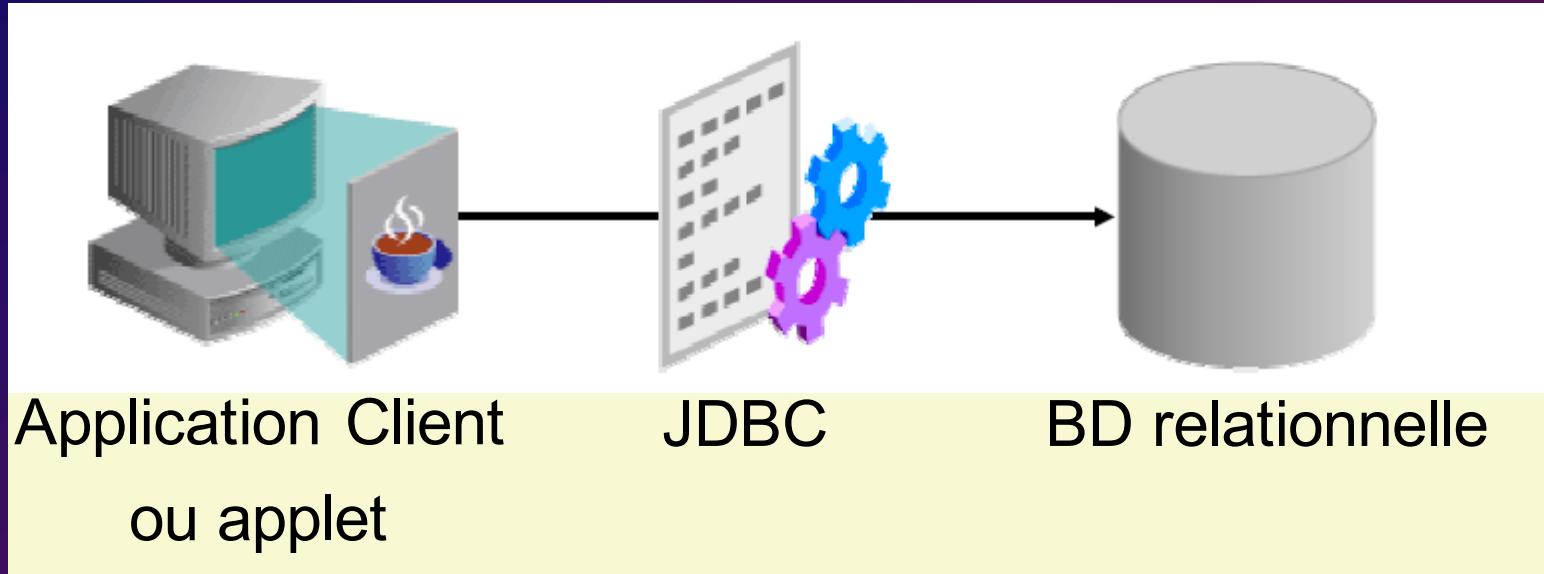


6

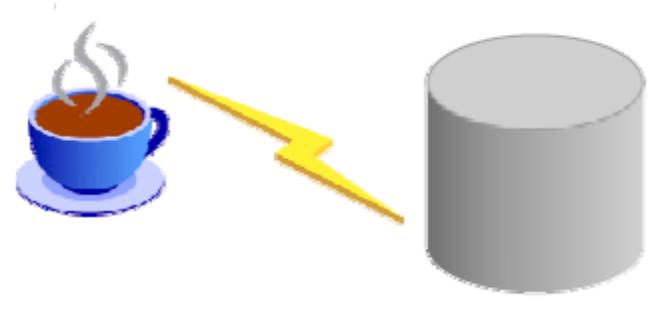
L'ACCÈS AUX BASES DE DONNÉES PAR JDBC



l'API JDBC (Java DataBase Connectivity)



permet d'exécuter des ordres SQL dans un programme écrit en Java



JDBC

- **Interface standard pour se connecter à des bases de données à partir de Java**
Le package java.sql contient un ensemble d'interfaces et quelques classes qui spécifient l'API JDBC
- **Les interfaces du package java.sql sont implémentées par un driver JDBC fourni par les éditeurs de SGBD ou autres vendeurs**
Le fichier d'archive du driver JDBC / source de données doit être inclus dans la variable CLASSPATH

Interfaces de l'API JDBC

JDBC permet :

- de se connecter à un ou plusieurs serveurs de données
- d'exécuter des ordres SQL
- d'obtenir des ensembles de résultats
- d'obtenir des métadonnées

Interfaces :

Package `java.sql` : `Driver`, `Connection`, `Statement`,
`PreparedStatement`, `CallableStatement`, `ResultSet`,
`DatabaseMetaData`, `ResultSetMetaData`, ...

