JDBC de base

Université Française d'Égypte
Richard Grin
Version O 0.9 – 17/9/12

Présentation

- JDBC (*Java Data Base Connectivity*), API de base pour l'accès aux BD relationnelles avec le langage SQL

Paquetage `java.sql`

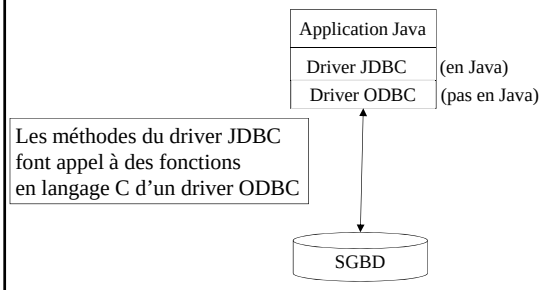
- Contient un grand nombre d'interfaces, et quelques classes
- JDBC ne fournit pas les classes qui implantent ces interfaces

Driver

- Pour travailler avec un SGBD il faut des classes adaptées au SGBD, qui implantent les interfaces de JDBC
- Un ensemble de telles classes est désigné sous le nom de *driver* JDBC

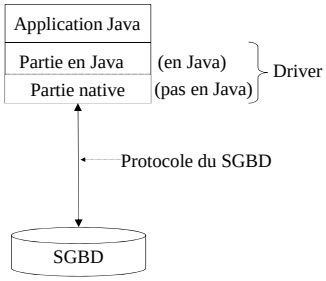
4 types de drivers JDBC

Type 1 : pont JDBC-ODBC

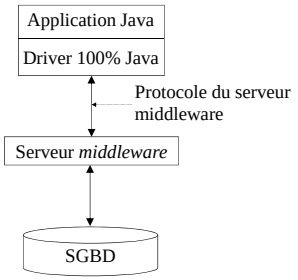


Type 2 : utilise une API native

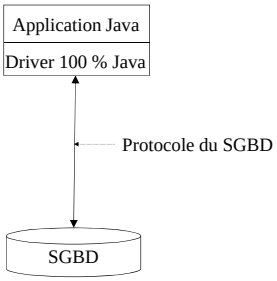
Les méthodes du driver JDBC font appel à des fonctions d'une API du SGBD écrite dans un autre langage que Java



Type 3 : accès à un serveur *middleware*



Type 4 : 100 % Java avec accès direct au SGBD



Travailler avec JDBC

Pour utiliser JDBC

- A l'exécution ajouter le chemin des classes du driver dans le *classpath* :

```
java -classpath /oracle/client/lib/ojdbc6.jar:... ...
```


Dans les classes qui utilisent JDBC

- Importer le paquetage java.sql :

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

- Charger en mémoire la classe du driver (automatique avec drivers compatibles JDBC 4)

Étapes du travail avec une BD

1. Ouvrir une connexion (**`Connection`**)
2. Créer des instructions SQL (**`Statement`**,…)
3. Lancer l'exécution de ces instructions : **`stmt.executeQuery`**, **`stmt.executeUpdate`**,…

(4. Valider ou non la transaction avec **`commit`** ou **`rollback`**)

5. Fermer la connexion (**`close`**)

Classes et interfaces de JDBC

Interfaces principales

- Driver : renvoie une instance de **`Connection`**
- Connection : connexion à une base
- Statement : ordre SQL
- PreparedStatement : ordre SQL paramétré
- CallableStatement : procédure stockée
- ResultSet : lignes récupérées par un ordre SELECT
- ResultSetMetaData : description des lignes récupérées par un SELECT
- DatabaseMetaData : informations sur la BD

Classes principales

- DriverManager : gère les drivers, lance les connexions aux BD
- Date : date SQL
- Time : heures, minutes, secondes SQL
- TimeStamp : date et heure, avec une précision à la microseconde
- Types : constantes pour désigner les types SQL

Exceptions

- SQLException : erreur SQL ; exception contrôlée
- SQLWarning : avertissement SQL

Interface **`Driver`**

- La méthode **`connect`** de **`Driver`** prend en paramètre
  - un URL qui désigne la BD
  - divers informations de connexion (login et mot de passe,…)
- **`connect`** renvoie une **`Connection`** ; renvoie **`null`** si le driver ne convient pas pour se connecter à la BD

