



Java Avancé – Cours 5

Accès aux BD via JDBC, DAO

V. DESLANDRES, I. GUIDARA

veronique.deslandres@univ-lyon1.fr

Mai 2018

Plan de ce cours

- Introduction : driver JDBC, API java
- Connexion avec DriverManager ----- [11](#)
- Connexion avec un DataSource ----- [16](#)
 - Fichier propriétés IUT ----- [24](#) et [25](#)
- Ecriture requêtes SQL ----- [26](#)
- SQL pré formaté : ----- [41](#)
- Commit / rollback : ----- [45](#)
- Mode synchrone / asynchrone ----- [47](#)
- Procédure stockée ----- [49](#)

Introduction

Java → monde des **OBJETS**

SGBD R → monde des **RELATIONS**

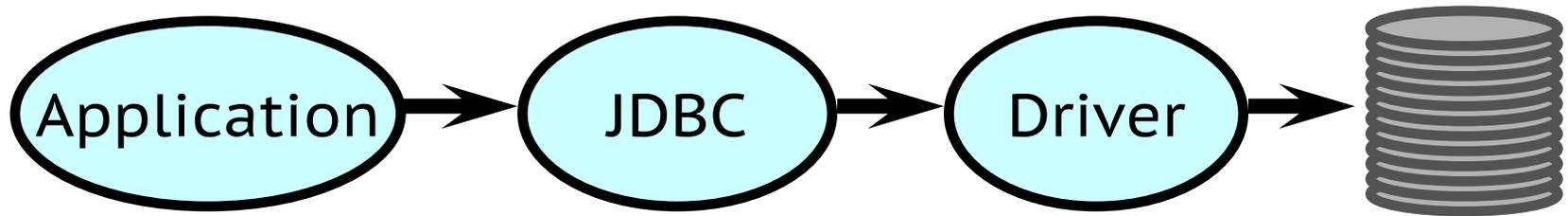
Un mapping O-R est nécessaire (ORM)

- API JDBC

Java Database Connectivity

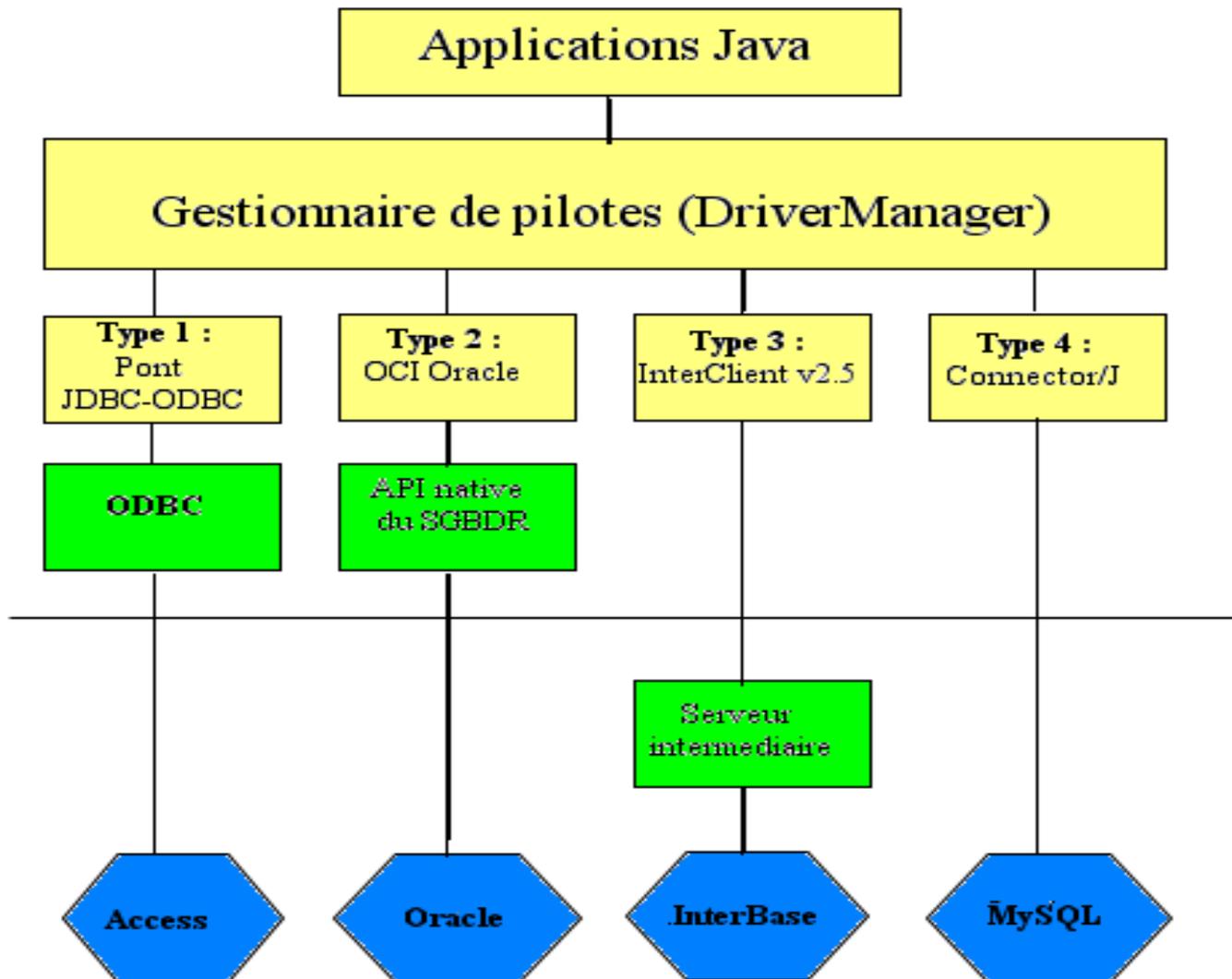
- Accès standardisé aux bases de données,
- Exploitation du SQL,
 - LMD,
 - LDD
- Support des protocoles réseaux

