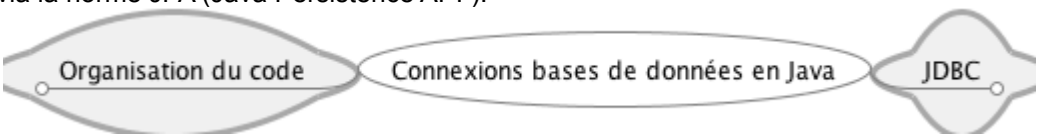
JDBC

Table des matières

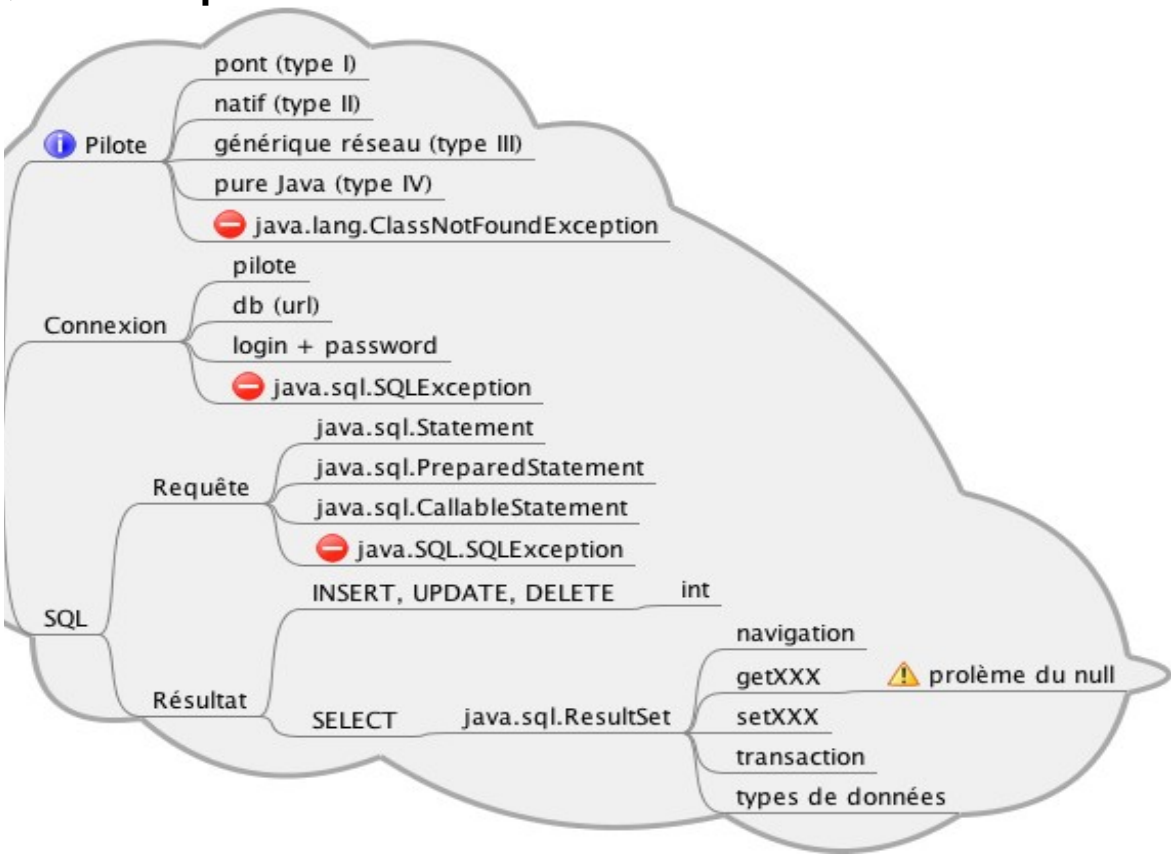
1 Introduction

L'objectif du cours est l'introduction à l'API JDBC: définition, fonctionnement. Nous verrons comment effectuer une connexion sur n'importe quelle système de base de données (si un pilote java existe) et y exécuter différentes requêtes SQL.

Dans ce cours, nous verrons aussi le design pattern fabrique abstraite pour introduire le pattern DAO (Data Access Object). Ce pattern permet de séparer la partie « persistance » (sauvegarde) des données du reste de l'application. Ce pattern nous permettra d'aborder les frameworks de type ORM (Object Relational Mapping) via la norme JPA (Java Persistence API[1]).



