# JAVA 8 – ACCES AUX BASES DE DONNEES

## JSE – SEQUENCE 6

LYCEE PASTEUR MONT-ROLAND
Julian.courbez@gmail.com

# Table des matières

1. Principe de fonctionnement d'une base de données ..... 2
1.2. Terminologie ..... 2
1.3. La langage SQL ..... 2
1.3.1. Recherche d'informations ..... 3
1.3.2. Ajout d'informations ..... 4
1.3.3. Mise à jour d'informations ..... 4
1.3.4. Suppression d'informations ..... 5
2. Accès à une base de données à partir de Java ..... 5
2.1. Présentation de jdbc ..... 6
2.2. Chargement du pilote ..... 7
2.3. Etablir et manipuler la connexion ..... 7
2.3.1. Etablir la connexion ..... 7
2.3.2. Manipuler la connexion ..... 8
2.4. Exécution d'instructions SQL ..... 12
2.4.1. Exécution d'instructions de base avec l'objet Statement ..... 12
2.4.2. Exécution d'instructions paramétrées avec l'objet PreparedStatement ..... 19
2.4.3. Exécution de procédures stockées avec l'objet CallableStatement ..... 22
2.5. Utilisation des jeux d'enregistrements avec l'interface ResultSet ..... 25
2.5.1. Positionnement dans un ResultSet ..... 27
2.5.2. Lecture des données dans un ResultSet ..... 29
2.5.3. Modification des données dans un ResultSet ..... 31
2.5.4. Suppression de données dans un ResultSet ..... 34
2.5.5. Ajout de données dans un ResultSet ..... 36
2.6. Gestion des transactions ..... 37
2.6.1. Mise en œuvre des transactions ..... 38
2.6.2. Points de sauvegarde ..... 40
2.6.3. Niveaux d'isolement ..... 40

# 1. Principe de fonctionnement d'une base de données

Les bases de données sont devenues des éléments incontournables de la majorité des applications. Elles se substituent à l'utilisation de fichiers gérés par le développeur lui-même. Cet apport permet un gain de productivité important lors du développement et une amélioration significative des performances des applications. Elles facilitent également le partage d'informations entre utilisateurs. Pour pouvoir utiliser une base de données, vous devez connaître un minimum de vocabulaire lié à cette technologie.

## 1.2. Terminologie

Dans le contexte des bases de données les termes suivants sont fréquemment utilisés :

- Base de données relationnelle : une base de données relationnelle est un type de base de données qui utilise des tables pour le stockage des informations. Elles utilisent des valeurs issues de deux tables pour associer les données d'une table aux données d'une autre table. En règle générale, dans une base de données relationnelle, les informations ne sont stockées qu'une seule fois.
- Table : une table est un composant d'une base de données qui stocke les informations dans des enregistrements (lignes) et dans des champs (colonnes). Les informations sont en général regroupées par catégorie au niveau d'une table. Par exemple, nous aurons la table des Clients, des Produits, ou des Commandes.
- Enregistrement : l'enregistrement est l'ensemble des informations relatives à un élément d'une table. Les enregistrements sont les équivalents au niveau logique des lignes d'une table. Par exemple un enregistrement de la table Clients contient les caractéristiques d'un client particulier.
- Champ : un enregistrement est composé de plusieurs champs. Chaque champ d'un enregistrement contient une seule information sur l'enregistre-ment. Par exemple un enregistrement Client peut contenir les champs CodeClient, Nom, Prénom...
- Clé primaire : une clé primaire est utilisée pour identifier de manière unique chaque ligne d'une table. La clé primaire est un champ ou une combinaison de champs dont la valeur est unique dans la table. Par exemple, le champ CodeClient est la clé primaire de la table Client. Il ne peut pas y avoir deux clients ayant le même code.
- Clé étrangère : une clé étrangère représente un ou plusieurs champs d'une table qui font référence aux champs de la clé primaire d'une autre table. Les clés étrangères indiquent la manière dont les tables sont liées.
- Relation : une relation est une association établie entre des champs communs dans deux tables. Une relation peut être de un à un, un à plusieurs ou plusieurs à plusieurs. Grâce aux relations, les résultats de requêtes peuvent contenir des données issues de plusieurs tables. Une relation de un à plusieurs entre la table Client et la table Commande permet à une requête de renvoyer toutes les commandes correspondant à un client.

## 1.3. La langage SQL

Avant de pouvoir écrire une application Java utilisant des données, vous devez être familiarisé avec le langage SQL (*Structured Query Language*). Ce langage permet de dialoguer avec la base de données. Il existe différentes versions du langage SQL en fonction de la base de données utilisée. Cependant SQL dispose également d'une syntaxe élémentaire normalisée indépendante de toute base de données.

### 1.3.1. Recherche d'informations

Le langage SQL permet de spécifier les enregistrements à extraire ainsi que l'ordre dans lequel vous souhaitez les extraire. Vous pouvez créer une instruction SQL qui extrait des informations de plusieurs tables simultanément, ou vous pouvez créer une instruction qui extrait uniquement un enregistrement spécifique. L'instruction `SELECT` est utilisée pour renvoyer des champs spécifiques d'une ou de plusieurs tables de la base de données.

L'instruction suivante renvoie la liste des noms et prénoms de tous les enregistrements de la table Client.

```
SELECT Nom,Prenom FROM Client
```

Vous pouvez utiliser le symbole * à la place de la liste des champs pour lesquels vous souhaitez la valeur.

```
SELECT * FROM Client
```

Vous pouvez limiter le nombre d'enregistrements sélectionnés en utilisant un ou plusieurs champs pour filtrer le résultat de la requête. Différentes clauses sont disponibles pour exécuter ce filtrage.

#### *Clause WHERE*

Cette clause permet de spécifier la liste des conditions que devront remplir les enregistrements pour faire partie des résultats retournés. L'exemple suivant permet de retrouver tous les clients habitant Nantes.

```
SELECT * FROM Client WHERE Ville='Nantes'
```

La syntaxe de la clause nécessite l'utilisation de simple cote pour la délimitation des chaînes de caractères.

#### *Clause WHERE ... IN*

Vous pouvez utiliser la clause `WHERE ... IN` pour renvoyer tous les enregistrements qui répondent à une liste de critères. Par exemple, vous pouvez rechercher tous les clients habitant en France ou en Espagne.

```
SELECT * FROM Client WHERE Pays IN ('France','Espagne')
```

#### *Clause WHERE ... BETWEEN*

Vous pouvez également renvoyer une sélection d'enregistrements qui se situent entre deux critères spécifiés. La requête suivante permet de récupérer la liste des commandes passées au mois de novembre 2005.

```
SELECT * from Commandes WHERE DateCommande BETWEEN '01/11/05' AND '30/11/05'
```

#### *Clause WHERE ... LIKE*

Vous pouvez utiliser la clause `WHERE ... LIKE` pour renvoyer tous les enregistrements pour lesquels il existe une condition particulière pour un champ donné. Par exemple, la syntaxe suivante sélectionne tous les clients dont le nom commence par un d :

```
SELECT * FROM Client WHERE Nom LIKE 'd%'
```

Dans cette instruction, le symbole % est utilisé pour remplacer une séquence de caractères quelconque.

*Clause ORDER BY …*

Vous pouvez utiliser la clause `ORDER BY` pour renvoyer les enregistrements dans un ordre particulier. L'option `ASC` indique un ordre croissant, l'option `DESC` indique un ordre décroissant. Plusieurs champs peuvent être spécifiés comme critère de tri. Ils sont analysés de la gauche vers la droite. En cas d'égalité sur la valeur d'un champ, le champ suivant est utilisé.

SELECT * FROM Client ORDER BY Nom DESC,Prenom ASC

Cette instruction retourne les clients triés par ordre décroissant sur le nom et en cas d'égalité par ordre croissant sur le prénom.

### 1.3.2. Ajout d'informations

La création d'enregistrements dans une table se fait par la commande `INSERT INTO`. Vous devez indiquer la table dans laquelle vous souhaitez insérer une ligne, la liste des champs pour lesquels vous spécifiez une valeur et enfin la liste des valeurs correspondantes. La syntaxe complète est donc la suivante :

INSERT INTO client (codeClient,nom,prenom) VALUES (1000,'Dupond','Pierre')

Lors de l'ajout de ce nouveau client, seul le nom et le prénom seront renseignés dans la table. Les autres champs prendront la valeur `NULL`. Si la liste des champs n'est pas indiquée, l'instruction `INSERT` exige que vous spécifiez une valeur pour chaque champ de la table. Vous êtes donc obligé d'utiliser le mot clé `NULL` pour indiquer que pour un champ particulier il n'y a pas d'information. Si la table Client est composée de cinq champs (codeClient,nom,prenom,adresse,pays) l'instruction précédente peut être écrite avec la syntaxe suivante :

INSERT INTO client VALUES (1000,'Dupond','Pierre',NULL,NULL)

À noter que dans ce cas les deux mot clés NULL sont obligatoires pour les champs adresse et pays.

### 1.3.3. Mise à jour d'informations

La modification des champs pour des enregistrements existants s'effectue par l'instruction `UPDATE`. Cette instruction peut mettre à jour plusieurs champs de plusieurs enregistrements d'une table à partir des expressions qui lui sont fournies. Vous devez fournir le nom de la table à mettre à jour ainsi que la valeur à affecter aux différents champs. La liste est indiquée par le mot clé `SET` suivi de l'affectation de la nouvelle valeur aux différents champs. Si vous voulez que les modifications ne portent que sur un ensemble limité d'enregistrements, vous devez spécifier la clause `WHERE` afin de limiter la portée de la mise à jour. Si aucune clause `WHERE` n'est indiquée la modification se fera sur l'ensemble des enregistrements de la table.

