Introduction

JDBC, Java Data Base Connectivity est un ensemble de classes (API – Application Programming Interface --JAVA) permettant de se connecter à une base de données relationnelle en utilisant des requêtes SQL ou des procédures stockées.

L'API JDBC a été développée de manière à pouvoir se connecter à n'importe quelle base de données avec la même syntaxe; cette API est dite indépendante du SGBD utilisé.

Introduction

Les classes JDBC font partie du package **java.sql** et

javax.sql JDBC permet:

1. L'établissement d'une connexion avec le SGBD.
2. L'envoi de requêtes SQL au SGBD, à partir du programme java: création de tables, sélection de données,...
3. Le traitement, au niveau du programme, des données retournées par le SGBD.
4. Le traitement des erreurs retournées par le SGBD lors de l'exécution d'une instruction.

Introduction

Pilote de bases de données ou driver JDBC

- Un pilote ou driver JDBC est un "logiciel" qui permet de convertir les requêtes JDBC en requêtes spécifiques auprès de la base de données.

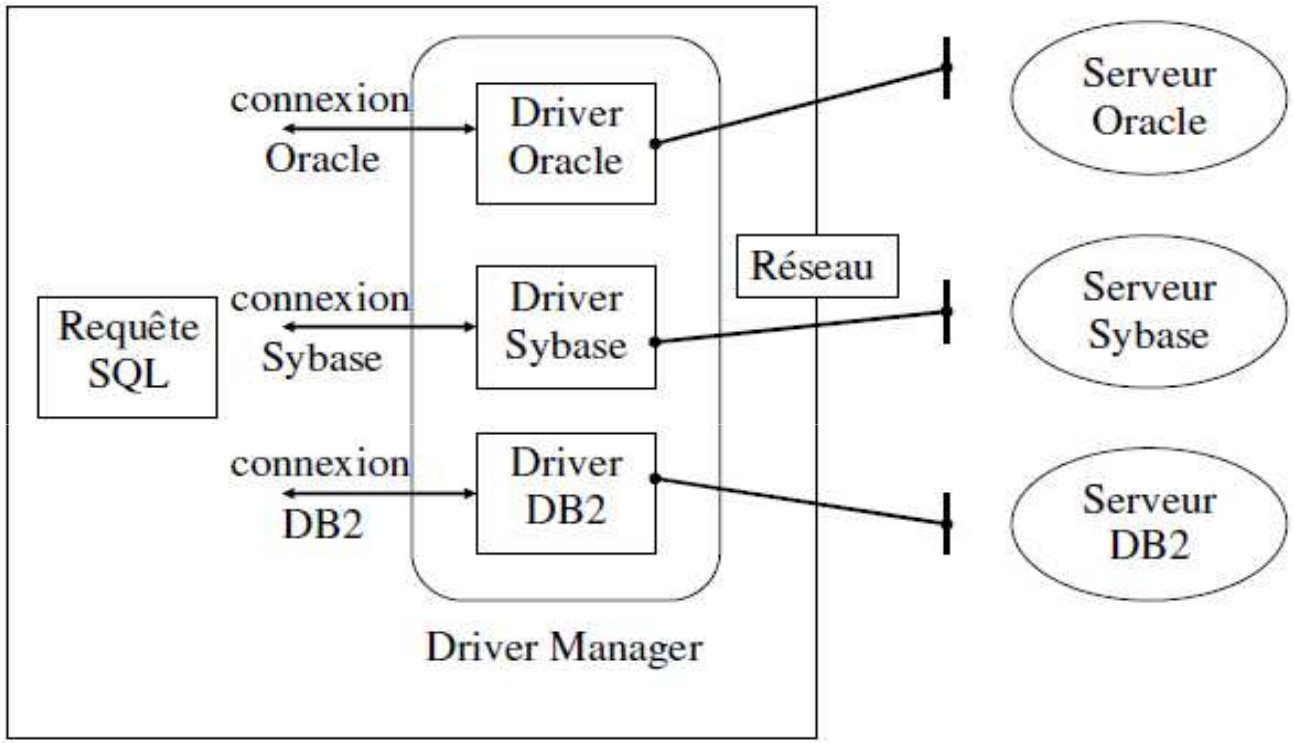
- Ce "logiciel" est en fait une implémentation de l'interface Driver, du package java.sql.