Architecture



- Un pilote : propre à UNE base de données
- Une même application peut utiliser plusieurs pilotes pour accéder à plusieurs BD
- Idéal : pouvoir changer de SGBD *sans modifier le code de l'application* (pas toujours possible)

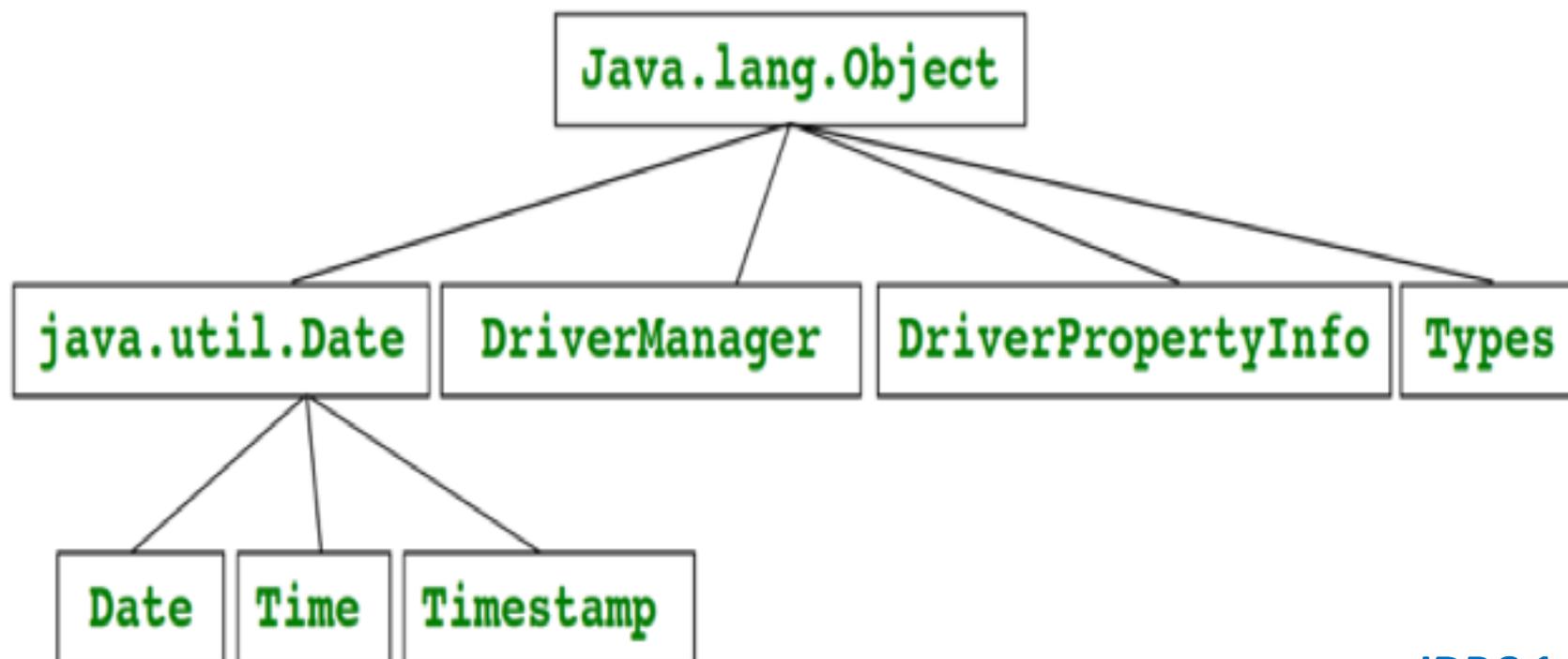
Les 4 niveaux de l'API JDBC



JDBC4 : standard de l'API Java

`java.sql` et `javax.sql`

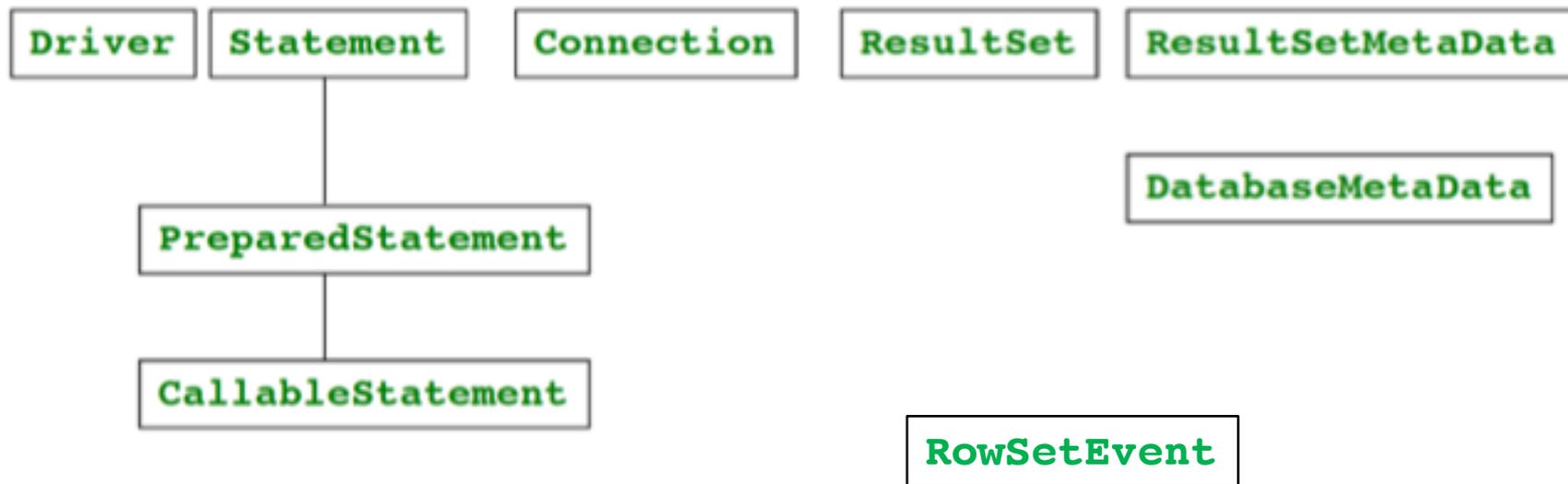
- Les **classes** de `java.sql`



JDBC 1.4.2

Les packages JDBC de l'API

Les interfaces de `java.sql`



Dans `javax.sql`, des classes :

`RowSetEvent`

`ConnectionEvent`

des interfaces :

