

# CHAPITRE 1

## ACCEDER A UNE BASE DE DONNEES AVEC JDBC

## 1. ENVIRONNEMENT DE DEVELOPPEMENT

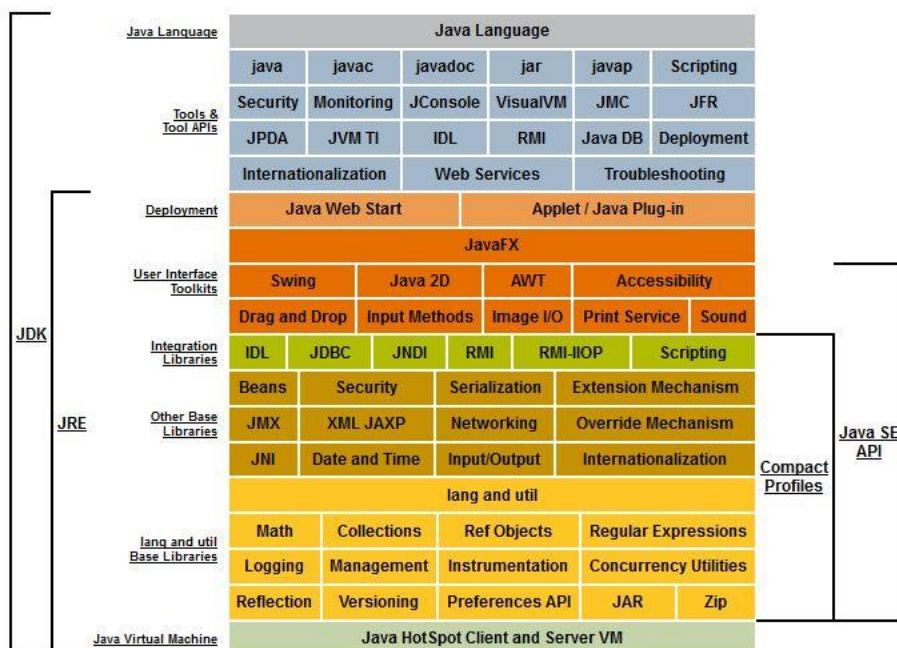
L'environnement de développement du ch.1 est décrit dans la liste suivante :

### Installer Java

La version 8 du JDK (Java Development Kit) de le version JavaSE (Standard Edition) de java est téléchargeable sur cette page :

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

Liste des APIs contenues dans JavaSE :



L'ensemble des classes java que toute application Java doit utiliser pour accéder à une BD dans un SGBD, est contenu dans l'API JDBC.

### Installer l'IDE

L'interface de développement utilisée dans ce cours sera l'IDE (Integrated Development Environment) NetBeans 8.2 téléchargeable sur cette page :

<https://netbeans.org/downloads/>

### Installer le SGBD

Le SGBD utilisé dans ce cours sera MySQL Community Server 5.x téléchargeable sur cette page :

<https://dev.mysql.com/downloads/mysql/>

### Installer un gestionnaire de BD

MySQL n'offre qu'un invite de commandes pour administrer ses bases de données, il est donc préférable d'utiliser un assistant graphique pour gérer nos bases de données et tester l'exécution des requêtes SQL. On utilisera dans ce cours l'outil Navicat version 10 ou ultérieure, téléchargeable sur cette page :

<https://www.navicat.com/en/products/navicat-for-mysql>

## Ajouter le pilote JDBC de MySQL dans le projet NetBeans

Quand une application (Java, PHP, C, ...) a besoin d'utiliser une BD hébergée dans un SGBD, elle doit se connecter à ce SGBD, pour pouvoir lui adresser des requêtes en SQL (**SELECT**, **UPDATE**, ...). Et pour cela elle doit utiliser les classes d'une bibliothèque qu'on appelle Pilote ou Driver.

Chaque SGBD – ex. MySQL – fournit un pilote pour chaque langage de programmation, y compris le pilote Java, qu'on appelle le pilote JDBC, qui est un ensemble de classes précompilées regroupées dans un fichier **jar**.

