

TP Archi-ntier : Programmation Java externe à Oracle

TP à rendre en fin de séance

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 J2SE (Java 2 Standard Edition) et donc par extension de J2EE (Java 2 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 pour ce premier 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 .

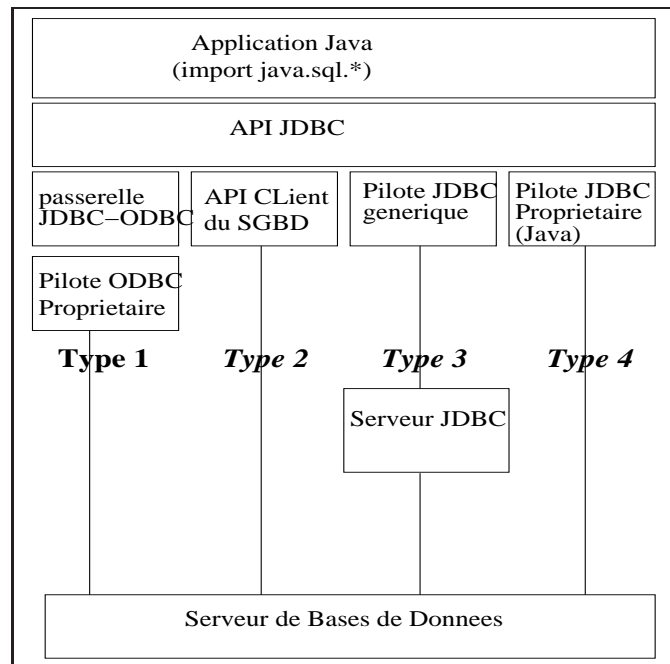


FIG. 1 – Diagramme schématisant les différences entre pilotes

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 (`ojdbc14_g.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 le jdk disponible.
Obligation de modifier le fichier `.bashrc`. en ajoutant la commande

```
export CLASSPATH=$HOME/Java :/usr/local/oracle/client_11/jdbc/lib/ojdbc14_g.jar
```

`$HOME/Java` est superflu, et n'a d'intérêt que si vous souhaitez organiser vos classes java dans un répertoire prévu à cet effet.
- Vous souhaitez travailler avec l'environnement de développement intégré Eclipse, vous aurez alors à référencer dans votre projet l'archive JDBC (`ojdbc14_g.jar`)

Vous travaillerez sur la base de données *master* (serveur neptune) avec des comptes (internes) Oracle de **int1** à **int40** (avec le mot de passe de int1 à int40) pour des raisons de sécurité.

Pour vous connecter via l'interpréteur `sqlplus`, vous aurez à invoquer la commande suivante :
`sqlplus user/password@database` (ex : `sqlplus int1/int1@master`).

La distribution des comptes se fera en début de séance, de manière à ne pas entrer en concurrence avec vos voisins, susceptibles d'utiliser le même compte que vous. Il n'est pas impossible cependant que vous vous trouviez en concurrence avec un de vos voisins, vous préfixerez donc les tables et les contraintes que vous créez par vos initiales. Le fichier de création des tables et d'insertion des tuples (script classique des employés nommé `Creation.sql`) est donné sur : <http://www.lirmm.fr/~mougenot/Jdbc-Exemples> (ce répertoire contient également les exemples traités dans ce tp, le compte exploité est le compte par défaut `scott/tiger`).

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 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 neptune et à la base de données master :

```
jdbc :oracle :thin :@neptune :1521 :master
```

- **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 ResultSet approprié. La plus simple de ces signatures (public Statement createStatement()) est sans argument et la méthode executeQuery associée va retourner 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