`DataSource`

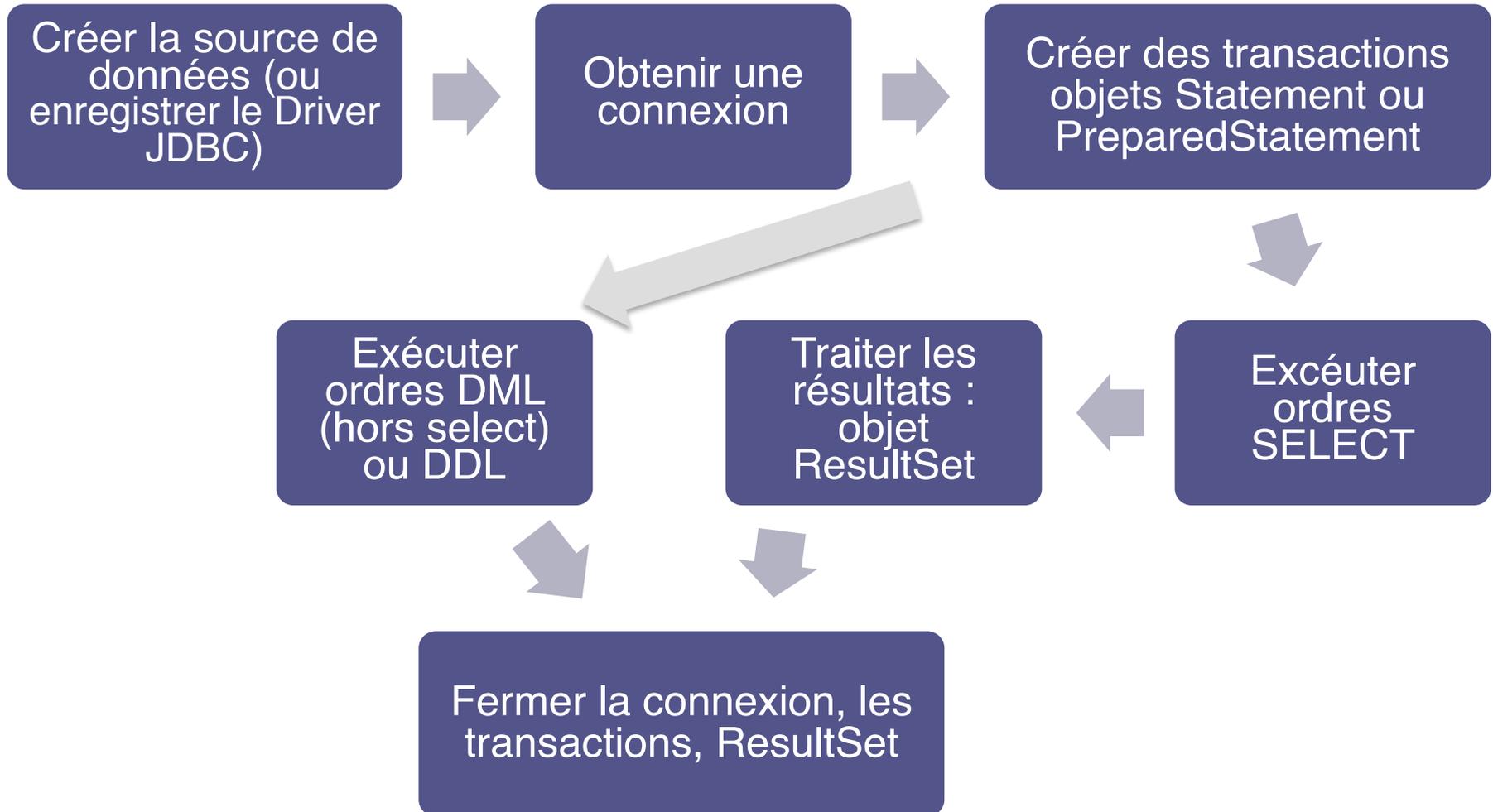
`RowSet`

<http://docs.oracle.com/javase/7/docs/api/javax/sql/package-summary.html>

Charger un pilote JDBC

- Chercher sur votre moteur de recherche :
JDBC + nom de votre BD
- Télécharger le **zip** ou **.jar**
 - Ex. mariadb-connector-java-*[version]*-bin.jar
- Sous IDE, ajouter **l'archive du pilote (.jar)** au projet
 - Sur le répertoire **Libraries**, **add .jar**
- Sans IDE, l'inclure dans le CLASSPATH de la machine

Processus JDBC



Différents types de connexion

- L'accès aux BD avec **l'API JDBC** est simple et rapide
 - Reste peu utilisé dans le milieu professionnel
 - Fournit les bases pédagogiques nécessaires en Bac+2
 - Avec `DriverManager` ou `DataSource` sans pool de connexions
- Utilisation d'un **pools de connexion** (à préférer)
 - Basé sur un `DataSource` gérant un vrai pool de connexions et des transactions distribuées
- **JNDI** pour des architectures avec un serveur d'application (ex. GlassFish) connecté à des serveurs de données
 - JNDI: **Java Naming and Directory Interface**
 - architecture J2EE, REST

(méthode ancienne bientôt obsolète : « Legacy code »)

CONNEXION AVEC DRIVER MANAGER

Processus de Connexion

```
import java.sql.DriverManager; // gestion des pilotes
import java.sql.Connection; // une connexion à la BD