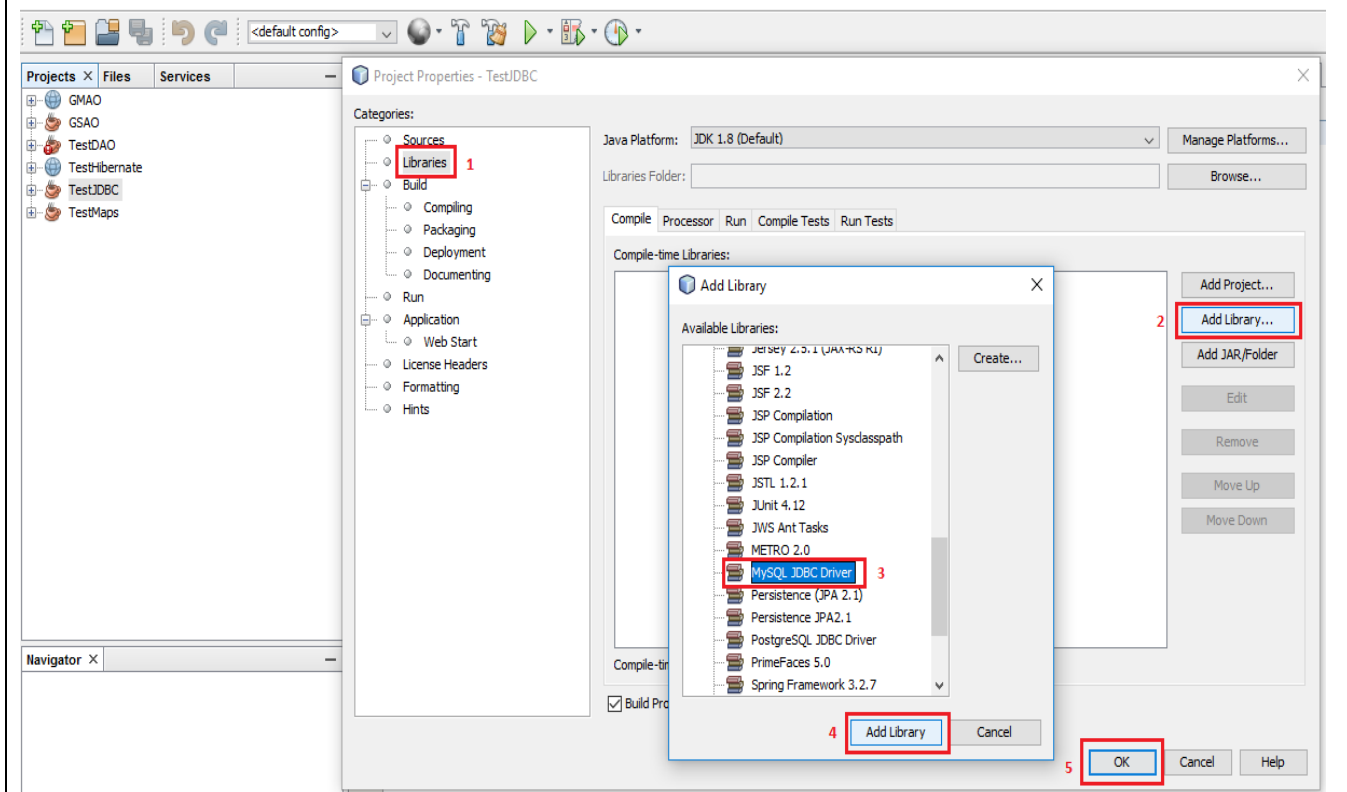
Le pilote JDBC de MySQL est téléchargeable sur cette page :

<https://dev.mysql.com/downloads/connector/j/>

Après avoir téléchargé le fichier jar, il faut l'ajouter dans les dépendances du projet NetBeans :

Ouvrir les propriétés du projet ciblé : Cliquez du bouton droit sur le projet, puis dans le menu contextuel choisir l'élément *properties*.

Puis dans la fenêtre des propriétés, choisir les paramètres suivants :



## 2. GENERALITES

### Définition du processus

Le processus standard qu'on doit suivre pour exécuter et exploiter une requête SQL est le suivant :



### IMPORTANT

Gérer les exceptions **ClassNotFoundException** et **SQLException**.

#### 1. Charger la classe du driver dans la mémoire de notre programme

Parmi les classes qui existent dans le *jar*, il y a la classe du driver, cette classe doit être chargée dans notre programme avant d'établir la connexion avec le serveur. Dans le cas de MySQL, cette classe s'appelle **Driver** et son chemin complet dans le fichier jar est :

**com.mysql.jdbc.Driver**

On doit charger cette classe grâce à la méthode statique **forName("chemin complet d'une classe")** de la classe **java.lang.Class**.

```
try {
    Class.forName("com.mysql.jdbc.Driver");
} catch (ClassNotFoundException ex) {
    System.err.println(ex.getMessage());
}
```

#### 2. Etablir la connexion avec le serveur

Créer un objet **java.sql.Connection** en utilisant la méthode statique **getConnection(connection\_string, mysql\_user, mysql\_password)** de la classe **java.sql.DriverManager**.

- Connection string en cas de connexion au serveur :

```
"jdbc:mysql://" + adress_ip_mysql + ":" + numéro_du_port + "/"
```

- Connection string en cas de connexion à une base de données :

```
"jdbc:mysql://" + adress_ip_mysql + ":" + numéro_du_port + "/" + nom_base
```

```
try {
    String adress_ip_mysql = "localhost", numero_du_port = "3306";
    String user = "root", pass = "1234";
    String connectionString = "jdbc:mysql://" + adress_ip_mysql + ":" + numéro_du_port + "/";
    Connection c = DriverManager.getConnection(connectionString, user, pass);
} catch (SQLException ex) {
    System.err.println(ex.getMessage());
}
```

#### 3. Créer la requête SQL

```
String sql = "une requete sql";
```

#### 4. Exécuter la requête SQL

Créer un objet `java.sql.Statement` en utilisant la méthode `createStatement()` de l'objet `Connection`.

