Génie Logiciel Avancé : JDBC (2/2)

Arnaud Labourel (arnaud.labourel@univ-amu.fr)

22 janvier 2025

amU Faculté des sciences Aix Marseille Université

Section 1

Gestion des Exceptions

Problème `Exception` pour lecture de fichier

```
String readFirstLine(String path) throws IOException {
  FileReader fr = new FileReader(path);
  BufferedReader br = new BufferedReader(fr);
  try {
    return br.readLine();
  } finally {
    br.close();
    fr.close();
  }
}
```

Si `br.close()` lève une `Exception` alors on sort de l'exécution et `fr.close()` n'est pas appelé (fuite de ressources).

Solution `try` with ressources

```
static String readFirstLineFromFileWithFinallyBlock(String path)
    throws IOException {
  try(FileReader fr = new FileReader(path);
      BufferedReader br = new BufferedReader(fr)) {
    return br.readLine();
  }
}
```

`close()` est automatiquement appelé pour tous les objets instanciés entre les parenthèses.

Syntaxe `try` *with resources*

```
try (AutoClosableType1 r1 = new  AutoClosableType1();
     AutoClosableType2 r2 = new  AutoClosableType2()){
/* use r1 and r2 in the block
   At the end of the execution (exception or end of the block)
   r2 then r1 are closed */
}
```

AutoClosableType1 et AutoClosableType2 doivent implémenter AutoCloseable

```
interface AutoCloseable {
  /**
   * Closes this resource, relinquishing any underlying resources.
   */
  void close()
}
```

Exemple try with ressources SQL

```
public static void viewTable(Connection con) throws SQLException {
  String query = "select COF_NAME, PRICE from COFFEES";
  try (Statement stmt = con.createStatement()) {
    ResultSet rs = stmt.executeQuery(query);
    while (rs.next()) {
      String coffeeName = rs.getString("COF_NAME");
      float price = rs.getFloat("PRICE");
      System.out.println(coffeeName + ", " + price);
    }
  } catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
  }
}
```

Exemple 2 try with ressources SQL

```
String sql = "SELECT id, username FROM users WHERE id = ?";
List<User> users = new ArrayList<>();
try (Connection con = DriverManager.getConnection(myConnectionURL);
     PreparedStatement ps = con.prepareStatement(sql)) {
    ps.setInt(1, userId);
    try (ResultSet rs = ps.executeQuery()) {
    while(rs.next()) {
      users.add(new User(rs.getInt("id"), rs.getString("name")));
    }
  }
} catch (SQLException e) {
  e.printStackTrace();
}
```

Gestion des transactions

De base, tous les `Statement` d'une `Connection` sont appliqués à la base dès leur exécution.

Pour ne pas exécuter directement les `Statement` il faut appeler :

```
connexion.setAutoCommit(false);
```

`connexion.commit()` : applique les `Statements` de `connection` pas encore appliqués `connexion.rollback()` : annule les opérations du dernier commit

Section 2

Utilisation avancée de JDBC

Récupérer les valeurs des clés auto-générées (1/2)

Dans la quasi-totalité des tables, on a une ou plusieurs clés auto-générées.

Par exemple, dans notre application de calendrier, chaque créneau a un identifiant unique.

Cet identifiant doit être donné par la base de données.

Comment le récupérer ?

Constante pour spécifier le retour des clés

Constante de type int
`Statement.RETURN_GENERATED_KEYS`
à utiliser comme deuxième argument dans :

```
Connection.prepareStatement(String sql, int autoGeneratedKeys)
statement.execute(String sql, int autoGeneratedKeys)
```

Récupérer les valeurs des clés auto-générées (2/2)

Une fois le `statement` exécuté, on peut récupérer les clés générées sous la forme d'un `ResultSet` :

```
ResultSet resultSet = statement.getGeneratedKeys();
```

Il faut ensuite parcourir le `resultSet` pour obtenir l'unique clé (dans le cas d'une insertion unique) :

```
resultSet.next();
int id = resultSet.getInt(1);
```

Exécuter des scripts SQL avec HyperSQL

Il est possible d'exécuter un fichier SQL avec HyperSQL

```
try(Connection connection = DriverManager
    .getConnection(db, user, password)) {
  SqlFile sf = new SqlFile(ClassLoader
    .getSystemResource("database_schema.sql"));
    /* le fichier doit se trouver dans le répertoire
        src/main/resources du projet */
  sf.setConnection(connection);
  sf.execute();
}
```

C'est particulièrement utile pour les scripts de création des tables.