import java.sql.Statement; // une instruction
import java.sql.ResultSet; // un résultat (lignes/colonnes)
import java.sql.SQLException; // les erreurs liées à la BD

public class JDBCSTable {
// 0- chargement du pilote (pour JDBC antérieur à JDBC4)
// 1- ouverture de connexion
// 2- exécution d'une requête
// 3- programme principal
}
```

Créer une connexion

- Méthode getConnection() de DriverManager :
- `Connection conn = DriverManager.getConnection(URL, userid, password);`
- Toute la difficulté réside dans la définition de la chaîne de connexion :

```
final String url =  
"oracle.jdbc.driver.OracleDriver";  
  
// "org.mariadb.jdbc.Driver" ; pour MariaDB  
// "sun.jdbc.odbc.JdbcOdbcDriver" ; pour ACCESS
```

À l'IUT

- URL pour MySQL sur le serveur IUT :

```
String url="jdbc:mysql://iutdoua-web.univ-lyon1.fr/pxxxxx";
```

```
String userid="pxxxxx";
```

```
String password="votrePwd";
```

- url pour Oracle sur le serveur IUT :

```
jdbc:oracle:thin:iutdoua-oracle.univ-lyon1.fr:1521:orcl
```

En local avec MySQL ou MariaDB

- **(MariaDB** est la BD issue de MySQL maintenue par le monde libre depuis qu'Oracle a racheté MySQL)
- ```
String url =
"jdbc:mariadb://localhost:3306/maBD";
Connection conn =
DriverManager.getConnection(url, "root",
"root");
```

