JDBC Tutoriel de Sun MicroSystem

traduction par Julien Guillard

1.0 Démarrer avec JDBC.

La première chose que vous devez faire, c'est de vérifier si votre système est correctement configuré:

1. Installez Java et JDBC sur votre machine.

   Pour installer votre plate-forme Java et l'API JDBC, suivez simplement les
   instructions pour télécharger la dernière version de JDK (Java Developement Kit).
   JDBC est fournit avec le JDK. Téléchargez là ici :

   http://java.sun.com/products/JDK/CurrentRelease

2. Installez un pilote sur votre machine

   Votre pilote devrait contenir des instructions d'installation. Certains pilote JDBC ont été écrit pour des SGBD (Systeme de Gestion de Base de Donnée) spécifiques, en général l'installation consiste à copier le pilote sur votre machine, aucune configuration n'est nécessaire.

3. Installez votre SGBD (si nécessaire)

   Si vous n'avez pas de SGBD installé sur votre machine, suivez les instructions
   fournis avec. La plupart des utilisateurs possèdent déjà un SGBD installé et
   travaillent avec une base de données déjà établie.

2.0 Implanter une base de données.

Nous considérerons que la base de données COFFEEBREAK existe déjà. (Créer une base de données n'est pas difficile, mais cela requière les permissions adéquates, tâche normalement effectuée par un administrateur de base de données.)

Quand vous créerez les tables utilisées dans ce tutoriel, elles seront dans la base de données par défaut. Nous avons expressément réduit la taille et le nombre des tables, gardant ainsi les choses raisonnables.

Supposez que notre exemple de base de données est utilisé par le propriétaire d'un petit bar appelé The Coffee Break, où les grains de café sont vendus à la livre et que le café torréfié et vendu à la tasse. Pour faire simple, nous imaginerons que le propriétaire a besoin de deux tables, une pour le type de café et une autre pour ses fournisseurs.

Tout d'abord, nous vous montrerons comment ouvrir une connexion avec votre SGBD, puis, comme tout ce que fait JDBC, c'est d'envoyer du code SQL à votre SGBD, nous utiliserons et expliquerons quelques instructions SQL. Après cela, nous vous montrerons avec quelle facilité d'utilisation JDBC passe ces instructions SQL à votre SGBD et comment il les retourne.

Ce code a été testé sur les SGBD principaux. Par contre, vous pourrez rencontrer quelques problèmes de compatibilité entre d'anciens pilotes OBDC et le JDBC-ODBC Bridge.

3.0 Établir une connexion.

La première chose que vous devez faire, est d'établir une connexion avec votre SGBD. Cela implique deux étapes :

- Charger les pilotes
- Créer la connexion

3.1 Charger les pilotes.

Charger le pilote ou les pilotes que vous voulez utiliser est très simple. Si par exemple, vous voulez utiliser le pilote JDBC-ODBC Bridge, ce code le chargera :

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

La documentation de votre pilote devrait vous fournir le nom de la classe à utiliser. Par contre, si le nom de la classe est jdbc.piloteXYZ, vous devrez charger le pilote avec cette ligne de code :

```
Class.forName("jdbc.piloteXYZ");
```

Vous n'avez pas besoin de créer une instance du pilote et de le référencer avec DriveManager, appeler Class.forName le fera pour vous automatiquement. Si vous aviez à créer votre propre instance, vous créeriez un duplicata inutile.

Une fois le pilote chargé, vous êtes prêt pour créer une connexion avec une SGDB.

3.2 Créer une connexion.

La deuxième étape pour établir une connexion est d'avoir le pilote approprié pour se connecter à votre SGBD. Cette ligne de code illustre l'idée :

```
Connection con = DriverManager.getConnection(url,
"MonLogin","MonMotDePasse");
```

Cette étape est aussi très simple, le plus dur est de fournir quelque chose pour l'URL. Si vous utilisez JDBC-ODBC Bridge, l'URL JDBC commencera par jdbc:odbc: . Le reste de l'URL, c'est généralement le nom de votre source de données ou du système de la base de données.

Donc, admettons que vous utilisiez ODBC pour accéder à une source de données ODBC appelé "Fred", par exemple, votre URL JDBC sera jdbc:odbc:Fred . À la place de " MonLogin", vous devrez mettre le nom que vous utilisez pour vous loguer auprès de votre SGBD, et à la place de "MonMotDePasse", votre mot de passe pour cet acompte. Donc si vous voulez vous identifiez à votre SGBD avec comme nom de login "Fernanda" et comme mot de passe "J8", ces deux lignes de code établiront la connexion:

```
String url = "jdbc:odbc:Fred";
Connection con =
DriverManager.getConnectio(url,"Fernanda","J8");
```

Si vous utilisez un pilote JDBC développé par un tiers, la documentation vous dira quel sous-protocole utiliser, donc, que mettre après jdbc: dans l'URL JDBC.

Si un des pilotes que vous avez chargé reconnaît l'URL JDBC fournit dans la méthode DriverManager.getConnection , ce pilote établira une connexion avec le SGBD spécifié dans l'URL JDBC.La class DriverManager, prendra en charge tout les détails afin d'établir, pour vous, la connexion. Si vous avez écrit votre propre pilote, vous n'aurez probablement jamais à utiliser aucune des méthodes de l'interface Driver, et la seule méthode dont vous aurez besoin est DriverManager.getConnection.

La connexion retournée par la méthode DriverManager.getConnection est une connexion ouverte, ce qui vous permettra de passer vos instructions SQL vers votre SGBD. Dans l'exemple précédant, conn est une connexion ouverte, et nous l'utiliserons pour les exemples qui suivront.

4.0 Configurer vos Tables

Tout d'abord, nous créerons les tables pour notre exemple de base de données. Cette table CAFE, contient les informations essentielles concernant les cafés vendus chez The Coffee Break, incluant le nom du café, son prix, le nombre de livres vendues dans la semaine courante, et le nombre de livres vendues jusqu'à maintenant. La table CAFE, que nous décrirons plus en détails plus tard, est la suivante :

NOM_CAFE	FO_ID	PRIX	VENTES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

La colonne qui contient le nom du café est NOM_CAFE, elle supporte des valeurs de type VARCHAR et a un maximum de 32 caractères de long. Comme nous utiliserons un nom différent pour chaque type de café vendu, le nom identifiera un café de façon unique, et pourra donc servir de clé primaire à notre table. La seconde colonne, FO_ID, contient un nombre qui identifie le fournisseur de café, cette variable SQL est de type INTEGER. La troisième colonne, appelée PRIX, contient des valeurs FLOAT, car nous avons besoin de valeurs supportants le point décimal. (Notez bien que cette valeur monétaire devrait être normalement conservée dans une variable SQL DECIMAL ou NUMERIC, mais, pour éviter les incompatibilités avec les anciennes versions de JDBC, nous utiliserons le type FLOAT (plus standard.). La colonne appelée VENTES contient des valeurs SQL de type INTEGER, et indique le nombre de livres de café vendues durant la semaine. La dernière colonne, TOTAL, contient un INTEGER SQL qui donne le nombre de livres de café vendues jusqu'à maintenant.