Par exemple pour modifier l'adresse d'un client particulier, vous pouvez utiliser l'instruction suivante :

UPDATE Client SET adresse= '4 rue de Paris 44000 Nantes'

WHERE codeClient=1000

Si la modification porte sur l'ensemble des enregistrements de la table, la clause `WHERE` est inutile. Par exemple si vous souhaitez augmenter le prix unitaire de tous vos articles, vous pouvez utiliser l'instruction suivante :

UPDATE CATALOGUE SET prixUnitaire=prixUnitaire*1.1

### 1.3.4. Suppression d'informations

L'instruction `DELETE FROM` permet de supprimer un ou plusieurs enregistre-ments d'une table. Vous devez au minimum fournir le nom de la table sur laquelle va se faire la suppression. Si vous n'indiquez pas plus de précisions, dans ce cas toutes les lignes de la table sont supprimées. En général une clause `WHERE` est ajoutée pour limiter l'étendue de la suppression. La commande suivante efface tous les enregistrements de la table Client :

```
DELETE FROM Client
```

La commande suivante est moins radicale et ne supprime qu'un enregistrement particulier :

```
DELETE FROM Client WHERE codeClient=1000
```

Le langage SQL est bien sûr beaucoup plus complet que cela et ne se résume pas à ces cinq instructions. Néanmoins, elles sont suffisantes pour la manipulation de données à partir de Java. Si vous souhaitez approfondir l'apprentissage du langage SQL je vous conseille de consulter un des ouvrages disponibles dans la même collection traitant de ce sujet de manière plus poussée.

## 2. Accès à une base de données à partir de Java

Lorsque l'on souhaite manipuler une base de données à partir d'un langage de programmation, deux solutions sont disponibles :

- Communiquer directement avec la base de données.
- Utiliser une couche logicielle assurant le dialogue avec la base de données.

La première solution comporte plusieurs exigences.

- Vous devez parfaitement maîtriser la programmation réseau.
- Vous devez également connaître en détail le protocole utilisé par la base de données.
- Ce type de développement est souvent très long et parsemé d'embûches. Par exemple les spécifications du protocole sont-elles accessibles ?
- Tout votre travail devra être recommencé si vous changez de type de base de données car les protocoles ne sont bien sûr pas compatibles d'une base de données à l'autre. Pire parfois même d'une version à l'autre d'une même base de données.

La deuxième solution est bien sûr préférable et c'est celle-ci que les concepteurs de Java ont choisie. Ils ont donc développé la bibliothèque jdbc pour l'accès à une base de données. Plus précisément la bibliothèque jdbc est composée de deux parties. La première partie contenue dans le package java.sql est essentiellement composée d'interfaces. Ces interfaces sont implé-mentées par le pilote jdbc. Ce pilote n'est pas développé par Sun mais en général par le concepteur de la base de données. C'est effectivement ce dernier qui maîtrise le mieux la technique pour communiquer avec la base de données. Il existe quatre types de pilotes jdbc avec des caractéristiques et des perfor-mances différentes.

- Type 1 : Pilote jdbc-odbc
  Ce type de pilote n'est pas spécifique à une base de données mais il traduit simplement les instructions jdbc en instructions odbc. C'est ensuite le pilote odbc qui assure la communication avec la base de données. Cette solution n'offre que des performances médiocres. Ceci est principalement lié au nombre de couches logicielles mises en œuvre. Les fonctionnalités jdbc sont également limitées par les fonctionnalités de la couche odbc. Ce type de pilote n'est plus disponible à partir de la version 8 de Java. Oracle recom-mande l'utilisation du pilote spécifique à la base de données que vous souhaitez utiliser. Il est généralement disponible sur le site du concepteur de la base de données.

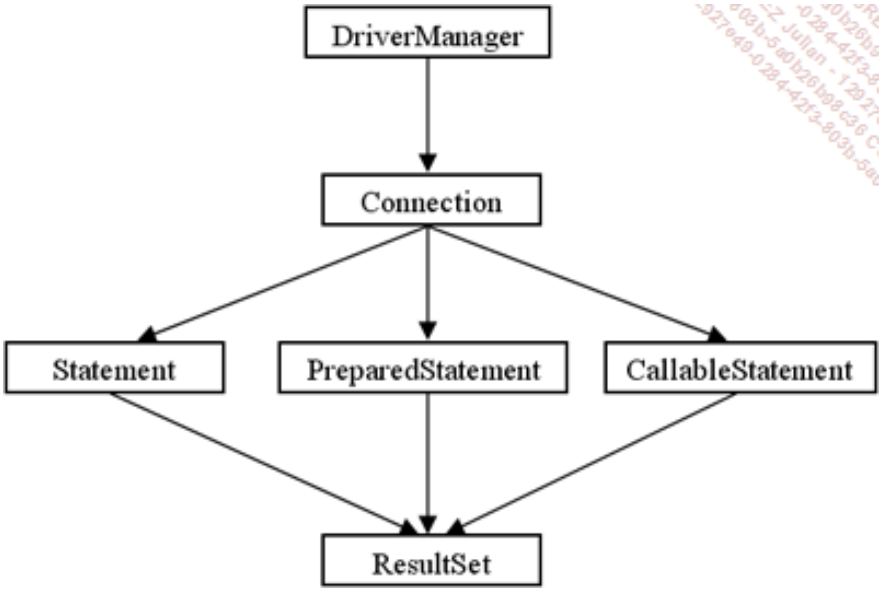
- Type 2 : Pilote natif
  Ce type de pilote n'est pas entièrement écrit en Java. La partie de ce pilote écrite en Java effectue simplement des appels vers des fonctions du pilote natif. Ces appels sont effectués grâce à l'API JNI (*Java Native Interface*). Comme pour les pilotes de type 1, il y a donc une traduction nécessaire entre le code Java et la base de données. Cependant, ce type de pilote est tout de même plus efficace que les pilotes jdbc-odbc.
- Type 3 : pilote utilisant un serveur intermédiaire
  Ce pilote développé entièrement en Java ne communique pas directement avec la base de données mais s'adresse à un serveur intermédiaire. Le dialogue entre le pilote et le serveur intermédiaire est standard quel que soit le type de base de données. C'est ensuite ce serveur intermédiaire qui transmet les requêtes en utilisant un protocole spécifique à la base de données.
- Type 4 : pilote entièrement écrit en Java
  Ce type de pilote représente la solution idéale puisqu'il n'y a aucun intermédiaire. Le pilote transmet directement les requêtes à la base de données en utilisant le protocole propre à la base de données. La majorité des pilotes sont maintenant de ce type.

Fggg

## 2.1. Présentation de jdbc

La bibliothèque jdbc fournit un ensemble de classes et surtout d'interfaces permettant la manipulation d'une base de données. Ces éléments représentent tout ce dont vous avez besoin pour accéder aux données à partir d'une application Java.

Le schéma ci-dessous reprend le cheminement logique depuis le pilote jusqu'aux données.




La classe `DriverManager` est notre point de départ. C'est elle qui assure la liaison avec le pilote. C'est par son intermédiaire que nous pouvons obtenir une connexion vers la base de données. Celle-ci est représentée par une instance de classe implémentant l'interface `Connection`. Cette connexion est ensuite utilisée pour transmettre des instructions vers la base. Les requêtes simples sont exécutées grâce à l'interface `Statement`, les requêtes avec paramètres le sont avec l'interface `PreparedStatement` et les procédures stockées avec l'interface `CallableStatement`.

Les éventuels enregistrements sélectionnés par l'instruction SQL sont accessibles avec un élément `Resultset`.

## 2.2. Chargement du pilote

La première étape indispensable est d'obtenir le pilote jdbc adapté à votre base de données. En général, ce pilote est disponible en téléchargement sur le site du concepteur de la base de données. Pour nos exemples, nous utiliserons le pilote fourni par MySQL AB / Oracle pour accéder à un serveur de base de données MySQL https://dev.mysql.com/downloads/connector/j/

Après décompression du fichier, vous devez obtenir un répertoire contenant le pilote lui-même sous forme d'une archive Java (mysql-connector-java-X.X.XX-bin.jar) et de nombreux fichiers d'aide sur l'utilisation de ce pilote. Le fichier archive contient les classes développées par Microsoft qui implémentent les différentes interfaces jdbc. Ce fichier devra bien sûr être accessible au moment de la compilation et de l'exécution de l'application.

Ce pilote doit ensuite être chargé grâce à la méthode `forName` de la classe `Class`. Cette méthode attend comme paramètre une chaîne de caractères contenant le nom du pilote. Il est à ce niveau indispensable de parcourir la documentation du pilote pour obtenir le nom de la classe. Dans notre cas, cette classe porte le nom suivant :

```
com.mysql.jdbc.Driver
```

L'enregistrement du pilote se fait donc avec l'instruction suivante :

```
Class.forName("com.mysql.jdbc.Driver");
```

Les chaînes de caractères passées comme paramètres représentent des noms de classes, elles sont donc sensibles à la casse. L'instruction `forName` doit d'ailleurs être protégée par un bloc `try catch`car elle est susceptible de déclencher une exception de type `ClassNotFoundException`.

## 2.3. Etablir et manipuler la connexion

Après son chargement, le pilote est maintenant capable de nous fournir une connexion vers le serveur de base de données.

### 2.3.1. Etablir la connexion

La méthode `getConnection` de la classe `DriverManager` se charge de ce travail. Celle-ci propose trois solutions pour établir la connexion. Chacune des versions de cette méthode attend comme paramètre une chaîne de caractères représentant une URL contenant les informations nécessaires pour établir la connexion. Le contenu de cette chaîne de caractères est spécifique à chaque pilote et il faut à nouveau consulter la documentation pour obtenir la bonne syntaxe. Le début de cette chaîne de caractères est cependant standard avec la forme suivante `jdbc:nomDuProtocole`. Le nom du protocole est propre à chaque pilote et c'est grâce à ce nom que la méthode `getConnection` est capable d'identifier le bon pilote. Le reste de la chaîne est lui complètement spécifique à chaque pilote. Il contient en général les informations permettant d'identifier le serveur et la base de données sur ce serveur vers laquelle la connexion doit être établie. Pour le pilote MySQL, la syntaxe de base est la suivante :

```
jdbc:mysql://hostname:port/dbname?user=username&password=userpassword ;
```

En cas de succès, la méthode `getConnection` retourne une instance de classe implémentant l'interface `Connection`. Dans certains cas, il est préférable d'utiliser la deuxième version de la méthode `getConnection` car celle-ci permet d'indiquer le nom et le mot de passe utilisés pour établir la connexion comme paramètres séparés de l'URL de connexion.

Voici un exemple permettant d'établir une connexion vers une base de données.