Puis exécuter la méthode `execute(code_sql)`, `executeQuery(code_sql)` ou `executeUpdate(code_sql)` (selon le type de la requête SQL) de l'objet `Statement`.

- **executeQuery** : exécuter une requête `SELECT`, cette fonction retourne les lignes récupérées par la requête, dans un objet `java.sql.ResultSet`.
- **executeUpdate** : exécuter une requête de manipulation de données (`INSERT`, `UPDATE`, `DELETE`) ou de définition de données (`CREATE`, `ALTER`, `DROP TABLE`), cette fonction retourne le nombre de lignes affectées par la requête
- **execute** : exécuter n'importe quel code SQL (sélection et/ou mise à jour), qui peut retourner plusieurs résultats (cas de script SQL qui peut contenir plusieurs instructions) :
  - Code SQL qui retourne des lignes : la fonction retourne `TRUE`, et on récupère les lignes dans un objet `ResultSet` grâce à la méthode `Statement.getResultSet()`
  - Code SQL qui retourne un nombre : la fonction retourne `FALSE`, et on récupère le nombre de lignes dans un entier grâce à la méthode `Statement.getUpdateCount()`
  - Pour passer d'un résultat à l'autre, utiliser la méthode `Statement.getMoreResults()`

```
try {  
    Statement stm = c.createStatement();  
    stm.executeQuery(sql); / stm.executeUpdate(sql); / stm.execute(sql);  
} catch (SQLException ex) {  
    System.err.println(ex.getMessage());  
}
```

#### 5. Fermer le statement et la connexion


Il est essentiel de libérer les ressources mémoire utilisées par les objets `statement` et `connexion`, immédiatement après terminé leur tâche, plutôt que d'attendre que java le fasse automatiquement.

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
    String adress_ip_mysql = "localhost", numero_du_port = "3306";  
    String user = "root", pass = "1234";  
    String connectionString = "jdbc:mysql://" + adress_ip_mysql + ":" + numero_du_port + "/";  
    Connection c = DriverManager.getConnection(connectionString, user, pass);  
    Statement stm = c.createStatement();  
    stm.executeQuery(sql); / stm.executeUpdate(sql); / stm.execute(sql);  
    stm.close();  
    c.close();  
} catch (SQLException ex) {  
    System.err.println(ex.getMessage());  
}
```


## TP

1. Créer une connexion MySQL dans Navicat.
2. Créer une base de données « gestion\_rh ».
3. Ecrire le script SQL pour créer les deux tables « employe » et « service ». Exécuter le script SQL sur Navicat.

employe :

Name	Type	Length	Decimals	Allow Null	
id	int	5		<input type="checkbox"/>	 1
nom	varchar	50		<input type="checkbox"/>	
prenom	varchar	50		<input type="checkbox"/>	
tel	varchar	50		<input type="checkbox"/>	
adresse	varchar	100		<input checked="" type="checkbox"/>	
salaire	decimal	7	2	<input checked="" type="checkbox"/>	
naissance	date			<input checked="" type="checkbox"/>	
service	int	5		<input checked="" type="checkbox"/>	

service :

Name	Type	Length	Decimals	Allow Null	
id	int	5		<input type="checkbox"/>	 1
nom	varchar	50		<input type="checkbox"/>	
date_creation	date			<input checked="" type="checkbox"/>	
parent	int	5		<input checked="" type="checkbox"/>	

Respecter les contraintes suivantes :

- Les clefs primaires sont auto-incrémentées pour les deux tables
- La table service est référencée dans la table « employe » par la clef étrangère « service »
- La table service se référence elle-même par la clef étrangère « parent »
- Utiliser les options suivantes pour les clefs étrangères :
  - « cascade » pour la mise à jour
  - « set null » pour la suppression

4. Créer un programme java qui fait l'insertion d'un employé, en respectant ceci :
  - Utiliser l'auto incrémentation pour l'identifiant.
  - L'employé n'appartient à aucun service.
  - Utiliser la méthode `executeUpdate(code_sql)` de l'objet Statement pour exécuter la requête d'insertion.

### 3. LES REQUETES DE MISE A JOUR

#### Définition

Du point de vue java, toute requête de :

- Création (**CREATE**) ou altération (**ALTER**, **DROP**, ...) de table ou de vue, ou bien :
- Création, suppression ou mise à jour de données (**INSERT**, **UPDATE**, **DELETE**)

est considérée comme une requête de **mise à jour**, et doit être exécutée avec la méthode **Statement.executeUpdate(code\_sql)**. Cette méthode retourne un entier qui représente le nombre de lignes affectées par la requête.

S'il n'y a pas de lignes affectées, la méthode retourne 0. Exemples :

- Cas de **CREATE / ALTER TABLE**
- Cas de **UPDATE** ou **DELETE** ayant un **WHERE** qui retourne **false**.

#### Exemple 1 : Supprimer toutes les lignes de la table « service »

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

    String adress_ip_mysql = "localhost", numéro_du_port = "3306",
        user = "root", pass = "", nom_base = "gestion_rh";
    String connectionString = "jdbc:mysql://" + adress_ip_mysql
        + ":" + numéro_du_port + "/" + nom_base;

    Connection c = DriverManager.getConnection(connectionString, user, pass);

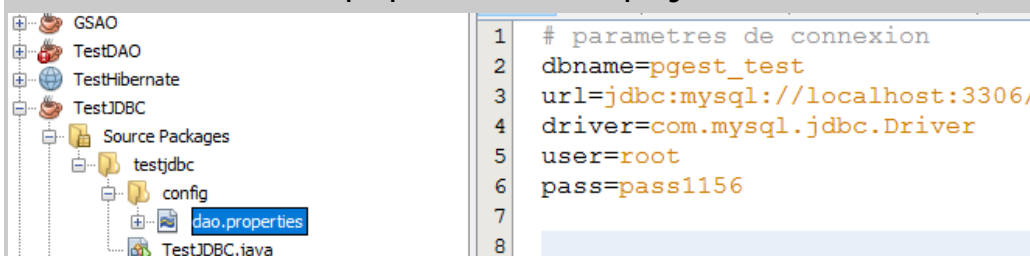
    String sql = "DELETE FROM SERVICE";
    Statement stm = c.createStatement();
    System.out.println("resultat (nombre de lignes) : " + stm.executeUpdate(sql));
} catch (ClassNotFoundException ex) {
    System.err.println(ex.getMessage());
} catch (SQLException ex) {
    System.err.println(ex.getMessage());
}
```

```
run:
resultat (nombre de lignes) : 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

#### Exemple 2 : Insérer une ligne dans la table « service ».

Les paramètres de connexion à la BD doivent être définis dans un fichier *properties*.

##### 1. création du fichier properties dans le projet



##### 2. lecture des paramètres contenus dans le fichier properties, puis exécution de la requête :

