TP JDBC et Programmation Java externe à MySQL

1. Connectivité avec une base de données Oracle au travers de l'interface JDBC

L'interface de programmation (API) JDBC (Java DataBase Connectivity) est une librairie Java d'accès aux SGBDs relationnels (Oracle mais aussi Postgresql, MySql, Informix, Access ...). L'idée générale est de faire abstraction des SGBDs et de fournir un accès homogène aux données quel que soit le SGBD cible. Les applications Java (applications basées sur une architecture J2EE par exemple) vont donc bénéficier de facilités pour tout ce qui concerne l'exploitation (consultation et mise à jour des données, évolution des schémas de données ...) de bases de données pouvant être hébergées sur différents serveurs de données ainsi que de facilités pour tout ce qui concerne la gestion des transactions (essentiel dans un environnement concurrent).
Pour plus de détails, JDBC est une API de Java SE (Java Standard Edition) et donc par extension de Java EE (Java Enterprise Edition), et comprend :

- un module central définissant les notions de connexion à une base de données pour différentes opérations de consultation, de mise à jour ou de possibles évolutions du schéma.
- une extension ajoutant les notions de **sources de données abstraites** accessibles via JNDI (Java Naming and Directory Interface) et potentiellement mutualisables (pooling) (définitions de **transactions globales**)

Nous nous concentrons dans ce TP sur l'utilisation du module central.
Plusieurs pilotes sont disponibles pour permettre à une application Java d'interagir avec un serveur de données. Nous n'aborderons ici que le pilote de type 4 écrit entièrement en Java. L'interface JDBC est composée de plusieurs classes d'accès et d'interfaces organisées dans différents paquetages dont un paquetage de base nommé java.sql.

2. Détails pratiques

De manière à pouvoir exploiter le paquetage java.sql (et autres paquetages nécessaires), il faudra ajouter le chemin de l'archive java (mysql-connector-java-8.0.21.jar) correspondante à votre chemin des classes (CLASSPATH).
Deux cas se présentent :

- Vous souhaitez travailler, avec un éditeur (xemacs par exemple) pour écrire les sources et avec le JDK disponible depuis votre poste de travail.
  Obligation de modifier le fichier .bashrc. en ajoutant la commande
  export CLASSPATH=/.../chemin/.../mysql-connector-java-8.0.21.jar :.

- Vous souhaitez travailler avec l'environnement de développement intégré Eclipse, vous aurez alors à référencer dans votre projet l'archive JDBC (mysql-connector-java-8.0.21.jar) ou utiliser Maven.

Vous travaillerez sur la base de données *Mysql* (de l'université) avec votre compte utilisateur.
Pour vous connecter via l'interpréteur sql de mysql, vous aurez à invoquer la commande suivante (p00000009432 est un identifiant enseignant dans sapiens) :

```
mysql -h mysql.etu.umontpellier.fr -u p00000009432 -p p00000009432
```

2.1 Description des principales classes et interfaces définies dans le package java.sql

Nous commençons par exploiter les classes et interfaces du paquetage java.sql. Un diagramme de classes UML donne une vision d'ensemble de ces classes.

- **DriverManager** gère la liste des drivers (ou pilotes), le choix et le chargement d'un des drivers ainsi que la création des connexions. Sa méthode la plus notable est la méthode getConnection(url, user, password) qui retourne une référence d'objet de type Connection.
- **Connection** représente une connexion avec le SGBD (canal de communication TCP vers une BD avec un format proche d'une URL) et offre des services pour la manipulation d'ordre SQL au travers de **Statement**, **PreparedStatement** et **CallableStatement**. Les ordres SQL relèvent soit du LDD (CREATE, DROP, ALTER, ...), soit du LMD (SELECT, INSERT, UPDATE, DELETE). Une connexion gère aussi les transactions (méthodes commit(), rollback(), setAutoCommit(boolean), setIsolationLevel(int) ...Par défaut, le mode activé est le mode commit automatique.
  Exemple de format d'une connexion au travers du pilote jdbc oracle de type 4 au serveur mysql.etu.umontpellier.fr et à la base de données p00000009432 :
  jdbc:mysql://mysql.etu.umontpellier.fr/p00000009432