FOURNISSEURS, la seconde table de notre base de données, donne des informations pour chaque fournisseur :

FO_ID	NOM_FO	RUE	VILLE	ETAT	CODE_POSTALE
101	Acme, Inc	99 Market Street	GroundVille	CA	95199
49	Superior Coffee	1 Party Place	Mendocino	CA	95460
150	The High Ground	100 Coffee Lane	Meadows	CA	93966

Les tables CAFE et FOURNISSEURS contiennent toutes deux la colonne FO_ID, cela veux dire que ces deux tables peuvent être utilisées dans une instruction SELECT pour prendre l'information basée sur ces deux tables. La colonne FO_ID est la clé primaire dans la table FOURNISSEURS, elle identifie de façon unique chacun des fournisseurs de café. Dans la table CAFE, FO_ID est appelée une clé étrangère.(Vous pouvez vous imaginez que la clé étrangère est étrangère dans le sens qu'elle est importée d'une autre table.) Notez que chacun des nombres dans FO_ID apparaît une seule fois

dans la table FOURNISSEURS; il faut pour cela qu'elle soit clé primaire. Dans la table CAFE, ou cette colonne est clé étrangère, cela démontre qu'un fournisseur peu vendre plusieurs types de café. Plus tard dans le chapitre, nous verrons un exemple qui montre comment utiliser une clé primaire et une clé étrangère dans une instruction SELECT.

L'instruction SQL qui suit crée la table CAFE. Les entrées entre les parenthèses sont le nom de chacune des colonnes suivit d'un espace puis du type de données que la colonne peut recevoir. Une virgule sépare chacune des colonnes les unes des autres. Le type VARCHAR est créer avec une longueur maximum de 32 caractères.

```
CREATE TABLE COFFEES
(COF_NAME VARCHAR(32),
SUP_ID INTEGER,
PRICE FLOAT,
SALES INTEGER,
TOTAL INTEGER)
```

Ce code ne se termine pas par une fin d'instruction car il peut varier d'un SGBD à un autre. Par exemple, Oracle utilise un point virgule (;) pour mettre fin à une instruction, Sybase utilise le mot go. Le pilote que vous utilisez apportera automatiquement le symbole qui mettra fin à l'instruction, et vous n'aurez pas besoin de l'inclure dans votre code JDBC.

Une autre chose à remarquer au sujet de l'instruction SQL dans sa forme. Dans l'instruction CREATE TABLE, les mots clés sont mis en lettres capitales, et chacun des items sont sur une ligne séparée. SQL ne nécessite pas de telles arrangements, mais ces conventions les rendent plus facile à lire. Le standard d'SQL est que les mots clés ne regardent pas la case, donc, par exemple, l'instruction SELECT qui suit peut être écrite de différentes façons :

```
SELECT Nom, Prenom
FROM Employees
WHERE Nom LIKE "Washington"

select Nom, Prenom from Employees where
Nom like "Washington"
```

Les chaînes de caractères par contre, sont sensibles à la casse : dans le nom " Washington, " "W" doit être en lettre capitale et le reste en minuscules.

Ces spécifications peuvent varier d'un SGBD à un autre quand il s'agit d'identifier des noms. Par exemple, certain SGBD demandent que les colonnes et les tables soient données exactement telles qu'elles ont été crées dans l'instruction CREATE TABLE, quand d'autres non. Par sécurité, nous utilisons les identifiants tel que CAFE et FOURNISSEURS en lettres majuscules car c'est ainsi que nous les avons crées.

Donc nous avons écrit l'instruction SQL pour créer la table CAFE. Maintenant, mettons le entre guillemet (pour en faire une chaîne de caractères) et assignons ce String à une variable creerTableCafe que nous utiliserons plus tard dans notre code JDBC. Tel que montré précédemment, les SGBD se fichent de savoir si l'instruction qui leurs est donné se fait sur plusieurs lignes, mais dans le langage de programmation Java, un objet String qui s'étend sur plus d'une ligne ne compilera pas. Par conséquent, quand vous attribuerez une chaîne de caractères, vous aurez besoin d'inclure chaques lignes entre guillemet et d'utiliser le signe plus (+) pour concaténer le tout.

```
String creerTableCafe = "CREATE TABLE CAFE" +
"(NOM_CAFE VARCHAR(32), FO_ID INTEGER, PRIX FLOAT,
" +
"VENTES INTEGER, TOTAL INTEGER)";
```

Le type de données que nous avons utilisé dans notre CREATE TABLE est le type SQL (aussi appelé type JDBC) générique qui est défini dans java.sql.Types. Les SGBD utilisent généralement ces types standards, donc quand sera venu le temps d'essayer certaines applications JDBC, vous aurez juste à utiliser l'application CreerCafe.java, qui utilise l'instruction CREATE TABLE.

Avant de lancer n'importe quelle application, nous allons traverser des bases de JDBC.

4.1 Créer une instruction JDBC

Un objet Statement est ce que votre instruction SQL envoie vers le SGBD. Vous créerez simplement un objet Statement puis, l'exécuterez, lui fournissant la méthode d'exécution appropriée avec l'instruction SQL que vous voulez envoyer.Pour une instruction SELECT, la méthode à utiliser est executeQuery. Pour les instructions visant à créer ou modifier des tables, la méthode est executeUpdate.