```

import java.util.Properties;
import java.io.IOException;
import java.io.InputStream;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class TestJDBC {
    protected static String PROPERTIES_FILE_PATH = "testjdbc/config/dao.properties";
    protected static String DBNAME;
    protected static String URL;
    protected static String USERNAME;
    protected static String PASSWORD;
    protected static String DRIVER;

    public static void main(String[] args) {
        try {
            Properties properties = new Properties();
            ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
            InputStream fichierProperties = classLoader.getResourceAsStream(PROPERTIES_FILE_PATH);
            properties.load(fichierProperties);

            # parametres de connexion
            dbname=pgest_test
            url=jdbc:mysql://localhost:3306/
            driver=com.mysql.jdbc.Driver
            user=root
            pass=pass1156

            DBNAME = properties.getProperty("dbname");
            URL = properties.getProperty("url");
            DRIVER = properties.getProperty("driver");
            USERNAME = properties.getProperty("user");
            PASSWORD = properties.getProperty("pass");

            Class.forName(DRIVER);
            Connection connex = DriverManager.getConnection(URL + DBNAME, USERNAME, PASSWORD);

            String sql = "INSERT INTO SERVICE(ID, NOM, DATE_CREATION, PARENT) " +
                "VALUES(NULL, 'SERVICE COMPTABILITE', '2012-10-15', NULL)";
            Statement stm = connex.createStatement();

```



```

        System.out.println("Nombre de lignes : " + stm.executeUpdate(sql));
    // fin block try
    } catch (Exception ex) {
        System.err.println(ex.getMessage());
    }
    stm.close(); connex.close();
    } // fin main
} // fin classe

```

Fermeture du statement puis de la connexion avant la fin du programme

```

run:
nombre de lignes : 1
BUILD SUCCESSFUL (total time: 0 seconds)

```

id	nom	date_creation	parent
1	SERVICE COMPTABILITE	2012-10-15	(Null)

### TP : gestion d'un cabinet d'avocat

1. Créer un nouveau projet NetBeans qui sera nommé « **avocat** », ayant l'arborescence suivante :

```

testjdbc (package racine)
testjdbc/TestJDBC.java (contient la méthode principale)
testjdbc/config
testjdbc/data
testjdbc/tools

```

2. Pour chacune de ces tables, créer avec NetBeans - dans le package **testjdbc/data** - un fichiers SQL (**dossier.sql**, **client.sql**, ...) qui va contenir le script **CREATE TABLE** :

```

DOSSIER (ID, DATE_CREATION)
CLIENT (ID, NOM, PRENOM, ADRESSE, TEL, #DOSSIER(DEFAULT NULL))
AFFAIRE (ID, MOTIF, DATE_CREATION, HONORAIRES, REST_PAIEMENT (DEFAULT 0),
    #DOSSIER(DEFAULT NULL))
AUDIENCE (ID, DATE_AUDIENCE, VERDICT(DEFAULT NULL), VILLE, INSTANCE,
    #AFFAIRE(DEFAULT NULL))
PIECE (ID, CHEMIN_IMG, #AFFAIRE(DEFAULT NULL))

```

COUR D'ASSISE /  
COUR D'APPEL

Respecter les contraintes suivantes :

- Les clefs primaires sont auto-incrémentées pour toutes les tables
- Utiliser les options suivantes pour les clefs étrangères :
  - « cascade » pour la mise à jour
  - « set null » pour la suppression

3. Dans le package nommé **tools**, créer une classe **FileUtil**, qui va contenir la méthode statique ***String readTextFile(String chemin)***. Cette méthode doit retourner le contenu du fichier dont le chemin est passé en paramètre.

Indication : Utiliser les classes **Path**, **Paths** et **Files** du package **java.nio.file** pour lire le fichier caractère par caractère.