Le problème de l'accès aux données

- Java oui mais, qu'avons nous besoin réellement ?
  - Faire des requêtes SQL. SQL étant le langage permettant de dialoguer avec un SGBD.
  - Prendre cette requête et la transférer telle quelle indépendamment du système et du SGBD

- mais il était difficile d'accéder à des bases de données SQL depuis Java :
  - obligé d'utiliser des API natives comme le driver ODBC

Le driver et son rôle …



- Chaque base de données utilise un pilote (*driver*) qui lui est propre et qui permet de convertir les requêtes dans le langage natif du SGBDR.

le driver ODBC de Microsoft

- **O**pen **D**ata**B**ase **C**onnectivity
  - Est un format défini par Microsoft permettant la communication entre des clients bases de données fonctionnant sous Windows et les SGBD du marché.
  - Est devenu un standard de fait du monde Windows
  - Est un ensemble API permettant la communication entre des clients de bases de données et les SGBD.
  - tous les constructeurs de SGBD fournissent un driver ODBC

Avantages et inconvénients de ODBC

- Avantages :
  - possibilité d'écrire des applications accédant à des données réparties entre plusieurs sources hétérogènes
- Inconvénients :
  - cette technologie reste une solution pour windows!
  - ODBC est fortement lié au langage C

Le but recherché

- Permettre aux programmeurs d'écrire un code indépendant de la base de données et du moyen de connectivité utilisé.

→ JDBC

Qu’est ce que JDBC ?

- **J**ava **D**ata**B**ase **C**onnectivity (*Core* API 1.1)
  - Fournit un ensemble de classes et d’interfaces permettant l’utilisation sur le réseau d’un ou plusieurs SGBD à partir d’un programme Java.
  - Ce paquetage permet de formuler et gérer les requêtes aux bases de données relationnelles
  - JDBC est fourni par le paquetage java.sql