Package `javax.sql` : `DataSource`, ...

Utilisation de JDBC pour exécuter des ordres SQL

1. Créer la source de données Oracle (ou enregistrer le driver JDBC) et Obtenir une connexion

2. Crée des objets *Statement* | *PreparedStatement*

3a. Exécuter des ordres SELECT

3b. Exécuter des ordres DML(hors SELECT) ou DDL

4. Traiter les résultats des requêtes

5. Fermer la connexion et les objets (*Statement*, *PreparedStatement*)

Gestion des exceptions

- Toutes les méthodes qui accèdent à la base de données lancent une exception `SQLException` en cas de problèmes
 - Ces exceptions doivent être traitées dans le code JDBC (**libérer les ressources si nécessaire** – dans le bloc `finally` par exemple)
- Quelques méthodes de la classe `java.sql.SQLException` :
 - `getMessage()` : renvoie un `String` décrivant l'erreur
 - `getErrorCode()` : renvoie un code d'exception fabriquant
 - `getSQLState()` : renvoie la valeur de l'état d'exécution de SQL
 - ...

Utilisation de JDBC pour exécuter des ordres SQL

1. Créer la source de données Oracle (ou enregistrer le driver JDBC)
et Obtenir une connexion

2. Crée des objets `Statement` | `PreparedStatement`

3a. Exécuter des ordres SELECT

3b. Exécuter des ordres DML(hors
SELECT) ou DDL

4. Traiter les résultats des requêtes

5. Fermer la connexion et les
objets (`Statement`, `PreparedStatement`)

Créer la source de données Oracle et obtenir une connexion

L'accès à une base de données via une **DataSource** est un mécanisme (depuis JDBC 3.0) désormais préféré au **DriverManager**

DataSource (package `javax.sql`) : interface représentant une "source de données"

Cette "source de données" est en fait une simple fabrique de connexions vers la source de données physique

DataSource

Avantages :

- Les drivers ne sont plus obligés de s'enregistrer (comme ils le faisaient avec DriverManager)
- Les objets DataSource possèdent des propriétés qui peuvent être modifiées (par exemple la source est déplacée sur un autre serveur)
 - le code d'accès à cette source n'a pas à être modifié
- Les instances de Connection fournies par les DataSource ont des capacités étendues (pool de connexion, transactions distribuées, etc.)
- ...

Implémentation de l'interface DataSource : Classe OracleDataSource (package oracle.jdbc.pool)

```
oracle.jdbc.pool
Class OracleDataSource
java.lang.Object
└ oracle.jdbc.pool.OracleDataSource
```

La classe OracleDataSource gère un pool de connexions : mécanisme permettant de réutiliser les connexions créées

Un pool de connexions ne ferme pas la connexion lors de l'appel à la méthode close () → celle-ci est "retournée" au pool et peut être utilisée ultérieurement

Gestion du pool :

```
OracleDataSource source = ...; //récupération d'une DataSource  
//récupération d'une connexion du pool  
Connection connexionBD = source.getConnection();  
//utilisation de la connexion ...  
connexionBD.close(); // connexion retornée au pool
```

Classe OracleDataSource : extrait de l'API

Constructor Summary	
<u>oracleDataSource ()</u>	
Method Summary	
<code>void</code>	<u>close()</u> Close DataSource API.
<code>java.sql.Connection</code>	<u>getConnection()</u> Attempt to establish a database connection.

`void setDriverType(java.lang.String dt)`
Set the JDBC driver type.

`void setPortNumber(int pn)`
Set the port number where a server is listening for requests.

`void setServiceName(java.lang.String svcname)`
Set the service_name of a database on a server.

`void setUser(java.lang.String user)`
Set the user name with which connections have to be obtained.

`void setServerName(java.lang.String sn)`
Set the name of the Server on which database is running.

`void setPassword(java.lang.String pd)`
Set the password with which connections have to be obtained.

Utilisation d'un fichier de configuration : classe Properties (java.util.Properties)

Représente un ensemble de propriétés

Classe Properties : extrait de l'API

Constructor Summary

Properties()

Creates an empty property list with no default values.

Method Summary

<code>void</code>	<u>load(InputStream inStream)</u>
-------------------	---

Reads a property list (key and element pairs) from the input byte stream.

