Documentation
Gestion de la persistance des données : JDBC

JDBC 3.0 (Java DataBase Connectivity 3.0) est une API Java 2 de Sun qui permet à une application (ou une applet) de se connecter et d'interagir avec une base de données relationnelle (ou objet-relationnelle) locale ou distante à l'aide de commandes SQL.
A l'origine JDBC était contenu dans JSE (Java Standard Edition). Depuis sa version 2, JDBC est maintenant contenu dans JEE (Java Entreprise Edition).

> D'importants fournisseurs de bases de données se sont regroupés pour créer une solution alternative à JDBC appelée SQLJ. SQLJ est une spécification pour l'écriture de SQL embarqué dans des applications Java qu'un préprocesseur peut lire et convertir en appels JDBC. Il est important de noter que SQLJ n'est pas une norme Java approuvée par l'OMG, mais une autre solution basée sur des formes anciennes et démodées d'accès aux bases de données. Le paradigme de SQLJ est familier aux programmeurs C et COBOL, mais très peu adapté à la nature orientée objets de Java.

JDBC doit charger un driver spécifique au SGBDR auquel on souhaite se connecter. Il existe quatre types de drivers (d'après une classification de Sun) :

- **Type 1** : *Ponts ODBC-JDBC*
  Le driver présent dans J2E accède à un SGBDR en passant par les drivers ODBC de Microsoft. Ce type de solution est utile lorsqu'on souhaite se connecter à BD du monde Microsoft.
- **Type 2** : *Driver d'API native*
  Fait appel à des fonctions natives (non Java et généralement en C) de l'API du SGBDR
- **Type 3** : *Driver 100% Java*
  Interagit avec une API réseau générique et communique avec une application intermédiaire (middleware) sur le serveur.
- **Type 4** : *Driver 100% Java utilisant le protocole réseau du SGBDR*
  Interagit directement avec la base de données via des sockets. Ces drivers sont généralement fournis par l'éditeur du SGBDR.

Une application Java peut utiliser les 4 types de drivers. Les drivers de type 1 et 2 sont censés être plus rapides (quoique Oracle assure que son driver *"thin"* de type 4 est, dans la plupart des cas, aussi performant que son driver *"OCI"* de type 2) mais ils ne peuvent pas être utilisés avec une Applet (car ils ne sont pas écrits en Java). Par contre, les drivers de type 3 et 4 qui sont "pur java" et donc plus portables le peuvent. Toutefois, même avec ces types de drivers, on peut être confronté à des difficultés pour se connecter à une base de données à cause de l'environnement d'exécution des applets (que l'on appelle bac à sable ou sandbox) qui, pour des raisons de sécurité, réduit volontairement l'accès aux ressources externes.

Pour contourner ces problèmes de sécurité, une alternative peut être l'utilisation des Servlets. Une autre serait d'utiliser un serveur d'application et de le solliciter grâce à des appels de procédures distantes (RPC). Si ce serveur d'application est écrit en Java, ces appels de procédures distantes peuvent être implémentés avec RMI. Dans le cas contraire on peut utiliser des technologies comme Corba, XML-RPC ou encore les Services Web.

Nous allons commencer par voir comment utiliser JDBC dans une application Java basique. Puis nous verrons comment utiliser des curseurs sophistiqués (navigable ou/et modifiables). Enfin, nous terminerons par des concepts avancés comme les requêtes préparées, les appels à des procédures stockées, les transactions ou l'utilisation des méta-données.