Différents modes d'hyperSQL

URL d'une base de données HyperSQL `"jdbc:hsqldb:mode:chemin"`

3 protocoles

- `mem` base de données en mémoire : pas de persistance après l'arrêt de la JVM
- `file` stockage dans des fichiers
- `res` stockage dans une ressource `Java` (non-modifiable)

Chemins

- nom de la base pour la mémoire, mode `mem`
- chemin des fichiers pour un stockage, mode `file` ou `res`

Fichiers

Un accès à `"jdbc:hsqldb:file:calendar/db"` peut créer les fichiers suivants dans le répertoire `Calendar` du projet :

- `db.properties` : numéros de version
- `db.script` : définition des tables et des données
- `db.log` : changements récents
- `db.backup` : dernière version consistante des données
- `db.lck` : fichier verrou
- `db.data`
- `db.lobs`

Méta informations sur les ResultSet

```
ResultSet rs = st.executeQuery();
ResultSetMetaData m = rs.getMetaData();
```

Informations disponibles :

- Nombre de colonnes : `int getColumnCount()`
- Libellé d'une colonne : `String getColumnLabel(int column)`
- Nom de la table d'origine d'une colonne : `String getTableName(int column)`
- Type associé à une colonne : `String getColumnTypeName(int column)`
- ...

Avantages

Code indépendant de la requête $\Rightarrow$ code réutilisable !

Méta informations sur la BD

```
DataBaseMetaData dbmd = connexion.getMetaData();
```

Informations disponibles :

- tables existantes dans la base : `ResultSet getTables(...)`
- nom d'utilisateur : `String getUserName()`
- version du pilote : `String getDriverVersion()`
- prise en charge des jointures externes : `boolean supportsOuterJoins()`
- ...

`ResultSet` depuis JDBC 2.0

Il existe quatre types de `ResultSet`:

- Réglage de la connexion à la base de données :
  - *Scroll-insensitive* : vision figée du résultat de la requête au moment de son évaluation (JDBC 1.0).
  - *Scroll-sensitive* : le `ResultSet` montre l'état courant des données (modifiées/détruites).
- Réglage des mises à jour :
  - *Read-only*: pas de modification possible (JDBC 1.0) donc un haut niveau de concurrence.
  - *Updatable*: possibilité de modification donc pose de verrou et faible niveau de concurrence.

Utilisation de `ResultSet` *Updatable*

```
Statement stmt = con.createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT * FROM Personne");
```

Ce `ResultSet` est modifiable, mais il ne reflète pas les modifications faites par d'autres transactions.

Méthode `updateXXX(col, newValue)` pour modifier des

Déplacement dans un `ResultSet`

- `boolean first()` : déplace le curseur à première ligne
- `void beforeFirst()` : déplace le curseur avant la première ligne
- `boolean next()` : déplace le curseur à la ligne suivante
- `boolean previous()` : déplace le curseur à la ligne précédente
- `void afterLast()` : déplace le curseur après la dernière ligne
- `boolean absolute(n)` : déplace le curseur à la ligne `n`
- `boolean relative(n)` : décale le curseur de `n` lignes

Modification d'un ResultSet (1/2)

```
Statement statement = connection.createStatement(/* ... */)
statement.execute("SELECT Nom, Age FROM users");
ResultSet resultSet = statement.getResultSet()
```

Mise à jour d'une ligne

```
resultSet.absolute(100); // déplacement
resultSet.updateString("Nom", "Fred");
resultSet.updateInt("Age", 30);
resultSet.updateRow();
```

Suppression d'une ligne

```
resultSet.deleteRow();
```

Modification d'un ResultSet (2/2)

```
Statement statement = connection.createStatement(/* ... */)
statement.execute("SELECT Nom, Age FROM users");
ResultSet resultSet = statement.getResultSet()
```