```
// 1- obtenir le chemin du fichier sous forme d'un objet Path
Path chemin = Paths.get(chemin_complet_du_fichier);

// 2- extraire tous les caractères contenus dans le fichier, sous forme d'un tableau de
// type byte, grâce à la méthode Files.readAllBytes()
byte[] bytes = Files.readAllBytes(chemin);

// 3- transformer le tableau de bytes en chaîne de caractères
```

4. (Optionnel) Dans la classe FileUtil, ajouter la méthode **`ArrayList<String> lireProperties(ArrayList<String>)`**, qui prend en paramètre la liste des noms des propriétés, et qui retourne la liste des valeurs des propriétés.
5. Dans le dossier config, créer un fichier **`params.properties`**, qui va contenir – en plus des paramètres de connexion – les chemins des fichiers SQL.
6. Écrire un programme java qui va réaliser les tâches suivantes :
  - Récupérer les propriétés enregistrées dans le fichier properties (Optionnel : en utilisant la méthode **`lireProperties`**).
  - Créer la base de données, qui sera nommée « avocat ».  
(Rappel SQL : `CREATE DATABASE nom_base`)
  - Lire les scripts SQL contenus dans les fichiers **`dossier.sql`**, **`client.sql`**, ... puis les exécuter un par un.

## 4. LES REQUETES PARAMETREES

Exemple : On souhaite insérer la ligne suivante dans la table « client »

ID	NOM	PRENOM	ADRESSE	TEL	DOSSIER
1	EL'YAHYAOUUI	OMAR	RABAT HAY RIAD	O537445566	(Null)

```
try {
    Class.forName("com.mysql.jdbc.Driver");
    String adress_ip_mysql = "localhost", numéro_du_port = "3306",
        user = "root", pass = "", nom_base = "avocat";
    String connectionString = "jdbc:mysql://" + adress_ip_mysql
        + ":" + numéro_du_port + "/" + nom_base;

    Connection c = DriverManager.getConnection(connectionString, user, pass);

    String sql =
        "INSERT INTO CLIENT VALUES(1,'EL'YAHYAOUUI','OMAR','RABAT HAY RIAD','0537445566', NULL)";

    Statement stm = c.createStatement();
    System.out.println("resultat (nombre de lignes) : " + stm.executeUpdate(sql));
} catch (ClassNotFoundException ex) {
    System.err.println(ex.getMessage());
} catch (SQLException ex) {
    System.err.println(ex.getMessage());
}
```

Le résultat est une exception lors de l'exécution de l'instruction `stm.executeUpdate(sql)`.

Le message contenu dans l'exception est :

**You have an error in your SQL syntax;**  
**check ..... near 'YAHYAOUUI','OMAR','RABAT HAY RIAD','0537445566', NULL) '**

La cause de cette erreur SQL est le fait que l'une des valeurs contient le caractère ' qui, en SQL, ne doit pas être utilisé à l'intérieur d'une chaîne de caractères.

On utilise pour ce genre de cas des requêtes qu'on appelle *paramétrées* ou *préparées*, grâce à un autre type de *statements* : **PreparedStatement**.

**Définition : utilisation de PreparedStatement au lieu de Statement**

```
// 1. Chaque valeur est remplacée par le caractère ?
String sql = "INSERT INTO nom_table VALUES(?, ?, ?, ?, ?, NULL)";
// 2. Création d'un objet PreparedStatement
PreparedStatement pstmt = connex.prepareStatement(sql);
// 3. Initialisation des paramètres de la requête paramétrée selon le type et l'ordre de
// chaque paramètre
pstmt.setInt(1, une_valeur);
pstmt.setString(2, une_valeur);
pstmt.setDouble(3, une_valeur);
.....
// 4. Exécution de la requête
pstmt.executeUpdate(); // executeUpdate() sans paramètre
```



## REMARQUES

- Les **PreparedStatement** sont utilisables dans tous les types de requêtes SQL.
- La valeur SQL « **NULL** » est représentée par l'attribut statique **NULL** de la classe **java.sql.Types**.

### Exemple : Refaire l'insertion précédente en utilisant une requête paramétrée.

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

    String adress_ip_mysql = "localhost", numéro_du_port = "3306",
        user = "root", pass = "", nom_base = "avocat";
    String connectionString = "jdbc:mysql://" + adress_ip_mysql
        + ":" + numéro_du_port + "/" + nom_base;

    Connection c = DriverManager.getConnection(connectionString, user, pass);

    String sql = "INSERT INTO CLIENT VALUES(?, ?, ?, ?, ?, ?)";
    PreparedStatement pstmt = c.prepareStatement(sql);
    pstmt.setInt(1, Types.NULL);
    pstmt.setString(2, "EL 'YAHYAQUI");
    pstmt.setString(3, "OMAR");
    pstmt.setString(4, "RABAT HAY RIAD");
    pstmt.setString(5, "0537445566");
    pstmt.setInt(6, Types.NULL);

    System.out.println("resultat (nombre de lignes) : " + pstmt.executeUpdate());
} catch (ClassNotFoundException ex) {
    System.err.println(ex.getMessage());
} catch (SQLException ex) {
    System.err.println(ex.getMessage());
}
```