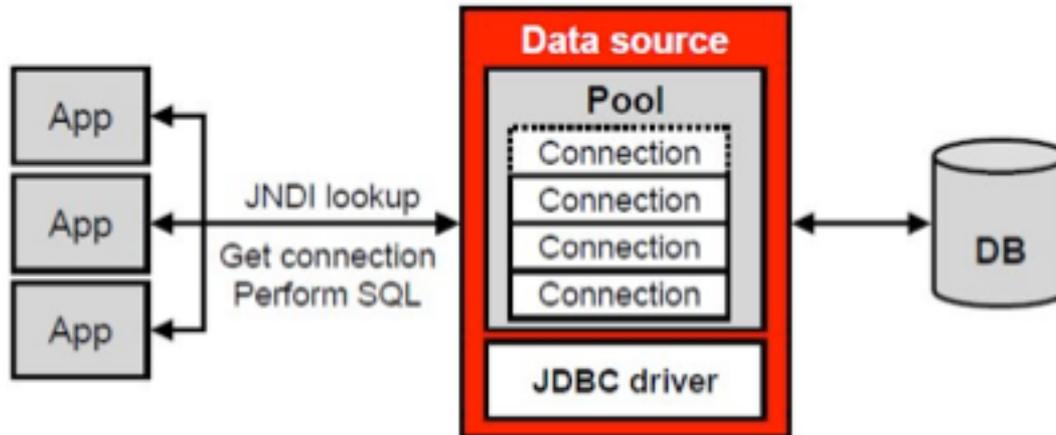
*Paramètres indiqués par votre envrt de serveur APACHE local*

## MySQL

MySQL peut être administrée via [phpMyAdmin](#).

Pour vous connecter au serveur MySQL dans vos propres scripts PHP, utilisez les paramètres suivants:

|              |                                         |
|--------------|-----------------------------------------|
| Hôte         | localhost                               |
| Port         | 3306                                    |
| Utilisateur  | root                                    |
| Mot de passe | root                                    |
| Socket       | /Applications/MAMP/tmp/mysql/mysql.sock |



Méthode plus robuste et plus riche

# CONNEXION AVEC UN DATASOURCE

# DataSource

- Un DataSource (package `javax.sql`) est une interface représentant une « source de données »
- Cette "source de données" est en fait une **fabrique de connexions** vers la source de données physique.
  - Ce mécanisme est apparu avec JDBC 3.0

# Avantages du Data Source

- Les drivers ne sont plus obligés de **s'enregistrer** eux-mêmes, en dur, comme avec DriverManager.
- Maintenance facilitée : on peut facilement **changer les propriétés** des sources de données sans modifier le code dans toutes les appli qui utilisent la BD
  - Par exemple, changement de serveur de base de données
- Les instances de Connection fournies par les DataSource ont des capacités étendues (**pool de connexions**, transactions distribuées, etc.) et on peut configurer au mieux le 'pool'
- Un pool de connexions permet :
  - $n$  connexions à la même BD,
  - 1 connexion à  $n$  différentes BD simultanément

# Fonctionnement d'un pool de connexions

- Le pool maintient un certain nombre **de connexions ouvertes** à disposition de l'application
- L'appel à `close()` ne ferme pas la connexion, mais **remet la connexion libérée dans le pool**, pour une utilisation ultérieure
- Il faut configurer correctement le pool de connexions utilisé, notamment **la taille du pool**

# Pool : interface DataSource

- Interface du package `javax.sql`

- Méthodes

- Connection **getConnection()** throws SQLException
    - Connection getConnection(String username, String password) throws SQLException

...

- Propriétés

- `databaseName` : String
    - `dataSourceName` : String
    - `networkProtocol` : String
    - `password` : String
    - `portNumber` : int
    - `serverName` : String
    - `User` : String

*Propriétés accessibles via les méthodes setXXX et getXXX*

# L'implémentation **OracleDataSource**

Classe du package `oracle.jdbc.pool`

- Méthodes :
  - Idem Interface DataSource
- Propriétés
  - Idem interface DataSource, plus :
    - `driverType` : String
    - `databaseName`: String
    - `url` : String

*Il y a d'autres propriétés moins importantes*

- **Gestion du pool :**

1. `OracleDataSource ods = ...` // définition
2. `Connection connBD = ods.getConnection();`
3. `connBD.insert(xxx);` // (actions sur la BD)
4. `connBD.close();`

# Ex. Création d'une source de données ORACLE

```
import oracle.jdbc.pool.OracleDataSource;
import java.io.FileInputStream;
import java.io.IOException;
import java.sql.SQLException;
import java.util.Properties;
```

Un SINGLETON

```
public class MonOracleDataSource extends OracleDataSource {
 // l'instance unique
 private static MonOracleDataSource ods;

 // Constructeur privé
 private MonOracleDataSource() throws SQLException {
 }

 public static MonOracleDataSource getOracleDataSourceDAO() {
 if (ods == null) { // on contrôle qu'il n'existe pas déjà
une source de données
 FileInputStream fichier = null;
 Properties props = new Properties();
```

## Ex. Création d'une source de données ORACLE

```
// On effectue une série de try / catch pour vérifier que :
// - Fichier de propriétés existe
// - Qu'on peut le charger, etc..

// alors on crée une instance vide de la classe (constr.
privé):
ods = new MonOracleDataSource();

// on la définit avec les paramètres fournis dans le fichier :
ods.setDriverType(props.getProperty("pilote"));
ods.setPortNumber(new Integer(props.getProperty("port")));
ods.setServerName(props.getProperty("serveur"));
ods.setServiceName(props.getProperty("service"));
ods.setUser(props.getProperty("user"));
ods.setPassword(props.getProperty("pwd"));

 } // sinon, un datasource existe déjà :
return ods;
} }
```