URL d'une BD

- jdbc:*sous-protocole*:*base de donnée*
- Pour Oracle :
  jdbc:oracle:thin:@sirocco.unice.fr:1521:INFO
  - oracle:thin sous-protocole (Oracle fournit aussi un autre type de driver)
  - @sirocco.unice.fr:1521:INFO BD INFO située sur la machine sirocco (le serveur écoute le port 1521)
- La forme exacte des parties *sous-protocole* et *base de données* dépend du SGBD cible

Gestionnaire de drivers

- La classe **`DriverManager`** gère les instances de **`Driver`** disponibles pour les différents SGBD utilisés
- La classe de chaque driver doit être chargée en mémoire (automatique avec JDBC 4)
- La classe crée alors une instance d'elle même et enregistre cette instance auprès de la classe **`DriverManager`**

Révision Java :
Comment est-ce possible ?

Obtenir une connexion

- On demande cette connexion à **`DriverManager`** :

```
String url =
 "jdbc:oracle:thin:@sirocco.unice.fr:1521:INFO";
Connection conn =
  DriverManager.getConnection(url,
                              "toto", "mdp");
```

- **`DriverManager`** s'adresse à tous les drivers enregistrés (méthode **`connect`**), jusqu'à ce qu'un driver lui fournisse une connexion (ne renvoie pas **`null`**)

Transactions

- Par défaut la connexion est en « *auto-commit* » : un commit est automatiquement lancé après chaque ordre SQL qui modifie la base
- Il est conseillé d'enlever l'auto-commit : **`conn.setAutoCommit(false)`**
- **`conn.commit()`** valide la transaction
- **`conn.rollback()`** annule la transaction

Instruction SQL simple

- Création d'un Statement :

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


Exécution de l'instruction SQL

- Consultation (select) : executeQuery renvoie un ResultSet pour récupérer les lignes une à une
- Modification des données (update, insert, delete) : executeUpdate renvoie le nombre de lignes modifiées
- Si on ne connaît pas la nature de l'ordre SQL : execute

Consultation des données (SELECT)

```
Statement stmt = conn.createStatement();
// rset contient les lignes renvoyées
ResultSet rset =
  stmt.executeQuery("SELECT nomE FROM emp");
// Récupère chaque ligne une à une
while (rset.next())
  System.out.println (rset.getString(1));
  // ou . . . (rset.getString("nomE"));
stmt.close();
```

La première colonne a le numéro 1

A mettre dans un bloc finally

Types JDBC/SQL

- Tous les SGBD n'ont pas les mêmes types SQL
- Pour cacher ces différences, JDBC définit ses propres types SQL dans la classe **`Types`**, sous forme de constantes nommées
- Ils sont utilisés dans le code quand un type SQL doit être précisé
- Le driver JDBC fait la traduction de ces types dans les types appropriés du SGBD

Types JDBC/SQL (classe **`Types`**)

- CHAR, VARCHAR, LONGVARCHAR
- BIT, TINYINT, SMALLINT, INTEGER, BIGINT
- REAL, DOUBLE, FLOAT
- DECIMAL, NUMERIC
- DATE, TIME, TIMESTAMP
- …

Correspondances entre types Java et SQL

- Il reste le problème de la correspondance entre les types Java et les types SQL
- Dans un programme JDBC, les méthodes getXXX, setXXX servent à préciser cette correspondance

Correspondances avec getXXX()

- On a une grande latitude ; ainsi, presque tous les types SQL peuvent être retrouvés par getString()
- Cependant, des méthodes sont recommandées :
  - CHAR et VARCHAR : getString
  - REAL : getFloat, DOUBLE et FLOAT : getDouble
  - DECIMAL et NUMERIC : getBigDecimal
  - DATE : getDate, TIME : getTime
  - …

Valeur NULL

```
Statement stmt = conn.createStatement();
ResultSet rset = stmt.executeQuery(
    "SELECT nomE, comm FROM emp");
while (rset.next()) {
  nom = rset.getString("nomE");
  commission = rset.getDouble("comm");
  if (rset.wasNull())
    System.out.println(nom + ": pas de comm");
  else
    System.out.println(nom + " a " + commission
                       + " € de commission");
}
```

Peut être null

Modification des données (INSERT, UPDATE, DELETE)

```
Statement stmt = conn.createStatement();
String ville = "NICE";
int nbLignesModifiees = stmt.executeUpdate(
    "INSERT INTO dept (dept, nomD, lieu) "
    + "VALUES (70, 'DIRECTION', '"
    + ville + "')" );
```

