# ACCEDER A UNE BASE DE DONNEES AVEC JDBC

# 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.

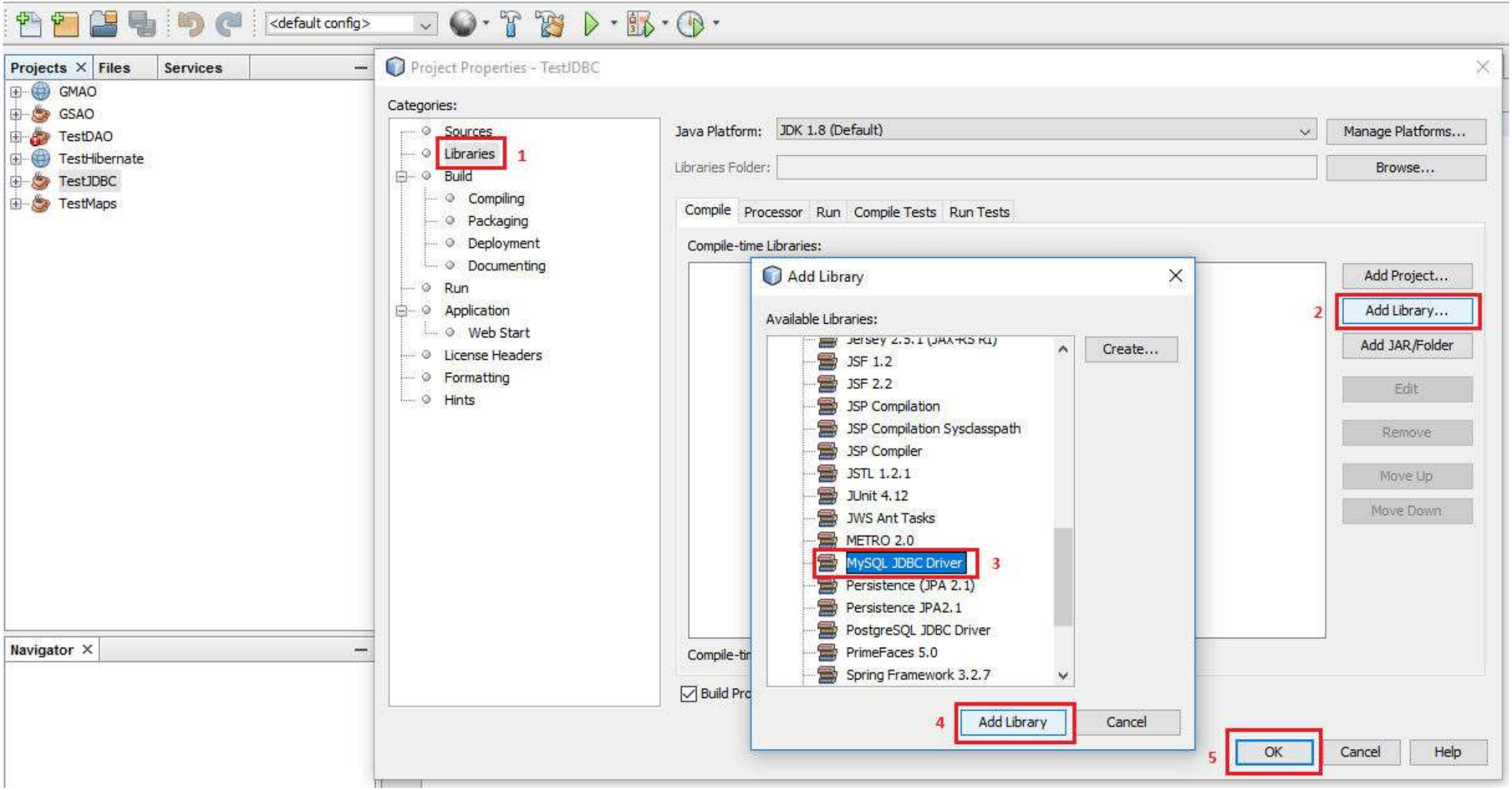
Chaque SGBD – ex. MySQL – fournit un pilote pour chaque langage de programmation, y compris le pilote Java, qu'on appelle le pilote **JDBC**, qui est un ensemble de classes précompilées regroupées dans un fichier jar. Le pilote **JDBC** de **MySQL** est téléchargeable sur cette page :

https://repo1.maven.org/maven2/mysql/mysql-connector-java/8.0.28/mysql-connector-java-8.0.28.jar