1 Utilisation basique de JDBC

1.1 Les différentes étapes pour utiliser JDBC

L'utilisation de JDBC se fait en 7 étapes :

- 1 importer le package `java.sql`
- 2 enregistrer le driver JDBC
- 3 établir une connexion à la base de données
- 4 créer une zone de description de la requête
- 5 exécuter la requête
- 6 traiter les données retournées

- 7 fermer les différents espaces

1.1.1 Importer le package `java.sql`

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

1.1.2 Enregistrer le driver JDBC adéquat

Méthode `forName()` de la classe `Class` : `Class.forName(nom de la classe du driver);`

Exemples :

```
Class.forName("oracle.jdbc.driver.OracleDriver");     // Oracle
Class.forName("org.postgresql.Driver");               // Postgres
Class.forName("com.mysql.jdbc.Driver");               // MySQL
Class.forName("org.sqlite.JDBC");                     // Sqlite
```

Attention : cette instruction est susceptible de déclencher une exception de type `ClassNotFoundException`

Attention : Dans Eclipse, pour pouvoir enregistrer un driver, il faut préalablement ajouter dans votre projet les fichiers ".jar" externes qui correspondent au driver en question. Par exemple, avec le JDK 1.1, pour utiliser le driver d'Oracle il faut ajouter le fichier classes111.jar (classes12.jar pour le JDK 1.3 ou encore, ojdbc14.jar pour le 1.4, ...)

sous Eclipse : clic droit de la souris sur le projet puis sélectionner – Properties – Java Build Path – Libraries – Add External JARs

1.1.3 Etablir une connexion à la base de données

```
Connection cn = DriverManager.getConnection(url,login,mdp);
```

url correspond à l'adresse du SGBDR où l'on souhaite se connecter
cette *url* est de la forme jdbc:sgbd:adresse
par exemple pour se connecter au serveur Oracle de l'IUT il faudra indiquer

```
"jdbc:oracle:thin:@gloin:1521:iut"
```

SGBD protocole machine n° de port base de données

Pour se connecter à Oracle depuis l'extérieur : `"jdbc:oracle:thin:@162.38.222.149:1521:iut"`
Pour se connecter à une BD MySQL locale on aura : `"jdbc:mysql://localhost/maBD"`

1.1.4 Créer une zone de description de la requête

A partir de l'instance de `Connection` (ici `cn`) on récupère le `Statement` associé.

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

☞ Comme nous le verrons au paragraphe §2, l'utilisation de la méthode `createStatement()` sans paramètre va donner plus tard des curseurs statiques qui seront ni navigables ni modifiables.
En effet, `st = cn.createStatement()` est équivalent à :

```
st=cn.createStatement(ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY);
```

1.1.5 Exécuter la requête

Il existe 2 méthodes qui permettent d'exécuter une requête sur un `Statement`.

- `executeQuery()` : pour les requêtes `SELECT` et qui retourne un `ResultSet` (c'est à dire un curseur)
- `executeUpdate()` : pour les requêtes de mise à jour `(INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, …)` et qui retourne un entier correspondant au nombre de tuples modifiés par la requête.

Exemples :

```
int nbInsertions;
nbInsertions = st.executeUpdate("INSERT INTO Livres (numISBN, titre, qteStock)
                                          VALUES ('1234567890', 'UML', 10)") ;
int nbSuppressions;
nbSuppressions = st.executeUpdate("DELETE FROM Livres
                                  WHERE titre LIKE '%Pokemon%'") ;
```

```
ResultSet rs;
rs = st.executeQuery("SELECT numISBN, titre, qteStock FROM Livres");
```

1.1.6 Traiter les données retournées dans le curseur

L'objet `ResultSet` permet d'accéder aux tuples qui composent le résultat de l'exécution d'une requête `executeQuery()`. On peut parcourir séquentiellement ce `ResultSet` avec la méthode `next()` et récupérer les valeurs des colonnes des tuples de l'élément courant avec la méthode `getXXX(int columnIndex)` ou `getXXX(String columnName)`

Exemple :

```
String num, titre ;
int qte;
while (rs.next()) {
   num = rs.getString(1);      // ou num = rs.getString("numISBN");
   titre = rs.getString(2);    // ou titre = rs.getString("titre");
   qte = rs.getInt(3);         // ou qte = rs.getInt("qteStock");
   System.out.println(num + " " + titre + " " + qte);
}
```

→ méthode next():boolean

- la méthode `next()` retourne `false` si on est sur le dernier tuple du `ResultSet`, `true` sinon
- chaque appel fait avancer le l'élément courant du curseur sur le tuple suivant
- initialement l'élément courant du curseur est positionné avant le premier tuple (`beforeFirst`)

→ méthode getXXX(int columnIndex):XXX

`getXXX(int columnIndex)` permet de récupérer la valeur d'une colonne de type `XXX` sur le tuple de l'élément courant du `ResultSet`. Cette méthode est équivalente à la méthode `getXXX(String columnName)`

`XXX` correspond au nom du type Java correspondant au type SQL / JDBC attendu.