Vous devez avoir l'instance d'une connexion active pour créer un objet Statement. Dans l'exemple suivant, nous utilisons notre objet Connection conn, pour créer l'objet Statement stmt :

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

A ce niveau, stmt existe, mais il n'a aucune instruction SQL à passer au SGBD. Nous devrons la fournir dans la méthode que nous utiliserons pour exécuter stmt. Par exemple, dans le bout de code suivant, nous proposons executeUpdate avec l'instruction SQL :

```
stmt.executeUpdate(("CREATE TABLE CAFE(NOM_CAFE
VARCHAR(32),FO_ID INTEGER,"+ "PRIX FLOAT, VENTES
INTEGER, TOTAL INTEGER)"));
```

Si nous avons fait de l'instruction SQL un string et que nous l'avons assigné à la variable creerTableCafe, nous pouvons écrire le code comme il suit :

```
stmt.executeUpdate(creerTableCafe);
```

4.2 Executer une instruction

Nous avons utilisé la méthode executeUpdate car l'instruction SQL contenue dans creerTableCafe est une DDL (Data Definition Language). Les instructions consistants à créer des tables, modifier des tables ou effacer des tables sont des exemples d'instructions DDL et sont exécuter avec la méthode executeUpdate. La méthode executeUpdate est utilisée pour exécuter les instruction SQL qui mettent à jour une table. En pratique, executeUpdate est utilisé le plus souvent pour mettre des tables à jour plutôt que pour les créer, car une table ne peut être créer qu'une seule fois, mais mise à jour plusieurs fois.

4.3 L'entrée de donnée dans une table

Nous vous avons montré comment créer la table CAFE en spécifiant le nom des colonnes et le type de données qu'elles contiennent, mais cela ne fait que construire la structure de la table. Elle ne contient aucune donnée. Nous allons entrer nos données dans une table une ligne à la fois, fournissant l'information à stocker dans chacune des colonnes de cette ligne. Notez que les valeurs insérées dans les colonnes doivent être dans le même ordre que les colonnes à leur création.

Le code suivant insère une ligne de données, Colombian dans la colonne NOM_CAFE, 101 dans FO_ID, 7.99 dans PRIX, 0 dans VENTES et 0 dans TOTAL.(The Coffee Break vient juste de commencer, c'est pourquoi certaines valeurs sont misent à 0.). Comme nous l'avons fait pour créer

des tables, nous allons déclarer un objet Statement, et l'exécuter en utilisant la méthode executeUpdate.

Portez attention qu'il faut un espace entre CAFE et VALUES. Cet espace doit être entre les guillemets, et peut être insérer après CAFE ou avant VALUES, sans cet espace, l'instruction serait erronée : " INSERT INTO CAFEVALUES . . ." , donc le SGBD cherchera la table CAFEVALUES.

```
Statement stmt = conn.createStatement();
stmt.executeUpdate(
"INSERT INTO CAFE VALUES ('Colombian', 101, 7.99,
0, 0)");
```

Le code qui suit insère une deuxième ligne dans la table CAFE. Nous réutilisons l'objet Statement stmt plutôt que d'en créer un nouveau pour chaque exécution.

```
stmt.executeUpdate("INSERT INTO CAFE" +
"VALUES ('French_Roast', 49, 8.99, 0, 0)");
```

Les valeurs pour les lignes restantes peuvent être insérées comme il suit :

```
stmt.executeUpdate("INSERT INTO CAFE" +
"VALUES ('Espresso', 150, 9.99, 0, 0)");
stmt.executeUpdate("INSERT INTO CAFE" +
"VALUES ('Colombian_Decaf', 101, 8.99, 0, 0)");
stmt.executeUpdate("INSERT INTO CAFE" +
"VALUES ('French_Roast_Decaf', 49, 9.99, 0, 0)");
```

4.4 Accéder aux données d'une table.

Maintenant que la table CAFE contient des valeurs, nous pouvons écrire une instruction SELECT pour avoir accès à ces valeurs. L'étoile (*) dans l'instruction SQL qui suit indique que toutes les colonnes devront être sélectionnées. N'ayant pas de clause WHERE définissant la restriction, nous effectuons la sélection sur toute la table.

```
SELECT * FROM CAFE
```

Le résultat, désignant la table entière, devrait ressembler à ceci :

NOM_CAFE	FO_ID	PRIX	VENTES	TOTAL
Colombian	101	7.99	0	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Le résultat ci-dessus est ce que vous devriez voir sur votre terminal si vous avez entré la requêtes SQL directement dans le système de la base de données. Lorsque nous accéderons à une base de données au travers d'une application Java, nous aurons besoin de prendre les résultats, pour que nous puissions les utiliser. Nous verrons comment faire tout ça dans la prochaine section.

5.0 Accéder aux valeurs d'un Result Set

Nous allons maintenant voir comment envoyer les requêtes écritent dans la section précédante depuis un programme écrit en Java, et comment accéder à ce résultat.
JDBC renvoie les résultats dans un objet ResultSet, donc, nous avons besoin de déclarer une

instance de la class ResultSet pour contenir nos résultats. Le code qui suit déclare l'objet ResultSet rs et lui assigne le résultat de notre requête précédente :

```
ResultSet rs = stmt.executeQuery(
"SELECT COF_NAME,PRICE FROM COFFEES");
```

5.1 Utiliser la méthode next

La variable rs, qui est une instance de ResultSet, contient les lignes de café et leurs prix. Pour pouvoir accéder à leurs noms et à leurs prix, nous irons dans chaque ligne et récupèrerons les valeurs selon leur type. La méthode next déplace ce qui est appelé curseur à la ligne suivante, et fait de cette ligne (appelée la ligne courante) celle sur laquelle nous travaillons. Initialement, ce curseur est positionné juste au dessus de la première ligne d'un objet ResultSet, donc le premier appel à la méthode next déplace le curseur à la première ligne et en fait la ligne courante. L'invocation successive de la méthode next déplace le curseur vers le bas, ligne par ligne. Notez qu'avec l'API JDBC 2.0, couvert dans la prochaine section, vous pouvez déplacer le curseur vers l'arrière, à une position spécifique, et à une position relative à la ligne courante.

5.2 Utiliser la méthode getXXX

Nous utilisons la méthode getXXX du type approprié pour accéder à une valeur dans chaque colonne. Par exemple, la première colonne de chaque ligne du result set est NOM_CAFE, qui contient une valeur de type SQL VARCHAR. La méthode pour accéder à une valeur de type VARCHAR est getString. La seconde colonne de chaque ligne contient une valeur de type SQL FLOAT, et la méthode pour accéder à ces valeurs est getFloat. Le code qui suit accède aux valeurs contenues dans la ligne courante du result set et affiche la ligne avec le nom suivit de trois espaces puis le prix. À chaque fois que la méthode next est invoquée, la ligne suivante devient la ligne courante, et la boucle continue jusqu'à ce qu'il n'y ai plus de ligne dans le result set.

```
String requete = "SELECT NOM_CAFE, PRIX FROM CAFE";
ResultSet rs = stmt.executeQuery(requete);
while(rs.next()){
String s = rs.getString("NOM_CAFE");
float n = rs.getFloat("PRIX");
System.out.println(s + " " + n);
```

Le programme affichera ceci :

```
Colombian 7.99
French_Roast 8.99
Espresso 9.99
Colombian_Decaf 8.99
French_Roast_Decaf 9.99
```

Voyons maintenant plus précisément comment la méthode getXXX fonctionne en examinant les deux instructions getXXX dans ce code. Premièrement, examinons getString.