2 Qu'est-ce que JDBC?



JDBC est une API de niveau SQL qui permet de passer des instructions SQL à partir de Java. JDBC impose que les fournisseurs de base de données offrent une implémentation de ses interfaces pour que les requêtes qui viennent de java soient indépendantes de la base de données. Les implémentations de l'API JDBC (les pilotes ou drivers) routent les appels SQL à la base de données en des appels propriétaires qu'elle reconnait. Le développeur java ne se préoccupe plus du routage des instructions spécifiques de telle ou telle base de données. Sans JDBC, il faudrait réécrire toutes les classes qui font des appels à la base de données si on devait changer de fournisseur de base de données.

La façade fournie par JDBC libère le développeur java des problèmes liés à une base de données particulière; on peut exécuter le même code quelque soit la base de données en présence.

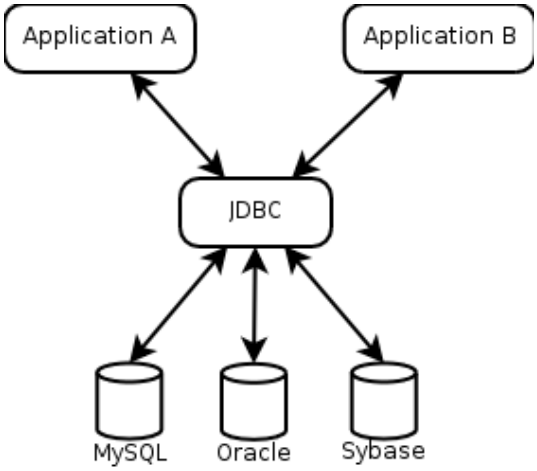
Pour être une API de niveau SQL, JDBC doit permettre de construire des instructions SQL et de les

1 http://java.sun.com/developer/technicalArticles/J2EE/jpa/

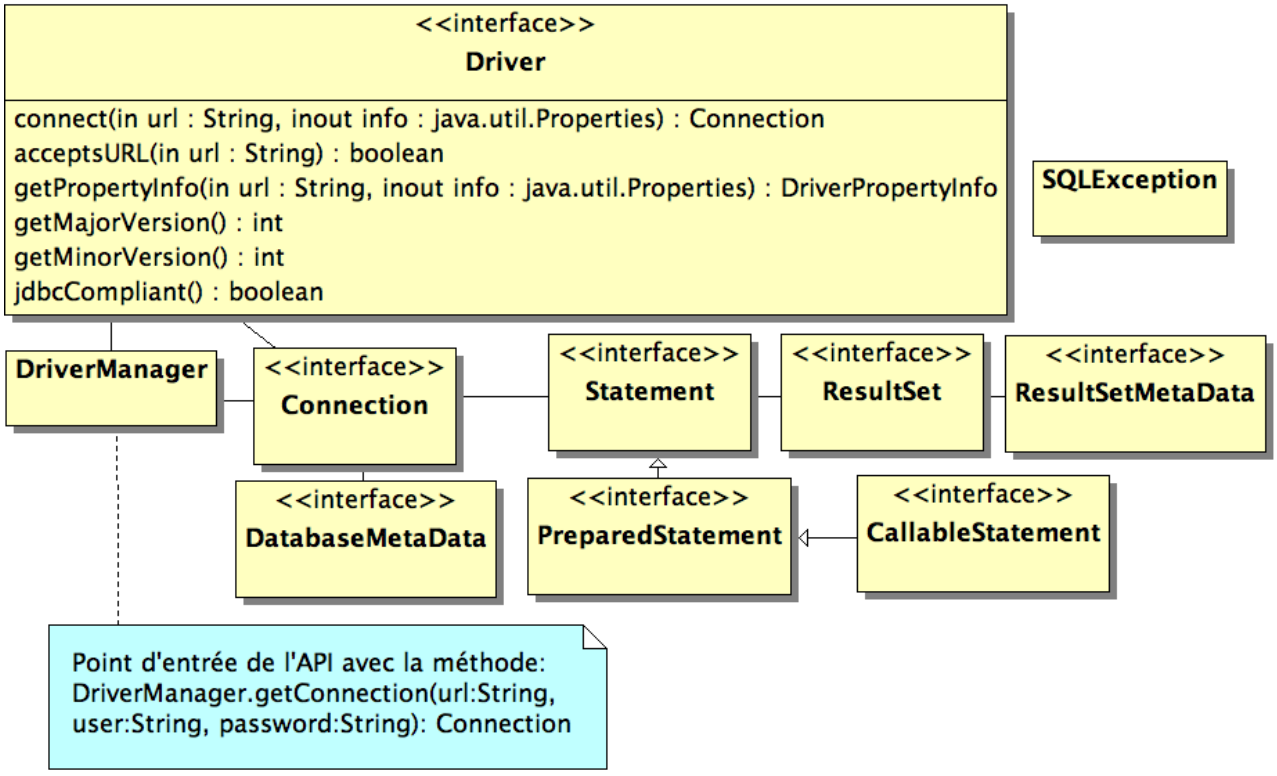
inclure dans les appels de l'API java. On utilise SQL, mais les réponses sont retournées en objets (ou types primitifs) java. Les problèmes soulèvent des exceptions.