- **Statement** Statement admet notamment trois méthodes executeQuery() pour ce qui concerne les requêtes de consultation (SELECT), executeUpdate() pour ce qui concerne les requêtes de mise à jour ou les ordres de LDD et execute() (plus rarement utilisée).
- **PreparedStatement** correspond à une instruction paramétrée, et en ce sens, contient un ordre SQL précompilé et des paramètres d'entrée (qui ne seront valués que peu avant l'exécution au travers de méthodes setXXX (setInt(1,5), setString(2,"Martin") par exemple) et qui sont représentés dans la requête par des points d'interrogation. L'exécution se fait de manière identique à celle des Statements mais les méthodes executeQuery() ou executeUpdate() sont alors dépourvues d'arguments.
- **ResultSet** L'application de la méthode executeQuery() sur un objet Statement (requête) retourne un objet de la classe **ResultSet** (résultats) qu'il convient ensuite d'exploiter pour la restitution des résultats de la requête. ResultSet gère l'accès aux tuples d'un résultat en offrant des services de navigation et d'extraction des données dans ce résultat (représentée par une table).
  L'accès aux colonnes se fait soit par le nom de colonne, soit par l'index (ordre la colonne dans la requête) au travers de 2 méthodes génériques (peuvent servir tous les types de données SQL) : getString() qui renvoie un String et getObject() qui renvoie un Object.
  ```
  String n = rset.getString("nomPers");
  String v = rset.getString(1);
  Object o = rset.getObject(1);
  ```
  Il existe aussi d'autres méthodes de type getXXX ou XXX représente le type de l'objet retourné (getInt, getBoolean, getFloat, ...). Pour chacune de ces méthodes, le driver doit effectuer une conversion entre le type SQL et le type Java approprié
  cf voir http ://java.sun.com/docs/books/tutorial/jdbc/basics/

Il est à noter qu'un statement concerne un seul ordre SQL à la fois. Si l'on veut réutiliser ce statement pour un autre ordre SQL, il faut s'assurer d'avoir fini d'exploiter la première requête. La méthode getRow() permet de connaître le nombre de tuples satisfaisant la requête. Attention, il faut se placer à la fin du curseur (ou ResultSet) pour avoir le nombre de tuples (un exemple est donné). Plusieurs catégories de ResultSet sont disponibles selon le sens de parcours et la possibilité de modifier ou non les valeurs présentes dans le ResultSet. Avec forward-only, il n'est pas possible de disposer de fonctionnalités de positionnement dans le curseur. Le updatable permet soit de modifier, soit de supprimer, soit d'insérer des tuples dans le ResultSet.

1. forward-only/read-only
2. forward-only/updatable
3. scroll-sensitive/read-only
4. scroll-sensitive/updatable
5. scroll-insensitive/read-only
6. scroll-insensitive/updatable

Différentes signatures de la méthode createStatement permettent de définir le ResulSet approprié. La plus simple de ces signatures (public Statement createStatement()) est sans argument et la méthode executeQuery associée va retourné un ResultSet non défilable, non modifiable et compatible avec les accès concurrents en lecture seule. La signature avec deux arguments createStatement(int,int) permet pour le premier argument, de gérer la possibilité de parcourir et de modifier le ResultSet et pour le deuxième argument, de gérer les accès concurrents en lecture/écriture.

— **ResultSetMetadata** permet d'exploiter les informations sur la structure d'un objet ResultSet, par exemple le nombre de colonnes renvoyées par le ResultSet, le type de chacune de ces colonnes, le nom de la table associée à chacune de ces colonnes ....
  — getColumnCount() : nombre de colonnes
  — getColumnName(int col) : nom d'une colonne
  — getColumnType(int col) : type d'une colonne
  — getTableName(int col) : nom de la table à laquelle appartient la colonne

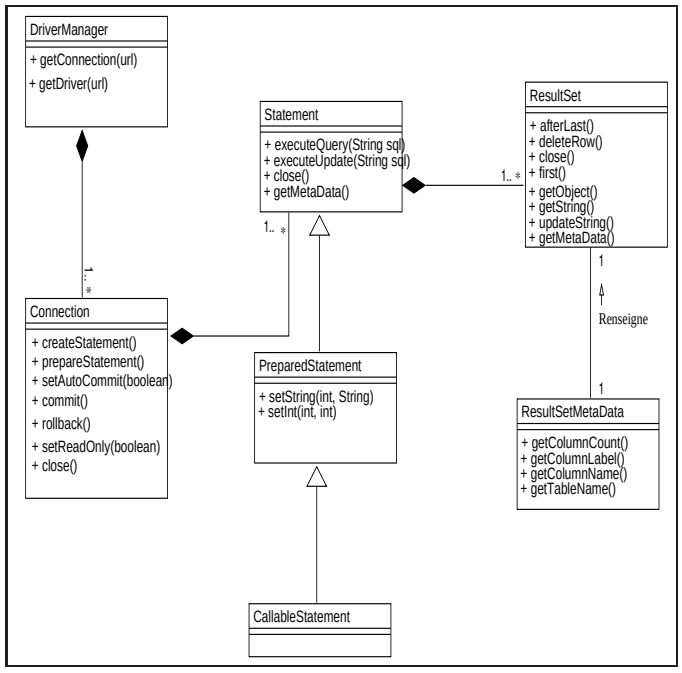

FIGURE 1 – Diagramme (non exhaustif) de classes/interfaces du paquetage java.sql

3. Exercice 1 : Consultation et manipulation des curseurs (ResultSet)

Vous travaillerez sur le script de création de tables de la BD Monument (seule une partie du script vous est donnée, vous aurez à terminer l'écriture de ce script).