String s = rs.getString("NOM_CAFE");

La méthode getString est invoquée sur l'objet ResultSet rs, donc getString doit accéder aux valeurs contenues dans la colonne NOM_CAFE de la ligne courante de rs. La valeur que getString rapporte a été convertie du SQL VARCHAR, au String de Java, et a été assignée à l'objet String s. Notez que nous avons utilisé la variable s dans l'expression println ci-dessus.

La situation est la même avec la méthode getFloat, à l'exception près, qu'il rapporte la valeur contenue dans la colonne PRIX, qui est un FLOAT SQL, et la convertir en FLOAT de Java avant de l'assigner à la variable n.

JDBC offre deux façons d'identifier une colonne dont une méthode getXXX prendra la valeur. Une des façons est de donner le nom de la colonne, comme dans l'exemple ci-dessus, et la seconde, est de donner l'index de la colonne (numéro de la colonne), où 1 désigne la première colonne, 2 la deuxième colonne et ainsi de suite. Utiliser le numéro de la colonne à la place de son nom donne le code suivant :

```
String s = rs.getString(1);
float n = rs.getFloat(2);
```

La première ligne de code prend la valeur de la première colonne de la ligne courante de rs (colonne NOM_CAFE), la convertie en String Java, et l'assigne à s. La deuxième ligne de code, prend la valeur contenue dans la deuxième colonne de la ligne courante de rs, la convertie en float Java, et l'assigne à n. Notez que le nombre de la colonne réfère au numéro de la colonne dans le result set, pas dans la table originale.

En résumé, JDBC vous autorise à utiliser le nom de la colonne ou son numéro comme argument dans une méthode getXXX. Utiliser le numéro de colonne est un peu plus pratique, mais dans certains cas, le nom de la colonne est requis. En général, fournir le nom de la colonne est essentiellement équivalent à fournir son numéro.

Grâce à la méthode getXXX de JDBC, vous pourrez alors accéder aux différents types de données SQL. Par exemple, la méthode getInt peut être utilisée pour accéder aux types numériques, ou caractères. Mais il est recommandé de n'utiliser getInt que pour accéder à des données SQL de type INTEGER. Il ne peut pas être utilisé pour rapporter des données de type BINARY, VARBINARY, LONGVARBINARY, DATE, TIME ou TIMESTAMP.

À cette adresse, vous trouverez les méthodes getXXX, et leurs conseils d'utilisation :

http://java.sun.com/docs/books/tutorial/jdbc/basics/_retrievingTable.html

6.0 Mettre les tables à jours

Supposons qu'après la première semaine, le propriétaire du The Coffee Break veuille mettre à jours la colonne VENTES de la table CAFE en entrant le nombre de livres vendues pour chaque type de café. L'instruction SQL pour mettre à jour est la suivante :

```
String updateString = "UPDATE CAFE" +
"SET VENTES = 75" +
"WHERE NOM_CAFE LIKE 'Colombian'";
```

Utilisant l'instruction stmt, le code JDBC exécute l'instruction SQL contenue dans updateString :

```
stmt.executeUpdate(updateString);
```

La table COFFES doit maintenant ressembler à ça :

NOM_CAFE	FO_ID	PRIX	VENTES	TOTAL

Colombian	101	7.99	75	0
French_Roast	49	8.99	0	0
Espresso	150	9.99	0	0
Colombian_Decaf	101	8.99	0	0
French_Roast_Decaf	49	9.99	0	0

Notez que nous n'avons pas encore mis à jour la colonne TOTAL, sa valeur est toujours 0.

Maintenant, sélectionnons la ligne mise à jour, rapportons la valeur dans les colonnes de NOM_CAFE et de VENTES, et affichons ces valeurs :

```
String requete = "SELECT NOM_CAFE, VENTES FROM
CAFE" +
"WHERE NOM_CAFE LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(requete);
while(rs.next()){
String s = rs.getString("NOM_CAFE");
int n = rs.getInt("VENTES");
System.out.println(n + " livres de" + s + " vendu
cette semaine.");
```

Ça affichera :

75 livres de Colombian vendu cette semaine

La clause WHERE limite la sélection à une seule ligne, il n'y avait qu'une seule ligne dans le result set rs, et donc une seule ligne a été affichée. Dans ce cas, il est possible d'écrire le code sans la boucle while :

```
rs.next();
String s = rs.getString(1);
int n = rs.getInt(2);
System.out.println(n + " livres de " + s + " vendu
cette semaine.");
```

Même si il n'y a qu'un seul résultat dans le result set, vous devez utiliser la méthode next pour y accéder. Un objet ResultSet est créé avec un curseur pointant au dessus de la première ligne. Le premier appel de la méthode next positionne le curseur sur la première (et dans ce cas, la seule) ligne de rs. Dans ce code, next est appelé juste une fois, donc si il y avait eu une autre ligne, elle n'aurait jamais pu être accédée.

Maintenant mettons à jour la colonne TOTAL en ajoutant le montant hebdomadaire au total existant, et affichons le nombre de livres vendues à ce jour :

```
String updateString = "UPDATE CAFE" +
"SET TOTAL = TOTAL + 75 " +
"WHERE NOM_CAFE LIKE 'Colombian'";
stmt.executeUpdate(updateString);
String query = "SELECT NOM_CAFE, TOTAL FROM CAFE" +
"WHERE NOM_CAFE LIKE 'Colombian'";
ResultSet rs = stmt.executeQuery(query);
while(rs.next()){

      String s = rs.getString(1);
      int n = rs.getInt(2);
```

```
            System.out.println(n+" livres de" + s +
            " vendu jusqu'à maintenant.");

        }
```

Notez que dans cet exemple, nous utilisons l'index de la colonne plutôt que son nom, fournissons l'index 1 à getString (la première colonne du result set est NOM_CAFE), et l'index 2 à getInt (la seconde colonne du result set est TOTAL). Il est important de faire la distinction entre l'index d'une colonne dans la table de la base de données comme opposée à l'index dans le result set. Par exemple, TOTAL est la cinquième colonne dans la table CAFE mais c'est la seconde colonne dans le résultat généré par la requête dans l'exemple ci-dessus.