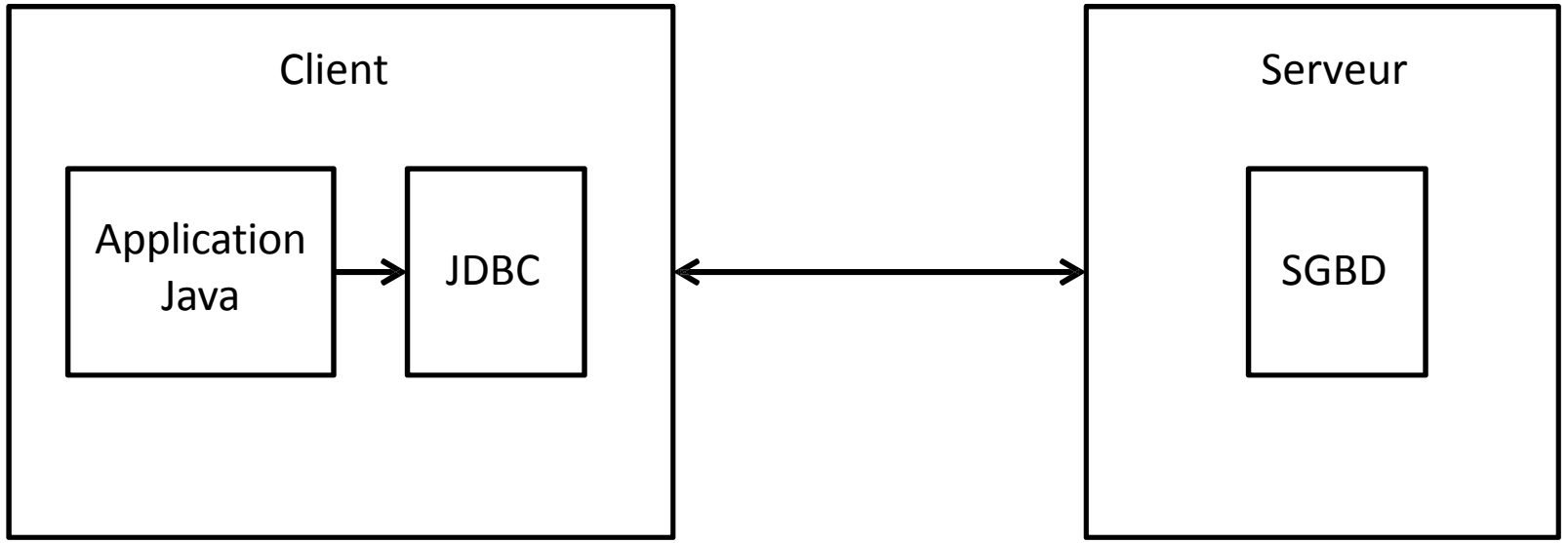
Avantages

- une interface uniforme permettant un accès homogène aux SGBD
- simple à mettre en œuvre
- JDBC est complètement indépendant de tout SGBD
- Avantages liése a Java :
  - portabilité sur de nombreux O.S. et sur de nombreuses SGBDR (Oracle, Informix, Sybase, ..)
  - uniformité du langage de description des applications et des accès aux bases de données

API JDBC

- Est fournie par le package **java.sql**
  - permet de formuler et gérer les requêtes aux bases de données relationnelles
  - 8 interfaces définissant les objets nécessaires :
    - à la connexion à une base éloignée
    - et à la création et exécution de requêtes SQL

Exemple d'utilisation de JDBC : Architectures 2-tiers



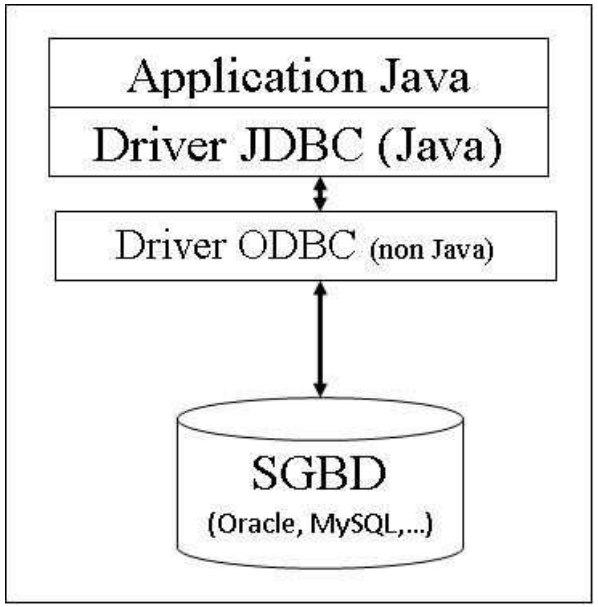
Principe de fonctionnement

- Chaque base de données utilise un pilote (*driver*) qui lui est propre et qui permet de convertir les requêtes JDBC dans le langage natif du SGBDR.

- 4 types de drivers (taxonomie de JavaSoft) :
  - Type I : JDBC-ODBC *bridge driver*
  - Type II : *Native-API, partly-Java driver*
  - Type III : *Net-protocol, all-Java driver*
  - Type IV : *Native-protocol, all-Java driver*
- Tous les drivers
  - http://www.javasoft.com/products/jdbc/drivers.html

Types de drivers JDBC(1/4)