# Fichier Propriétés ORACLE IUT

- Le fichier de paramètres

- `port=1521`
- `service=orcl`
- `user=p1503944 // mettez le votre !`
- `pwd=YYYY`
- `serveur=iutdoua-oracle.univ-lyon1.fr`
- `pilote=thin`

*Le fichier **properties** est un fichier texte dont les lignes respectent un certain format (« clef = valeur »)*

- Le pilote JDBC Oracle

- `ojdbc8.jar`

**Attention** : sur vos PC perso en WIFI, utiliser EDUROAM  
Sinon connexion non autorisée

- Le référencement des bibliothèques

- `import java.sql.*; et import javax.sql.*;`
- `import oracle.jdbc.*;`

# Fichier de propriétés pour MariaDB IUT

port=3306

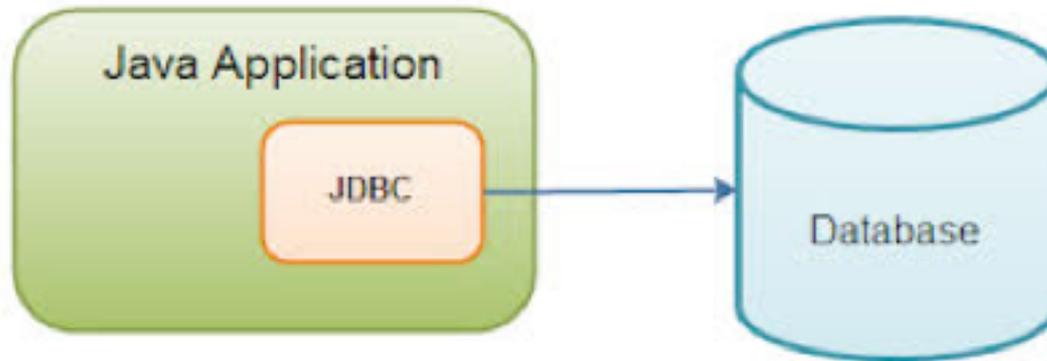
serveur=iutdoua-web.univ-lyon1.fr

pilote=thin

user=pxxxxx

pwd=xxx

base=pxxxxx

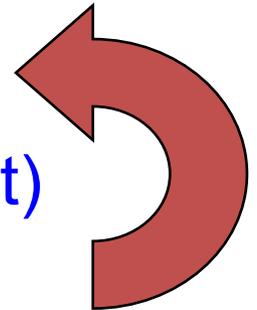


Langage de manipulation des données (LMD)

# ECRITURE DE REQUÊTES SQL

# Déroulement

- Ouvrir la connexion
- Créer un objet **Statement**
- Exécuter une requête sur ce **Statement**
- Si SELECT, traiter le résultat obtenu (**ResultSet**)
- Fermer le **Statement**
- Fermer la connexion



*Nota : un seul ResultSet par Statement*

# Les requêtes

- La connexion retournée par la méthode `getConnection()` est une connexion ouverte, ce qui permet de passer les instructions SQL vers le SGBD.
- Chaque requête s'exécute par un objet `Statement` et retourne un `ResultSet` :

```
Statement st =
conn.createStatement();
ResultSet rs =
st.executeQuery(maRequete); // (par ex.)
```

# Statement

3 méthodes selon le type de requête effectuée :

- Pour des **select**, retour d'un **ResultSet**:

```
ResultSet rs = ps.executeQuery()
```

On traite ensuite les lignes retournées, en séquence

- Pour des ordres DML (requêtes **insert**, **delete** ou **update**, ou dropTable ou ordre create ), retour d'un **int** :

```
int n = ps.executeUpdate() ;
```

Le nb de lignes traitées pour les ordres DML, 0 pour dropTable ou create

- Pour tout ordre SQL, retour d'un **booleén** :

```
boolean b = ps.execute(String req)
```

Renvoie vrai si le résultat est un ResultSet

## Ex. d'exécution 'Create table'

```
final String MA_REQUETE = "create
table Employes as select * from
scott.emp";
```

```
conn = newConnection();
```

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

```
int r = st.executeUpdate(MA_REQUETE);
```

# Ex. de requête 'Select'

- Exemple d'utilisation :

```
final String MA_REQUETE = "SELECT nom,prenom,age FROM personne
ORDER BY age";
```

```
public void listPersons() throws SQLException {
 Connection conn = null;
 try { // crée une connexion et un statement
 conn = newConnection();
 Statement st = conn.createStatement();
 ResultSet rs = st.executeQuery(MA_REQUETE);
 while (rs.next()) {
 System.out.printf("%-20s | %-20s | %3d\n", //
rs.getString(1), rs.getString("prenom"), rs.getInt(3));
 }
 }
 finally { // close result, statement and connection
 if (conn != null) conn.close(); ... } }
}
```