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnexionDirecte
{
public static void main(String[] args)
{
     try
     {
          Class.forName("com.mysql.jdbc.Driver");
     }
     catch (ClassNotFoundException e)
     {
          System.out.println("erreur pendant le chargement du pilote");
     }
     Connection cnxDirect=null;
     try
     {
           cnxDirect=DriverManager.getConnection("jdbc:mysql://localhost:8889/s6dat
abase?user=root&password=root");
     }
     catch (SQLException e)
     {
          System.out.println("erreur pendant la connexion : " + e);
     }
     }
}
```

### 2.3.2. Manipuler la connexion

Une connexion est ouverte dès sa création. Il n'y a donc pas de méthode permettant d'ouvrir une connexion. Par contre, une connexion peut être fermée en appelant la méthode `close`. Après la fermeture d'une connexion, il n'est plus possible de l'utiliser et elle doit être recréée pour pouvoir être à nouveau utilisable. La fonction `isClosed` permet de vérifier si une connexion est fermée. Si cette fonction retourne un boolean qui est égal à `false`, on peut donc légitimement penser que la connexion est ouverte et permet donc de dialoguer avec la base de données. Ce n'est en fait pas toujours le cas. La connexion peut parfois être dans un état intermédiaire : elle n'est pas fermée mais elle ne peut pas être utilisée pour transférer des instructions vers le serveur. Pour vérifier la disponibilité de la connexion, vous pouvez utiliser la méthode `isValid`. Cette méthode teste réellement la disponibilité de la connexion en essayant d'envoyer une instruction SQL et en vérifiant qu'elle obtient bien une réponse de la part du serveur. Cette méthode n'est pas implémentée dans tous les pilotes et si elle ne l'est pas, son appel déclenche une exception du

type `java.lang.UnsupportedOperationException` ou une erreur du type `java.lang.AbstractMethodError`.

Si vous effectuez uniquement des opérations de lecture sur la base de données vous pouvez optimiser la communication en précisant que la connexion est en lecture seule. La méthode `setReadOnly` permet de modifier ce paramétrage de la connexion. L'état peut ensuite être testé en utilisant la méthode `isReadOnly`. Pour certains pilotes cette fonctionnalité n'est pas implémentée et la méthode `setReadOnly` n'a aucun effet. La fonction suivante vérifie si cette fonctionnalité est disponible pour la connexion qui lui est passée comme paramètre.

```
public static void testLectureSeule(Connection cnx)
  {
      boolean etat;
      try
      {
            etat = cnx.isReadOnly();
            cnx.setReadOnly(!etat);
            if (cnx.isReadOnly() != etat)
            {
                  System.out.println("le mode lecture seule est pris en
charge par ce pilote");
            }
            else
            {
                  System.out.println("le mode lecture seule n\'est pas
pris en charge par ce pilote");
            }
            cnx.setReadOnly(etat);
      }
      catch (SQLException e)
      {
           e.printStackTrace();
      }
  }
```

Lors de l'exécution d'une instruction SQL, le serveur peut détecter des problèmes et de ce fait générer des avertissements. Ces avertissements peuvent être récupérés par la méthode `getWarnings` de la connexion. Cette méthode retourne un objet `SQLWarning` représentatif du problème rencontré par le serveur. Si plusieurs problèmes sont rencontrés par le serveur celui-ci génère plusieurs objets `SQLWarning` chaînés les uns aux autres. La méthode `getNextWarning` permet d'obtenir l'élément suivant ou null si la liste est terminée. La liste peut être vidée avec la méthode `clearWarnings`. La fonction suivante affiche tous les avertissements reçus par la connexion.

```
public static void affichageWarnings(Connection cnx)
  {
      SQLWarning avertissement;
      try
      {
            avertissement=cnx.getWarnings();
            if (avertissement==null)
            {
                System.out.println("il n\'y a pas d\'avertissement");
            }
            else
            {
                 while (avertissement!=null)
                 {
                   System.out.println(avertissement.getMessage());
                   System.out.println(avertissement.getSQLState());
                   System.out.println(avertissement.getErrorCode());
                   avertissement=avertissement.getNextWarning();
                 }
            }
            cnx.clearWarnings();
      }
      catch (SQLException e)
      {
                 e.printStackTrace();
      }
  }
```

En général, lors de la création de l'url de connexion, l'un des paramètres de cette URL détermine le nom de la base vers laquelle vous souhaitez établir une connexion. La modification du nom de la base de données à laquelle vous êtes connecté se réalise avec la méthode `setCatalog` à laquelle il faut fournir le nom d'une autre base présente sur le même serveur. Il faut bien sûr que le compte avec lequel vous avez ouvert la connexion dispose des droits d'accès suffisants pour cette base de données. À noter qu'avec cette méthode, nous changeons de base de données mais la connexion concerne toujours le même serveur. Il n'y a aucun moyen de changer de serveur sans créer une nouvelle connexion.

Le code suivant :

```
System.out.println("base actuelle : " +cnxDirect.getCatalog());
  System.out.println("changement de base de données");
  cnxDirect.setCatalog("s6database2");
```

```
System.out.println("base actuelle : " +cnxDirect.getCatalog());
```

nous affiche :

```
base actuelle : s6database
changement de base de données
base actuelle : s6database2
```

La structure de la base peut également être obtenue avec la méthode `getMetaData`. Cette méthode retourne un objet `DatabaseMetaData`fournissant de très nombreuses informations sur la structure de la base. La fonction suivante affiche une analyse rapide de ces informations

```
public static void infosBase(Connection cn)
{
    ResultSet rs;
    DatabaseMetaData dbmd;
    try
    {
         dbmd=cn.getMetaData();
         System.out.println("type de base : " +
dbmd.getDatabaseProductName());
         System.out.println("version: " +
dbmd.getDatabaseProductVersion());
         System.out.println("nom du pilote : " + dbmd.getDriverName());
         System.out.println("version du pilote : " +
dbmd.getDriverVersion());
          System.out.println("nom de l\'utilisateur : " +
          System.out.println("url de connexion : " + dbmd.getURL());
          rs=dbmd.getTables(null,null,"%",null);
          System.out.println("structure de la base");
          System.out.println("base\tschema\tnom table\ttype table");
          while(rs.next())
          {
            for (int i = 1; i <=4 ; i++)
            {
                 System.out.print(rs.getString(i)+"\t");
            }
            System.out.println();
          }
          rs.close();
          rs=dbmd.getProcedures(null,null,"%");
          System.out.println("les procédures stockées");
          System.out.println("base\tschema\tnom procedure");
```

```
            while(rs.next())
            {
              for (int i = 1; i <=3 ; i++)
              {
                    System.out.print(rs.getString(i)+"\t");
              }
              System.out.println();
            }
            rs.close();
      }
      catch (SQLException e)
      {
              e.printStackTrace();
      }
}
```

Toutes ces méthodes peuvent un jour ou l'autre vous rendre service mais le but principal d'une connexion est de permettre l'exécution d'instructions SQL. C'est donc elle qui va nous fournir les objets nécessaires pour l'exécution de ces instructions. Trois types d'instructions SQL peuvent être exécutés :

- Les requêtes simples
- Les requêtes précompilées
- Les procédures stockées.

À chacun de ces types correspond un objet JDBC :

- `Statement` pour les requêtes simples
- `PreparedStatement` pour les requêtes précompilées
- `CallableStatement` pour les procédures stockées.

Et enfin chacun des ces objets peut être obtenu par une méthode différente de la connexion :

- `createStatement` pour les objets `Statement`
- `prepareStatement` pour les objets `PreparedStatement`
- `prepareCall` pour les objets `CallableStatement`.

## 2.4. Exécution d'instructions SQL

### 2.4.1. Exécution d'instructions de base avec l'objet Statement

Cet objet est obtenu par la méthode `createStatement` de la connexion. Deux versions de cette méthode sont disponibles ; la première n'attend aucun paramètre. Dans ce cas, si l'objet `Statement` est utilisé pour exécuter une instruction SQL générant un jeu d'enregistrements (`select`), celui-ci sera en lecture seule et avec un défilement en avant.

Les informations présentes dans ce jeu d'enregistrements ne pourront pas être modifiées et le parcours du jeu d'enregistrements ne pourra se faire que du premier enregistrement vers le dernier enregistrement. La deuxième version permet de choisir les caractéristiques du jeu d'enregistrements généré. Elle accepte deux paramètres. Le premier détermine le type du jeu d'enregistre-ments.

Les constantes suivantes sont définies :

- `ResultSet.TYPE_FORWARD_ONLY` : le jeu d'enregistrements sera à défilement en avant seulement.
- `ResultSet.TYPE_SCROLL_INSENSITIVE` : le jeu d'enregistrements pourra être parcouru dans les deux sens mais sera insensible aux changements effectués dans la base de données par d'autres utilisateurs.
- `ResultSet.TYPE_SCROLL_SENSITIVE` : le jeu d'enregistrements pourra être parcouru dans les deux sens et sera sensible aux changements effectués dans la base de données par d'autres utilisateurs.

Le second détermine les possibilités de modification des informations contenues dans le jeu d'enregistrements. Les deux constantes suivantes sont définies :

- `ResultSet.CONCUR_READ_ONLY` : les enregistrements sont en lecture seule.
- `ResultSet.CONCUR_UPDATABLE` : les enregistrements peuvent être modifiés dans le jeu d'enregistrements.

Cet objet est le plus élémentaire permettant l'exécution d'instructions SQL. Il peut prendre en charge l'exécution de n'importe quelle instruction SQL. Vous pouvez donc exécuter aussi bien des instructions DDL (*Data Definition Language*) que des instructions DML (*Data Manipulation Language*). Il faut simplement choisir dans cet objet la méthode la plus adaptée pour l'exécution du code SQL. Le choix de cette méthode est dicté par le type de résultat que doit fournir l'instruction SQL. Quatre méthodes sont disponibles :

- `public boolean execute(String sql)` : cette méthode permet l'exécution de n'importe quelle instruction SQL. Le `boolean`retourné par cette méthode indique si un jeu d'enregistrements a été généré (`true`) ou si simplement l'instruction a modifié des enregistrements dans la base de données (`false`). Si un jeu d'enregistrements est généré, il peut être obtenu par la méthode `getResultSet`. Si des enregistrements ont été modifiés dans la base de données, la méthode `getUpdateCount` retourne le nombre d'enregistrements modifiés. La fonction suivante illustre l'utilisation de ces méthodes.