<code>String</code>	<u>getProperty(String key)</u>
---------------------	--

Searches for the property with the specified key in this property list.

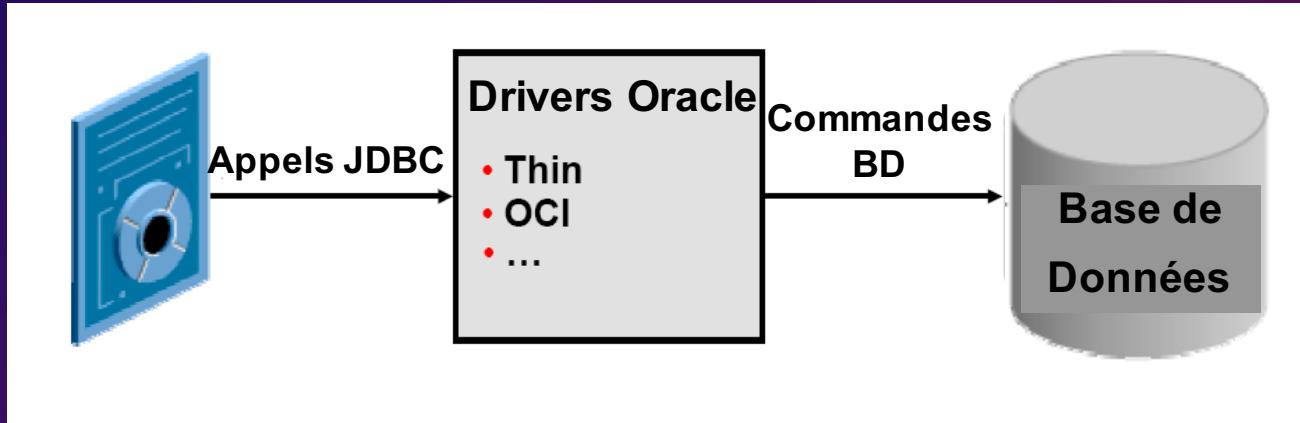
Exemple avec utilisation des patterns Factory et Singleton

Fichier "connexion.properties"

```
port=1521  
service=orcl.univ-lyon1.fr  
user=p11025478  
pwd=monmdp  
serveur=iutdoua-oracle.univ-lyon1.fr  
pilote=thin
```

Package oracle.jdbc.driver

→ Oracle fournit les drivers Thin, OCI,...



- Driver Thin : 100% Java, utilise le protocole TCP/IP, connexion à un serveur Oracle à partir d'une application ou d'une applet
- Driver OCI : écrit en C et Java, utilise le protocole Oracle Net, doit être installé sur le client, ne convient pas aux applets

UTILISATION DU DESIGN PATTERN [persistance] Data Access Object

Problématique :

Isoler la couche d'accès aux données de la couche métier d'une application pour :

- faciliter le changement de sources de données
- avoir une meilleure maîtrise des changements éventuels apportés à la couche métier
- faciliter la migration d'un SGBD vers un autre ou la mise à jour des nouvelles implémentations des fournisseurs de SGBD
- masquer les connexions aux différentes sources de données
- factoriser le code d'accès aux sources de données et donc faciliter pour le spécialiste des BD l'optimisation des accès

Sans doute le modèle de conception le plus utilisé dans le monde de la persistance

Solution :

Tout le code lié à la persistance est isolé dans des objets spécifiques : les objets DAO

Quand l'application a besoin d'effectuer une opération liée à la persistance, elle fait appel à un objet DAO