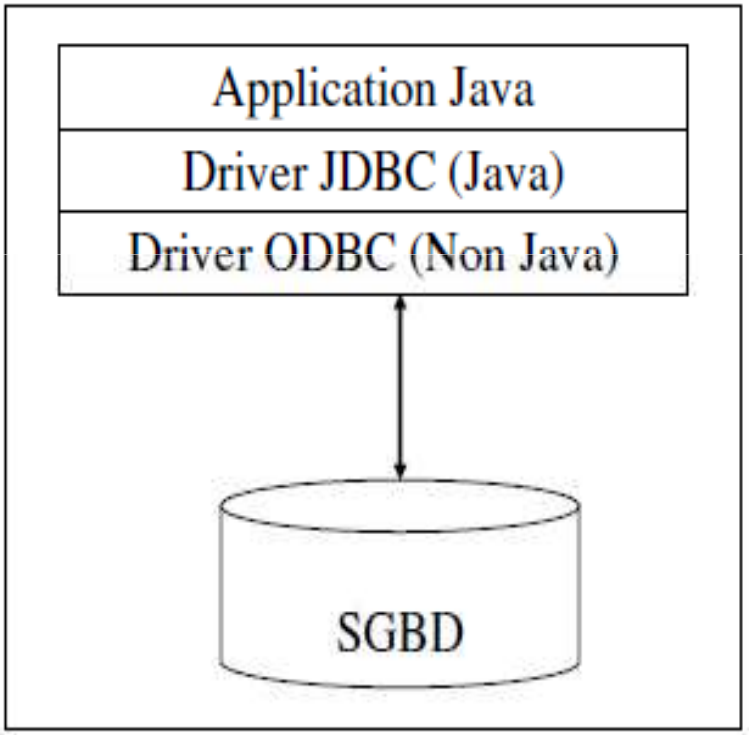
Les drivers de Type 1 : ODBC-JDBC bridges, ODBC (Open Data Base Connectivity) est une interface propre à Microsoft et qui permet l'accès à n'importe quelle base de données



Chaque requête JDBC est convertie par ce pilote en requête ODBC qui est par la suite convertie une seconde fois dans le langage spécifique de la base de donnée.

Exemple de Driver JDBC-ODBC

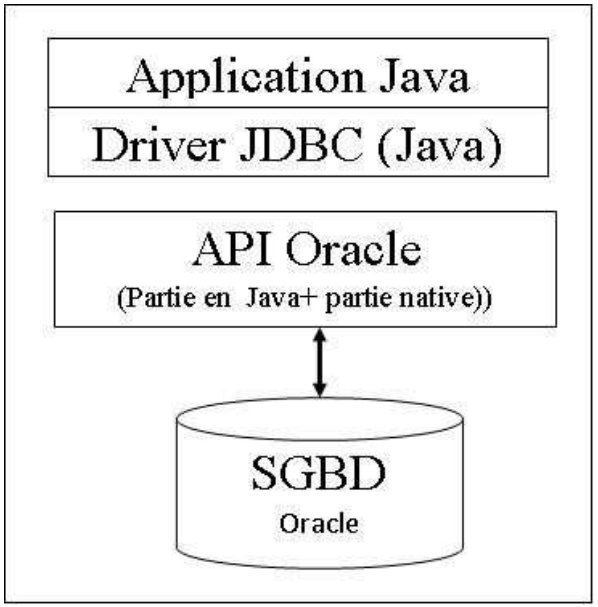
- Le driver JDBC accède à un SGBDR en passant par les drivers ODBC :
  - les appels JDBC sont traduits en appels ODBC
  - est fourni par SUN avec le JDK
    - **sun.jdbc.odbc.JdbcOdbcDriver**



Types de drivers JDBC(2/4)

Les drivers de Type 2

Ce type de driver traduit les appels de JDBC à un SGBD particulier, grâce à un mélange d'API java et d'API natives. (propre au SGBD).