```
public static void testExecute(Connection cnx)
   {
      Statement stm;
      BufferedReader br;
      String requete;
      ResultSet rs;
      boolean resultat;
      try
      {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
            br=new BufferedReader(new InputStreamReader(System.in));
            System.out.println("saisir votre instruction SQL :");
```

```
            requete=br.readLine();
            resultat=stm.execute(requete);
            if (resultat)
            {
                  System.out.println("votre instruction a généré
un jeu d\'enregistrements");
                  rs=stm.getResultSet();
                  rs.last();
                  System.out.println("il contient "
+ rs.getRow() + " enregistrements");
            }
            else
            {
                  System.out.println("votre instruction a modifié
des enregistrements dans la base");
                  System.out.println("nombre d'enregistrements
modifiés :"
+ stm.getUpdateCount());
            }
      }
      catch (SQLException e)
      {
         System.out.println("votre instruction n\'a pas fonctionné
correctement");
      }
      catch (IOException e)
      {
            e.printStackTrace();
      }
   }
```

`public Resultset executeQuery(String requete)` : cette méthode est conçue spécialement pour l'exécution d'instructions `select`. Le jeu d'enregistrements est disponible directement comme valeur retournée par la fonction.

- `public int executeUpdate(String requête)` : cette méthode est parfaitement adaptée pour l'exécution d'instructions modifiant le contenu de la base de données comme les instructions `insert`, `update`, `delete`. L'entier retourné par cette fonction indique le nombre d'enregistrements affectés par la modification.
- `public int[] executeBatch()` : cette méthode permet d'exécuter un ensemble d'instructions SQL par lot. Le lot d'instructions à exécuter doit être préparé au préalable avec les méthodes `addBatch` et `clearBatch`. La première reçoit comme paramètre une chaîne

de caractères représentant une instruction SQL à ajouter au lot. La seconde permet de réinitialiser le lot d'instructions. Il n'est pas possible de supprimer une instruction particulière du lot. Vous ne devez pas ajouter au lot d'instruction SQL générant un jeu de résultats, car dans ce cas la méthode `updateBatch` déclenche une exception de type `BatchUpdateException`. Cette fonction retourne un tableau d'entiers permettant d'obtenir une information sur l'exécution de chacune des requêtes du lot. Chaque case du tableau contient un entier représentatif du résultat de l'exécution de la requête correspondante dans le lot. Une valeur supérieure ou égale à 0 indique un fonctionnement correct de l'instruction et représente le nombre d'enregistrements modifiés. Une valeur égale à la constante `Statement.EXECUTE_FAILED` indique que l'exécution de l'instruction a échoué. Dans ce cas, certains pilotes arrêtent l'exécution du lot alors que d'autres continuent avec l'instruction suivante du lot. Une valeur égale à la constante `Statement.SUCCESS_NO_INFO` indique que l'instruction a été exécutée correctement mais que le nombre d'enregistrements modifiés ne peut pas être déterminé. Cette méthode est très pratique pour exécuter des modifications sur plusieurs tables liées. Ce pourrait par exemple être le cas dans une application de gestion de commerciale avec une table pour les commandes et une table pour les lignes de commande. La suppression d'une commande doit dans ce cas entraîner la suppression de toutes les lignes correspondantes. La fonction suivante vous permet de saisir plusieurs instructions SQL et de les exécuter par lot.

```
public static void testExecuteBatch(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String requete="";
    int[] resultats;
    try
    {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
      br=new BufferedReader(new InputStreamReader(System.in));
           System.out.println("saisir vos instructions SQL puis
run pour exécuter le lot :");
           requete=br.readLine();
           while (!requete.equalsIgnoreCase("run"))
           {
                 stm.addBatch(requete);
                 requete=br.readLine();
           }
           System.out.println("exécution du lot d\'instructions");
           resultats=stm.executeBatch();
           for (int i=0; i<resultats.length;i++)
```

```
            {
                  switch (resultats[i])
                  {
                  case Statement.EXECUTE_FAILED:
                        System.out.println("l\'exécution de
l'instruction " + i + " a échoué");
                        break;
                  case Statement.SUCCESS_NO_INFO:
                        System.out.println("l\'exécution de
l'instruction " + i + " a réussi");
                        System.out.println("le nombre
d\'enregistrements modifiés est inconnu");
                        break;
                  default:
                        System.out.println("l\'exécution de
l'instruction " + i + " a réussi");
                        System.out.println("elle a modifié " +
resultats[i] + " enregistrements");
                        break;
                  }



            }
      }
      catch (SQLException e)
      {
            e.printStackTrace();
      }
      catch (IOException e)
      {
            e.printStackTrace();
      }
}
```

Plusieurs autres méthodes vont intervenir sur le comportement de l'objet `Statement` :

- `public void setQueryTimeOut(int duree)` : cette méthode indique la durée maximale allouée pour l'exécution d'une instruction SQL avant le déclenchement d'une exception.
- `public void close()` : lorsqu'un objet `Statement` n'est plus utile dans une application, il est préférable de le fermer explicitement en appelant cette méthode. Celle-ci provoque la

libération de toutes les ressources utilisées par cet objet. La fermeture d'un objet `statement` provoque également la fermeture du jeu d'enregistrements associé.

- `public void setMaxRows(int nombre)` : cette méthode limite le nombre de lignes des jeux d'enregistrements générés par cet objet `Statement`. Si une instruction SQL génère un jeu d'enregistrements comportant plus de lignes, les lignes excédentaires sont tout simplement ignorées (sans plus d'information).
- `public void setMaxFieldSize(int taille)` : cette méthode limite la taille de certains types de champs dans le jeu d'enregistrements. Les types de champs concernés sont les champs d'une base de données pouvant avoir une taille variable comme par exemple les champs caractères ou binaires. Les données excédentaires sont simplement ignorées. Les champs concernés peuvent de ce fait être inutilisables dans l'application.
- `public void setFetchSize(int nbLignes)` : lorsqu'une instruction SQL génère un jeu d'enregistrements, les données correspondantes sont transférées du serveur de base de données vers la mémoire de l'application Java. Ce transfert est effectué par blocs en fonction de l'utilisation des données. Cette méthode indique au pilote le nombre de lignes de chaque bloc transféré de la base de données vers l'application.
- `public boolean getMoreResults()` : si la méthode `execute` est utilisée pour exécuter plusieurs instruction SQL, par exemple deux instructions `select`, il y a dans ce cas génération de deux jeux d'enregistrements. Le premier est obtenu par la méthode `getResultSet`. Le second ne sera accessible qu'après avoir appelé la méthode `getMoreResults`. Cette fonction permet de déplacer le pointeur sur le résultat suivant et retourne un `boolean` égal à `true` si le résultat suivant est un jeu d'enregistrements. Si c'est le cas, il est lui aussi accessible par la méthode `getResultSet`. Si la fonction `getMoreResults` retourne un boolean false, c'est que le résultat suivant n'est pas un jeu d'enregistrements ou qu'il n'y a simplement pas de résultat suivant. Pour lever le doute, il faut donc appeler la fonction `getUpdateCount` et vérifier la valeur retournée par cette fonction. Si cette valeur est égale à -1 c'est qu'il n'y a pas de résultat suivant sinon la valeur retournée représente le nombre d'enregistrements modifiés dans la base de données.

La fonction suivante permet l'exécution de plusieurs instructions SQL séparées par des points-virgules :

```
public static void testExecuteMultiple(Connection cnx)
{
    Statement stm;
    BufferedReader br;
    String requete;
    ResultSet rs;
    boolean resultat;
    try
    {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
          br=new BufferedReader(new InputStreamReader(System.in));
          System.out.println("saisir vos instruction SQL séparées
par ; :");
```

```
          requete=br.readLine();
          resultat=stm.execute(requete);
          int i=1;
          // traitement du résultat généré par la première
          // instruction
          if (resultat)
          {
                System.out.println("votre instruction N° " + i + " a
généré un jeu d\'enregistrements");
                rs=stm.getResultSet();
                rs.last();
                System.out.println("il contient " + rs.getRow() + "
enregistrements");
          }
          else
          {
                System.out.println("votre instruction N° " + i + " a
modifié des enregistrements dans la base");
                System.out.println("nombre d'enregistrements
modifiés :" + stm.getUpdateCount());
          }
          i++;
          // déplacement du pointeur sur un éventuel résultat
          // suivant
          resultat=stm.getMoreResults();
          // boucle tant qu'il y a encore un résultat de type jeu
          // d'enregistrement -> resultat==true
          // ou qu'il y a encore un resultat de type nombre
          // d'enregistrements modifiés -> getUpdateCount != -1
          while (resultat || stm.getUpdateCount()!=-1)
          {
                if (resultat)
                {
                     System.out.println("votre instruction N° " +
i + " a généré un jeu d\'enregistrements");
                     rs=stm.getResultSet();
                     rs.last();
                     System.out.println("il contient " + rs.getRow()
+ " enregistrements");
```

```
                }
                else
                {
                     System.out.println("votre instruction N° " +
i + " a modifié des enregistrements dans la base");
                     System.out.println("nombre d'enregistrements
modifiés :" + stm.getUpdateCount());
                }
                i++;
                // déplacement du pointeur sur un éventuel résultat
                // suivant
                resultat=stm.getMoreResults();
          }
    }
    catch (SQLException e)
    {
          System.out.println("votre instruction n\'a pas fonctionné
correctement");
          e.printStackTrace();
    }
    catch (IOException e)
    {
          e.printStackTrace();
    }
}
```

### 2.4.2. Exécution d'instructions paramétrées avec l'objet PreparedStatement

Il arrive fréquemment d'avoir à faire exécuter plusieurs fois une requête SQL avec juste une petite modification entre deux exécutions. L'exemple classique correspond à une requête de sélection avec une restriction.

```
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
resultat=stm.execute("select * from customers where customerId=
'ALFKI'");
```

La valeur sur laquelle porte la restriction est en général saisie par l'utilisateur de l'application et dans ce cas disponible dans une variable. La première solution qui vient à l'esprit consiste à construire la requête SQL par concaténation de plusieurs chaînes de caractères.

