de conception (design pattern) **Singleton**.

```
public class MonOracleDataSource extends OracleDataSource {
    //L'instance unique
    private static MonOracleDataSource ods;
    //Constructeur privé
    private MonOracleDataSource() throws SQLException{
    }
    public static MonOracleDataSource getOracleDataSource() throws
            SQLException {
        if (ods == null) {// on contrôle qu?il n?existe pas déjà une source de données
            ods = new MonOracleDataSource();
            // on la définit avec les paramètres suivants :
            ods.setDriverType("thin");
            ods.setPortNumber(1521);
            ods.setServerName("iutdoua-oracle.univ-lyon1.fr");
            ods.setServiceName("orcl.univ-lyon1.fr");
            ods.setUser("pxxxxxx");
            ods.setPassword("xxxxxx");
        } // sinon, un datasource existe déjà :
        return ods;
    }
}
```

Connexions avec un DataSource (4/4)

```
public class MonMariadbDataSource extends MariaDbDataSource{

   //L'instance unique
    private static MonMariadbDataSource ods;
    //Constructeur privé
    private MonMariadbDataSource() throws SQLException{
    }
    public static MonMariadbDataSource getOracleDataSourceDAO() throws
            SQLException {
        if (ods == null) {// on contrôle qu?il n?existe pas déjà une source de données
            ods = new MonMariadbDataSource();
            // on la définit avec les paramètres :
            ods.setPortNumber(3306);
            ods.setServerName("iutdoua-web.univ-lyon1.fr");
            ods.setDatabaseName("pxxxxx");
            ods.setUser("pxxxxx");
            ods.setPassword("xxxxxx");
        } // sinon, un datasource existe déjà :
        return ods;
    }
}
```

MariaDbDataSource

Package org.mariadb.jdbc (dans mariadb-java-client-2.4.1.jar)

Les classes métier (1/3)

- Java permet de manipuler des objets
- Un SGBD relationnel permet de manipuler des relations

Un mapping O-R est nécessaire (ORM)

Exemple :
TCLIENT(IdClient, Nom, TauxRemise)
TCOMMANDE(NumCommande, DateCommande, #IdClient)

Correspondances des types SQL en Java

Number -> int
Number(5,2) -> float ou double
Varchar2 -> String
Date -> java.sql.Date

Les classes métier (2/3)

```
public class Client {
    private final int id;
    private final String nom;
    private float remise;
    public Client(int id, String nom)
    {
        this.id = id;
        this.nom = nom;
    }
    public int getId()
    {
        return id;
    }
    public float getRemise()
    {
        return remise;
    }
    @Override
    public String toString()
    {
        return "Id=" + id + ", nom=" + nom;
    }
}
```

Les classes métier (3/3)

```
public class Commande {
    private String numCommande;
    private Date dateCommande;
    private int idClient; // lien par valeur vers le client de la commande
    public Commande (String numCommande, Date dateCommande, int idClient)
    {
        this.numCommande = numCommande;
        this.dateCommande = dateCommande;
        this.idClient = idClient;
    }
    public String getNumCommande(){
        return numCommande;
    }
    public DategetDateCommande(){
        return dateCommande;
    }
    public int getIdClient(){
        return idClient;
    }
}
```

Les objets persistants

- Qu'est-ce que la persistance ?
  - notion qui caractérise la capacité d'un objet à conserver son état à l'arrêt du programme
  - il s'agit en fait de sauvegarder les objets
- Plusieurs mécanisme
  - sérialisation dans un fichier binaire
  - sérialisation dans un fichier XML
  - stockage en base de données