JDBC va donc servir au programmeur java à se connecter à n'importe quelle base de données, à exécuter des requêtes SQL et à récupérer des résultats. Il s'agit d'une interface entre le langage Java et le SQL qui permet d'obtenir des informations sur la base de données, de la base de données. Il s'agit d'une abstraction par rapport au SGBDR semblable à l'ODBC du monde Windows.

3 Structure de JDBC

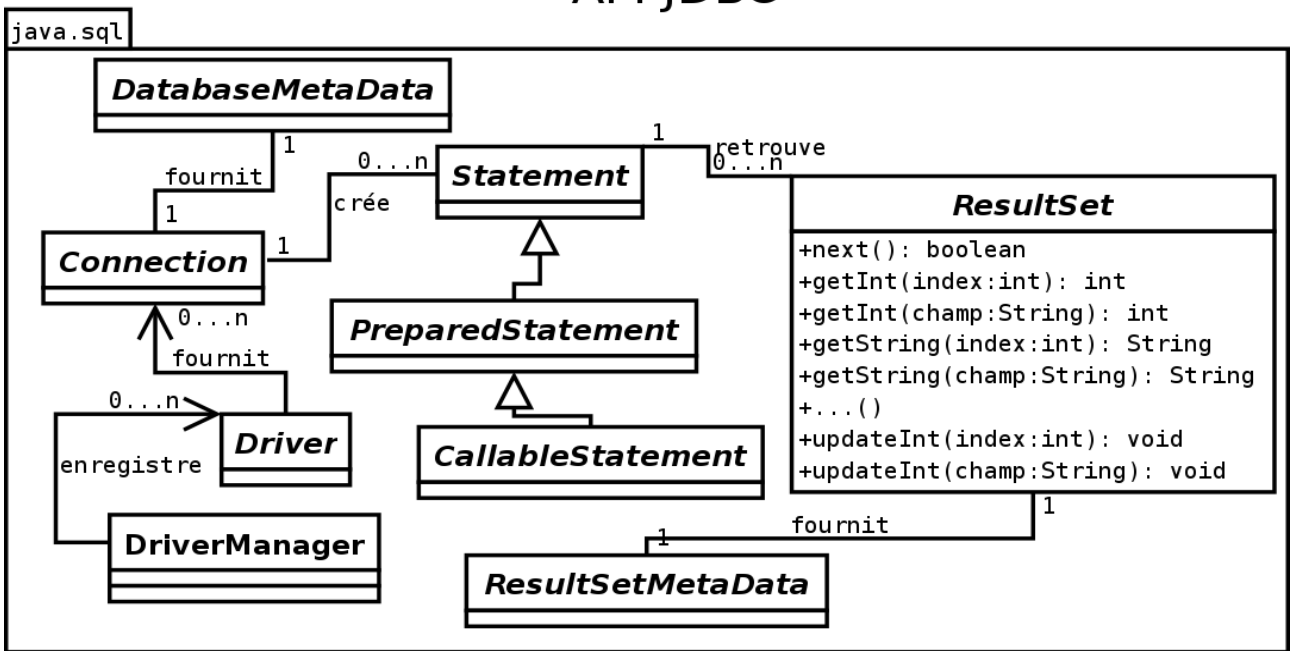


4 L'API[1]

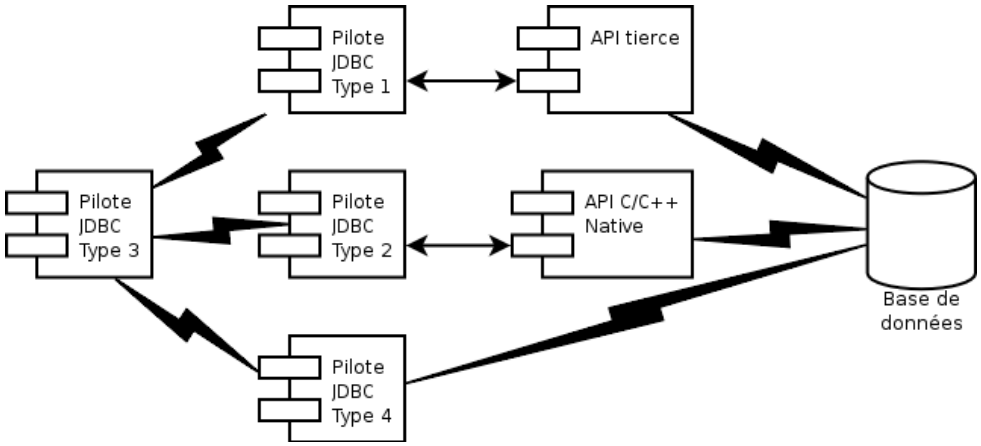


1 http://java.sun.com/javase/6/docs/api/java/sql/package-summary.html

API JDBC



5 Les pilotes JDBC



5.1 Type 1

Ces pilotes utilisent une technologie de pont pour accéder à la base de données. Exemple: JDBC-ODBC. Dans ce cas, il faut installer un logiciel sur le client.

5.2 Type 2

Pilotes natifs. Il contient du code java qui appelle des méthodes en C ou C++ implémentées par chaque fournisseur de la base de données et réalisant l'accès à la base. Dans ce cas, il faut installer un logiciel sur le client.

5.3 Type 3

Pilotes donnent au client une API réseau générique qui est ensuite traduite en accès spécifiques à la base de données sur le serveur. L'utilisation de socket pour appeler une application intermédiaire sur le serveur qui traduit les requêtes du client en une API spécifique au pilote voulu. Extrêmement souple, aucune installation sur le client et un seul pilote peut fournir l'accès à de multiples base de données.

5.4 Type 4

Ces pilotes utilisent les protocoles réseaux intégrés au moteur de la base de données. Ils conversent directement avec la base de données via sockets java. Il s'agit de la solution la plus directe. Les protocoles réseaux des base de données sont rarement documentés, ces pilotes proviennent souvent du fournisseur.

6 Fabrique Abstraite

L'API JDBC est conçue sous la forme d'une fabrique abstraite (cf Design Pattern Tête la première, page 109: Les fabriques; p156 AbstractFactory). JDBC fournit une interface pour créer des familles d'objets apparentés ou dépendants sans avoir à spécifier leurs classes concrètes (voir l'exemple des familles de Pizzas). Les pilotes doivent implémenter toutes les interfaces de JDBC pour s'intégrer dans l'API et permettre au développeur d'interroger la base de données pour laquelle le pilote a été conçu.