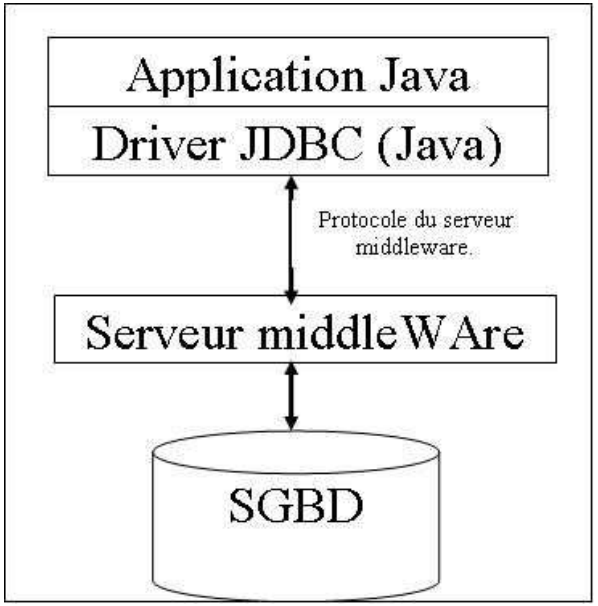
Ce Driver est fourni par l'éditeur de SGBD

Il est de ce fait nécessaire de fournir au client l'API native de la base de données.

Si on change le type de la base de données, on doit changer le pilote.

Types de drivers JDBC(3/4)

Drivers de type 3
(complètement écrit en JAVA)



Permet la connexion à une base de données via un serveur intermédiaire régissant l'accès aux multiples bases de données

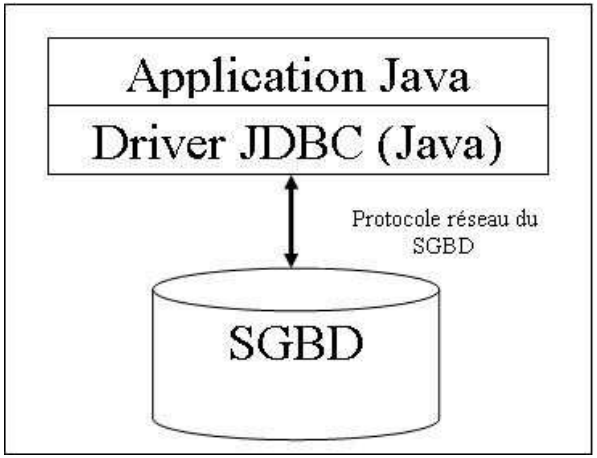
Ce type de driver est portable car écrit entièrement en java. Il est adapté pour le Web. Cela exige une autre application serveur à installer et à entretenir.

Ce type de driver peut être facilement utilisé par une applet, mais dans ce cas le serveur intermédiaire doit obligatoirement être installé sur la machine contenant le serveur Web.

Types de drivers JDBC(4/4)

Drivers de type 4 (complètement écrit en JAVA)

Ce type de driver est connu sous le nom Direct Database Pure Java Driver), permet d'accéder directement à la base de données (sans ODBC ni Middleware). C'est le type le plus optimal.



Liste des interfaces de l'API JDBC

- Driver : renvoie une instance de **Connection**
- Connection : connexion à une base
- Statement : requête SQL
- PreparedStatement : requête SQL précompilé
- CallableStatement : procédure stockée sur le SGBD
- ResultSet : lignes récupérées par un ordre SELECT
- ResultSetMetaData : description des lignes récupérées par un SELECT
- DatabaseMetaData : informations sur la base de données

Liste des classes principales

- DriverManager : gère les drivers, lance les connexions aux bases
- Date : date SQL
- Time : heures, minutes, secondes SQL
- TimeStamp : comme Time avec une précision à la microseconde
- Types : constantes pour désigner les types SQL (pour les conversions avec les types Java)

Liste des exceptions

- SQLException : erreurs SQL est levée dès qu'une connexion ou un ordre SQL ne se passe pas correctement
  - la méthode **getMessage()** donne le message en clair de l'erreur
- SQLWarning : avertissements SQL
- DataTruncation : avertit quand une valeur est tronquée lors d'un transfert entre Java et le SGBD