**TP : Suite TP précédent - gestion d'un cabinet d'avocat**

**7.** Dans le même projet nommé « **avocat** », écrire un programme qui fait l'insertion des lignes suivantes dans la table CLIENT.

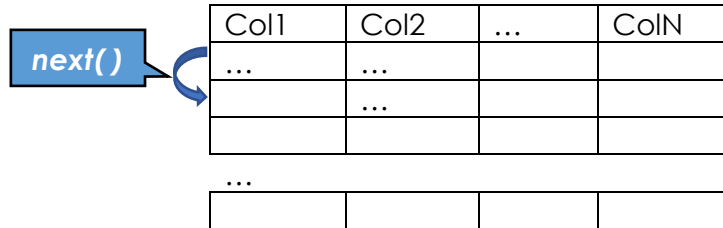
ID	NOM	PRENOM	ADRESSE	TEL	DOSSIER
1	EL'YAHYAOU	OMAR	RABAT HAY RIAD	0537445566	(Null)
2	SABER	ASMAE	RABAT HAY RIAD	0537445566	(Null)
3	MOUMEN	WALID	RABAT HASSAN	(Null)	(Null)
4	BELHAJ	YOUNES	RABAT HASSAN	(Null)	(Null)
5	ENHARI	YOUNES	RABAT HASSAN	(Null)	(Null)
6	FOUNOUNE	FATIMA	MOHAMEDIA	(Null)	(Null)
7	MAMADOU	BACHIR-ALI	RABAT	(Null)	(Null)

Les données doivent être lues depuis le clavier (sauf l'identifiant).

## 5. LES REQUETES SELECT

### Définition : utilisation de la classe ResultSet

Un objet **ResultSet** est une grille qui contient le résultat retourné par la requête **SELECT** exécutée par la méthode **executeQuery()**.



Pour parcourir cette grille (lignes/colonnes) il faut pouvoir passer de ligne en ligne, et pouvoir récupérer la valeur d'une cellule en fonction du nom (ou l'indice) de la colonne. La classe contient un ensemble de méthodes prévues à cet effet :

<b>next()</b>	<p>Passe de la ligne courante à la suivante et retourne <b>true</b> s'il y a une ligne suivante. Si le ResultSet ne contient plus de lignes, elle retourne <b>false</b>.</p> <p>Cette méthode est utilisée autant de fois que nécessaire jusqu'à la fin des lignes du ResultSet.</p>
<b>first()</b>	Place le curseur du ResultSet à la 1 <sup>ère</sup> ligne.
<b>last()</b>	Place le curseur du ResultSet à la dernière ligne.
<b>beforeFirst()</b>	<p>Place le curseur du ResultSet avant la 1<sup>ère</sup> ligne.</p> <p><b>Dès la création de l'objet ResultSet avec la méthode executeQuery, le curseur est placé par défaut avant la 1<sup>ère</sup> ligne.</b></p>
<b>afterLast()</b>	Place le curseur à la fin du ResultSet, après la dernière ligne.
<b>absolute(int ligne)</b>	Place le curseur exactement à la ligne dont l'indice est passé en paramètre. L'indexation commence par 1.
<b>getInt(int indice)</b>	<p>Retourne la valeur contenue dans une cellule, qui se trouve dans une colonne de type <b>int</b>, en fonction de la ligne courante, et de l'indice de colonne, passé en paramètre. L'indexation commence par 1.</p> <p>Accepte aussi en paramètre le nom de la colonne au lieu de l'indice. (<b>getInt("nom_colonne")</b>)</p>
<b>getString(int indice)</b>	<p>Idem que <b>getInt()</b>, pour les colonnes de type <b>String</b>.</p> <p>De même que <b>getDouble</b>, <b>getDate</b>, <b>getBlob</b>, ...</p>
<b>getRow()</b>	Retourne l'indice de la ligne courante. L'indexation commence par 1.
<b>getMetaData()</b>	Retourne un objet <b>ResultSetMetaData</b> . Cet objet contient toutes les métadonnées concernant l'objet ResultSet (nombre de colonnes, type & nom de chaque colonne, ...)
<b>setFetchDirection()</b>	<p>Détermine la direction dans laquelle la grille sera parcourue :</p> <p>Vers le haut : <b>rs.setFetchDirection(ResultSet.FETCH_REVERSE)</b></p> <p>Vers le bas : <b>rs.setFetchDirection(ResultSet.FETCH_FORWARD)</b></p>

```
// 0. Chargement du pilote et création de la connexion
// 1. Création de la requête (simple ou paramétrée)
// 2. Création d'un objet Statement ou PreparedStatement
```

```
// 3. Initialisation des paramètres en cas de requête paramétrée
// 4. Exécution de la requête
ResultSet rs = stm.executeQuery(sql); // ou executeQuery() en cas de requête paramétrée
// 5. Parcourir le résultat de la requête ligne par ligne
while(rs.next()) {
    // 6. Récupérer la valeur d'une cellule (Ex. : cellule de type int)
    int age = rs.getInt(indice_de_la_colonne_age);
}
```