7.0 Fin des bases de JDBC

Avec ce que nous avons fait jusqu'ici, vous avez appris les bases de JDBC. Vous avez vu comment créer des tables, insérer des valeurs à l'intérieur, faire des requêtes, accéder à un résultat, et mettre à jour une table. C'est là la façon la plus simpliste d'utiliser une base de données, et vous pouvez maintenant les utiliser dans un programme écrit en Java utilisant l'API JDBC 1.0. Nous avons utilisé des requêtes très simples dans nos exemples, mais en admettant que votre SGBD les supporte, vous pouvez en envoyer de bien plus complexes en utilisant les bases couvertes jusqu'à maintenant.

Le reste de ce tutoriel, est axé sur l'utilisation de notions un peu plus avancées : Prepared statements, stored procedures, et transactions.

8.0 Utilisation des Prepared Statements.

Parfois, il est plus utile et plus efficace d'utiliser l'objet PreparedStatement pour envoyer une instruction SQL à la base de données. Ce type spécial d'instruction est dérivé de la classe plus générale, Statement, que vous connaissez déjà.

8.1 Quand utiliser un objet PreparedStatement.

Si vous voulez exécuter un objet Statement plusieurs fois, le temps d'exécution sera normalement réduit si vous utilisez à la place un objet PreparedStatement.

La fonctionnalité principale d'un objet PreparedStatement, contrairement à l'objet Statement, est de lui fournir une instruction SQL dès sa création. L'avantage de ceci dans plusieurs cas, est que l'instruction SQL sera directement envoyée au SGBD, où elle y sera compilée. Ayant pour résultat que l'objet PreparedStatement ne contient plus seulement une instruction SQL, mais bien une instruction SQL précompilée. Cela signifie que quand le PreparedStatement est exécuté, le SGBD a juste à lancer l'instruction SQL du PreparedStatement sans avoir à le compiler avant.

Un objet PreparedStatement peut être aussi utilisé dans une instruction SQL sans paramètre, mais vous allez plus probablement les utiliser pour des instructions SQL avec paramètre. L'avantage de ceci, quand vous utilisez une instruction SQL requérant des paramètres, est que vous pouvez utiliser la même instruction en fournissant différentes valeurs à chaque fois que vous l'exécutez. Nous verrons un exemple dans la prochaine section.

8.2 Créer un objet PreparedStatement.

Tout comme l'objet Statement, vous devez créer l'objet PreparedStatement avec une méthode Connexion. Utilisant notre précédante connexion ouverte conn, vous aurez à écrire un code ressemblant à celui ci-dessous pour créer un objet PreparedStatement recevant deux paramètres.

```
        PreparedStatement updateVentes =
```

```
conn.preparedStatement(
"UPDATE COFFES SET VENTES = ? WHERE N LIKE ?");
```

La variable updateVentes contient maintenant une instruction SQL, "UPDATE CAFE SET VENTES = ? WHERE NOM_CAFE LIKE ?", qui a été envoyée à SGBD puis précompilée.

8.3 Paramètres vers un PreparedStatement.

Vous devrez fournir des valeurs pour être utilisées à la place des points d'interrogation, si il y en a, avant que vous puissiez exécuter un objet PreparedStatement. Pour faire ça, il faudra utiliser l'une des méthodes setXXX définies dans la classe PreparedStatement. Si la valeur que vous voulez substituer au point d'interrogation est un int Java, vous devrez appeler la méthode setInt. Si la valeur est un String Java, vous devrez appeler la méthode setString, et ainsi de suite. En général, il y a un setXXX pour chaque type Java.

En utilisant l'objet PreparedStatement de l'exemple précédant, la ligne de code qui suit devrait remplacer le point d'interrogation en un int Java, la valeur 75 :

```
updateVentes.setInt(1,75);
```

Comme vous pouvez le voir dans l'exemple, le premier argument donné à la méthode setXXX est la position du point d'interrogation, et le deuxième argument, la valeur par laquelle on veut le remplacer. L'exemple suivant substitue le deuxième paramètre avec le String "Colombian".

```
updateVentes.setString(2, "Colombian");
```

Après que ces valeurs aient été déposées dans les deux paramètres d'entrées, l'instruction SQL contenue dans updateVentes sera équivalente à l'instruction SQL dans l'objet String updateString que nous avons utilisé dans un des exemples précédents. Les deux fragments de codes suivants font la même chose :

Fragment de Code 1 :

```
String updateString = "UPDATE CAFE SET VENTES = 75
" +
"WHERE NOM_CAFE LIKE 'Colombian'";
stmt.executeUpdate(updateString);
```

Fragment de Code 2 :

```
PreparedStatement updateVentes =
conn.preparedStatement(
"UPDATE CAFE SET VENTES = ? WHERE NOM_CAFE
LIKE ?");
updateVentes.setInt(1,75);
updateVentes.setString(2, "Colombian");
updateVentes.executeUpdate();
```

Nous avons utilisé la méthode executeUpdate pour exécuter les objets Statement stmt et PreparedStatement updateVentes. Remarquez qu'aucun argument n'est fourni à executeUpdate quand ils sont utilisés pour exécuter updateVentes. Et ceci est juste, car updateVentes contient déjà l'instruction SQL devant être exécutée.

Prêtez attention à ces exemples, vous verrez pourquoi vous devriez utiliser un objet PreparedStatement avec paramètres, plus qu'une simple instruction, et ce, même si elle nécessite

moins d'étapes. Si vous mettiez une ou deux fois à jour la colonne VENTES, cela ne vaudrait pas le coup d'utiliser une instruction SQL avec paramètres. Mais si vous la mettez souvent à jour, il serait plus simple d'utiliser un objet PreparedStatement, spécialement en situation ou vous avez à utiliser une boucle for ou une boucle while pour définir un paramètre à une succession de données. Nous verrons un exemple plus tard dans cette section.