Mettre en œuvre JDBC

- 0. Importer le package **java.sql**
- 1. Enregistrer le driver JDBC
- 2. Etablir la connexion à la base de données
- 3. Créer une zone de description de requête
- 4. Exécuter la requête
- 5. Traiter les données retournées
- 6. Fermer les différents espaces

Importer le package java.sql

- il faut définir la variable d'environnement CLASSPATH pour inclure le répertoire contenant les classes du driver

  CLASSPATH=/ora0/app/oracle/product/8.1.5/jdbc/lib/classes12.zip

- import java.sql.*;

Enregistrer le driver JDBC

- Méthode **forName()** de la classe **Class** :
  - **Class.forName("sun.com.mysql.jdbc.Driver"); // pour Mysql**
  - **Class.forName("oracle.jdbc.driver.OracleDriver"); //pour oracle**
  - **Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver"); // pour Sql Server**

- Cette méthode charge en mémoire la classe demandée et elle lève des exceptions
- La méthode **Class.forName(string driver)** fait partie du pckage **java.lang et** peut lancer une exception de type «ClassNotFoundException ».

Exemple d'enregistrement d'un driver

Quand une classe **Driver** est chargée, elle doit créer une instance d'elle même et s'enregistrer auprès du **DriverManager.**

```
Try
    {
     Class.forName ("oracle.jdbc.driver.OracleDriver");

    System.out.println("Pilote chargé");
    }

  catch(ClassNotFoundException cnfe)
    {
    System.out.println("ERREUR : Driver manquant.");
    }
```

Url de connexion (1/2)

- Accès à la base via un URL de la forme :

  protocole:sous_protocole :{hôte ou ip}:{port}**:nom**

- qui spécifie :
  - <Protocole>: Le protocole dans une URL JDBC est toujours jdbc
  - <sous_protocole> Cela correspond au nom du driver ou au mécanisme de connexion à la base de données.
  - <nom >correspond au nom de la base de donnée

Url de connexion (2/2)

- Exemples
  - jdbc:mysql://localhost/maBase
  - jdbc:oracle:oci8@170.33.43.123:1521:maBase
  - jdbc:oracle:thin@:maBase
  - jdbc:sybase:Tds:localhost:5020/maBase
  - jdbc:postgresql://199.201.123.22:5400/database1

Connexion à la base

- Méthode **getConnection()** de **DriverManager**
- 3 arguments :
  - l'URL de la base de données
  - le nom de l 'utilisateur de la base
  - son mot de passe
- **Connection** connexion = **DriverManager.getConnection**(url,user,password);

- Exemple :

Connection conn = DriverManager.getConnection(url, "toto", "mdp");

Exemple de connexion à la base

```
public static void main(String[] args) {
String url = "jdbc:oracle:thin:@172.17.200.251:1521:orcl";  String usager ="usager1";
String motdepasse="oracle1";
        try
      {Class.forName ("oracle.jdbc.driver.OracleDriver");  System.out.println("Pilote chargé");}
      catch(ClassNotFoundException cnfe)
      {System.out.println("ERREUR : Driver manquant.");}
try{
       Connection connexion = DriverManager.getConnection(url,usager, motdepasse);
      System.out.println("connécté");
      }
      catch (SQLException se)
      {System.out.println("ERREUR : bd  manquante ou connexion
invalide.");
      }}
```

Charger le pilote

Établir la connexion

Créer une zone de description de requête (1/2)

- L'objet **Statement** possède les méthodes nécessaires pour réaliser les requêtes sur la base associée à la connexion dont il dépend
- 3 types de **Statement** :
  - **Statement** : requêtes statiques simples
  - **PreparedStatement** : requêtes dynamiques précompilées (avec paramètres d'entrée/sortie)
  - **CallableStatement** : procédures stockées

Créer une zone de description de requête (2/2)