Type SQL / JDBC	Type Java correspondant	Méthode getXXX()
CHAR, VARCHAR, LONGVARCHAR	String	getString()
NUMERIC, DECIMAL	java.math.BigDecimal	getBigDecimal()
BIT	boolean	getBoolean()
INTEGER	int	getInt()
BIGINT	long	getLong()
REAL	float	getFloat()
DOUBLE, FLOAT	double	getDouble()
DATE	java.sql.Date	getDate()

1.1.7 Fermer les différents espaces

Pour terminer proprement un traitement il faut fermer les différents espaces ouverts (les `Connection`, les `Statement` et les `ResultSet`) avec les méthodes `close()`.

```
rs.close();
st.close();
cn.close();
```

Il est à noter que si on ferme un `Statement`, les `ResultSet` qui lui sont associés sont fermés automatiquement (et il ne sera plus possible de parcourir les `ResultSet`). De même, si on ferme une `Connection`, les `Statement` qui lui sont associés sont également fermés automatiquement. Ainsi, fermer la `Connection` permet de fermer automatiquement tous les `Statement` et les `ResultSet`. Toutefois, il est conseillé de fermer un à un tous les objets.

1.2 JDBC pour les nulls

Comme toujours, la gestion des `NULL` pose des problèmes ; notamment pour les colonnes qui vont être transcrites dans l'application Java par une variable de type primitif. Cela va généralement être le cas avec les types primitifs numériques (`int`, `float`, …). En effet, en utilisant la méthode `getInt()`, un `ResultSet` Java est incapable de retourner dans un type primitif `int` une valeur équivalente au `NULL` de SQL, et va retourner la valeur par défaut 0. Cela peut être une source de confusion et d'erreurs. Pour remédier à cela, il suffit d'appeler la méthode `wasNull()` de `ResultSet`. Cette méthode renvoie `true` si la dernière valeur qui a été lue par une méthode `getXXX()` était un `NULL` de SQL ; `false` sinon.

Exemple :
```
int qte = rs.getInt(3);
if (rs.wasNull()) {
      System.out.println("le stock est inconnu");
      qte = -99 ;
    }
```

1.3 JDBC et les exceptions

Toutes les instructions vues précédemment sont susceptibles de déclencher des Exceptions SQL.
Il existe deux types d'exceptions SQL :

- les SQLException levées dès qu'une connexion ou un ordre SQL ne se passe pas correctement.
- les SQLWarning : avertissement SQL. Cette classe étend SQLException mais l'erreur n'est pas fatale ; l'exception n'est pas levée mais stockée. La liste des avertissements est disponible grâce à la méthode `getWarning()`

Exemple :
```
   try(
     //Création et ouverture des ressources nécessaires (connexion,
statement, result set…). Les placer dans cette partie du bloc catch permet
d'automatiquement les fermer  à la fin du traitement.
      ...
      ResultSet rs = st.executeQuery("SELECT numISBN, titre, qteStock FROM
Livres");
      ...

   ) { // dans le try toutes les autres instructions JDBC
      ...
      if (rs.getWarning() != null) System.out.println(rs.getWarning().getMessage()) ;
      ...
   }
   catch(SQLException e) {
      System.out.println(e.getMessage());
      e.printStackTrace();
   }
```

- Normalement, dans une application objet structurée, l'ouverture et la fermeture de la connexion ne devrait pas se faire dans la même méthode ; et on ne devrait donc pas avoir besoin du bloc finally.