Etudiez l'exemple EX1_20.java (donné en fin de ce document) qui consulte une partie du contenu de la table Monument afin de répondre aux deux questions suivantes

- consulter tout le contenu de la table Monument
- retourner les informations sur les monuments (code, nom et type) localisés sur la commune (lieu) de Montpellier (requête portant sur une jointure).

4. Exercice 2 : Manipulation de requêtes paramétrées au travers de l'interface PreparedStatement

Cette interface se concentre sur l'exploitation des ordres qui vont se révéler répétitifs et que le concepteur aura tout intérêt à envisager comme étant paramétrés. Les différences avec Statement portent sur :

- l'instruction SQL va être pré-compilée de manière à améliorer les performances puisque cette instruction est amenée à être appelée de nombreuses fois
- les instructions SQL des instances de PreparedStatement contiennent un ou plusieurs paramètres d'entrée, non spécifiés lors de la création de l'instruction. Ces paramètres sont représentés par des points d'interrogation (?). Ces paramètres doivent être valués avant l'exécution.

Nous utilisons la classe utilitaire nommée Scanner pour faciliter les saisies clavier (paquetage java.util).

Des exemples sont donnés.
Vous exploiterez plusieurs requêtes paramétrées :

- Une requête de consultation sur la condition de sélection portant sur le type de monument (EGLISE, EDIFICE, ...) qui renvoie le code, le nom, et les coordonnées géographiques éventuelles de tous les monuments décrits par ce type.
- Une requête de mise à jour (UPDATE) qui effectue une mise à jour des coordonnées géographiques (qui peuvent être quelconques et donc paramétrée) sur la base d'un code de monument passé en paramètre. Jusqu'à maintenant, vous étiez en mode autocommit, désactivez ce mode autocommit (setAutoCommit(false) et ne validez la transaction qu'après demande de validation par l'usager (méthodes commit() et rollback()), testez les effets sur la base de données en vous connectant par ailleurs via une session mysql. Vous pouvez également passer des ordres de modification de tuples sur les tables que vous être en train de manipuler au travers des classes Java sans faire de validation. Dans quel mode transactionnel, pensez vous être ? Vous pouvez vérifier votre réponse en appelant la méthode getTransactionIsolation() : int sur votre objet Connection. La méthode setIsolationLevel(int) permet de modifier le niveau d'isolation.

5. Exercice 3 : Jeu de requêtes

Vous définirez un jeu de requêtes textuel et au format SQL qui vous semble pertinent dans le contexte du projet. Vous commencer à définir des classes Java qui permettent d'exécuter ces ordres SQL.

6. Exemples de codes

6.1 Exemple 1

N'oubliez pas de fermer proprement les connexions. Utilisez les mécanismes de gestion des exceptions à cet effet. Dans le cas contraire, une exception de type SLQException pourrait nuire à une fermeture correcte de la connexion.

```
import java.sql.*;

 public class EX1_20
   {

 public static void main (String[] args)
      {

   Connection c = null;
   Statement st = null;
   ResultSet rset = null;


try {
   c =
       DriverManager.getConnection("jdbc:mysql://mysql.etu.umontpellier.fr/p00000009432?"
           +
                           "user=p00000009432&password=XXXX");

    st = c.createStatement();
```

```
            rset = st.executeQuery ("select nom from Monument ");
            ResultSetMetaData rsetSchema = rset.getMetaData();
            int nbCols = rsetSchema.getColumnCount();
            for (int i=1; i<=nbCols;i++)
            {
             System.out.print(rsetSchema.getColumnName(i)+ " | ");
            }
            System.out.println();
            System.out.println("--------------------------------");
            while (rset.next ())
            {
             for (int i=1; i<=nbCols;i++)
             {
                System.out.print(rset.getObject(i)+ " | ");;
             }
              System.out.println();
            }
         System.out.println("--------------------------------");

} catch (SQLException ex) {
   // gestion des erreurs
   System.out.println("SQLException: " + ex.getMessage());
   System.out.println("SQLState: " + ex.getSQLState());
   System.out.println("VendorError: " + ex.getErrorCode());
}
}}
```