# L'interface `java.sql.ResultSet`

- Accès aux valeurs :

```
TYPE getType(int numeroDeColonne);
```

```
TYPE getType(String nomDeColonne);
```

```
boolean next();
```

- Le TYPE peut être

|            |           |               |
|------------|-----------|---------------|
| Byte       | Boolean   | AsciiStream   |
| Short      | String    | UnicodeStream |
| Int        | Bytes     | BinaryStream  |
| Long       | Date      | Object        |
| Float      | Time      |               |
| BigDecimal | TimeStamp |               |

# Correspondance des types Java SQL

| SQL                               | Java                 |
|-----------------------------------|----------------------|
| CHAR VARCHAR<br>LONGVARCHAR       | String               |
| NUMERIC DECIMAL                   | java.math.BigDecimal |
| BIT                               | boolean              |
| TINYINT                           | byte                 |
| SMALLINT                          | short                |
| INTEGER                           | int                  |
| BIGINT                            | long                 |
| REAL                              | float                |
| FLOAT DOUBLE                      | double               |
| BINARY VARBINARY<br>LONGVARBINARY | byte[]               |

# Correspondances date / heures

| SQL       | Java               | Obs.                            |
|-----------|--------------------|---------------------------------|
| DATE      | java.sql.Date      | codage de la date               |
| TIME      | java.sql.Time      | codage de l'heure               |
| TIMESTAMP | java.sql.TimeStamp | codage de la date et de l'heure |

*REGLE - tjrs donner le maximum de travail en SQL et le minimum en Java (cf procédures stockées)*

# ResultSet: récupération des données

- On ne peut pas récupérer toute une ligne de table en une fois
  - On procède colonne par colonne
- Méthodes **getXXX()** du **ResultSet**
  - `variable = getType (indice | "nom_colonne")`

```
cpCli = rs.getString("cpcli");
caCli = rs.getDouble("caclic");
noCat = rs.getInt(4);
```

Remarques :

- l'indice commence à **1**,
- Si le type de la colonne est différent, il faut transtyper.

# ResultSet: modification des données

- Méthodes setXXX() du ResultSet
  - *setType* (indice | "nom\_colonne", valeur)

```
updateVentes.setInt(1, 75);
updateVentes.setString(2, "Columbian");
```

Remarques :

- l'indice commence à **1**,
- Il fait référence au numéro de colonne du ResultSet (celui défini dans l'ordre SELECT) et **non au numéro de colonne** de la table.
- Si le type de la colonne est différent, il faut transtyper.

# ResultSet

- Le **ResultSet** est la table résultat issue de l'exécution d'une requête d'un **Statement**
- Un seul **ResultSet** par **Statement** peut être ouvert à la fois.
- On peut faire plusieurs requêtes sur un même **Statement**

## Parcours du ResultSet

- Méthode `next()` du `ResultSet`
  - La requête ne doit retourner qu'une ligne :  
`if (rs.next()) {`
  - La requête peut retourner plusieurs lignes :  
`while (rs.next()) {`

Remarque : à l'instanciation, le pointeur est placé **juste avant** la première ligne.

# ResultSet : valeur `null`

- En SQL, NULL signifie que le champ est vide
- Ça n'est pas pareil que `0` ou `""`
- En JDBC, on peut **explicitement** tester si le dernier champ lu est `null` avec la méthode :
  - `ResultSet.isNull(column)`
- Les méthodes `getXXX()` de `ResultSet` convertissent les valeurs **NULL SQL** en valeur *acceptable* par le type d'objet demandé :
  - `getString()`, `getObject()`, `getDate()` : retourne `null` java
  - `getByte()`, `getInt()`, ... : retourne `0`
  - `getBoolean()` : retourne `false`

# Valeur `null` (suite)

- Pour insérer des valeurs *null* dans un Prepared Statement :
  - Utiliser la méthode `setNull(index, Types.sqlType)` pour les types primitifs (ex.: `INTEGER`, `REAL`);
  - On peut aussi utiliser `setType(index, null)` pour les objets (ex.: `STRING`, `DATE`).

# Utilisation de SQL préformaté

**Obj.: éviter l'injection de code, plus fiables, ordres pré compilés**

- Interface : PreparedStatement
  - Méthode : conn.prepareStatement(req)
  - A utiliser pour des requêtes **qui sont exécutées plusieurs fois**
  - Compilés (*parsed*) par le SGBD une seule fois
  - Au lieu de valeurs, on utilise '?' (passage de code binaire, plus facile)
- ➔ Ce sont donc des *statements* avec variables, dont les **valeurs réelles sont données dans un 2e temps**

# Exécutions avec Select et Delete

Ex. de Select : on récupère un ou plusieurs ResultSet

```
PreparedStatement ps =
 conn.prepareStatement("select * from
 client where nocli = ?");
ps.setInt(1, numcli);
ResultSet rs = ps.executeQuery();
```

Ex. de Delete : on récupère un Entier

```
ps = conn.prepareStatement("delete from
 client where nocli = ?");
ps.setInt(1, numcli);
int nbLignes = ps.executeUpdate();
```