2 Curseurs navigables et modifiables

Les `ResultSet` que nous avons vus jusqu'à présent sont des curseurs statiques. Ils ne sont pas navigables (c'est-à-dire qu'on peut uniquement les parcourir séquentiellement avec des `next()`, il n'est pas possible par exemple de les parcourir en arrière) et ne sont pas non plus modifiables (c'est-à-dire qu'ils sont uniquement en lecture , il n'est pas possible de modifier la base de données à travers ces curseurs).

Si on le souhaite, on peut également manipuler des curseurs navigables, modifiables, ou les deux à la fois.

2.1 Curseurs navigables

Depuis JDBC 2.0, il est possible de manipuler des curseurs navigables qui peuvent être parcourus en avant ou en arrière, en se déplaçant de manière absolue ou relative. Pour pouvoir manipuler ces curseurs navigables, il faut lors de la création du `Statement`, ne pas utiliser la méthode `createStatement()` sans paramètre. Il faut donc plutôt écrire :

```
st=cn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,ResultSet.CONCUR_READ_ONLY);
```

ou encore

```
st=cn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
```

- `SENSITIVE` ou `INSENSITIVE` indique si le ResultSet est sensible aux modifications de la base de données.
- `CONCUR_READ_ONLY` indique qu'on aura des curseurs non modifiables. Pour utiliser des curseurs modifiables, voir le paragraphe §2.2
- attention, cette instruction ne fonctionne pas avec un driver se connectant à une base de données Sqlite !

En effet comme indiqué précédemment (§1.1.4), l'utilisation de `createStatement()` sans paramètre `st = cn.createStatement()` est équivalent à :

```
st=cn.createStatement(ResultSet.TYPE_FORWARD_ONLY,ResultSet.CONCUR_READ_ONLY);
```

Une fois cela fait, les `ResultSet` qui dépendent du `Statement` pourront bénéficier des méthodes suivantes (en plus des méthodes `next()` et `getXXX()` vues précédemment :

```
boolean previous()                    void beforeFirst()
boolean absolute(int row)             boolean isFirst()
boolean first()                       boolean isLast()
boolean last()                        boolean isBeforeFirst()
boolean relative(int row)             boolean isAfterLast()
void afterLast()                      int getRow()
```

2.2 Curseurs modifiables

Lorsqu'on manipule un curseur modifiable, il est possible de réaliser des mises à jour (`UPDATE`, `DELETE`, `INSERT`) dans la table référencée par le curseur, en modifiant les lignes du `ResultSet`. Pour pouvoir manipuler ces curseurs modifiables, il faut lors de la création du `Statement`, ne pas utiliser la méthode `createStatement()` sans paramètre et plutôt écrire :

```
st=cn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

- `TYPE_SCROLL_SENSITIVE` indique que le curseur est navigable et sensible aux modifications qui sont apportées dans la table. Si on indique `TYPE_FORWARD_ONLY` on a alors un curseur non navigable (c.f. §2.1)
- attention, cette instruction ne fonctionne pas avec un driver se connectant à une base de données Sqlite !

2.2.1 Création d'un curseur modifiable

La création d'un `ResultSet` modifiable se fait grâce à la méthode `executeQuery()`, tout comme pour un `ResultSet` statique.

Exemple :

```
ResultSet rs = st.executeQuery("SELECT numISBN, titre, qteStock FROM Livres");
```

Pour que le `ResultSet` soit modifiable, il y a quand même quelques restrictions :

- la requête SQL ne doit porter que sur une seule table ; il ne doit pas y avoir de jointure
- **si on souhaite récupérer toutes les colonnes d'une table, il ne faut pas indiquer `SELECT *` mais `SELECT aliasTable.*`**. Par exemple :
  `ResultSet rs = st.executeQuery("SELECT l.* FROM Livres l");`
- la requête ne doit pas avoir de fonction (`AVG, MIN, MAX, SUM`, etc.) dans le `SELECT`.
- elle ne doit pas utiliser les clauses `GROUP BY, HAVING`, ou `CONNECT BY`.
- lorsqu'on souhaite insérer des tuples dans une table à travers un curseur, il faut que la clé primaire de la table soit incluse dans le curseur (ainsi que les attributs `NOT NULL`)

2.2.2 Suppression de données à travers un curseur modifiable

Pour supprimer une ligne de la table à travers le curseur, il faut placer l'élément courant du curseur sur le tuple qu'on souhaite supprimer puis appeler la méthode `deleteRow()`.