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

Ouvrir les **propriétés** du projet ciblé : Clique 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 :

# Ajouter le pilote JDBC de MySQL dans le projet NetBeans



# Ajouter le pilote JDBC de MySQL

**Définition du processus**

- Le processus standard qu'on doit suivre pour exécuter et exploiter une requête SQL est le suivant :
  1. **Charger la classe du driver dans la mémoire de notre programme**
  2. **Etablir la connexion avec le serveur**
  3. **Créer la requête SQL**
  4. **Exécuter la requête SQL**
  5. **Fermer le statement et la connexion**

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 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 et l'Exécuter

**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( )**

# 3. Créer la requête SQL et l'Exécuter

```
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 {
  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);
  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());
}
```

# LES REQUETES DE MISE A JOUR

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.

# LES REQUETES DE MISE A JOUR

**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());
}
```

# LES REQUETES DE MISE A JOUR

**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**


```
1  # parametres de connexion
2  dbname=pgest_test
3  url=jdbc:mysql://localhost:3306/
4  driver=com.mysql.jdbc.Driver
5  user=root
6  pass=pass1156
7
8
```

# Ressources

Java permet la récupération des ressources basées sur des fichiers stockées dans un fichier JAR à côté des classes compilées. Cette rubrique se concentre sur le chargement de ces ressources et leur mise à disposition dans votre code.
**Une ressource** est une donnée de type fichier avec un nom de chemin d'accès, qui réside dans le chemin de classe. L'utilisation la plus courante des ressources consiste à regrouper des images d'application, des sons et des données en lecture seule (telles que la configuration par défaut).
Les ressources sont accessibles avec les méthodes **ClassLoader.getResource** et **ClassLoader.getResourceAsStream**. Le cas d'utilisation le plus courant est d'avoir des ressources placées dans le même package que la classe qui les lit; les méthodes **Class.getResource** et **Class.getResourceAsStream** servent ce cas d'utilisation commun.
La seule différence entre une méthode **getResource** et la méthode **getResourceAsStream** est que la première renvoie une **URL**, tandis que la seconde **ouvre** cette **URL** et renvoie un **InputStream**.

# Ressources

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

```
String URL,DRIVER,USERNAME,PASSWORD,DBNAME;
Properties defaults = new Properties();
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
InputStream defaultsStream = classLoader.getResourceAsStream("jdbc/Config/Default_Config.properties");
defaults.load(defaultsStream);
DBNAME = defaults.getProperty("dbname");
URL = defaults.getProperty("url");
DRIVER = defaults.getProperty("driver");
USERNAME = defaults.getProperty("user");
PASSWORD = defaults.getProperty("password");
```

```
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();
```

# LES REQUETES PARAMETREES

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

3. Initialisation des paramètres de la requête paramétrée selon le type et l'ordre de chaque paramètre

```
pstm.setInt(1, une_valeur);
pstm.setString(2, une_valeur);
pstm.setDouble(3, une_valeur);
……
```

4. Exécution de la requête