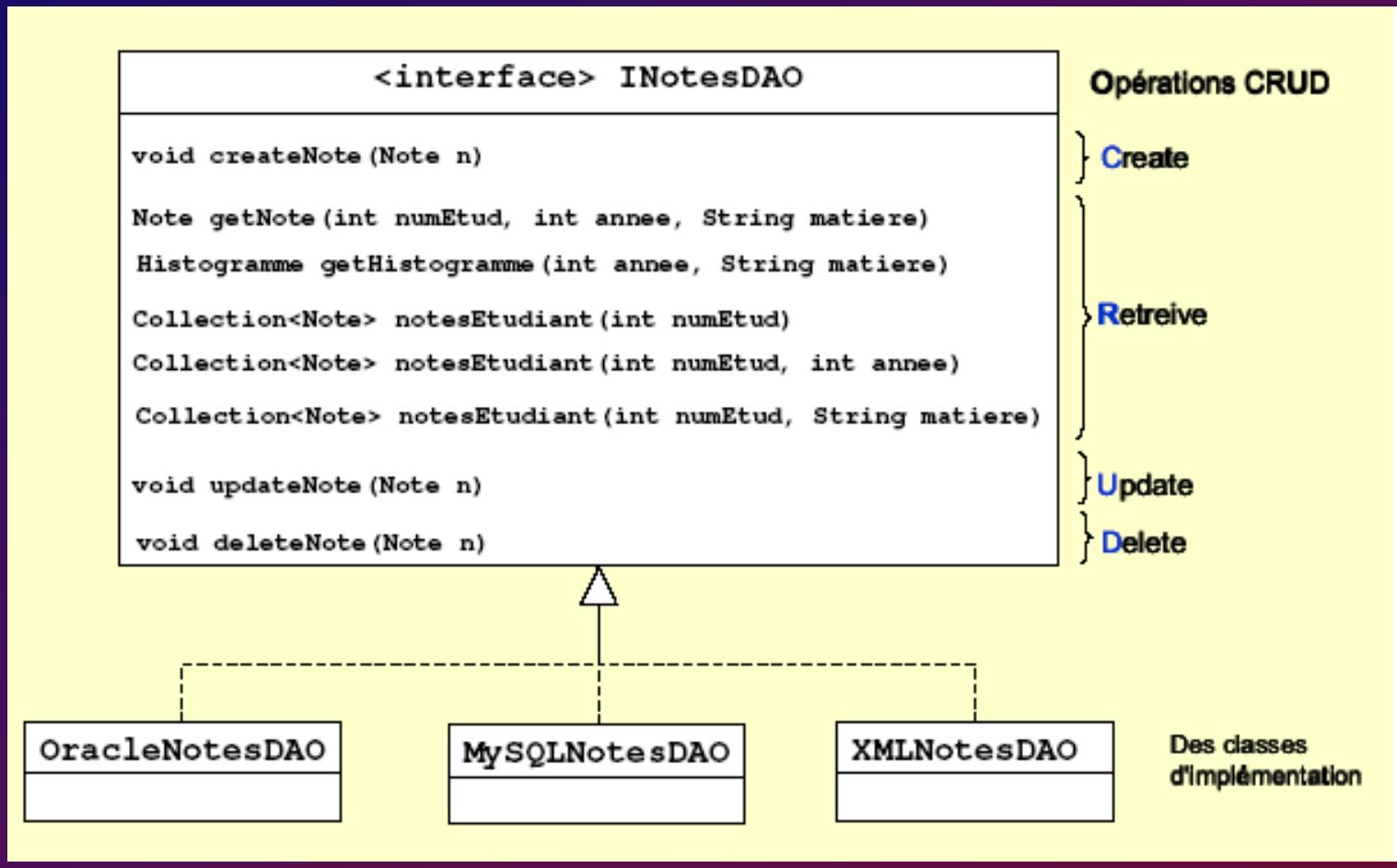
L'interface des objets DAO est indépendante du support de la persistance

Chaque classe d'objet métier a son propre type de DAO (IClientDAO, ICompteDAO, ...)

Le même objet DAO peut être utilisé pour tous les objets d'une classe d'objet métier

Les DAOs sont placés dans la couche dite "d'accès aux données"

Exemple



Utilisation de JDBC pour exécuter des ordres SQL

1. Créer la source de données Oracle (ou enregistrer le driver JDBC) et Obtenir une connexion

2. Crée des objets **Statement** | **PreparedStatement**

3a. Exécuter des ordres SELECT

3b. Exécuter des ordres DML(hors SELECT) ou DDL

4. Traiter les résultats des requêtes

5. Fermer la connexion et les objets (Statement, PreparedStatement)

ÉTAPE 2 : Créer un objet Statement

- Les objets Statement sont créés à partir de l'instance de connexion
- La méthode `java.sql.Connection.createStatement` fournit un contexte pour l'exécution d'un ordre SQL
 - Méthode `public Statement createStatement()`

```
Statement nomOrdre = nomConnexion.createStatement();
```

EXEMPLE :

```
private Connection connexionBD;  
...  
Statement stmt = connexionBD.createStatement();
```

UTILISATION DE L'INTERFACE Statement

L'interface Statement fournit 3 méthodes pour exécuter des ordres SQL :

- **executeQuery (String sql) pour des ordres SELECT**
 - Renvoie un objet ResultSet pour traiter les lignes
- **executeUpdate (String sql) pour des ordres DML (hors select) ou DDL**
 - Renvoie un entier
- **execute (String) pour n'importe quel ordre SQL**
 - Renvoie une valeur booléenne