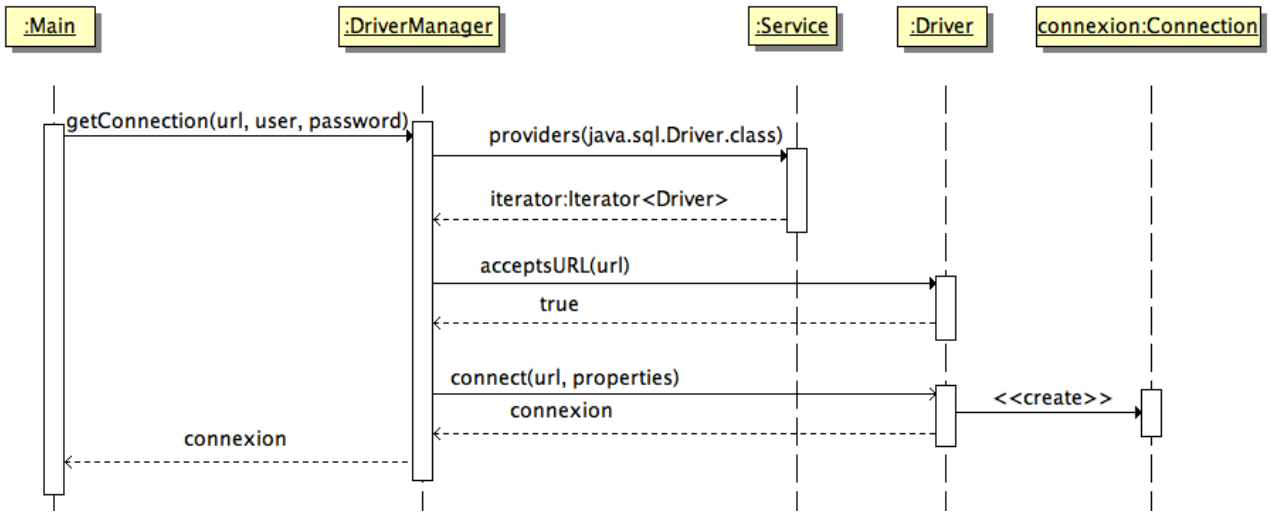
7 Connexion à une base de données

7.1 Déroulement

1. La seule information JDBC spécifique au pilote que nous devons donner est **l'URL de la base de données** et les propriétés **login** et **mot de passe** pour s'y connecter.
2. L'application va commencer par demander une implémentation de `java.sql.Connection` auprès du `java.sql.DriverManager`. Celui-ci va rechercher parmi toutes les implémentations connues de `java.sql.Driver` celle qui correspond à l'URL fournie. Si il n'en trouve aucune, il lance une exception.
3. Quand un driver (pilote) reconnaît l'URL, il crée une connexion à la base de données en utilisant les propriétés données. Il donne ensuite au `java.sql.DriverManager` une implémentation de `java.sql.Connection` représentant la connexion. Le `java.sql.DriverManager` retourne ensuite cet instance `java.sql.Connection` à l'application. En clair:
   `Connection conn = DriverManager.getConnection(url, login, pass);`

Comment le `DriverManager` connaît-il les pilotes?

Si le pilote est compatible JDBC 4.0, et qu'on travaille avec Java6, le DriverManager demande à la classe (sun.misc.)Service tous les services qui fournissent l'interface java.sql.Driver. Avant, c'est en chargeant la classe du pilote que le pilote s'enregistrait auprès du `DriverManager` grâce à la méthode anonyme `static: registerDriver(Driver)`. Si le driver n'est pas compatible JDBC 4.0, il faut donc, avant de demander une `java.sql.Connection` au `DriverManager`, exécuter cette ligne (qui peut soulever une `java.lang.ClassNotFoundException`): `Class.forName(`*`nomCompletDuPilote`*`);`)



7.2 Les quatre classes qui entrent en jeu dans la connexion

7.2.1java.sql.Driver

À moins d'écrire son propre pilote JDBC, on n'a jamais besoin de cette classe dans l'application. Elle donne simplement à JDBC un point de lancement pour la connectivité à la base de données en répondant aux demandes de connexion du `java.sql.DriverManager` et en fournissant l'information sur l'implémentation en question.

7.2.2java.sql.DriverManager

Contrairement à la majorité des autres parties de JDBC, `DriverManager` est une classe et non une interface. Sa principale responsabilité est de maintenir une liste d'implémentations de `Driver` et de donner aux applications celle qui correspond à l'URL demandée. `registerDriver()` et `deregisterDriver()` permettent à une implémentation de `Driver` de s'enregistrer auprès du `DriverManager` ou de se supprimer de la liste. `getDrivers()` permet d'obtenir une énumération des pilotes enregistrés.

7.2.3java.sql.Connection

Représente une seule connexion logique à la base de données. On utilise la classe `Connection` pour envoyer une série d'instructions SQL à la base de données et pour valider ou annuler ces instructions.