```
pstm.executeUpdate(); // executeUpdate() sans paramètre
```

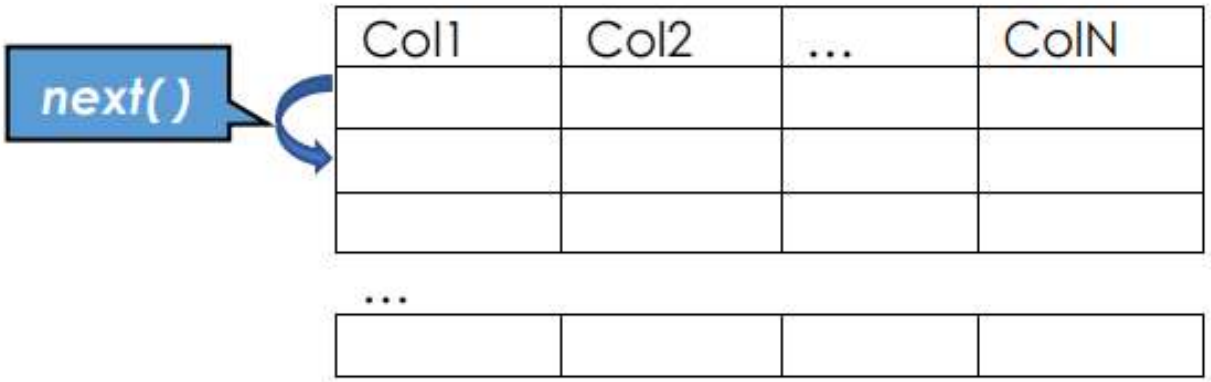
**Remarque:**

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**

# LES REQUETES SELECT

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 :

# LES REQUETES SELECT

| | |
|---|---|
| ***next( )*** | Passe de la ligne courante à la suivante et retourne `true` s'il y a une ligne suivante. Si le ResultSet ne contient plus de lignes, elle retourne `false`.<br>Cette méthode est utilisée autant de fois que nécessaire jusqu'à la fin des lignes du ResultSet. |
| ***first( )*** | Place le curseur du ResultSet à la 1iere ligne. |
| ***last( )*** | Place le curseur du ResultSet à la dernière ligne. |
| ***beforeFirst( )*** | Place le curseur du ResultSet avant la 1iere ligne.<br>**Dès la création de l'objet ResultSet avec la méthode *executeQuery*, le curseur est placé par défaut avant la 1iere ligne.** |
| ***afterLast( )*** | Place le curseur à la fin du ResultSet, après la dernière ligne. |
| ***absolute(int ligne)*** | Place le curseur exactement à la ligne dont l'indice est passé en paramètre. L'indexation commence par 1. |
| ***getInt(int indice)*** | Retourne la valeur contenue dans une cellule, qui se trouve dans une colonne de type **int**, en fonction de la ligne courante, et de l'indice de colonne, passé en paramètre. L'indexation commence par 1.<br>Accepte aussi en paramètre le nom de la colonne au lieu de l'indice. (***getInt("nom_colonne")***) |

# LES REQUETES SELECT

| | |
|---|---|
| ***getString(int indice)*** | Idem que ***getInt( )***, pour les colonnes de type **String.**<br>De même que ***getDouble***, ***getDate***, ***getBlob***, … |
| ***getRow( )*** | Retourne l'indice de la ligne courante. L'indexation commence par 1. |
| ***getMetaData( )*** | Retourne un objet **ResultSetMetaData**. Cet objet contient toutes les métadonnées concernant l'objet ResultSet (nombre de colonnes, type & nom de chaque colonne, …) |
| ***setFetchDirection( )*** | Détermine la direction dans laquelle la grille sera parcourue :<br>Vers le haut : `rs.setFetchDirection(ResultSet.FETCH_REVERSE)`<br>Vers le bas : `rs.setFetchDirection(ResultSet.FETCH_FORWARD)` |

```
// 0. Chargement du pilote et création de la conncexion
// 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);
}
```

# LES REQUETES SELECT

**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());
}
```

# RECUPERER UNE CLEF GENEREE PAR AUTO-INCREMENTATION

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.) ?

## Solution : Statement.RETURN_GENERATED_KEYS

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);**

## 1iere 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);**

# RECUPERER UNE CLEF GENEREE PAR AUTO-INCREMENTATION

**2ieme 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));
}
```

# LES RESULTSETS MODIFIABLES

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_SENSITIVE :
// 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();
```

**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.**

# LES RESULTSETS MODIFIABLES

**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();
```