Une fois qu'on a assigné une valeur à un paramètre, il retiendra la valeur jusqu'à ce qu'elle soit réinitialisée avec une autre valeur, ou bien jusqu'à ce que la méthode clearParameters soit appelée. Utilisant l'objet PrepareStatement updateVentes. Le code suivant illustre la réutilisation d'un prepared statement après avoir réinitialisé la valeur de l'un des paramètres et en conservant la valeur de l'autre paramètre :

```
updateVentes.setInt(1, 100);
updateVentes.setString(2, "French_Roast");
updateVentes.executeUpdate();
//Change la valeur de la colonne VENTES à la ligne
French Roast pour la valeur //100
updateVentes.setString(2, "Espresso");
updateVentes.executeUpdate();
//Change la valeur de la colonne VENTES à la ligne
Espresso pour la valeur //100(le premier paramètre
est resté à 100, et le second paramètre est
//réinitialisé par "Espresso"
```

8.4 Utilisation d'une boucle pour assigner des valeurs.

Vous pouvez souvent rendre le code plus facile en utilisant une bouche for ou while pour assigner des valeurs aux paramètres d'entrées.

Le fragment de code qui suit présente l'utilisation d'une boucle for pour assigner des valeurs aux paramètres dans un objet PreparedStatement updateVentes. Le tableau VentesDeLaSemaine contient les montants des ventes hebdomadaires. Ces montants de ventes correspondent aux noms des cafés listés dans le tableau cafe, donc le premier montant dans VentesDeLaSemaine(175) s'applique au premier café dans cafe (" Colombian "), le second montant dans VentesDeLaSemaine (150) s'applique au second café dans cafe (" French_Roast "), et ainsi de suite. Ce fragment de code présente la mise à jour de la colonne VENTES pour tout les cafés de la table CAFE :

```
PreparedStatement updateVentes;
String updateString = "update CAFE" +
"set VENTES = ? WHERE NOM_CAFE LIKE ?";
updateVentes = conn.prepareStatement(updateString);
int []VentesDeLaSemaine = {175 , 150, 60, 155, 90};
String [] cafes
={"Colombian","French_Roast","Espresso",
"Colombian_Decaf", "French_Roast_Decaf"};
int len = cafes.length;
for(int i = 0 ; i < len ; i ++){
updateVentes.setInt(1, VentesDeLaSemaine[i]);
updateVentes.setString(2, cafes[i]);
updateVentes.executeUpdate();

}
```

Quand le propriétaire désire faire une mise à jour sur les montants des ventes de la semaine suivante, il peut utiliser ce code comme modèle. Il ne lui reste plus qu'à entrer de nouveaux montants de vente dans l'ordre adéquat dans le tableau VentesDeLaSemaine. Le nom du café dans le tableau cafe reste constant, donc il n'a pas besoin d'y apporter de modifications. (Dans une vraie application, les valeurs seraient probablement entrées par l'utilisateur plutôt que d'être initialisées dans le tableau).

8.5 Valeurs retournées par la méthode executeUpdate.

La méthode executeQuery retourne un objet ResultSet contenant les résultats de la requête envoyée au SGBD, la valeur retournée pour executeUpdate est un int qui indique combien de lignes de la table ont été mises à jour. Le code qui suit montre la valeur retournée par executeUpdate en l'assignant à la variable n :

```
updateVentes.setInt(1,50);
updateVentes.setString(2, "Espresso");
int n = updateVentes.executeUpdate();
//n = 1 car une ligne procède à un changement.
```

La table CAFE a été mise à jour en remplaçant la valeur contenue dans la colonne VENTES de la ligne Espresso par 50. La mise à jour affecte une seule ligne dans la table, donc n est égal à 1.

Quand la méthode executeUpdate est utilisée pour exécuter une instruction DDL, comme créer une table, elle retourne la valeur 0. Dans le fragment de code suivant, qui exécute une instruction DDL pour créer la table CAFE, n recevra la valeur 0 :

```
int n = executeUpdate(createTableCafe); // n = 0
```

Notez que quand la valeur retournée de executeUpdate est 0, cela peut vouloir dire deux choses :

- L'instruction exécutée est une mise à jour et que 0 ligne sont affectées.
- L'instruction est une instruction DDL

9.0 Utiliser les jointures.

Parfois, il vous faut utiliser deux ou plusieurs tables pour accéder à l'information que vous voulez. Par exemple, supposons que le propriétaire du The Coffee Break veut une liste des cafés qu'il a achetés chez Acme, Inc. Cela demande des informations dans la table CAFE ainsi que dans la table FOURNISSEURS. C'est un des cas ou une jointure est nécessaire. Une jointure est une opération qui relie deux ou plusieurs tables par les valeurs qu'elles ont en commun. Dans notre base de données, les tables CAFE et FOURNISSEURS ont toutes deux la colonne FO_ID, qui pourra être utilisée pour joindre les deux tables.

Mais avant d'aller plus loin, nous devons créer la table FOURNISSEURS et la remplir de ses valeurs. Le code ci-dessous créera la table FOURNISSEURS :

```
String createFournisseurs = "create table
FOURNISSEURS" +
"(FO_ID INTEGER, NOM_FO VARCHAR(40),RUE
VARCHAR(40), VILLE VARCHAR(20), " +
"ETAT CHAR(2), CODE_POSTAL CHAR(5))";
stmt.executeUpdate(createFournisseurs);
```

Le code suivant insère les lignes pour les trois fournisseurs dans FOURNISSEURS :

```
stmt.executeUpdate("insert into FOURNISSEURS values
(101,'Acme, Inc.',"+ "'99 Market Street',
'Groundsville', 'CA', '95199'");
stmt.executeUpdate("Insert into FOURNISSEURS values
(49,'Superior " + "Coffee', '1 Party Place',
'Mendocino', 'CA','95460'");
```

```
stmt.executeUpdate("Insert into FOURNISSEURS values
(150,'The High"+ "Ground', '100 Coffee Lane',
'Meadows', 'CA','93966'");
```

Maintenant que nous avons les tables CAFE et FOURNISSEURS, nous pouvons procéder au scénario où le propriétaire veut obtenir la liste des cafés qu'il a achetés auprès d'un fournisseur en particulier. Le nom des fournisseurs se trouve dans la table FOURNISSEURS et le nom des cafés se trouve dans la table CAFE. Deux des tables possèdent maintenant la colonne FO_ID, cette colonne peut être alors utilisée comme jointure. Mais vous devez pouvoir distinguer de quelle table on parle lorsque la colonne FO_ID est utilisée. On utilise pour ça, le nom de la table devant la colonne séparée par un point, par exemple " CAFE.FO_ID" réfère que vous utilisez la colonne FO_ID de la table CAFE. Le code suivant, dans lequel stmt est un objet Statement, sélectionne les cafés achetés chez Acme, Inc :

```
String requete = "
SELECT CAFE.NOM_CAFE" +
"FROM CAFE, FOURNISSEURS " +
"WHERE FOURNISSEURS.NOM_FO LIKE 'Acme,Inc.''" +
"and FOURNISSEURS.FO_ID = CAFE.FO_ID";

ResultSet rs = stmt.executeQuery(requete);
System.out.println("Les café acheté à Acme, Inc. :
");
while(rs.next()){
String nomCafe= rs.getString("NOM_CAFE");
System.out.println(" " + nomCafe);
}
```