Insertion d'une ligne

```
resultSet.moveToInsertRow();
resultSet.updateString("Nom", "Fred");
resultSet.updateInt("Age", 30);
resultSet.insertRow();
resultSet.first();
```

Batch updates

Regroupement de plusieurs mise à jour

```
connexion.setAutoCommit(false);
Statement st = connexion.createStatement();

st.addBatch("INSERT ...");
st.addBatch("INSERT ...");

int[] nb = st.executeBatch();
```

On peut combiner des `PreparedStatement` et des « Batch updates ».

JDBC 3.0

- nouveau package javax.sql.*
- Save point: pose de point de sauvegarde.
- Connection Pool: Gestion des ensembles de connexions partagées.
- Support des séquences (auto génération de valeurs).
- Augmentation et mise à jour des types :
  - CLOB : Character Large OBject (texte long stocké via une référence)
  - BLOB : Binary Large OBject (données binaires via une référence)
- Prise en compte de SQL-3.

Exemple d'utilisation de point de sauvegarde

```
public void modifyPricesByPercentage(String coffeeName,
                                     float priceModifier,
                                     float maximumPrice)
                                      throws SQLException {
  con.setAutoCommit(false);
  ResultSet rs = null;
  String priceQuery = "SELECT COF_NAME, PRICE FROM COFFEES " +
                      "WHERE COF_NAME = ?";
  String updateQuery = "UPDATE COFFEES SET PRICE = ? " +
                       "WHERE COF_NAME = ?";
  // suite au prochain transparent
```

Exemple d'utilisation de point de sauvegarde

```
public void update(/* ... */) throws SQLException {
  con.setAutoCommit(false);
  ResultSet rs = null;
  try (PreparedStatement getPrice =
        con.prepareStatement(priceQuery,
          ResultSet.TYPE_SCROLL_INSENSITIVE,
          ResultSet.CONCUR_READ_ONLY);
       PreparedStatement updatePrice =
         con.prepareStatement(updateQuery)){
            // suite au prochain transparent
         }
```

Exemple d'utilisation de point de sauvegarde

```
try(/* ... */){
  Savepoint save1 = con.setSavepoint();
      getPrice.setString(1, coffeeName);
      getPrice.execute();
      rs = getPrice.getResultSet();
      rs.first();
      float oldPrice = rs.getFloat("PRICE");
      float newPrice = oldPrice + (oldPrice * priceModifier);
      updatePrice.setFloat(1, newPrice);
      updatePrice.setString(2, coffeeName);
      updatePrice.executeUpdate();
      // suite au prochain transparent
```

Exemple d'utilisation de point de sauvegarde

```
    CoffeesTable.viewTable(con);
    if (newPrice > maximumPrice) {
      con.rollback(save1);
    }
    con.commit();
  } catch (SQLException e) {
    JDBCTutorialUtilities.printSQLException(e);
  } finally {
    con.setAutoCommit(true);
  }
}
```

Les DataSource

L'interface `javax.sql.DataSource` permet :

- d'obtenir une connexion JDBC,
- de gérer un pool de connexion,
- de faire disparaître les constantes (placées dans un annuaire JNDI ou un fichier de configuration).

Les RowSet (1/2)

```
javax.sql.rowset.RowSetFactory factory = RowSetProvider.newFactory();
javax.sql.rowset.RowSet rs = factory.createCachedRowSet();

rs.setUrl("jdbc:mysql://localhost/dbessai");
rs.setCommand("SELECT * FROM person");
rs.setUsername("alaboure");
rs.setPassword("...");
rs.setConcurrency(ResultSet.CONCUR_UPDATABLE);
rs.execute();

while (rs.next()) {
    System.out.printf("Nom : %s\n", rs.getString("nom"));
}
rs.close();
```

Les RowSet (2/2)

Il existe trois types de RowSet:

- JDBCRowSet (basé sur JDBC),
- CachedRowSet (déconnecté de la base),
- WebRowSet (échange basé sur des flux XML),