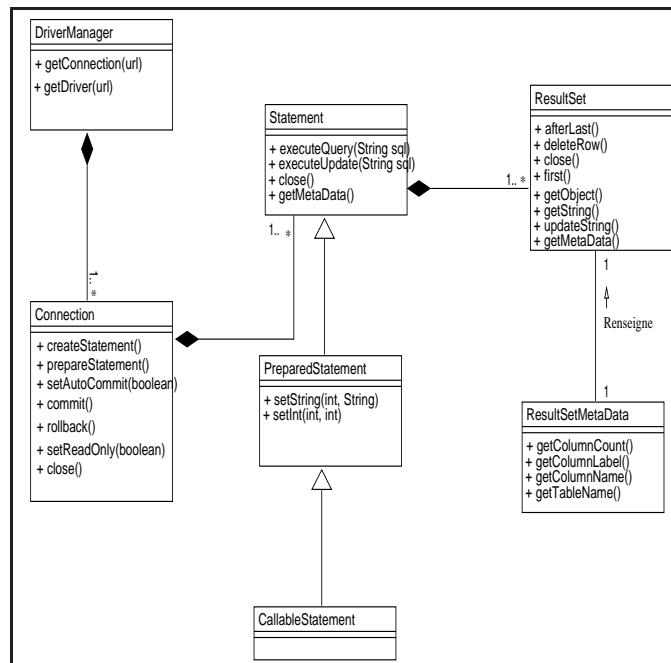


FIG. 2 – Diagramme (non exhaustif) de classes/interfaces du paquetage java.sql

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

Etudiez l'exemple EmpConsult (donné en fin de ce document) qui consulte le contenu de la table EMP afin de répondre aux deux questions suivantes

- construire la classe EmpDescription qui permet de consulter la description du schéma de la table EMP à partir de la table du méta-schéma appropriée (user_tab_columns). Vous vous assurerez de renvoyer le nombre de tuples (nombre d'attributs) retourné par la requête en définissant un curseur (ResultSet) permettant le parcours avant/arrière dans les tuples et en vous positionnant sur le dernier tuple.
- construire la classe DescEmp qui retourne non seulement les attributs et le type de ces attributs mais également le nom des contraintes et le type des contraintes portant éventuellement sur ces attributs. Cette classe doit donner un résultat amélioré par rapport à la commande sqlplus desc nom_de_table. Il vous faut travailler dans ce cadre sur plusieurs tables du méta-schéma (user_tab_columns mais aussi user_constraints et user_cons_columns) et faire appel aux jointures externes. Il vous faut également passer par des vues car les mécanismes de jointure externe ne sont pas efficaces sur les tables du méta-schéma.

4. Exercice 2 : Se positionner et modifier un objet ResultSet

Les objets ResultSet (curseurs) peuvent être parcourus dans les deux sens et peuvent également servir d'appui à la mise à jour dans une table (suppression, modification, insertion de tuples). Ces propriétés de curseurs sont à renseigner dès la création du statement :

```
Statement stmt = c.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
    ResultSet.CONCUR_UPDATABLE);
```

Vous écrirez une classe qui exploite un ResultSet afin d'insérer de nouveaux tuples dans la table Dept. Un exemple (ConsultationMDUpdateCommit) vous est donné qui permet notamment de se positionner en fin de curseur et de compter le nombre de tuples contenus dans la table EMP, d'afficher ces tuples puis de rajouter un tuple dans la table via les méthodes moveToInsertRow(), updateXXX(), insertRow().

5. Exercice 3 : 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 porte sur :

- l'instruction SQL va être pré-compilée de manière à améliorer les performances puisque cette instruction doit 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 une classe utilitaire nommée ConsoleStandard pour faciliter les saisies clavier. Un exemple (DepRepParamSaisieClavier) vous est donné, qui propose de modifier la localisation du

département et le budget sur la base du numéro de département renseigné passé en argument par l'utilisateur.

Vous exploiterez plusieurs requêtes paramétrées :

- Une requête de consultation sur la condition de sélection portant sur la fonction (commercial, secrétaire, etc) qui renvoie le nom, le prénom, le salaire et la commission éventuelle de tous les membres de l'entreprise appartenant à cette fonction.
- Une requête de mise à jour (UPDATE) qui effectue une augmentation de salaire (qui peut être quelconque et donc paramétrée) sur la base d'un nom de salarié passé en paramètre. Jusqu'à maintenant, vous étiez en mode autocommit, désactivez ce mode auto-commit (`setAutoCommit(false)`) et ne validez la transaction qu'après demande de validation par l'utilisateur (méthodes `commit()` et `rollback()`), testez les effets sur la base de données en vous connectant par ailleurs via une session sqlplus. Vous pouvez également passer des ordres de modification de tuples sur les tables que vous êtes en train de manipuler au travers des classes Java sans faire de validation. Dans quel mode d'isolation pensez vous être ? Vous pouvez vérifier votre réponse en appelant la méthode `getIsolationLevel()` :int sur votre objet `Connection`. La méthode `setIsolationLevel(int)` permet de modifier le niveau d'isolation.