N'oubliez pas l'espace !

```
INSERT INTO dept (dept, nomD, lieu)
VALUES(70, 'DIRECTION', 'NICE')
```


Instruction SQL paramétrée

- Les SGBD peuvent n'analyser qu'une seule fois une requête exécutée un grand nombre de fois durant une connexion
- Les requêtes paramétrées permettent de profiter de cette fonctionnalité
- Elles sont représentées par l'interface PreparedStatement qui hérite de Statement

Création d'une requête paramétrée

```
PreparedStatement pstmt =
  conn.prepareStatement("UPDATE emp SET sal = ?"
                    + " WHERE nome = ?");
```

- « ? » indique un paramètre
- Cette requête pourra être exécutée avec plusieurs couples de valeurs :
  (2500, 'DUPOND'),
  (3000, 'DURAND'),
  etc.

Requête paramétrée – Valeurs des paramètres

- Les valeurs des paramètres sont données par les méthodes setXXX(n, valeur) (setDouble, setString,…)
- On choisit la méthode setXXX d'après le type Java de la valeur
- Le driver JDBC fait la conversion dans le bon format pour le SGBD

Requête paramétrée - Exemple

```
PreparedStatement pstmt =
  conn.prepareStatement(
    "UPDATE emp SET sal = ? "
  + "WHERE nomE = ?");
for (int i = 0; i < 10; i++) {
  pstmt.setDouble(1, employe[i].getSalaire());
  pstmt.setString(2, employe[i].getNom());
  pstmt.executeUpdate();
}
```

commence à 1 et pas à 0

Requête paramétrée - NULL

- Pour passer la valeur NULL à la base de donnée
  - utiliser la méthode setNull(n, type) (type de la classe **`Types`**)
  - ou passer la valeur Java **`null`** si la méthode setXXX attend un objet en paramètre

Avantages des PreparedStatement

- Traitement plus rapide s'ils sont utilisés plusieurs fois avec plusieurs paramètres
- Améliorent la portabilité car les méthodes setXXX gèrent les différences entre SGBD (format des dates par exemple)
- Évitent l'injection de code SQL

Procédures stockées

- Permettent non seulement de précompiler des ordres SQL mais aussi de les regrouper
- D'où souvent des performances améliorées
- Attention à la portabilité…

Exemple de procédure stockée (Oracle)

```
create or replace procedure augmenter
  (unDept in integer, pourcentage in number,
   cout out number) is
begin
 select sum(sal) * pourcentage / 100
    into cout
    from emp
    where dept = unDept;
  update emp
    set sal = sal * (1 + pourcentage / 100)
    where dept = unDept;
end;
```


Création d'une procédure stockée

```
CallableStatement cstmt =
  conn.prepareCall("{? = call proc1(?,?)}");
```


Création d'une procédure stockée

Classe des procédures Stockées. Hérite de **`PreparedStatement`**

```
CallableStatement cstmt =
  conn.prepareCall("{? = call proc1(?,?)}");
```


Création d'une procédure stockée

```
CallableStatement cstmt =
  conn.prepareCall("{? = call proc1(?,?)}");
```

Création de la procédure stockée

Création d'une procédure stockée

```
CallableStatement cstmt =
  conn.prepareCall("{? = call proc1(?,?)}");
```

Syntaxe JDBC.
Sera traduit dans la syntaxe
Du SGBD par le driver

Exécution d'une procédure stockée

1. Passage des paramètres « in » et « in/out » par les méthodes setXXX
   Types des paramètres « out » et « in/out » indiqués par la méthode registerOutParameter
2. Exécution par une des méthodes executeQuery, executeUpdate ou execute, suivant le type des commandes SQL que la procédure contient
3. Paramètres « out », « in/out », et la valeur éventuelle retournée, récupérés par les méthodes getXXX

Utilisation d'une procédure stockée

```
CallableStatement csmt = conn.prepareCall(
  "{ call augmenter(?, ?, ?) }");
// 2 chiffres après la virgule pour 3ème paramètre
csmt.registerOutParameter(3, Types.DECIMAL, 2);
// Augmentation de 2,5 % des salaires du dept 10
csmt.setInt(1, 10);
csmt.setDouble(2, 2.5);
csmt.executeQuery(); // ou execute
double cout = csmt.getDouble(3);
System.out.println("Cout total augmentation : "
                   + cout);
```