**Exemple : Gestion cabinet d'avocat - Récupérer et afficher les infos nom et prénom de tous les clients qui habitent Rabat.**

```
try {
    // Création de la requête
    String sql = "SELECT NOM, PRENOM FROM CLIENT WHERE ADRESSE LIKE '%RABAT%'";
    // Création d'un objet Statement
    Statement stm = c.createStatement();
    // Exécution de la requête SELECT
    ResultSet rs = stm.executeQuery(sql);
    // Parcourir le résultat
    while(rs.next()) {
        System.out.println(rs.getString(1) + " - " + rs.getString(2));
    }
} catch (Exception ex) {
    System.err.println(ex.getMessage());
}
```

```
run:
EL'YAHYAOU I - OMAR
SABER - ASMAE
MOUMEN - WALID
BELHAJ - YOUNES
ENHARI - YOUNES
MAMADOU - BACHIR-ALI
BUILD SUCCESSFUL (total time: 0 seconds)
```

**TP : Suite TP précédent - gestion d'un cabinet d'avocat**

8. Dans un nouveau package **testjdbc/data/model**, créer la classe modèle qui correspond à chaque table, en respectant les types de colonnes.

9. Dans un nouveau package **testjdbc/data/dao**, créer la classe **ClientDAO** qui va offrir les actions du CRUD sur la table CLIENT, y compris l'option de recherche (recherche par id, par nom de famille ou par ville).

Cette classe doit avoir la structure suivante :

- **ArrayList<Client> clients** : attribut (privé) qui contiendra la liste des clients.
- **int insert(Client)** : méthode qui sert à ajouter un client dans la table. Si l'ajout se fait avec succès, le client est ajouté aussi dans la liste **ArrayList<Client> clients**.
- **int update(Client)** : même comportement que **insert**, mais pour la mise à jour.
- **int delete(int)** : même comportement que **insert**, mais pour la suppression. Cette méthode prend en paramètre l'identifiant du client au lieu de prendre tout l'objet client.
- **ArrayList<Client> getClients()** : *getter* de l'attribut **clients**. Si l'attribut est nul, la méthode récupère tous les clients depuis la table, et les charge dans l'attribut **clients** avant de le retourner.

10. Créer aussi la classe **DossierDAO** dans le même package. Cette classe doit offrir les mêmes services que la classe **ClientDAO**.



**IMPORTANT**

Les méthodes des classes DAO ne doivent pas créer les objets *Connection*, *Statement*, ..., elles doivent les prendre en paramètres.

11. Toujours dans le nouveau package **testjdbc/data/dao**, créer la classe **DAOUtils** qui va fournir les objets (*statement*, *resultset*, *connexion*, ...) nécessaires au bon fonctionnement des méthodes des classes DAO. Cette classe doit contenir les méthodes suivantes :

- **Connection getConnection()** : retourne un objet *Connection*
- **Statement getStatement()** : retourne un objet *Statement*
- **PreparedStatement getStatement(String sql)** : retourne un objet *PreparedStatement*

12. Pour le fonctionnement des méthodes de la classe **DAOUtils**, il faut la lecture des paramètres contenus dans le fichier *properties*. Réaliser et utiliser donc la méthode décrite dans la question 4, page 11.

13. La classe *TestJDBC* ne contiendra plus que le menu principal.



## 6. RECUPERER UNE CLEF GENeree PAR AUTO-INCREMENTATION

### Problème

Rappel SQL :

Lors de l'insertion d'une ligne dans une table **ayant une clef primaire auto-increment**, le SGBD génère une valeur automatiquement pour cette clef.

Problème :

Comment récupérer cette valeur au niveau de JDBC (dans une application de CRUD par ex.) ?