Utilisation de JDBC pour exécuter des ordres SQL

1. Créer la source de données Oracle (ou enregistrer le driver JDBC) et Obtenir une connexion

2. Crée des objets *Statement* | *PreparedStatement*

3a. Exécuter des ordres SELECT

3b. Exécuter des ordres DML(hors SELECT) ou DDL

4. Traiter les résultats des requêtes

5. Fermer la connexion et les objets (*Statement*, *PreparedStatement*)

ÉTAPE 3a : Exécuter un ordre SQL

SELECT

Argument de la méthode `executeQuery()`: ordre SELECT sous forme de chaîne de caractères sans ";"

- Renvoie un objet `ResultSet`
- Création d'un objet la classe `ResultSet` qui reçoit le résultat de l'exécution de la requête

Méthode (`java.sql.Statement`) :

`public ResultSet executeQuery(String sql)`

```
ResultSet nomResultSet = nomOrdre.executeQuery("requête SQL  
SELECT ...");
```

EXEMPLE :

```
Statement stmt = connexionBD.createStatement();  
ResultSet rset = stmt.executeQuery ("select ename  
from scott.emp");
```

L'objet ResultSet

- Le driver JDBC renvoie les résultats d'un ordre SQL dans un objet ResultSet
- ResultSet
 - ✓ maintient un curseur pointant sur la ligne courante des données résultats

L'interface ResultSet fournit des méthodes pour obtenir les valeurs des colonnes

Utilisation de JDBC pour exécuter des ordres SQL

1. Créer la source de données Oracle (ou enregistrer le driver JDBC) et Obtenir une connexion

2. Crée des objets *Statement* | *PreparedStatement*

3a. Exécuter des ordres SELECT

3b. Exécuter des ordres DML(hors SELECT) ou DDL

4. Traiter les résultats des requêtes

5. Fermer la connexion et les objets (*Statement*, *PreparedStatement*)

ÉTAPE 3b : Soumettre un ordre DML (hors select) ou DDL

Rappel étape 3 : créer un objet Statement

```
Statement nomOrdre = nomConnexion.createStatement();
```

- Utiliser `executeUpdate` pour exécuter l'ordre

```
public int executeUpdate(String sql)  
int count =  
    nomOrdre.executeUpdate("OrdreDML/DDL");
```

Valeur renvoyée :

- ✓ 0 pour un ordre DDL create (ou nombre de tuples créés)
- ✓ 0 pour drop table
- ✓ le nombre de lignes traitées pour un ordre DML (insert, update, delete)

EXEMPLE :

```
Statement stmt= connexionBD.createStatement();
int r =stmt.executeUpdate("create table emp as select
* from scott.emp");
```

Utilisation de JDBC pour exécuter des ordres SQL

1. Créer la source de données Oracle (ou enregistrer le driver JDBC) et Obtenir une connexion

2. Crée des objets *Statement* | *PreparedStatement*

3a. Exécuter des ordres SELECT

3b. Exécuter des ordres DML(hors SELECT) ou DDL

4. Traiter les résultats des requêtes

5. Fermer la connexion et les objets (*Statement*, *PreparedStatement*)

ÉTAPE 4 : Traiter les résultats

Utiliser les méthodes de l'interface `ResultSet`:

- La méthode `public boolean next()` dans une boucle pour exploiter les différentes lignes résultats
- Les méthodes `getXXX()` pour obtenir les valeurs des colonnes avec `XXX` type de données Java (`public String getString(String nomCol)` , `public float getFloat(int colIndex)` ,`public Object getObject(int colIndex)` ,...)

EXEMPLE :

```
while (nomResultSet.next()) { ...  
  
String nom = nomResultSet.getString("ename");  
int numero = nomResultSet.getInt(1);  
...}
```

Utilisation de JDBC pour exécuter des ordres SQL

1. Créer la source de données Oracle (ou enregistrer le driver JDBC) et Obtenir une connexion

2. Crée des objets *Statement* | *PreparedStatement*

3a. Exécuter des ordres SELECT

3b. Exécuter des ordres DML(hors SELECT) ou DDL

4. Traiter les résultats des requêtes

5. Fermer la connexion et les objets (Statement, PreparedStatement)

ÉTAPE 5 : Fermer la connexion et les objets Statement et PreparedStatement

Fermeture explicite des objets Connection, Statement et ResultSet pour libérer les ressources