```
public static void testRequeteConcat(Connection cnx)
{
    Statement stm;
```

```
    BufferedReader br;
    String requete;
    String code;
    ResultSet rs;
    boolean resultat;
    try
    {
stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
         br=new BufferedReader(new InputStreamReader(System.in));
         System.out.println("saisir le code du client recherché :");
         code=br.readLine();
         requete="select * from customers where customerID=\'" + code
+"\'";
         resultat=stm.execute(requete);
    }
    catch (SQLException e)
    {
         e.printStackTrace();
    } catch (IOException e) {
          // TODO Bloc catch autogénéré
          e.printStackTrace();
    }
}
```

Cette solution présente plusieurs inconvénients :

- La concaténation de chaînes de caractères consomme beaucoup de ressources.
- Le serveur doit analyser à chaque fois une nouvelle requête.
- La syntaxe va rapidement devenir complexe si plusieurs chaînes doivent être concaténées. Il faut toujours conserver à l'esprit qu'au final nous devons obtenir une instruction SQL correcte.

Les concepteurs de JDBC ont prévu une solution efficace pour pallier ces inconvénients. L'objet `PreparedStatement` apporte une solution efficace à ce problème en nous permettant la création de requêtes avec paramètres. Dans ce type de requête, les paramètres sont remplacés par des points d'interrogation. Avant l'exécution de la requête, il faut fournir à l'objet `PreparedStatement` les valeurs qu'il doit utiliser pour remplacer les différents points d'interrogation. Un objet `PreparedStatement` peut être créé en utilisant le même principe que pour la création d'un objet `Statement`.

La méthode `prepareStatement` accessible à partir d'une connexion retourne un objet `PreparedStatement`. Cette méthode est disponible sous deux formes. La première attend comme argument une chaîne de caractères représentant la requête SQL. Dans cette requête, l'emplacement des paramètres doivent être réservé par des points d'interrogation. Si un jeu

d'enregistrements est créé par l'exécution de cette requête, celui-ci sera en lecture seule et à défilement en avant seulement. La deuxième forme de la méthode `prepareStatement` attend en plus de la chaîne de caractères un argument indiquant le type de jeu d'enregistrements et un autre déterminant les possibilités de modification des informations contenues dans le jeu d'enregistrements.

Les mêmes constantes que pour la méthode `createStatement` peuvent être utilisées.

```
stm=cnx.prepareStatement("select * from customers where
customerID=?",ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

Avant l'exécution de l'instruction SQL, vous devez fournir une valeur pour chacun des points d'interrogation représentant un paramètre. Pour cela, l'objet `PreparedStatement` dispose de nombreuses méthodes permettant l'affectation d'une valeur à un paramètre . Chacune de ces méthodes correspond au type de données SQL à insérer à la place d'un point d'interrogation. Ces méthodes sont nommées selon le même schéma : setXXXX où XXXX représente un type de données SQL. Chacune de ces méthodes attend comme premier argument un entier correspondant au rang du paramètre dans l'instruction SQL. Le premier paramètre est situé au rang 1. Le deuxième argument correspond à la valeur à transférer dans le paramètre. Le type de cet argument correspond bien sûr au type de données SQL à transférer vers le paramètre. Le pilote jdbc convertit ensuite le type Java en type SQL.

Par exemple la méthode `setInt (int indiceParam, int value)` effectue une conversion du type `int` Java en type INTEGER sql. Les valeurs stockées dans les paramètres sont conservées d'une exécution à l'autre de l'instruction SQL. La méthode `clearParameters` permet de réinitialiser l'ensemble des paramètres.

Les paramètres peuvent être utilisés dans une instruction SQL en remplacement de valeurs mais jamais pour remplacer un nom de champ, encore moins un opérateur. La syntaxe suivante est bien sûr interdite :

```
stm=cnx.prepareStatement("select * from customers where
?=?",ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
stm.setString(1,"customerID");
stm.setString(2,"AFK");
rs=stm.executeQuery();
```

Les autres méthodes disponibles avec un objet `PreparedStatement` sont parfaitement identiques à celles définies pour un objet `Statement`puisque l'interface `PreparedStatement` hérite directement de l'interface `Statement`.

```
public static void testPreparedStatement(Connection cnx)
  {
      {
            PreparedStatement stm;
            BufferedReader br;
            String code;
```

```
            ResultSet rs;
            try
            {
                  stm=cnx.prepareStatement("select * from customers
where customerID like ?",ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
                  br=new BufferedReader(new
InputStreamReader(System.in));
                  System.out.println("saisir le code du client
recherché :");
                  code=br.readLine();
                  stm.setString(1,code);
                  rs=stm.executeQuery();
                  while (rs.next())
                  {
                        for (int i = 1; i
<=rs.getMetaData().getColumnCount(); i++)
                        {
                        System.out.print(rs.getString(i)+"\t");
                        }
                        System.out.println();
                  }
            }
            catch (SQLException e)
            {
                  e.printStackTrace();
            }
            catch (IOException e)
            {
                  e.printStackTrace();
            }
      }
  }
```

### 2.4.3. Exécution de procédures stockées avec l'objet CallableStatement

Une procédure stockée représente du code SQL stocké sur le serveur. Cette approche procure plusieurs avantages :

- Les traitements et les temps de réponse sont améliorés.
- Le code SQL peut être partagé entre plusieurs applications.

Par contre, les applications deviennent beaucoup plus dépendantes du serveur de base de données. Le changement de serveur vous obligera certainement à réécrire vos procédures stockées car la syntaxe d'une procédure stockée est propre à chaque serveur. Les procédures stockées sont accessibles à partir de Java grâce à l'objet `CallableStatement`. Comme pour les objets `Statement` et `PreparedStatement`, c'est encore une fois la connexion qui va nous fournir une instance de classe correcte. C'est dans ce cas la méthode `prepareCall` qui doit être utilisée. Cette méthode attend comme argument une chaîne de caractères identifiant la procédure stockée à appeler. Par contre la syntaxe de cette chaîne de caractères est un petit peu spéciale puisqu'il ne suffit pas d'indiquer le nom de la procédure stockée.

Deux cas de figures sont à prendre en compte suivant que la procédure stockée retourne ou non une valeur. Si la procédure stockée ne retourne pas de valeur, la syntaxe de cette chaîne de caractères est la suivante :

```
{call nomProcedure( ?, ?,...)}
```

Dans cette syntaxe, les points d'interrogation représentent les paramètres attendus par la procédure stockée. Comme pour l'objet `PreparedStatement` les valeurs de ces paramètres doivent être fournies par les méthodes `setXXXX` correspondantes au type du paramètre.

Si la procédure stockée retourne une valeur, il faut utiliser la syntaxe suivante qui ajoute un paramètre supplémentaire pour accueillir le retour de la procédure stockée.

```
{ ?=call nomProcedure( ?, ?,...)}
```

Le premier paramètre étant utilisé en sortie de la procédure stockée, c'est l'exécution de la procédure stockée qui va y stocker une valeur, vous devez en informer l'objet `CallableStatement` en appelant la méthode `registerOutParameter`. Cette méthode attend comme premier argument, l'indice du paramètre 1 pour la valeur de retour de la procédure stockée, mais n'importe quel autre paramètre peut être utilisé en sortie, puis comme deuxième argument le type SQL du paramètre. Ce type peut être indiqué avec une des constantes définies dans l'interface `java.sql.Types`. Après exécution de la procédure stockée, la valeur des paramètres utilisés en sortie est accessible par les méthodes `getXXXX` où `XXXX` représente le type SQL du paramètre. Ces méthodes attendent comme argument l'indice du paramètre dans l'appel de la procédure stockée.

Comme pour les méthodes `createStatement` et `prepareStatement`, la méthode `prepareCall` propose une deuxième syntaxe permettant d'indiquer les caractéristiques d'un éventuel jeu d'enregistrements généré par l'exécution de la procédure stockée.

Pour illustrer l'utilisation de l'objet `CallableStatement`, nous allons utiliser les deux procédures stockées suivantes :

```
create PROCEDURE commandesParClient @code nchar(5)
AS
SELECT OrderID,
  OrderDate,
  RequiredDate,
  ShippedDate
FROM Orders
```

```
WHERE CustomerID = @code
ORDER BY OrderID

CREATE procedure nbCommandes @code nchar(5) as
declare @nb int
select @nb=count(*) from orders where customerid=@code
return @nb
```

La première retourne la liste de toutes les commandes du client dont le code est passé comme paramètre. La deuxième retourne un entier correspondant au nombre de commandes passées par le client dont le code est fourni comme paramètre.

```
public static void testProcedureStockee(Connection cnx)
  {
     CallableStatement cstm1,cstm2;
     BufferedReader br;
     String code;
     ResultSet rs;
     int nbCommandes;
     try
     {
          br=new BufferedReader(new InputStreamReader(System.in));
          System.out.println("saisir le code du client recherché :");
          code=br.readLine();
          cstm1=cnx.prepareCall("{ ?=call nbCommandes ( ? )}");
          cstm1.setString(2,code);
          cstm1.registerOutParameter(1,java.sql.Types.INTEGER);
          cstm1.execute();
          nbCommandes=cstm1.getInt(1);
          System.out.println("nombre de commandes passées par le
client " + code + " : " + nbCommandes );
          cstm2=cnx.prepareCall("{ call commandesParClient ( ?
)}",ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_READ_ONLY);
          cstm2.setString(1,code);
          rs=cstm2.executeQuery();
          System.out.println("détail des commandes");
          System.out.println("numéro de commande\tdate de commande");
          while (rs.next())
          {
                  System.out.print(rs.getInt("OrderID") + "\t");
                  System.out.println(new
```

```
SimpleDateFormat("dd/MM/yy").format(rs.getDate("OrderDate")));
            }
      }
      catch (SQLException e)
      {
            e.printStackTrace();
      }
      catch (IOException e)
      {
            e.printStackTrace();
      }
}
```

## 2.5. Utilisation des jeux d'enregistrements avec l'interface ResultSet

Lorsqu'une instruction SQL `select` est exécutée par la méthode `executeQuery` d'un objet `Statement`, `PreparedStatement` ou `CallableStatement`, celle-ci retourne un objet `ResultSet`. C'est par l'intermédiaire de cet objet `ResultSet` que nous allons pouvoir intervenir sur le jeu d'enregistrements. Nos possibilités d'action sur ce jeu d'enregistrements sont déterminées par les caractéristiques de l'objet `ResultSet`. Ces caractéristiques sont fixées au moment de la création des objets `Statement`, `PreparedStatement` ou `CallableStatement` en fonction des arguments passés lors de l'appel des méthodes `createStatement`, `prepareStatement` ou `prepareCall`.