Cela devrait produire le résultat suivant :

```
Les café acheté à Acme, Inc. :
Colombian
Colombian_Decaf
```

10.0 Utilisation des Transactions

Parfois, vous ne voulez pas qu'une instruction prenne effet sans qu'une autre lui succède. Par exemple, quand le propriétaire du The Coffee Break met à jour le montant de café vendu chaque semaine, il aimerait aussi mettre à jour le montant total des ventes jusqu'à maintenant. Donc, il ne veut pas mettre à jour l'un sans mettre à jour l'autre, sinon les données ne seraient pas crédibles. La manière pour être sûr que toutes les actions et interactions voulues s'effectuent est d'utiliser une transaction. Une transaction est un jeu de une ou plusieurs instructions qui sont exécutées ensembles de façon unitaire, donc toutes les instructions sont exécutées, ou aucune.

10.1 Désactiver le mode Auto-commit

Quand une connexion est crée, elle est en mode auto-commit. Ce qui veut dire que chaque instruction SQL est traitée comme une transaction et prendra automatiquement effet après sa bonne exécution. (Pour être plus précis, une instruction SQL prend automatiquement effet sur la base de données lorsqu'elle est terminée, et non pas quand elle est exécutée. Une instruction est terminée quand tous ses résultats et toutes ses mises à jour ont été rapportées. Dans la plupart des cas, une instruction est terminée, puis les changements prennent effets sur la base de données, puis elle est exécutée).

La manière d'autoriser deux ou plusieurs instructions à être groupées dans une transaction est de mettre hors service le mode auto-commit. C'est ce que nous faisons à la ligne suivante, où conn est notre connexion active :

```
conn.setAutoCommit(false);
```

10.2 Pour que la transaction prenne effet sur la BD

Une fois que le mode auto-commit est désactivé, aucune instruction SQL ne prendra effet jusqu'à ce que la méthode commit soit employée. Toutes les instructions exécutées après l'appel de la méthode commit seront inclues dans la transaction et donc, pourront prendre effet en tant qu'unité. Le code qui suit, où conn est la connexion active, illustre la transaction :

```
conn.setAutoCommit(false);
PreparedStatement updateVentes =
conn.preparedStatement(
"UPDATE CAFE SET VENTE = ? WHERE NOM_CAFE LIKE ?");
updateVentes.setInt(1,50);
updateVentes.setString(2, "Colombian");
updateVentes.executeUpdate();
PreparedStatement updateTotal =
conn.preparedStatement(
"UPDATE CAFE SET TOTAL = TOTAL + ? WHERE" +
"NOM_CAFE LIKE ?");
updateTotal.setInt(1,50);
updateTotal.setString(2,"Colombian");
updateTotal.executeUpdate();
conn.commit();
conn.setAutoCommit(true);
```

Dans cet exemple, le mode auto-commit est désactivé pour la connexion conn, ce qui veut dire que les deux prepared statements updateVentes et updateTotal prendront effet dans la base de données quand la méthode commit est appelée. Quand la méthode commit est appelée (automatiquement quand auto-commit est activée ou explicitement quand il est désactivé), tous les changements résultants des instructions de la transaction seront permanent. Dans ce cas, cela signifie que les colonnes VENTES et TOTAL pour le café Colombian ont été modifiées à 50 (Si TOTAL était 0) et retiendra cette valeur jusqu'à ce qu'elle soit modifié par une instruction.

La dernière ligne de l'exemple précédent active le mode auto-comit, ce qui veut dire que chaques instructions prendront dès lors effet automatiquement sur la base de données quand elle sera terminée. Vous reviendrez donc à l'état par défaut, celui où vous n'avez plus à appeler la méthode commit. Il est conseillé de désactiver le mode auto-commit uniquement quand vous être en mode transaction. De cette façon, vous réduisez les chances de conflit au niveau des entrées dans la base de données avec les autres utilisateurs.

10.3 Utiliser les transactions pour préserver l'intégrité des données

En plus de grouper les instructions ensembles pour être exécutées unitairement, les transactions peuvent aider à préserver l'intégrité de données dans une table. Par exemple, imaginons qu'un employé était supposé entrer les nouveaux prix du café dans la table CAFE, mais ne l'a pas fait depuis quelques jours. Pendant ce temps, les prix ont augmentés, et le propriétaire décide de mettre à jour les nouveaux prix dans la table. L'employé se décide finalement à entrer les prix n'étant plus à jour dans la base de données en même temps que le propriétaire met à jour la table. Après avoir insérer les prix non à jour, l'employé réalise qu'ils ne sont plus valides, et donc appel la méthode rollback de l'objet Connexion afin d'annuler les effets de ses manipulations. (La méthode rollback met fin à une transaction et restaure les valeurs présentes avant qu'elles aient été mises à jour) Mais en même temps, le propriétaire exécute une instruction SELECT puis affiche les nouveaux prix. Dans cette situation, il est possible que les prix affichés par le propriétaire proviennent d'un rollback contenant les anciens prix, rendant l'affichage des prix incorrects.

Ce genre de situation peut être évitée en utilisant les transactions. Si un SGBD supporte les transactions, et la plupart le font, il fournira certains niveaux de protection contre les conflits qui peuvent survenir quand on accède à des données au même moment.

Pour éviter les conflits durant une transaction, un SGBD utilise des verrous, mécanisme bloquant l'accès aux données utilisées par la transaction aux autres utilisateurs. Une fois qu'un verrou est posé, il restera en place jusqu'à ce qu'une transaction soit envoyée vers la base de données pour qu'elle puisse y prendre effet, ou vers un rollback. Par exemple, un SGBD pourra verrouiller la ligne d'une table jusqu'à ce que les mises à jour aient été validées. L'effet de ce verrou peut être de prévenir un utilisateur contre les valeurs erronées.

La manière dont les verrous sont posés détermine le niveau d'isolation de la transaction, pouvant aller jusqu'à bloquer totalement les transactions.