Listing 1 – Consultation de la table Monument

```
import java.sql.*;

 public class EX1_20_I
   {

 public static void main (String[] args) throws SQLException
      {

   Connection c = null;
   Statement st = null;
   ResultSet rset = null;


try {
   c =
       DriverManager.getConnection("jdbc:mysql://mysql.etu.umontpellier.fr/p00000009432?"
           +
                           "user=p00000009432&password=silong");

    st = c.createStatement();
// insertion
   st.executeUpdate("INSERT INTO lieu VALUES ('12266','SEGUR',2.834599,44.291329,'12')");
   st.executeUpdate("INSERT INTO lieu VALUES
       ('12145','MILLAU',3.077801,44.100575,'12')");

      System.out.println("--------------------------------");

} catch (SQLException ex) {
   // gestion des erreurs
   System.out.println("SQLException: " + ex.getMessage());
```

```
   System.out.println("SQLState: " + ex.getSQLState());
   System.out.println("VendorError: " + ex.getErrorCode());
}
finally {
     st.close();
     c.close();
   }
}}
```

Listing 2 – Insertion dans la table lieu

```
import java.sql.*;
import java.util.Scanner;

public class EX1_20_P {

   public static void main(String[] args)
   throws SQLException
      {
     Connection c = null;
     ResultSet rset = null;
     PreparedStatement pstmt = null;
     Scanner saisie = null;


   try {
   c =
       DriverManager.getConnection("jdbc:mysql://mysql.etu.umontpellier.fr/p00000009432?"
           +
                           "user=p00000009432&password=xxxx");

   c.setAutoCommit(false);
   // curseur parametre pouvant etre parcouru et modifie
   System.out.println("Entrez un code insee de lieu ");
   saisie = new Scanner(System.in);
   String codeinsee = saisie.next();
   System.out.println("Entrez un nom de lieu");
   String nom = saisie.next();
   System.out.println("Entrez une longitude");
   Float longitude = saisie.nextFloat();
   System.out.println("Entrez une latitude");
   Float latitude = saisie.nextFloat();
   String sql = "insert into lieu values (?,?,?,?,null)";
   pstmt = c.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE,
         ResultSet.CONCUR_UPDATABLE);
   pstmt.setString(1,codeinsee);
   pstmt.setString(2,nom);
   pstmt.setFloat(3,longitude);
   pstmt.setFloat(4,latitude);
   int etat = pstmt.executeUpdate();
   // attention scanner prend des virgules et non points pour la decimale
  // 66008 ARGELES-SUR-MER 3.022911 42.546214
   System.out.println("ok "+etat);
   c.commit();
   // testez sans commit
     }
   catch (Exception e) {
                System.err.println("Erreur SQL "+e);
              }
```

```
   finally {
     pstmt.close();
     c.close();
   }
     } }
```

Listing 3 – Requête paramétrée

6.2 Exemple Oracle

```
import java.sql.*;
public class InsertionTuples {
   public static void main(String[] args) throws SQLException {
     try {
        Class.forName("oracle.jdbc.driver.OracleDriver"); }
     {System.err.println("Erreur de chargement du driver "+e);}
     Connection c = null;
     Statement stmt = null;
     ResultSet rset =null;
     try {
   String url = "jdbc:oracle:thin:@venus:1521:master";
   c = DriverManager.getConnection (url, "mast1","mast1");
   // par defaut mode autocommit ici on le desactive
   c.setAutoCommit(false);
   // curseur pouvant se parcourir et se modifier
   stmt = c.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
         ResultSet.CONCUR_UPDATABLE);
   rset = stmt.executeQuery ("select num, nom ,"
                   + " fonction, salaire from emp");
   ResultSetMetaData rsetSchema = rset.getMetaData();
   if (!rset.isAfterLast())
      rset.afterLast();
   rset.previous ();
   System.out.println("nbre de tuples :" + rset.getRow());
   int nbCols = rsetSchema.getColumnCount();
   for (int i=1; i<=nbCols;i++)
   {
      System.out.print(rsetSchema.getColumnName(i)+ " | ");
   }
   System.out.println();
   System.out.println("_____________________");
   if (!rset.isAfterLast())
      rset.afterLast();
   while (rset.previous ())
   {
      for (int i=1; i<=nbCols;i++)
      {
         System.out.print(rset.getObject(i)+ " | ");
      }
       System.out.println();
   }
   rset.moveToInsertRow();
   rset.updateInt(1,37);
   rset.updateString(2,"Voltaire");
   rset.updateString(3,"drh");
   rset.updateFloat(4,2000);
   rset.insertRow();
```

```
  c.commit();
  // testez aussi avec rollback
            }
  catch (Exception e) {
    System.err.println("Erreur SQL "+e); }
  finally {
    rset.close();
    stmt.close();
    c.close();
  } }   }
```

Listing 4 – Manipulation du curseur avec insertion de tuples