Le premier argument détermine le type du jeu d'enregistrements. Les constantes suivantes sont définies :

- `ResultSet.TYPE_FORWARD_ONLY` : le jeu d'enregistrements sera à défilement en avant seulement.
- `ResultSet.TYPE_SCROLL_INSENSITIVE` : le jeu d'enregistrements pourra être parcouru dans les deux sens mais sera insensible aux changements effectués dans la base de données par d'autres utilisateurs.
- `ResultSet.TYPE_SCROLL_SENSITIVE` : le jeu d'enregistrements pourra être parcouru dans les deux sens et sera sensible aux changements effectués dans la base de données par d'autres utilisateurs.

Le second argument détermine les possibilités de modification des informations contenues dans le jeu d'enregistrements. Les deux constantes suivantes sont définies :

- `ResultSet.CONCUR_READ_ONLY` : les enregistrements sont en lecture seule.
- `ResultSet.CONCUR_UPDATABLE` : les enregistrements peuvent être modifiés dans le jeu d'enregistrements.

Il faut bien sûr que les actions exécutées sur l'objet `Resultset` soit compatibles avec ces caractéristiques sinon une exception sera déclenchée. Il est possible de vérifier les caractéristiques d'un objet `Resultset` en utilisant les méthodes `getType` et `getConcurrency`.

```
public static void infosResultset(ResultSet rs)
      {
          try {
              switch (rs.getType())
              {
                      case ResultSet.TYPE_FORWARD_ONLY:
                          System.out.println("le Resultset est à
défilement en avant seulement");
                          break;
                      case ResultSet.TYPE_SCROLL_INSENSITIVE:
                          System.out.println("le Resultset peut être
parcouru dans les deux sens");
                          System.out.println("il n\'est pas sensible aux
modifications faites par d\'autres utilisateurs");
                          break;
                      case ResultSet.TYPE_SCROLL_SENSITIVE:
                          System.out.println("le Resultset peut être
parcouru dans les deux sens");
                          System.out.println("il est sensible aux
modifications faites par d\'autres utilisateurs");
                          break;
              }
              switch (rs.getConcurrency())
              {
                      case ResultSet.CONCUR_READ_ONLY:
                          System.out.println("les données contenues dans
le ResulSet sont en lecture seule");
                          break;
                      case ResultSet.CONCUR_UPDATABLE:
                          System.out.println("les données contenues dans
le ResulSet sont modifiables");
                          break;
              }
          }
          catch (SQLException e)
          {
              e.printStackTrace();
          }
     }
```

### 2.5.1. Positionnement dans un ResultSet

L'objet `ResultSet` gère un pointeur d'enregistrement déterminant sur quel enregistrement vont intervenir les méthodes exécutées sur le `ResultSet` lui-même. Cet enregistrement est parfois appelé enregistrement actif ou enregistrement courant. L'objet `ResultSet` contient toujours deux enregistrements fictifs servant de repère pour le début du `ResultSet` (BOF) et pour la fin du `ResultSet` (EOF). Le pointeur d'enregistrement peut être positionné sur l'un de ces deux enregistrements mais jamais avant l'enregistrement BOF ni après l'enregistrement EOF. Ces enregistrements ne contiennent pas de données et une opération de lecture ou d'écriture sur ces enregistrements déclenche une exception. À la création du `ResultSet` le pointeur est positionné avant le premier enregistrement (BOF). De nombreuses méthodes sont disponibles pour gérer le pointeur d'enregistrements :

- `boolean absolute(int position)` : déplace le pointeur d'enregistrement sur l'enregistrement spécifié. La numérotation des enregistrements débute à 1. Si la valeur de l'argument position est négative, le déplacement est effectué en partant de la fin du `ResultSet`. Si le numéro d'enregistrement n'existe pas le pointeur est positionné sur l'enregistrement BOF si la valeur est négative et inférieure au nombre d'enregistrements ou sur l'enregistrement EOF si la valeur est positive et supérieure au nombre d'enregistrements. Cette méthode retourne `true` si le pointeur est positionné sur un enregistrement valide et `false` dans le cas contraire (BOF ou EOF).
- `boolean relative(int deplacement)` : déplace le curseur du nombre d'enregistrements spécifié par l'argument `deplacement`. Si la valeur de cet argument est positive le curseur descend dans le jeu d'enregistrements et si la valeur est négative le curseur remonte dans le jeu d'enregis-trements. Cette méthode retourne `true` si le pointeur est positionné sur un enregistrement valide et `false` dans le cas contraire (BOF ou EOF).
- `void beforeFirst()` : déplace le pointeur d'enregistrements avant le premier enregistrement (BOF).
- `void afterLast()` : déplace le pointeur d'enregistrements après le dernier enregistrement (EOF).
- `boolean first()` : déplace le pointeur d'enregistrements sur le premier enregistrement. Cette méthode retourne `true` s'il y un enregistrement dans le `ResultSet` et `false` dans le cas contraire.
- `boolean last()` : déplace le pointeur d'enregistrements sur le dernier enregistrement. Cette méthode retourne `true` s'il y un enregistrement dans le `ResultSet` et `false` dans le cas contraire.
- `boolean next()` : déplace le pointeur d'enregistrements sur l'enregistrement suivant l'enregistrement courant. Cette méthode retourne `true` si le pointeur est sur un enregistrement valide et `false` dans le cas contraire (EOF).
- `boolean previous()` : déplace le pointeur d'enregistrements sur l'enregistrement précédant l'enregistrement courant. Cette méthode retourne `true` si le pointeur est sur un enregistrement valide et `false` dans le cas contraire (BOF).

Pour toutes ces méthodes sauf pour la méthode `next`, il faut obligatoirement que le `ResultSet` soit de type `SCROLL_SENSITIVE` ou `SCROLL_INSENSITIVE`. Si le `ResultSet` est de type `FORWARD_ONLY`, seule la méthode `next` fonctionne et dans ce cas les autres méthodes déclenchent une exception. Les méthodes suivantes permettent de tester la position du pointeur d'enregistrements :

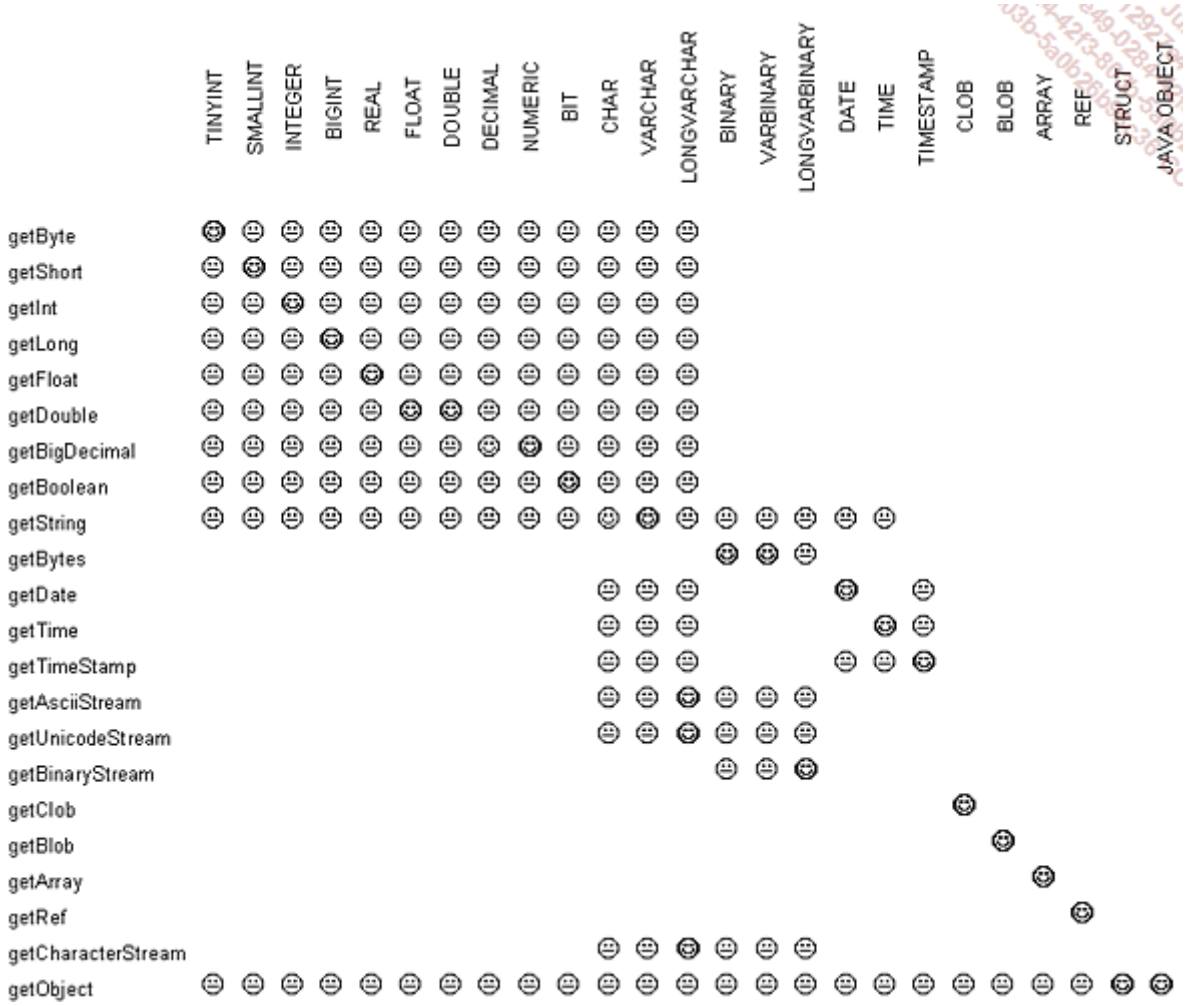
- `boolean isBeforeFirst()` : retourne `true` si le pointeur est placé avant le premier enregistrement (BOF).
- `boolean isAfterLast()` : retourne `true` si le pointeur est placé après le dernier enregistrement (EOF).
- `boolean isFirst()` : retourne `true` si le pointeur est placé sur le premier enregistrement.
- `boolean isLast()` : retourne `true` si le pointeur est placé sur le dernier enregistrement.
- `int getRow()` : retourne le numéro de l'enregistrement sur lequel se trouve le pointeur d'enregistrements. La valeur 0 est retournée s'il n'y a pas d'enregistrement courant (BOF ou EOF).

```
     public static void positionRs(ResultSet rs)
     {
           try {
               if (rs.isBeforeFirst())
               {
                    System.out.println("le pointeur est avant le
premier enregistrement");
               }
               if (rs.isAfterLast())
               {
                    System.out.println("le pointeur est après le
dernier enregistrement");
               }
               if (rs.isFirst())
               {
                    System.out.println("le pointeur est sur le premier
enregistrement");
               }
               if (rs.isLast())
               {
                    System.out.println("le pointeur est sur le dernier
enregistrement");
               }
               int position;
               position=rs.getRow();
               if (position!=0)
               {
                    System.out.println("c'est l\'enregistrement numéro
" + position);
```

```
                }
            } catch (SQLException e) {
                // TODO Bloc catch autogénéré
                e.printStackTrace();
            }
        }
}
```

### 2.5.2. Lecture des données dans un ResultSet

L'objet `ResultSet` fournit de nombreuses méthodes permettant la lecture des champs d'un enregistrement. Chacune de ces méthodes est spécifique à un type de données SQL. Il faut bien sûr utiliser en priorité la méthode adaptée au type du champ dont on souhaite obtenir la valeur. Cependant certaines de ces méthodes sont relativement souples et permettent la lecture de plusieurs types de données. Le tableau ci-après reprend les principaux types de données SQL et les méthodes en permettant la lecture à partir d'un `ResultSet`. Les méthodes marquées avec le symbole ☺ sont les méthodes conseillées. Les méthodes marquées avec le symbole K sont possibles mais avec des risques de perte d'informations.