Un exemple de niveau d'isolation de transaction est TRANSACTION_READ_COMMITTED qui n'autorise l'accès à aucune valeur jusqu'à ce qu'elles aient été validées dans la base de données. En d'autre terme, si le niveau d'isolation de la transaction est TRANSACTION_READ_COMMITED, le SGBD ne permettra donc pas de pouvoir lire des valeurs erronées. L'interface Connexion inclue cinq niveaux d'isolation que vous pourrez utiliser avec JDBC.

Normalement vous n'avez pas besoin de faire quoi que ce soit en ce qui concerne le niveau d'isolation de transaction, vous pouvez toujours utiliser celui par défaut de votre SGBD. JDBC vous permet de savoir quel niveau d'isolation votre SGBD utilise (par le biais de la méthode getTransactionIsolation) et vous permet aussi de définir un autre niveau que celui utilisé (grâce à setTransactionIsolation). Gardez à l'esprit que même si JDBC vous permet de définir un niveau de transaction, il faut quand même que les drivers utilisés et le SGBD utilisé le supporte.

10.4 Quand appeler la méthode RollBack

Comme mentionné plus tôt, appeler la méthode rollback annule une transaction et remet les valeurs qui ont été modifiées à leurs valeurs originales. Si vous essayez d'exécuter une ou plusieurs instructions dans une transaction et que vous avez une SQLExeption, vous devrez utilisez la méthode rollback pour annuler la transaction et la recommencer depuis le début. C'est la seule façon d'être sûr de ce qui a été pris en compte sur la base de données, et ce qui ne l'a pas été. Une SQLException vous indique que quelque chose ne fonctionne pas, mais elle ne vous indique pas ce qui à été pris en compte ou pas. Donc vous ne pouvez pas compter sur le fait que rien n'a été mis à jour sur la BD, appeler la méthode rollback est la seule façon d'être sûr.

11 Les Procédure Stockées ou Stored Procedures

Une procédure stockée est un groupe d'instructions SQL qui forme une unité logique et effectue une tâche particulière. Les procédures stockées sont utilisées pour encapsuler un jeu d'opérations ou de requêtes à exécuter sur un serveur de base de données. Par exemple, les opérations sur une base de données d'employées (embauche, augmentation etc...) peuvent être codées en procédures stockées puis exécutées. Elles peuvent être compilées puis exécutées avec différents paramètres et différents résultats, et elles possèdent plusieurs combinaisons d'entrées, de sorties et de paramètres entrées/sorties.

Les procédures Stockées sont supportées par la plupart des SGBD, mais il y a quand même quelques variations dans leur syntaxe. Pour cette raison, nous vous montrerons un simple exemple de ce à quoi une procédure stockée ressemble et comment l'invoquer avec JDBC, mais cet exemple n'est pas fait pour être exécuté.

11.0 Instructions SQL pour créer des procédures Stockées.

Cette section traite des procédures stockées très simplistes sans paramètres. Même si les procédures stockées effectuent généralement des taches bien plus complexes que celles présentées dans cet exemple, il sert tout de même à illustrer certains points à leurs propos. Comme nous l'avons dit précédemment, la syntaxe est différente pour chaque SGBD. Par exemple, certains utilisent begin . . .

end ou d'autres mots clés pour indiquer le commencement et la fin de la définition de la procédure. Dans certain SGBD, cette instruction SQL créera une procédure stockée :

```
create procedure SHOW_FOURNISSEURS
as
SELECT FOURNISSEURS.NOM_FO, CAFE.NOM_CAFE
FROM FOURNISSEURS, CAFE
WHERE FOURNISSEURS.FO_ID = CAFE.FO_ID
order by NOM_FO
```

Le code qui suit met l'instruction SQL dans une chaîne de caractères et l'assigne à la variable createProcedure, que nous utiliserons plus tard :

```
String createProcedure =
"create procedure SHOW_FOURNISSEURS" +
"as" +
"select FOURNISSEURS.NOM_FO, CAFE.NOM_CAFE" +
"from FOURNISSEURS, CAFE" +
"where FOURNISSEURS.FO_ID = CAFE.FO_ID" +
"order by NOM_FO";
```

Le fragment de code suivant utilise l'objet Connexion conn pour créer un objet Statement, qui sera utilisé pour envoyer l'instruction SQL afin de créer la procédure stockée sur la base de données :

```
Statement stmt = conn.createStatement();
stmt.executeUpdate(createProcedure);
```

La procédure SHOW_FOURNISSEURS sera compilée et stockée dans la base de données comme un objet pouvant être appelé de façon similaire à l'appel d'une méthode.

11.1 Appeler une procédure stockée avec JDBC

JDBC vous permet d'appeler une procédure stockée sur la base de données depuis une application écrite en Java. La première étape est de créer un objet CallableStatement. Comme avec les objets Statement et PreparedStatement, ceci est fait avec une connexion ouverte. Un objet CallableStatement contient l'appel d'une procédure, il ne contient pas la procédure elle-même. La première ligne de code ci-dessous crée un appel à la procédure stockée SHOW_FOURNISSEURS en utilisant la connexion conn. La partie qui est entre accolade est la syntaxe pour la procédure stockée. Quand le driver rencontre "{call SHOW_FOURNISSEURS}", il traduira cette syntaxe en SQL natif utilisé par la base de données pour appeler la procédure stockée nommée SHOW_FOURNISSEURS :

```
CallableStatement cs = conn.prepareCall("{call
SHOW_FOURNISSEURS}");
ResultSet rs = cs.executeQuery();
```

Notez que la méthode pour exécuter cs est executeQuery car cs appelle une procédure stockée qui contient une requête et produit un resultset. Si la procédure avait contenue une mise à jour ou une des instructions DDL, la méthode executeUpdate aurait été utilisée. Comme c'est parfois le cas, une procédure stockée contient plus d'une instruction SQL, qui pourrait produire plus d'un résultset, plus d'une mise à jour, ou une combinaison de result set et de mise à jour. Dans ce cas, lorsqu'il y a de multiples résultats, la méthode execute devra être utilisé pour exécuter CallableStatement.

La classe CallableStatement est une classe dérivée de PrepareStatement, donc un objet CallableStatement peut avoir des paramètres d'entrées tout comme l'objet PreparedStatement. En plus, un objet CallableStatement peut avoir des paramètres de sorties ou des paramètres qui sont fait pour l'entrée et la sortie. Les paramètres INOUT et la méthode execute sont rarement utilisées.