Appel des méthodes **close()** respectives :

```
Connection connexionBD = ...;  
Statement stmt = ...;  
ResultSet rset = stmt.executeQuery...;  
...  
// Fermeture  
if (stmt != null) stmt.close();  
if (rset != null) rset.close();  
if (connexionBD != null) connexionBD.close();
```

Cas des valeurs nulles

- Utilisation de la méthode `wasNull()` de `ResultSet`
 - Renvoie `true` si l'on vient de lire un `NULL`, `false` sinon
- Les méthodes `getXXX()` de `ResultSet` convertissent une valeur `NULL` SQL en une valeur acceptable par le type d'objet demandé
 - `getString()`, `getObject()`, `getDate()` : "null" java
 - `getByte()`, `getInt()`, ... : "0"
 - `getBoolean()` : "false"

Soumettre un ordre SQL inconnu

1. Créer un objet Statement

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

2. Utiliser **execute** pour exécuter l'ordre

```
boolean isQuery = stmt.execute(String sql)
```

Vrai si le résultat est un objet ResultSet

3. Traiter l'ordre SQL en conséquence (méthodes de l'interface Statement) :

```
if (isQuery) { // ordre SELECT - traiter les résultats  
    ResultSet r = stmt.getResultSet(); ...}  
else { // ordre DML (!=select) ou DDL - traiter le résultat  
    int count = stmt.getUpdateCount(); ...}
```

Gestion des transactions

- Par défaut les connexions sont en mode autocommit
- Pour désactiver l'autocommit :

```
nomConnexion.setAutoCommit(false)
```

- Pour contrôler les transactions sans le mode autocommit :

```
nomConnexion.commit()
```

```
nomConnexion.rollback()
```

- La fermeture d'une connexion valide la transaction même si l'option autocommit est à off

Accès aux méta données :

`java.sql.ResultSetMetaData`

- Informations sur le `ResultSet`
- La méthode `getMetaData()` sur un objet `ResultSet` renvoie un objet `ResultSetMetaData`
- Informations :
 - Nombre de colonnes : `int getColumnCount()`
 - Nom d'une colonne : `String getColumnName(int col)`
 - Type d'une colonne : `int getColumnType(int col)`
 - Nom de la table : `String getTableName(int col)`
 - Si un NULL SQL peut être stocké dans une colonne : `int isNullable(int col)`
 - ...

EXEMPLE :

```
...  
  
ResultSet rset = stmt.executeQuery("SELECT * FROM emp");  
ResultSetMetaData rsetmd = rset.getMetaData();  
int nbCol = rsetmd.getColumnCount();  
for (int i = 1; i <= nbCol; i++) {  
    // numérotation des col. à partir de 1  
    String typeCol = rsetmd.getColumnType(i);  
    ...  
}
```

Accès aux métadonnées :

`java.sql.DatabaseMetaData`

- Informations sur la base de données.
- La méthode `getMetaData()` sur un objet `Connection` renvoie un objet `DatabaseMetaData`.
- Informations :
 - Version du produit : `String getDatabaseProductVersion()`
 - Nom de l'utilisateur : `String getUsername()`
 - Nom du driver JDBC : `String getDriverName()`
 - URL de la base : `String getURL()`
 - ...

L'objet PreparedStatement ordre SQL pré-compilé

- L'ordre SQL d'un **PreparedStatement** est analysé une seule fois même s'il est exécuté plusieurs fois
 - Plus efficace qu'un **Statement**
- Il contient des variables dont les valeurs seront fournies à chacune de ses exécutions (ordre dynamique)
- Utilisé également pour des ordres SQL exécutés plus d'une fois

CRÉATION D'UN PreparedStatement

- Utilisation de la méthode (`java.sql.PreparedStatement`)

```
public PreparedStatement  
    prepareStatement(String sql)
```

et

- Du caractère "?" pour représenter une valeur d'une variable

```
PreparedStatement nomOrdre =  
    nomConnexion.prepareStatement("ordre SQL contenant des ?");
```

EXÉCUTION D'UN PreparedStatement

1. Positionnement des variables :

Méthodes de PreparedStatement: setXXX(), avec
xxx type java de la variable (public void setInt(int index, int x), public void setString(int index, String x), public setObject(int index, Object x),...)

```
nomOrdre.setXXX(rang, valeur);
```

(rang : position du paramètre dans l'ordre SQL)

2. Exécution de l'ordre

```
nomOrdre.executeQuery(); //sans argument  
nomOrdre.executeUpdate();  
nomOrdre.execute();
```