```
if (rs.next())          // on se place sur la première ligne du curseur
   rs.deleteRow();      // on supprime la ligne dans la table et le curseur
```

- il est à noter que le tuple va disparaître dans la table *Livres* ainsi que dans le curseur `rs`.

2.2.3 Modification de données à travers un curseur modifiable

Pour modifier une ligne de la table à travers le curseur, il faut placer l'élément courant du curseur sur le tuple qu'on souhaite mettre à jour, puis modifier les valeurs des colonnes de ce tuple avec les méthodes `updateXXX(int, XXX)`, puis appeler la méthode `updateRow()`.

```
if (rs.next()) {        // on se place sur la première ligne du curseur
   rs.updateString(2, "JDBC pour les gens bons");
   rs.updateInt(3,10);  // on modifie la valeur des colonnes
   rs.updateRow();      // on propage les modifications dans la table
}
```

- il est à noter que le tuple va être modifié dans la table *Livres* mais aussi dans le curseur `rs`.

2.2.4 Insertions de données à travers un curseur modifiable

Pour insérer une nouvelle ligne dans la table à travers le curseur, il faut placer l'élément courant du curseur sur une nouvelle ligne avec la méthode `moveToInsertRow()`, puis donner des valeurs aux colonnes avec les méthodes `updateXXX()`, puis appeler `insertRow()`.

```
rs.moveToInsertRow();   // on place l'enregistrement courant sur une nouvelle ligne
rs.updateString(1, "0246813579");
rs.updateString(2, "JDBC pour l'effort");
rs.updateInt(3,20);     // on donne des valeurs aux colonnes de cette ligne
rs.insertRow();         // on propage les modifications dans la table
```

- **lors de l'insertion, le tuple va être ajouté dans la table *Livres* mais pas dans le curseur `rs` !** (même si le curseur est déclaré sensitif avec `TYPE_SCROLL_SENSITIVE`).
  Pour obtenir les lignes insérées dans un curseur, il faut faire une nouvelle requête `SELECT`.
- une fois la ligne insérée, il est possible de revenir à l'enregistrement courant avec la méthode `moveToCurrentRow()`

3 JDBC avancé

3.1 Requêtes pré-compilées

L'interface `PreparedStatement` étend l'interface `Statement`. Un objet `PreparedStatement` envoie une requête paramétrée à la base de données (mais sans renseigner la valeur des paramètres) pour une pré-compilation et spécifiera le moment voulu la valeur des paramètres.
L'utilisation des requêtes **pré-compilées améliore les performances** notamment lorsqu'on souhaite lancer plusieurs fois la même requête (avec des paramètres différents). De plus, elle permet d'**éviter l'injection de code SQL**.

Exemple :

```
PreparedStatement pst = cn.prepareStatement("SELECT * FROM Livres
                                             WHERE titre = ?
                                             AND qteStock = ?");
...
pst.setString(1, "eXtreme Programming pour Windows XP");   // valeur du premier "?"
pst.setInt(2, 0);                                          // valeur du second "?"
rs = pst.executeQuery();
...
```

3.2 Appel à des procédures stockées

L'interface `CallableStatement` étend l'interface `PreparedStatement`. Un objet `CallableStatement` permet d'exécuter une procédure stockée déjà écrite et pré-compilée sur le SGBDR.

Exemple :

Soit la procédure suivante écrite en PL/SQL sur le SGBDR Oracle :

```
CREATE OR REPLACE PROCEDURE insere (p_num IN Livres.numIsbn%TYPE,
                                    p_titre IN Livres.titre%TYPE,
                                    p_qte IN Livres.qteStock%TYPE) IS
BEGIN
   INSERT INTO Livres (numISBN, titre, qteStock) VALUES(p_num, p_titre, p_qte);
END ;
```

Il suffira d'écrire dans le code Java