### Définition : `Statement.RETURN_GENERATED_KEYS`

Rappel :

Lors de la préparation d'un objet `PreparedStatement` pour une requête **INSERT** paramétrée, il est créé par défaut avec la syntaxe suivante :

```
pstm = connex.prepareStatement(sql);
```

#### 1<sup>iere</sup> partie :

Si on souhaite récupérer la valeur de la clef générée automatiquement, un paramètre supplémentaire doit être ajouté lors de l'appel de la méthode **`prepareStatement()`**. On doit prévenir l'objet **connex**, qu'il doit nous préparer un statement destiné à faire des insertions auto-increment, et ce grâce au paramètre **`RETURN_GENERATED_KEYS`**, qui est défini (de manière statique) dans la classe **Statement**.

```
pstm = connex.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);
```

#### 2<sup>ieme</sup> partie :

Après exécution de la méthode, on récupère la clef générée (ou les clefs générées, en cas d'insertion multiple) avec la méthode **`getGeneratedKeys()`** qui retourne un objet **resultSet**, ayant une seule colonne, contenant la liste des identifiants générés.

```
pstm = connex.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);  
pstm.executeUpdate();  
ResultSet keys = pstm.getGeneratedKeys();  
while(keys.next()) {  
    System.out.println(keys.getInt(1));  
}
```

### TP : Gestion d'un cabinet d'avocat – Suite

14. Modifier le programme écrit dans la question 9 :

L'identifiant de chaque objet java inséré dans la table, doit être initialisé par la valeur générée par auto-incrémentation.

## 7. LES RESULTSETS MODIFIABLES

### Définition : utilisation de la classe ResultSet

Un **ResultSet** dit *modifiable* ou *updatable* est un resultset qui permet non seulement de récupérer le contenu d'une table, mais aussi de le modifier.  
Cet objet permet de récupérer une « copie » de la table en mémoire, et chaque opération INSERT / UPDATE / DELETE effectuée sur l'objet resultset, impacte directement la table en temps réel.  
Ce mode ne nécessite pas d'effectuer les actions [connexion → création de statement → exécution de requête] à chaque opération. Ces actions sont gérées par l'objet resultset.

```
// 0. Chargement du pilote et création de la connexion
// 1. Création de la requête « SELECT * FROM ... » pour récupérer toute la table dans
// l'objet resultSet
// 2. Création de l'objet Statement, en passant à la méthode createStatement( ) les 2
// paramètres suivants :
// ResultSet.TYPE_SCROLL_INSENSITIVE :
// ResultSet.CONCUR_UPDATABLE :
stm = c.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
// 3. Exécution de la requête
ResultSet rs = stm.executeQuery(sql);
// 4. Parcourir le resultSet : idem que pour un simple resultSet
// 5. Effectuer une action CRUD
// 5.1 Modifier une ligne :
// parcourir les lignes jusqu'à la ligne ciblée (next(), absolute(), ...)
// mettre à jour les cellules
rs.updateInt(indice_colonne, nouvelle_valeur); // (ou updateString, updateDouble, ...)
// appliquer les modifications
rs.updateRow();
// 5.2 Supprimer une ligne :
// parcourir les lignes jusqu'à la ligne ciblée (next(), absolute(), ...)
// appliquer la suppression
rs.deleteRow();
// 5.3 Ajouter une ligne :
// créer une nouvelle ligne vierge et positionner le curseur sur cette ligne
rs.moveToInsertRow();
// mettre à jour les cellules
rs.updateInt(indice_colonne, nouvelle_valeur); // (ou updateString, updateDouble, ...)
// appliquer l'insertion
rs.insertRow();
```



### IMPORTANT

L'objet ResultSet utilise un grand nombre de ressources mémoire, surtout si on l'utilise pour des actions de CRUD. Il est plus conseillé d'utiliser la méthode traditionnelle : connexion → création du statement → exécution de requête → fermeture des statement & connexion.

**Exemple : Gestion cabinet d'avocat - Utiliser un resultSet modifiable pour supprimer le client numéro 7 et ajouter un nouveau client.**

```
// Récupérer toute la table
String sql = "SELECT * FROM CLIENT";
// Création d'un objet Statement qui donnera des ResultSets updatables
Statement stm =
    c.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);

// Création du resultSet
ResultSet rs = stm.executeQuery(sql);

// Supprimer le client numero 7
while(rs.next()) {
    if(rs.getInt("id") == 7)
        rs.deleteRow();
}

// Ajouter un nouveau client
rs.moveToInsertRow();
rs.updateString("nom", "HILALI");
rs.updateString("prenom", "MOHCINE");
rs.updateString("adresse", "KENITRA");
rs.insertRow();
```

ID	NOM	PRENOM	ADRESSE	TEL	DOSSIER
1	EL'YAHYAOU	OMAR	RABAT HAY RIAD	0537445566	(Null)
2	SABER	ASMAE	RABAT HAY RIAD	0537445566	(Null)
3	MOUMEN	WALID	RABAT HASSAN	(Null)	(Null)
4	BELHAJ	YOUNES	RABAT HASSAN	(Null)	(Null)
5	ENHARI	YOUNES	RABAT HASSAN	(Null)	(Null)
6	FOUNOUNE	FATIMA	MOHAMEDIA	(Null)	(Null)
8	HILALI	MOHCINE	KENITRA	(Null)	(Null)