Chacune de ces méthodes est disponible sous deux formes. La première accepte comme argument le numéro de la colonne dont on souhaite obtenir la valeur. La numérotation commençant à 1. La deuxième version accepte une chaîne de caractères représentant le nom de la colonne dans la base

de données. Si la requête ayant été utilisée pour créer le ResultSet contient des alias alors les colonnes portent le nom de l'alias et non le nom du champ dans la base de données. Pour une meilleure lisibilité du code, il est bien sûr préférable d'utiliser les noms des colonnes plutôt que leurs indices. Lorsque dans la base de données un champ ne contient pas de valeur (Dbnull), les méthodes retournent une valeur égale à 0 pour les champs numériques, une valeur false pour les champs booléens et une valeur null pour les autres types. Dans certains cas, il y a donc un doute possible sur la valeur réellement présente dans la base. Par exemple, la méthode getInt peut retourner une valeur égale à zéro parce qu'il y a effectivement cette valeur dans la base de données ou parce que ce champ n'est pas renseigné dans la base de données. Pour lever ce doute, la méthode wasNull retourne un boolean égal à true si le champ sur lequel la dernière opération de lecture dans le `ResultSet` contenait effectivement une valeur `null`.

```
    public static void lectureRs(Connection cnx)
    {
         Statement stm;
         String requete;
         ResultSet rs;
         try
         {
    stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_READ_ONLY);
                requete="select *   from products " ;
                rs=stm.executeQuery(requete);
                System.out.println("code produit\tdésignation\tprix
unitaire\tstock\tépuisé\tDLC");
                while(rs.next())
                {
                           System.out.print(rs.getInt("ProductID")+"\t");

    System.out.print(rs.getString("ProductName")+"\t");

    System.out.print(rs.getDouble("UnitPrice")+"\t");
                           rs.getShort("UnitsInStock");
                           if (rs.wasNull())
                           {
                                System.out.print("inconnu\t");
                           }
                           else
                           {

    System.out.print(rs.getShort("UnitsInStock")+"\t");
                           }
```

```
    System.out.print(rs.getBoolean("Discontinued")+"\t");
                            if (rs.getDate("DLC")!=null)
                            {
                                  System.out.println(rs.getDate("DLC"));
                            }
                            else
                                  System.out.println("non perissable");
                      }
                      rs.close();
                      stm.close();
                }
                catch (SQLException e)
                {
                      e.printStackTrace();
                }
    }
```

### 2.5.3. Modification des données dans un ResultSet

La modification des données est effectuée simplement en utilisant les méthodes `updateXXX` où `XXX` correspond au type de données de la colonne à mettre à jour. Comme pour les méthodes `getXXX` celles-ci sont disponibles en deux versions, l'une attend comme argument l'indice de la colonne à mettre à jour, la deuxième attend elle comme argument une chaîne de caractères représentant le nom de la colonne dans la base de données. Si la requête ayant été utilisée pour créer le `ResultSet` contient des alias, alors les colonnes portent le nom de l'alias et non le nom du champ dans la base de données. Pour une meilleure lisibilité du code, il est bien sûr préférable d'utiliser les noms des colonnes plutôt que leurs indices.

Le type du deuxième argument attendu par ces méthodes correspond bien sûr au type de données à mettre à jour dans le `ResultSet`. Les modifications doivent ensuite être validées par la méthode `updateRow` ou annulées par la méthode `cancelRowUpdates`. Le `ResultSet` doit obligatoirement être de type `CONCUR_UPDATABLE` pour pouvoir être modifié. Dans le cas contraire, l'exécution d'une méthode `updateXXX`déclenche une exception.

```
public static void modificationRs(Connection cnx)
     {
          Statement stm;
          String requete;
          ResultSet rs;
          int num=0;
          BufferedReader br;
          String reponse;
          try
```

```
          {
     stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
                requete="select *   from products " ;
                rs=stm.executeQuery(requete);
                System.out.println("numéro de ligne\tcode
produit\tdésignation\tprix unitaire\tstock\tépuisé\tDLC");
                while(rs.next())
                {
                              num++;
                              System.out.print(num + "\t");
                              System.out.print(rs.getInt("ProductID")+"\t");

     System.out.print(rs.getString("ProductName")+"\t");

     System.out.print(rs.getDouble("UnitPrice")+"\t");
                              rs.getShort("UnitsInStock");
                              if (rs.wasNull())
                              {
                                   System.out.print("inconnu\t");
                              }
                              else
                              {

     System.out.print(rs.getShort("UnitsInStock")+"\t");
                         }

     System.out.print(rs.getBoolean("Discontinued")+"\t");
                              if (rs.getDate("DLC")!=null)
                              {
                                   System.out.println(rs.getDate("DLC"));
                              }
                              else
                                   System.out.println("non perissable");
                }
                br=new BufferedReader(new InputStreamReader(System.in));
                 System.out.println("quelle ligne voulez-vous modifier ? ");
                reponse=br.readLine();
                rs.absolute(Integer.parseInt(reponse));
```

```
                System.out.println("désignation actuelle " +
rs.getString("ProductName"));
                System.out.println("saisir la nouvelle valeur ou enter
pour conserver la valeur actuelle");
                reponse=br.readLine();
                if (!reponse.equals(""))
                {
                        rs.updateString("ProductName",reponse);
                }
                System.out.println("prix unitaire actuel " +
rs.getDouble("UnitPrice"));
                System.out.println("saisir la nouvelle valeur ou enter
pour conserver la valeur actuelle");
                reponse=br.readLine();
                if (!reponse.equals(""))
                {

     rs.updateDouble("UnitPrice",Double.parseDouble(reponse));
                }
                rs.getShort("UnitsInStock");
                if (rs.wasNull())
                {
                System.out.println ("quantité en stock actuelle inconnue");
                }
                else
                {
                System.out.println("quantité en stock actuelle " +
rs.getShort("UnitsInStock"));
                }
                System.out.println("saisir la nouvelle valeur ou enter
pour conserver la valeur actuelle");
                reponse=br.readLine();
                if (!reponse.equals(""))
                {

     rs.updateShort("UnitsInStock",Short.parseShort(reponse));
                }
                System.out.println("voulez-vous valider ces modifications
o/n");
```

```
                reponse=br.readLine();
                if (reponse.toLowerCase().equals("o"))
                {
                     rs.updateRow();
                }
                else
                {
                     rs.cancelRowUpdates();
                }
                System.out.println("les valeurs actuelles ");
                System.out.print(rs.getString("ProductName")+"\t");
                System.out.print(rs.getDouble("UnitPrice")+"\t");
                rs.getShort("UnitsInStock");
                if (rs.wasNull())
                {
                     System.out.print("inconnu\t");
                }
                else
                {
                System.out.print(rs.getShort("UnitsInStock")+"\t");
                }
                rs.close();
                stm.close();
            }
            catch (SQLException e)
            {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
    }
```

### 2.5.4. Suppression de données dans un ResultSet

La suppression d'une ligne est effectuée très simplement en positionnant le pointeur sur la ligne à supprimer puis en appelant la méthode `deleteRow`. La ligne est immédiatement supprimée du `ResultSet` et dans la base de données. La position du pointeur d'enregistrement après la suppression dépend du pilote de base de données utilisé. Certains pilotes déplacent le pointeur sur l'enregistrement suivant, d'autres le déplacent sur l'enregistrement précédent et enfin quelques-uns ne modifient pas la position du pointeur. Il faut dans ce cas utiliser une des méthodes de déplacement pour positionner le pointeur sur un enregistrement utilisable. Le `ResultSet` doit obligatoirement être de type `CONCUR_UPDATABLE` pour pouvoir y supprimer des données. Dans le cas contraire, l'exécution de la méthode `deleteRow` déclenche une exception.

```
public static void suppressionRs(Connection cnx)
     {
         Statement stm;
         String requete;
         ResultSet rs;
         int num=0;
         BufferedReader br;
         String reponse;
         try
         {
     stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
               requete="select *  from products " ;
               rs=stm.executeQuery(requete);
               System.out.println("numéro de ligne\tcode
produit\tdésignation\tprix unitaire\tstock\tépuisé\tDLC");
               while(rs.next())
               {
                         num++;
                         System.out.print(num + "\t");
                         System.out.print(rs.getInt("ProductID")+"\t");

     System.out.print(rs.getString("ProductName")+"\t");

     System.out.print(rs.getDouble("UnitPrice")+"\t");
                         rs.getShort("UnitsInStock");
                         if (rs.wasNull())
                         {
                              System.out.print("inconnu\t");
                         }
                         else
                         {

     System.out.print(rs.getShort("UnitsInStock")+"\t");
                         }

     System.out.print(rs.getBoolean("Discontinued")+"\t");
                         if (rs.getDate("DLC")!=null)
                         {
```

```
                              System.out.println(rs.getDate("DLC"));
                         }
                         else
                              System.out.println("non périssable");
               }
               br=new BufferedReader(new InputStreamReader(System.in));
               System.out.println("quelle ligne voulez-vous supprimer ? ");
               reponse=br.readLine();
               rs.absolute(Integer.parseInt(reponse));
               rs.deleteRow();
               System.out.println("le pointeur est maintenant sur la
ligne " + rs.getRow());
          }
          catch (Exception e)
          {
               e.printStackTrace();
          }
     }
```