```
CallableStatement cst = cn.prepareCall("{call insere(?,?,?)}");
…
cst.setString(1,"1357902468");           // valeur de 1357902468 dans le premier "?"
cst.setString(2, "JDBC pour les mazettes");    // "JDBC …" dans le second "?"
cst.setInt(3, 10);                       // 10 dans le troisième "?"
cst.execute();
```

3.3 Les transactions

Comme nous l'avons vu dans le dernier fascicule du dossier II, les transactions permettent de rendre un ensemble d'instructions atomique. Elles doivent être utilisées lorsqu'on souhaite qu'un ensemble de requêtes soient exécutées de façon indivisible. Une transaction peut être :

- validée (`commit`)
- ou annulée (`rollback`)

Dans JDBC, par défaut, toutes les instructions d'une `Connection` qui modifient les données de la base sont commitées automatiquement (`auto-commit`). Si on veut empêcher cela, il faut indiquer explicitement que la `Connection` n'est pas `auto-commit`. Cela peut être fait avec la méthode `setAutoCommit()`.

Exemple

```
cn.setAutoCommit(false) ;     // demande de ne pas valider automatiquement
                              // toutes les modifications dans les tables
Statement st = cn.createStatement();
st.executeUpdate("INSERT INTO livres (numISBN, titre, qteStock)
                  VALUES ('8642097531', 'JDBC pour les balaises', 3)") ;
st.executeUpdate("INSERT INTO livres (numISBN, titre, qteStock)
                  VALUES ('9753108642', 'JDBC pour les nimois', 13)") ;

cn.commit(); // ou encore, cn.rollback() ;
```

3.4 Les méta-données

Les méta-données permettent d'obtenir des informations sur la structure des tables de la base de données. L'utilisation des méta-données s'avère indispensable lorsqu'on souhaite interroger une base de données dont on ne connaît rien. Grâce à elles, il n'est pas utile d'avoir recours aux tables (ou vues) Système qui ont le désavantage d'être propres au SGBD interrogé.

Il existe deux objets méta-données intéressants :

- l'objet `ResultSetMetaData` qui permet de trouver des informations sur la structure d'un `ResultSet` ; on peut alors répondre à des questions du type : combien y a-t-il de colonnes, les noms des colonnes sont-ils sensibles à la casse, une colonne peut elle avoir la valeur NULL, quel est le type de donnée d'une colonne, de quelle table provient une colonne, etc. …

  Exemple d'utilisation d'un `ResultSetMetaData` qui affiche dans la console les types des différentes colonnes d'un `ResultSet`

  ```
  ResultSet rs = st.executeQuery("SELECT * FROM Livres");
  …
  ResultSetMetaData rsmd = rs.getMetaData();
  int nbColonnes = rsmd.getColumnCount();

  String nomColonne;
  String typeColonne;
  int tailleColonne;
  boolean estClePrimaire;

  for (int i=1; i<=nbColonnes ; i++) {
     nomColonne = rsmd.getColumnName(i);
     typeColonne = rsmd.getColumnTypeName(i);
     tailleColonne = rsmd.getPrecision(i);
     estClePrimaire = (rsmd.isNullable(i) == ResultSetMetaData.columnNoNulls);
     System.out.println(nomColonne + " " + typeColonne + " " +
                        tailleColonne + " " + estClePrimaire);
  }
  ```

- l'objet `DatabaseMetaData` qui permet de trouver des informations sur la structure globale d'une base de données ; on peut alors répondre à des questions du type : quelles sont les tables de la BD visibles par l'utilisateur, quel est le nom de l'utilisateur utilisé par la connexion, les jointures externes sont-elles supportées, quelles sont les clés primaires d'une table, etc. …

  Exemple d'utilisation d'un `DatabaseMetaData` qui affiche dans la console les noms de toutes les tables de la base de données.

  ```
  DatabaseMetaData dbmd = cn.getMetaData();
  String tab[] = {"TABLE"};
  ```

```
rs = dbmd.getTables(" ", dbmd.getUserName(), null, tab);
while (rs.next())
   System.out.println(rs.getString("TABLE_NAME"));
```