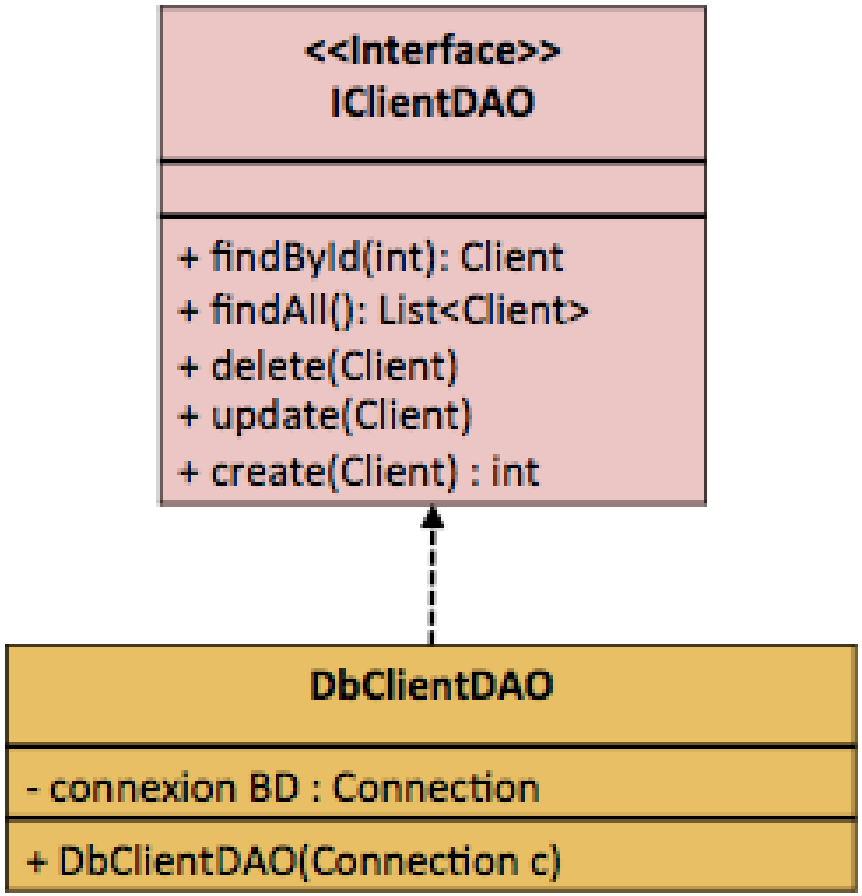
Le pattern DAO (Data Access Object)(1/2)

Propose pour les objets métiers :

- une interface de gestion de la persistance (CRUD)
  - Create
  - Read
  - Update
  - Delete
- et d'autres fonctionnalités métier
- indépendante de la source de données (BD, fichiers XML ...) ou de la table.

Le pattern DAO (Data Access Object) (2/2)

La connection sera fournie à partir du pool de connections d'un DataSource créé avec :
ds.getConnection() ;



Statement : interface pour les requêtes

Syntaxe

```
Statement st = con.createStatement();
```

où con est une connection obtenue à partir d'un DataSource.
java.sql.Statement est une interface pour exécuter une requête SQL. Elle permet :

- d'interroger la base avec la méthode **executeQuery** (SELECT)
- de modifier la base avec la méthode **executeUpdate** (UPDATE, INSERT, CREATE ...)
- l'appel de procédures ou de fonctions stockées avec la méthode **execute**

```
st.close(); /* fermera aussi le ResultSet associé si il existe */
```

ResultSet : récupération des données (1/3)

java.sql.ResultSet est une interface pour accéder aux résultats d'une requête de type SELECT.

- On peut faire plusieurs requêtes sur un même Statement
- Un seul ResultSet par Statement peut être ouvert à la fois

Syntaxe

```
ResultSet rs = st.executeQuery("SELECT...");
...
rs.close();
```

Remarque

à l'instanciation, le pointeur du ResulSet est placé **juste avant** la première ligne.

ResultSet : récupération des données (2/3)

- Parcours d'un ResultSet :
  - La requête ne doit retourner qu'une ligne :

```
if (rs.next()){
```

  - La requête doit retourner plusieurs lignes :

```
while (rs.next()){
```

La méthode next fait avancer le pointeur du ResultSet sur la ligne suivante. La méthode retourne true si une ligne existe, false sinon.

ResultSet : récupération des données (3/3)

- On ne récupère pas l'ensemble des données d'une ligne. On procède colonne par colonne.
- Méthode getXXX() du ResultSet où XXX est un Type Java.

Syntaxe

variable = getType (indice | "nom_colonne")

Exemple

```
nom = rs.getString("nomCli");
remise = rs.getDouble("remise");
noCat = rs.getInt(4);
```

Remarque

Dans une ligne de ResultSet, l'indice des colonnes commence à 1.

ResultSet : valeur null

- En SQL, NULL signifie que le champ est vide
- Ce n'est pas la même chose que 0 ou ""
- Les méthodes getXXX() de ResultSet convertissent les valeurs NULL SQL en valeur *acceptable* pour le type d'objet java demandé.

```
getString(), getDate(),... : retournent null
getByte(), getInt(),... : retournent 0
getBoolean() : retourne false
```

- En JDBC, on peut **explicitement** tester si la dernière colonne lue était à NULL dans la BD

```
ResultSet.wasNull() : retourne true ou false
```

Description et Manipulation de la base

Syntaxe

int nb = st.executeUpdate(ordre LMD ou LDD)

Exemple

```
int nb = st.executeUpdate("DELETE FROM EMPLOYE");
```

Valeur retournée

0 pour un ordre DDL create (ou nombre de tuples créés)
0 pour drop table
le nombre de lignes traitées pour un ordre DML (insert, update, delete).