6. Exercice 4 : Classe pour supprimer les tables du compte utilisateur

Vous écrirez une classe exploitant deux statements : un qui permet la consultation de toutes les tables de l'utilisateur au travers de la table du méta-schéma `user_tables`) et un statement qui à partir des noms de table renvoyés, les supprime au travers d'une ordre DDL `drop`. Si vous en avez le temps, vous pouvez améliorer votre programme en prenant également en charge la suppression moins drastique des contraintes portant sur les tables.

7. Exemples de codes

7.1 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 `SQLException` pourrait nuire à une fermeture correcte de la connexion.

```
import java.sql.*;

public class EmpDescription {

public static void main(String[] args)
throws SQLException
{
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");

    }
    catch (ClassNotFoundException e)
```

```

{System.err.println("Erreur de chargement du driver "+e);}
    Connection c = null;
    ResultSet rset = null;
    Statement stmt = null;
    try {
String url = "jdbc:oracle:thin:@neptune:1521:master";
    c =
DriverManager.getConnection (url, "scott","tiger");
    stmt = c.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
    rset = stmt.executeQuery ("select num , nom "
        + " from EMP");
    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();
    }
    if (!rset.isAfterLast())
        rset.afterLast(); rset.previous();
    System.out.println("-----");
    System.out.println("nbre de tuples dans la table "+rset.getRow());
    System.out.println("-----");
    }
    catch (Exception e) {
        System.err.println("Erreur SQL "+e);
    }

    finally { rset.close();
    stmt.close();
    c.close();
    }
}
}

```

7.2 Exemple 2

```

import java.sql.*;

public class DepReqParamSaisieClavier {

```

```

public static void main(String[] args)
throws SQLException
{
    try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
    }
    catch (ClassNotFoundException e)
    {System.err.println("Erreur de chargement du driver "+e);}
    Connection c = null;
    PreparedStatement pstmt = null;
    ConsoleStandard cs = new ConsoleStandard();
    try {
    String url = "jdbc:oracle:thin:@neptune:1521:master";
    c =
    DriverManager.getConnection (url, "scott","tiger");
    // par default mode autocommit ici on le desactive
    c.setAutoCommit(false);
    // curseur parametre defilable et modifiable
    System.out.println("Entrez un numero de departement");
    int dep = cs.lireEntier();
    System.out.println("Entrez une localisation");
    String local = cs.lireChaine();
    System.out.println("Entrez un montant de budget");
    double budg = cs.lireDouble();
    String sql = "update dept set lieu=?, budget=? where n_Dept=?";
    pstmt = c.prepareStatement(sql, ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
    pstmt.setString(1,local);
    pstmt.setDouble(2,budg);
    pstmt.setInt(3,dep);
    int etat = pstmt.executeUpdate ();
    System.out.println("nombre de tuples modifies "+etat);
    c.commit();
    // testez sans commit
    }
    catch (Exception e) {
    System.err.println("Erreur SQL "+e);
    }
    finally {
    pstmt.close();
    c.close();
    }
}
}

```

7.3 Exemple 3

```
import java.sql.*;
```

```

public class InsertionTuples {

public static void main(String[] args)
throws SQLException
{
    try {
Class.forName("oracle.jdbc.driver.OracleDriver");
    }
    catch (ClassNotFoundException e)
{System.err.println("Erreur de chargement du driver "+e);}
    Connection c = null;
    Statement stmt = null;
    ResultSet rset =null;

    try {
String url = "jdbc:oracle:thin:@neptune:1521:master";
c =
DriverManager.getConnection (url, "scott","tiger");
// par default mode autocommit ici on le desactive
c.setAutoCommit(false);
// curseur defilable et modifiable
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(1,36);
rset.updateString(2,"Drouet");
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();
}
}
```