Procédure stockée contenant plusieurs ordres SQL

- On la lance par **`execute`**
- Pour retrouver tous les résultats de ces ordres, on utilise la méthode **`getMoreResults()`** de la classe **`Statement`**
- Ainsi, si elle contient 2 ordres SELECT, on récupère le 1er ResultSet par **`getResultSet`** ; on passe à la 2ème requête par **`getMoreResults`** et on récupère son ResultSet par **`getResultSet`**

Syntaxe spéciale de JDBC (« *SQL Escape syntax* »)

- Pour ne pas dépendre de particularités des différents SGBD
- dates : **`{d '2000-10-5'}`**
- appels de fonction : **`{fn concat("Hot", "Java")}`**
- **`...`**

Les Meta données

- Informations sur les données récupérées par un SELECT (interface ResultSetMetaData),
- mais aussi sur la base de données elle-même (interface DatabaseMetaData)

ResultSetMetaData

```
ResultSet rs =
  stmt.executeQuery("SELECT * FROM emp");
ResultSetMetaData rsmd = rs.getMetaData();
int nbColonnes = rsmd.getColumnCount();
for (int i = 1; i <= nbColonnes; i++) {
  String typeColonne = rsmd.getColumnTypeName(i);
  String nomColonne = rsmd.getColumnName(i);
  System.out.println("Colonne " + i + " de nom "
                     + nomColonne + " de type "
                     + typeColonne);
}
```


DatabaseMetaData

```
private DatabaseMetaData metaData;
private java.awt.List listTables = new List(10);
. . .
  metaData = conn.getMetaData();
  String[] types = { "TABLE", "VIEW" };
  ResultSet rs =
    metaData.getTables(null, null, "%", types);
  String nomTables;
  while (rs.next()) {
    nomTable = rs.getString(3);
    listTables.add(nomTable);
  }
```

Joker pour noms des tables et vues

Fermer les ressources

- Toutes les ressources utilisées par JDBC doivent être fermées pour les rendre au SGBD
- Le plus souvent la fermeture doit se faire dans un bloc **`finally`**
- Les ressources à fermer :
  - **`Connection`** (le plus important)
  - **`Statement`** (et sous-interfaces)

Transactions et exceptions

Exception ⇒ Rollback ?

- Non
- Le code Java doit explicitement appeler **`rollback()`** si le problème n'est pas récupérable par le programme

Quelques règles usuelles

- Les exceptions non contrôlées correspondent souvent à des erreurs non récupérables
- En ce cas le **`catch`** lance un *rollback*
- Les exceptions contrôlées sont parfois récupérables
- Par exemple, si la clé existe déjà, on peut changer de clé, ou faire un update à la place d'un insert

Exemple avec rollback

```
try {
  ...
  conn.commit();
}
catch(SQLException e) {
  // Impossible de corriger le problème
  conn.rollback();
}
```


Exemple de récupération

```
try {
  ...
  conn.commit();
}
catch(SQLException e) {
  // Situation récupérable
  // Correction du problème
  . . .
  conn.commit();
}
```


Fermeture de connexion et transactions

- Si une transaction est en cours lors de la fermeture d'une connexion, le comportement dépend du SGBD et du driver JDBC
- Il peut y avoir un commit automatique, ou un rollback, ou un autre comportement
- Il faut donc absolument terminer explicitement une transaction par un commit ou un rollback

Quel est le problème principal si on ne termine pas une transaction ?

Schéma global de code

```
try {
  ...
  conn.commit();
}
catch(SQLException e) {
  // Traiter l'exception ;
  // conn.commit() ou conn.rollback() selon les cas
  ...
}
finally {
  // Fermeture des ressources plus utilisées par la
  // suite (stmt.close(), conn.close(),...)
  ...
}
```

Adapter si c'est une méthode appelante qui gère la transaction ou la connexion

Fermeture automatique des connexions

- Le JDK 7 permet de fermer automatiquement à la sortie du bloc **`try`** des ressources indiquées au début du bloc **`try`**
- **`Connection`** et **`Statement`** implémentent **`AutoCloseable`**

Code schématique global

```
try (Connection c =
        DriverManager.getConnection(url,"toto","mdp");) {
  // Transaction 1
  try (Statement stmt = c.createStatement();) {
    ...
    c.commit();
  }
  catch(SQLException e) {
    // Traiter exception et c.commit() ou c.rollback()
    ...
  }
  // Autres transactions éventuelles (semblables à la 1ère)
  ...
}
```