7.2.4java.sql.SQLException

Permet d'intercepter tous les problèmes que pourrait soulever la base de données. Dès que quelque chose se passe mal entre JDBC et la base de données, cela soulève une `SQLException`.

En plus des informations que l'on trouve dans les exceptions java, `SQLException` fournit une information d'erreur propre à la base de données comme la valeur `SQLState` et le code d'erreur du fournisseur. En cas d'erreurs multiples, le pilote JDBC « chaîne » les exceptions. Autrement dit, on peut demander à toute `SQLException` si une autre exception le précède en appelant `getNextException()`.

7.3 Exemple de connexion JDBC

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
public class SimpleConnection {
      public static void main(String[] args) {
            Connection connection = null;
            if (args.length != 3) {
                  System.out.println(« Syntaxe: java SimpleConnection  » +
                        « URL IDENTIFIANT MOT_DE_PASSE »);
                        System.exit(1);
            }
            try {
                  connection = DriverManager.getConnection
                        (args[0], args[1], args[2]);
                  System.out.println(« Connexion réussie »);
                  // les requêtes et mises à jour se font ici
            } catch(SQLException e){
                  e.printStackTrace();
            } finally {
```

```
                    if (connection != null) {
                        try { connection.close(); }
                        catch(SQLException e) { e.printStackTrace(); }
                    }
                }
            }
        }
```

8 Requêtes sur une base de données

On doit utiliser un objet `Connection` pour générer des implémentations de `java.sql.Statement` liées à la même transaction (à la même connexion) de la base de données. Après avoir utilisé un ou plusieurs objets `Statement` générés par la connexion, on peut utiliser cette dernière pour valider ou annuler les objets `Statement` qui lui sont associées. Un `Statement` représente un instruction SQL.

On attribue des instructions SQL au statement lorsqu'on est prêt à les envoyer à la base de données. Lorsqu'on envoie l'instruction SQL, il faut savoir si c'est une requête (SELECT), dans ce cas on obtient un `java.sql.ResultSet` en retour (il permettra d'obtenir les différentes lignes du SELECT); ou si il s'agit d'une mise à jour (UPDATE, INSERT, DELETE, ...), dans ce cas, on obtient un entier en retour. Cet entier correspond au nombre de lignes qui ont été modifiées par la requête.

8.1 Les classes mises en oeuvre pour l'accès à la base de données

8.1.1java.sql.Connection

Déjà vue.

8.1.2java.sql.Statement

Représente une instruction SQL. Elle est obtenue grâce à l'objet `Connection`.

- Si il s'agit d'un SELECT, il faut appeler la méthode: `executeQuery(query: String): java.sql.ResultSet`
- Si il s'agit d'une mise à jour (INSERT, UPDATE, DELETE, ...): `executeUpdate(query: String): int` (la valeur de retour correspond au nombre de lignes qui ont été modifiées par la requête).

Dans le cas où on ne sait pas de quel type sera la requête (requête dynamique), on peut appeler la méthode: `execute(query: String): boolean`. Si la méthode retourne `vrai`, il s'agissait d'un SELECT et on peut obtenir le `ResultSet` grâce à `getResultSet()`; sinon, on peut obtenir le nombre de lignes modifiées par la requête grâce à la méthode: `getUpdateCount()`.

8.1.3java.sql.ResultSet

Représente une ou plusieurs lignes retournées par une requête. La classe fournit simplement un ensemble de méthodes pour retrouver les colonnes à partir des résultats d'une requête.