## Ex. de PreparedStatement Update utilisé dans une boucle

interface

```
PreparedStatement updateVentes;

String updateString = "update CAFE" + "set VENTES = ?
WHERE NOM_CAFE LIKE ?";

updateVentes = conn.prepareStatement(updateString); méthode

int[] VentesDeLaSemaine = {175 , 150, 60, 155, 90};

String[] cafes ={"Colombian", "French_Roast", "Espresso",
"Colombian_Decaf", "French_Roast_Decaf"};

for(int i = 0 ; i < cafes.length; i ++) {
 updateVentes.setInt(1, VentesDeLaSemaine[i]);
 updateVentes.setString(2, cafes[i]);
 updateVentes.executeUpdate();
}
```

# Meta informations du ResultSet

```
ResultSetMetaData m = rs.getMetaData();
```

Informations disponibles :

- nombre de colonnes : `int getColumnCount()`
- libellé d'une colonne,
- table d'origine,
- type associé à une colonne : `getColumnType(int col)`,
- la colonne peut-elle avoir une valeur null :
  - `int isNullable(int col)`
- etc. (**mais pas le nb de lignes !**)
  - *On utilisera `rs.last()` puis `rs.getRow()` pour l'avoir*

Cf aussi `java.sql.DatabaseMetaData`

# AutoCommit()

- Par défaut, les connexions sont en mode auto-commit
  - chaque ordre SQL sera exécuté et validé séparément
- Pour désactiver l'autocommit :  
`maConnexion.setAutoCommit(false)`
- Permet de regrouper plusieurs ordres SQL en une **transaction**
  - Obj. : éviter les conflits d'accès à la BD

## AutoCommit (suite)

- Avec `setAutoCommit(false)`, l'objet `Statement` prend fin en appelant :
  - La méthode `commit` de l'interface `Connection` (pour valider les changements apportés à la base de données)
  - La méthode `rollback` (pour remettre la base de données dans le même état qu'avant la transaction)
- Attention : la fermeture d'une connexion valide la transaction même si `autoCommit` est à ***false***

# Modes de communication avec la BD

- Mode synchrone
  - Chaque action effectuée sur les données de l'application (ajout, modification, suppression) est propagée directement à la BD
  - **C'est le mode privilégié**
  - Avantage: moins de risque de conflit d'accès
  - Restriction : peu de requêtes pendant l'exécution, sinon utiliser du multithread
- Mode asynchrone
  - On travaille en local avec un conteneur de données, chargées au lancement de l'application, et on sauvegarde quand on quitte l'application
  - Avantage : moins d'accès réseau
  - On peut coder soi-même ou utiliser l'interface **CachedRowSet** et sa méthode **acceptChanges ( )**

<https://docs.oracle.com/javase/7/docs/api/javax/sql/rowset/CachedRowSet.html>

# Pool de connexions : un mixte

- Le coût de connexion à une BD étant élevé, on a un moyen simple de procéder qui permet d'avoir les bénéfices des modes synchrone / asynchrone
- Un pool de connexions dispose d'un ensemble de connexions, qui sont attribuées aux méthodes de l'application selon leur disponibilité
  - Quand on 'ferme' une connexion, on la rend disponible pour le pool, elle n'est pas 'fermée'

# Appeler une procédure stockée

- La méthode `prepareCall(String proc)`
  - Méthode de la classe `Connection`,
  - Paramètre sous la forme :
    - `"{? = call nom_fonction([?,?,...])}"`
    - `"{call nom_procédure([?,?,...])}«`
- La classe `CallableStatement` est dédiée à l'appel des procédures stockées

Comme pour la méthode `prepareStatement`, les paramètres sont définis par des ?.

```
CallableStatement cs = connection.prepareCall("{? = call inc_parametre(?) }");
```

# Paramètres en entrée d'un CallableStatement

- Méthodes `setXXXX(indice, valeur)`

Le passage de paramètre à un CallableStatement est identique à `prepareStatement()`.

```
CallableStatement cs =
 connection.prepareCall("{? = call
 inc_parametre(?) }");
cs.setString(2, nopar);
```

# Paramètres en sortie d'un CallableStatement

- Méthode `registerOutParameter(indice, type)`
  - Indice : position du paramètre,
  - Type : entier (contante ou constante nommée) identifiant le type.

```
CallableStatement cs = connection.prepareCall("{?
 = call inc_parametre(?) }");
cs.registerOutParameter(1, Types.INTEGER);
```

Les constantes nommées :

```
Types.VARCHAR, Types.DATE, Types.REAL ...
```

## Récupération d'un paramètre en sortie d'un CallableStatement

- Méthode `getXXXX ( )`
  - `variable = getType (indice | "nom_colonne")`

```
int nb = cs.getInt(1)
```

# Exécuter un CallableStatement

- Méthode `execute ( )`
  - Retourne un booléen :
    - `true` : l'exécution a produit un ResultSet
    - `false` : pas de retour, ou mise à jour.

```
cs.execute () ;
```