### 2.5.5. Ajout de données dans un ResultSet

Chaque objet `ResultSet` contient une ligne spéciale destinée à l'insertion de données. Le pointeur d'enregistrement doit au préalable être positionné sur cette ligne spéciale avec l'instruction `moveToInsertRow`. Une fois que le pointeur est positionné sur cette ligne, celle-ci peut être mise à jour avec les méthodes `updateXXX`. L'insertion de la ligne doit ensuite être validée avec la méthode `insertRow`. Cette méthode provoque la mise à jour de la base de données. La ligne d'insertion devient à ce moment une ligne normale du `ResultSet` et le pointeur d'enregistrement est positionné sur cette ligne. Vous pouvez revenir sur la ligne sur laquelle vous étiez avant l'insertion grâce à la méthode `moveToCurrentRow`. Si vous ne fournissez pas de valeur pour toutes les colonnes, des valeurs `null` seront insérées dans la base de données pour les colonnes non renseignées. La base de données doit accepter les valeurs nulles pour ces champs sinon une exception est déclenchée. Le `ResultSet` doit obligatoirement être de type `CONCUR_UPDATABLE` pour pouvoir y insérer des données. Dans le cas contraire, l'exécution de la méthode `moveToInsertRow` déclenche une exception.

```
public static void ajoutRs(Connection cnx)
     {
          Statement stm;
          String requete;
          ResultSet rs;
          int num=0;
          BufferedReader br;
          String reponse;
          try
```

```
          {
     stm=cnx.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
              requete="select *  from products " ;
              rs=stm.executeQuery(requete);
              br=new BufferedReader(new InputStreamReader(System.in));
              System.out.println("saisir les valeurs de la nouvelle ligne");
              rs.moveToInsertRow();
              System.out.print("code produit : ");
              reponse=br.readLine();
              rs.updateInt ("ProductID",Integer.parseInt(reponse));
              System.out.print("Désignation : ");
              reponse=br.readLine();
              rs.updateString ("ProductName",reponse);
              System.out.print("Prix unitaire : ");
              reponse=br.readLine();
              rs.updateDouble("UnitPrice",Double.parseDouble(reponse));
              System.out.print("Quantité en stock : ");
              reponse=br.readLine();
              rs.updateDouble("UnitsInStock",Short.parseShort(reponse));
              rs.insertRow();
          }
          catch (Exception e)
          {
               e.printStackTrace();
          }
}
```

## 2.6. Gestion des transactions

Les transactions vont permettre de garantir qu'un ensemble d'instructions SQL sera exécuté avec succès pour chacune d'elles ou qu'aucune d'entre elles ne sera exécutée. Le transfert d'une somme d'argent entre deux comptes bancaires représente l'exemple classique où une transaction est nécessaire. Imaginez la situation suivante : notre banque doit effectuer l'encaissement d'un chèque de 1000 € à débiter sur le compte numéro 12345 et à créditer sur le compte 67890. Par mesure de sécurité après chaque opération effectuée sur un compte (débit ou crédit), un rapport est édité. Voici ci-dessous un extrait du code pouvant effectuer ces opérations.

```
public static void mouvement(String compteDebit,String compteCredit,double
somme)
    {
          try
          {
```

```
    Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
              Connection cnx=null;

    cnx=DriverManager.getConnection("jdbc:sqlserver://localhost;database
Name=banque; user=sa;password=;");
              PreparedStatement stm;
              stm=cnx.prepareStatement("update comptes set solde=solde
+ ? where numero=?");
              stm.setDouble(1,somme * -1);
              stm.setString(2,compteDebit);
              stm.executeUpdate();
              impressionRapport(compteDebit, somme);
              stm.setDouble(1,somme);
              stm.setString(2,compteCredit);
              stm.executeUpdate();
              impressionRapport(compteCredit, somme);
          }
          catch (Exception e)
          {
          e.printStackTrace();
          }
    }
```

Pour 99,9999 % des mouvements effectués avec ce code, il n'y a aucun problème, sauf qu'un beau jour l'imprimante chargée des éditions se bloque et ce blocage déclenche une exception dans la méthode `impressionRapport`. A priori cette exception ne pose pas de problème puisque l'appel de cette méthode est placé dans un bloc `try` et qu'un bloc `catch` traite l'exception. Il ne faut cependant pas oublier que si une exception est déclenchée et qu'un bloc `catch` est exécuté, l'exécution se poursuit par l'instruction suivant le bloc `catch`. Dans ce cas de figure, les instructions placées entre celle ayant déclenché l'exception et la fin du bloc `try` ne sont tout simplement pas exécutées. Dans notre cas, ceci peut être très problématique si cette exception est déclenchée dans la méthode `impressionRapport` exécutée aussitôt après l'opération de débit. L'opération de crédit n'a tout simplement pas lieu. La somme est donc perdue. Une solution consiste à annuler l'effet des instructions SQL ayant déjà été exécutées en effectuant le traitement inverse, dans notre cas en créditant le compte.

C'est en fait ce mécanisme qui est mis en œuvre dans une transaction mais bien sûr de manière automatique. C'est au niveau de la connexion vers le serveur de base de données que sont gérées les transactions.

### 2.6.1. Mise en œuvre des transactions

Jusqu'à présent, nous ne nous sommes pas préoccupés des transactions et pourtant nous en réalisons depuis notre première instruction SQL exécutée via JDBC. Le fonctionnement par défaut de jdbc consiste effectivement à inclure chaque instruction exécutée dans une transaction puis valider la transaction si l'instruction a été exécutée correctement (`commit`) ou à annuler l'instruction dans le cas contraire (`rollBack`). Ce mode de fonctionnement est appelé mode `autoCommit`. Si vous souhaitez gérer vous-même la fin d'une transaction en validant ou en annulant toutes les instructions qu'elle contient, vous devez désactiver le mode `autoCommit` en appelant la méthode `setAutoCommit(false)` sur l'objet `Connection`. Vous êtes maintenant responsables de la fin des transactions. Les méthodes `commit` et `rollback` de l'objet `Connection`permettent de valider ou d'annuler les instructions exécutées depuis le début de la transaction. Une nouvelle transaction débute automatiquement dès la fin de la précédente ou dès l'ouverture de la connexion. Le code de notre méthode permettant de transférer un montant entre deux comptes doit donc prendre la forme suivante.

```
public static void mouvement1(String compteDebit,String compteCredit,double
somme)
     {
          Connection cnx=null;
          try
          {

     Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
     cnx=DriverManager.getConnection("jdbc:sqlserver://localhost;database
Name=banque; user=sa;password=;");
               cnx.setAutoCommit(false);
               PreparedStatement stm;
               stm=cnx.prepareStatement("update comptes set solde=solde
+ ? where numero=?");
               stm.setDouble(1,somme * -1);
               stm.setString(2,compteDebit);
               stm.executeUpdate();
               impressionRapport(compteDebit, somme);
               stm.setDouble(1,somme);
               stm.setString(2,compteCredit);
               stm.executeUpdate();
               impressionRapport(compteCredit, somme);
               cnx.commit();
          }
          catch (Exception e)
          {
               try {
                    cnx.rollback();
               }
               catch (SQLException e1)
```

```
            {
                e1.printStackTrace();
            }
            e.printStackTrace();
        }
    }
```

### 2.6.2. Points de sauvegarde

Lors de l'appel de la méthode `rollback`, l'ensemble des instructions SQL exécutées depuis le début de la transaction sont annulées. Cette méthode propose une deuxième version acceptant comme paramètre un objet `SavePoint`. Cet objet représente un repère dans l'exécution des instructions SQL. Il est créé par la méthode `setSavePoint`. L'appel de la méthode `rollback` avec comme argument un objet `SavePoint` provoque l'annulation de toutes les instructions exécutées jusqu'à ce point de sauvegarde.

### 2.6.3. Niveaux d'isolement

Pendant qu'une transaction est active, les données modifiées par les instructions exécutées au sein de la transaction peuvent être verrouillées par la base de données pour éviter les conflits et les incohérences. Différents types de verrous sont disponibles pour une transaction. Pour bien comprendre leurs effets, il faut au préalable identifier les types de problèmes pouvant être rencontrés lorsqu'une transaction est active.

- **Lecture erronée** : cette anomalie se produit lorsqu'une application accède à des données qui sont en train d'être modifiées par une transaction qui n'a pas encore été validée.
- **Lecture non reproductible** : cette anomalie se produit lorsque les exécutions successives d'une même instruction `select` ne produisent pas le même résultat. C'est le cas si les données que vous lisez sont en cours de modification par une autre transaction.
- **Lecture fantôme** : cette anomalie se produit si des exécutions successives d'une même requête renvoient des données en plus ou en moins. Cela peut être le cas si une autre transaction est en train de supprimer ou d'ajouter des données à la table.

JDBC prévoit plusieurs niveaux d'isolement. Ces niveaux d'isolation déterminent la manière dont sont verrouillées les données durant la transaction. Ce verrouillage peut être placé en lecture, en écriture ou encore en lecture et en écriture sur les données accédées par les instructions de la transaction.

La méthode `setTransactionIsolationLevel` permet de fixer les types de problèmes pouvant être évités. Cette méthode accepte comme argument une des constantes présentées dans le tableau suivant. Ce tableau présente également l'effet de chaque niveau d'isolement sur les données.

| Constante | Lecture erronée | Lecture non reproductible | Lecture fantôme |
|---|---|---|---|
| TRANSACTION_READ_UNCOMMITTED | possible | possible | possible |
| TRANSACTION_READ_COMMITTED | impossible | possible | possible |
| TRANSACTION_REPEATABLE_READ | impossible | impossible | possible |
| TRANSACTION_SERIALIZABLE | impossible | impossible | impossible |

Le fait de spécifier un niveau d’isolement pour une transaction peut parfois avoir un effet sur les autres applications accédant aux données manipulées dans la transaction puisque celles-ci peuvent être verrouillées et donc être inaccessibles aux autres applications.