On peut parcourir les résultats grâce à la méthode `next(): boolean` qui permet de passer à la ligne suivante (tant qu'il y en a). Un premier appel à la méthode est nécessaire même si il n'y a qu'une seule ligne.

Ensuite, pour obtenir les informations d'une ligne, on utilise les méthodes qui ont la forme:

`getXXX(numéroColonne: int): XXX`

`ou getXXX(nomColonne: String): XXX`

Si la première colonne est un entier, on appellera `getInt(1)` et on obtiendra en retour un `int`. Si la colonne 6 est une chaîne de caractère, on appellera: `getString(6)` et on obtiendra en retour une `String`. Au lieu d'utiliser le numéro de la colonne (la première colonne est la colonne **1**), on peut utiliser le nom de la

colonne. Dans ce cas, on ne passe plus un entier mais une chaîne de caractères.

8.2 Remarques supplémentaires

8.2.1Le problème du `null`

En java, les types primitifs ne peuvent être `null`. Or, en SQL, ils le peuvent. Pour savoir si une valeur, après avoir appeler une méthode `getXXX()`, il suffit d'appeler la méthode `wasNull(): boolean`. Si le retour est `true`, la dernière valeur était nulle, sinon, elle ne l'était pas.

8.2.2Nettoyage des connexions

Par sécurité, pour être certain de libérer **toutes** les ressources, il vaut mieux appeler la méthode `close()` lorsqu'on a fini d'utiliser un `ResultSet`, un `Statement`, et une `Connection`.

8.2.3Transactions

JDBC valide automatiquement chaque instruction SQL au fur et à mesure qu'il les envoie à la base de données. On peut changer ce comportement (si on veut faire des transactions) en appelant: `con.setAutoCommit(false)`

Il faudra alors, lorsqu'on veut valider un ensemble de requêtes, appeler `con.commit()`; sinon, il faudra appeler `con.rollBack()`.

8.2.4Mise à jour via le ResultSet

Si le `ResultSet` a été créé comme `ResultSet.`*`CONCUR_UPDATABLE`*, il est possible de modifier la valeur d'un champ en appelant la méthode: `rs.updateXXX( nomChamp: String, valeur: XXX)`. Pour que la valeur soit mise à jour, il faut appeler la méthode: `rs.updateRow()`.

9 Les types de données

9.1 Correspondance SQL vers Java.

Type SQL	Type Java
BIT	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long
REAL	float
FLOAT	double
DOUBLE	double
DECIMAL	java.math.BigDecimal
NUMERIC	java.math.BigDecimal
CHAR	java.lang.String
VARCHAR	java.lang.String
LONGVARCHAR	java.lang.String
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp

BINARY	byte[]
VARBINARY	byte[]
LONGVARBINARY	byte[]
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
REF	java.sql.Ref
STRUCT	java.sql.Struct

9.2 Correspondance Java vers SQL.

Type Java	Type SQL
boolean	BIT
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	FLOAT
double	DOUBLE
java.math.BigDecimal	NUMERIC
java.lang.String	VARCHAR ou LONGVARCHAR
byte[]	VARBINARY ou LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
java.sql.Blob	BLOB
java.sql.Clob	CLOB
java.sql.Array	ARRAY
java.sql.Ref	REF
java.sql.Struct	STRUCT

Ces correspondances sont simplement la spécification JDBC pour une correspondance directe des types et non une loi imposant le format de vos données SQL que vous devez utiliser en Java. Autrement dit, vous pouvez récupérer une colonne INTEGER sous forme d'un long Java, ou bien mettre un objet Java Date dans un champ TIMESTAMP. Cependant, certaines conversions n'ont aucun sens. Vous ne pouvez pas mettre un booléen Java dans un champ DATE de la base de données.

10 En pratique et en accéléré

10.1 Connexion à une base de données