- A partir de l'instance de l'objet **Connection**, on récupère le **Statement** associé en utilisant la méthode :

Statement req1 = connexion.**createStatement**();

PreparedStatement req2 = connexion.**prepareStatement**(str);

CallableStatement req3 = connexion.**prepareCall**(str);

Création d'un statement

- A partir de l'instance de l'objet **Connection**, on récupère le **Statement** associé en utilisant la méthode **createStatement** :

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

- Permet d'exécuter deux types de requêtes :
  - Les requêtes de modification de la base
  - Les requêtes de consultation de la base
- Toutes les méthodes doivent prendre en compte l'exception SQLException

Exécution d'une requête (1/3)

- 2 types d'exécution :
  - **executeQuery()** : pour les requêtes (SELECT) qui retournent un **ResultSet** (tuples résultants)
  - **executeUpdate()** : pour les requêtes (INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE) qui retournent un entier (nombre de tuples traités)

Exécution d'une requête (2/3)

- **executeQuery()** et **executeUpdate()** de la classe **Statement** prennent comme argument une chaîne (**String**) indiquant la requête SQL à exécuter :

```
Statement st = connexion.createStatement();
ResultSet rs = st.executeQuery ("chaines de
                                 caractères");

int nb = st.executeUpdate("chaines de caractères");
```

•Toutes les méthodes doivent prendre en compte l'exception SQLException

Exemple d’exécution d’une requête

```
ResultSet résultas=null;

String Req1 = "INSERT INTO client
    VALUES(3,'client_Nom',‘client_prenom')";

String Req2 = "SELECT * FROM client";  try {
    Statement stmt = con.createStatement();
     int nbMaj = stmt.executeUpdate(Req1);
    affiche("nb mise a jour = "+nbMaj);
     résultats = stmt.executeQuery(Req2);
              catch (SQLException e) {
//traitement de l'exception
}
```

Manipuler les données retournées

- L'interface ResultSet permet d'accéder ligne par ligne au résultat retourné par une requête.
- Seules les données demandées sont transférées en mémoire par le driver JDBC.
- La méthode next()
  - retourne false si le dernier enregistrement est lu
  - fait avancer le curseur sur l'enregistrement suivant
  - while(rs.next()) {// Traitement de chaque ligne}

Les méthodes de la classe ResultSet

- La classe ResultSet dispose des méthodes suivantes :

  //on peut parcourir le ResultSet d'avant en arrière :
  - next() vs. previous()

  // en déplacement absolu : aller à la n-ième ligne
  - absolute(int row), first(), last(), ...

  //en déplacement relatif : aller à la n-ième ligne à partir de la position courante du curseur, …
  - relative(int row), afterLast(), beforeFirst(), ...

  Date getDate(int i); // retourne l'objet Date de la colonne i int getInt(String col); //retourne un entier de la colonne col
  String getString(int i); //retourne l'objet String de la colonne i

Traitements des résultats

- La classe ResultSet fournit des méthodes pour récupérer dans le code Java les valeurs des colonnes des lignes renvoyées par le SELECT:
  - getXXX(int numéroColonne)
  - getXXX(String nomColonne)

- Les colonnes sont référencées par leur numéro ou par leur nom

- L'accès aux valeurs des colonnes est réalisé par des méthodes de la forme get*Type*() où *Type* est un type Java compatible avec le type de la colonne

```
int id = rs.getInt(" Aut_id");
String nom = rs.getString("aut_nom");
String prenom = rs.getString(3);
```

Exemple de manipulation des données

```
String Req2 = "SELECT * FROM client";
try {
    Statement stmt = con.createStatement();
    resultset res = stmt.executeQuery(Req2);
        while(res .next()) {
            int i = res.getInt("id");
            String n = res.getString("Nom");
             String p = res.getString("prenom");
        }
catch (SQLException e) {
//traitement de l'exception
System.out.println(e.getmessage());
}
```