PreparedStatement : interface pour les requêtes préparées (1/2)

Objectifs

Eviter les injections de code SQL et utiliser des ordres pré-compilés.

Syntaxe

```
PreparedStatement pst = con.prepareStatement("requête");
```

Il s'agit d'un Statement avec variables dont les valeurs seront précisées ultérieurement.
A utiliser :

- pour des requêtes exécutées plusieurs fois
  - Elle sera compilée (*parsed*) une seule fois par le SGBD
- pour des requêtes qui dépendent du contenu de variables, spécialement celles dont la valeur est fournie par l'utlisateur

PreparedStatement : interface pour les requêtes préparées (2/2)

Syntaxe

PreparedStatement pst = con.prepareStatement("ordre sql contenant au moins un ?") ;

un ? représente une valeur qui sera fournie dans un deuxième temps.

Renseignement des valeurs

rps.setXXX(rang, valeur)

XXX le type de la variable (String, Int, Float, Double...)
Le rang est la position du paramètre (?) dans l'ordre SQL. Il commence à **1**.

Exemple de requête préparée

```
String sql = "UPDATE Stocks SET prix = ?, quantite
    = ? WHERE nom = ?";
//préparation de la requête
PreparedStatement ps = con.prepareStatement(sql);
//on assigne un décimal au 1er paramètre
ps.setFloat(1,15.6);
//on assigne un entier au 2nd paramètre
ps.setInt(2,256);
//on assigne une chaîne de caractères au 3ème
ps.setString(3,"café");
//exécution de la requête
int nb = ps.executeUpdate();
```

Exemple de requête préparée dans un for

```
PreparedStatement updateVentes;
String req = "UPDATE cafe SET ventes = ? WHERE
    nom_cafe LIKE ?");
updateVentes = con.prepareStatement(req);
int[] ventesDeLaSemaine = {175, 150, 155};
String[] cafes = {"Colombian", "Expresso", "
    Decaféiné"};
for(int i=0; i<cafes.length; i++){
 updateVentes.setInt(1, ventesDeLaSemaine[i]);
 updateVentes.setString(2, cafes[i]);
 updateVentes.executeUpdate();
}
```

CallableStatement : interface pour appels de fonctions et procédures

Objectif

Appeler une fonction ou une procédure stockée dans le SGBD.

Syntaxe

CallableStatementt cst = con.prepareCall("chaine");

où "chaine" est de la forme :

```
"{? = call nom_fonction([?,?,...])}"
OU
"{call nom_procédure([?,?,...])}"
```

CallableStatement : paramètres en entrée

Le passage de paramètres en entrée est similaire à celui vu pour un PreparedStatement

- Méthodes setXXX (rang, valeur)

```
CallableStatement cst = con.prepareCall("{?= call
    ma_fonction(?)}");
cst.setString(2,ma_variable_entree);
```

CallableStatement : sortie(s) (1/2)

- Pour récupérer la valeur de retour des fonctions
- Pour récupérer les variables des paramètres en mode OUT ou INOUT pour les procédures et les fonctions
- Méthode registerOutParameter(rang, type)

Le rang est la position du paramètre (?) dans la chaîne d'appel. Il commence à **1** Le type est un code (constante) défini dans l'API JDBC (package java.sql)

```
CallableStatement cst = con.prepareCall("{?= call
    ma_fonction(?)}");
cst.registerOutParameter(1,Type.INTEGER);
```

D'autres types génériques SQL

Type.VARCHAR, Type.DATE, Type.REAL...

CallableStatement : sortie(s) (2/2)

- Méthode getXXX(rang | "nom_colonne") où XXX est un type Java.

Syntaxe

variable = getType (indice | "nom_colonne")

```
CallableStatement cst = con.prepareCall("{?= call
    ma_fonction(?)}");
cst.registerOutParameter(1,Type.INTEGER);
int nb=cst.getInt(1);
```

nb contient la valeur de retour de ma_fonction.

CallableStatement : exemple

Appel de fonction

```
CallableStatement cst = con.prepareCall("{? = call
    getUserNameById(?)}");
cst.setInt(2,13291);
cst.registerOutParameter(1,Types.VARCHAR);
cst.execute();
String username = cst.getString(1)
cst.close();
```

Exercices

Finir le TD5

JDBC : Je retiens

- Les principes : l'API Java, les pilotes des SGBD
  - Les principaux éléments, leur nom, leurs interactions
- La connexion avec un DataSource
- L'utilisation de DAO, le pattern Singleton
- Le principe d'exécution des requêtes :
  - Les interfaces Java : Statement, PreparedStatement, CallableStatement
  - Les méthodes execute(), executeQuery(),executeUpdate()
  - L'exploitation des résultats avec un ResultSet
- Les exceptions à gérer de type SQLException.