```
// Chargement du Driver (plus nécessaire avec les pilotes compatibles JDBC 4.0)
// Class.forName(« nom.du.driver »).newInstance();
// Connexion à la base de données
```

```
Connection c = DriverManager.getConnection(« url », « login », « password »);
```

La méthode `forName()` de la classe `java.lang.Class` peut soulever une exception `java.lang.ClassNotFoundException`.

L'appel à `java.sql.DriverManager.getConnection()` peut soulever une `java.sql.SQLException`.

10.2 SELECT

```
// Création d'un Statement
Statement stmt = conn.createStatement();
// Obtention du ResultSet
ResultSet rs = stmt.executeQuery(« SELECT * ... »);
// Parcours du ResultSet
while(rs.next()) { int id = rs.getInt(1); }
```

10.3 INSERT, UPDATE, DELETE, CREATE...

```
// Création d'un Statement
Statement stmt = conn.createStatement();
// Exécution de la requête
int result = stmt.executeUpdate(« INSERT * ... »);
```

11 Les résultats flottants (et la navigation)

11.1 Résultats flottants

Depuis JDBC 2.0, on peut parcourir les résultats dans les deux sens. Il faut utiliser un autre appel `createStatement()` en passant comme arguments: `ResultSet.CONSTANT`.

- argument 1:
  - `TYPE_FORWARD_ONLY`: n'est pas défilant, les autres oui;
  - `TYPE_SCROLL_SENSITIVE`: voir les modifications;
  - `TYPE_SCROLL_INSENSITIVE`: ne s'occupe pas des modifications. Sinon, on relance la requête et on voit les modifications.
- argument 2:
  - `CONCUR_READ_ONLY`
  - `CONCUR_UPDATEABLE`

11.2 Navigation dans un ensemble de résultats

Au moment de sa création, un `ResultSet` est positionné *avant* la première ligne. Les méthodes de positionnement comme `next()` font pointer un `ResultSet` sur des lignes réelles. `next()` retourne `true` tant qu'il y a des lignes à analyser.

On peut aussi parcourir le `ResultSet` avec `previous()`. Si on arrive avant la première ligne, cette méthode renvoie `false`.

Il y a aussi des méthodes pour se positionner:

Déplacement	Position

```
beforeFirst(): void
afterLast(): void
first(): void
last(): void
absolute(int): void
relative(int): void
```

```
isBeforeFirst(): boolean
isAfterLast(): boolean
isFirst(): boolean
isLast(): boolean
getRow(): int
```

11.3 Gérer les ensembles résultats défilants

Donner le sens de la navigation au pilote:

- `setFetchDirection(ResultSet.FETCH_FORWARD)`
- `setFetchDirection(ResultSet.FETCH_REVERSE)`
- `setFetchDirection(ResultSet.FETCH_UNKNOWN)`

Selon le pilote, les performances peuvent être augmentées.

Nombre de ligne à pré-charger:

`setFetchSize(int)` (par défaut: 0)

Une optimisation est possible, mais la valeur n'est pas une limite. On évite de charger 1000 lignes si on n'en consulte qu'une centaine.

12 Les différents types d'instructions (statement)

12.1 java.sql.Statement

Notez qu'un Statement (comme un PreparedStatement ou un CallableStatement) représentent **une seule requête**. Si vous devez effectuer plusieurs requêtes dans une méthode, il faut créer ou utiliser plusieurs Statement (ou enfant)

12.2 java.sql.PreparedStatement

Chaque instruction SQL envoyée à la base de données doit être analysée par le moteur de la base de données avant de pouvoir être exécutée. Pendant l'analyse de l'instruction SQL, la base de données lit le code afin de déterminer ce qu'on veut qu'elle fasse; elle formule ensuite un plan d'exécution des instructions: **construction d'un plan de requête**.

Si on exécute plusieurs fois la même instruction ayant le même plan de requête sans utiliser de `PreparedStatement`, on gaspille de la puissance de traitement.

Les bases de données fournissent deux sortes de SQL préparé:

- les instructions préparées;
- les procédures stockées.

L'avantage du SQL préparé par rapport aux instructions SQL simples, c'est que la base de données peut prendre le code SQL par avance et créer un plan de requête pendant qu'on s'occupe d'une autre partie de l'application.

Cela signifie que notre code SQL va s'exécuter plus vite et qu'on dispose d'une référence générique à une instruction qu'on peut réutiliser au lieu de recréer à chaque fois de nouvelles instructions SQL pour chaque nouvel accès à la base de données.

Comme le pilote va directement envoyer la requête pour que la base de données établisse son *plan de requête*, il faut, au moment de créer le `PreparedStatement`, envoyer la requête SQL:

```
PreparedStatement pstmt = conn.prepareStatement
          (« INSERT INTO contacts (prenom, nom, pays) VALUES (?, ?, ?) »);
```

Remarquez qu'il n'y a pas d'apostrophe pour entourer les points d'interrogation. Le PreparedStatement va pouvoir servir autant de fois que l'on veut. Il pourrait donc être instancié au début de l'application et fermer seulement à la fin de l'application.

Pour effectuer les requêtes, on va utiliser la séquence suivante: effacer les paramètres, mettre à jour les paramètres, exécuter l'update (dans ce cas un INSERT). Remarque: on peut (**doit**) utiliser le PreparedStatement pour toutes les instructions SQL.

```
pstmt.clearParameters();
pstmt.setString(1, « Arnold »);
pstmt.setString(2, « D'Angelo »);
pstmt.setString(3, « BE »);
pstmt.executeUpdate();
```