



Architecture des applications Client-Serveur en Java et J2EE

DUT Informatique
Semestre 4

Mourad Ouziri
Mourad.Ouziri@parisdescartes.fr

Plan du cours

Plan

-  Architecture 2-tiers (client lourd) : Accès aux bases de données via *JDBC*
-  Architecture 3-tiers (web) : Servlet, JSP, Architecture MVC

Intervenants :

Jean-François Brette et Mourad Ouziri

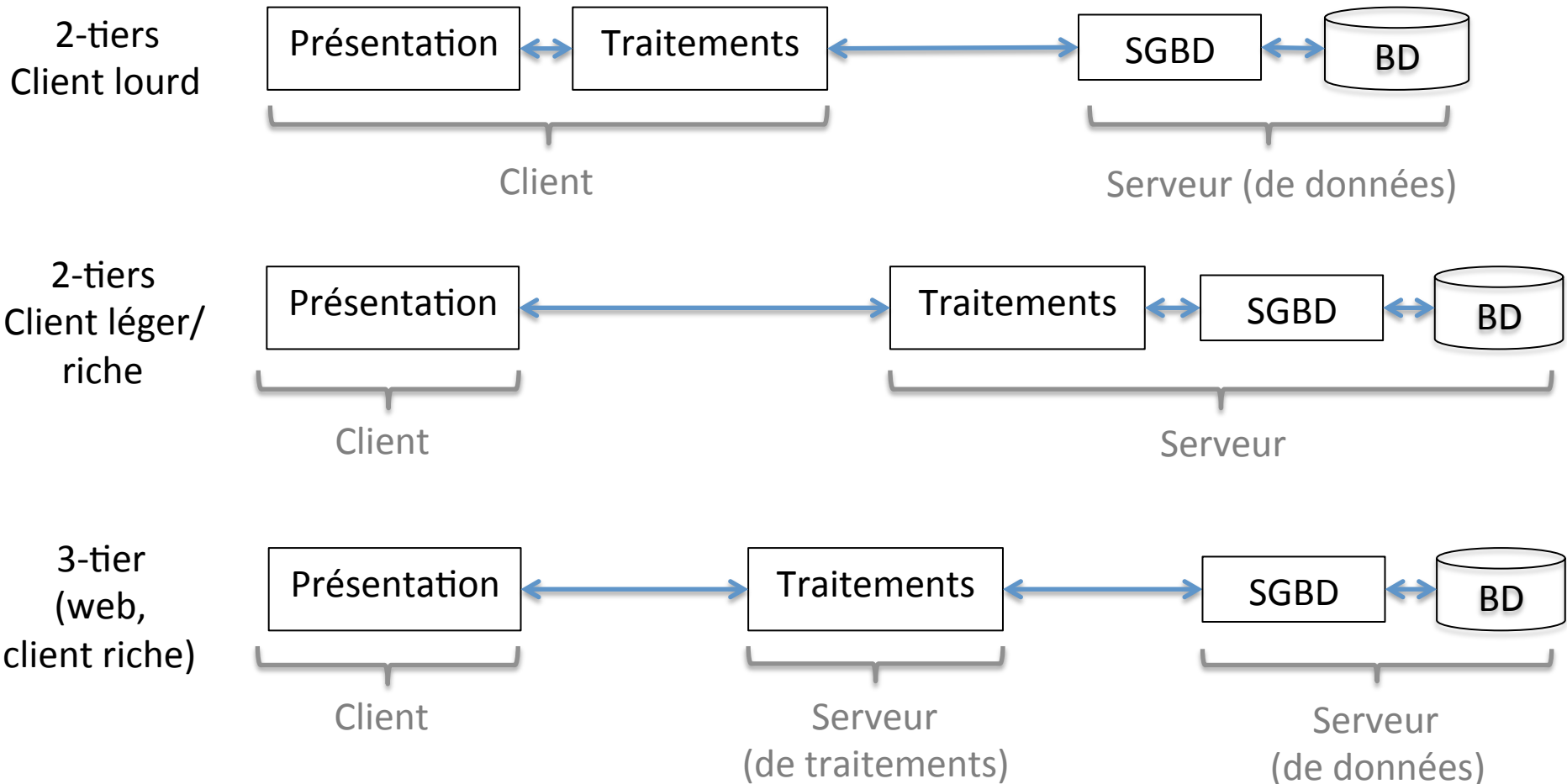
Evaluation :

Sur projet seulement qui se déroulera du 27/02 au 13/03 !

Introduction :

Architectures d'accès à la base de données

- Architecture : composants, rôles, interactions



Architecture Client-lourd/Serveur :

Accès au serveur de données via JDBC

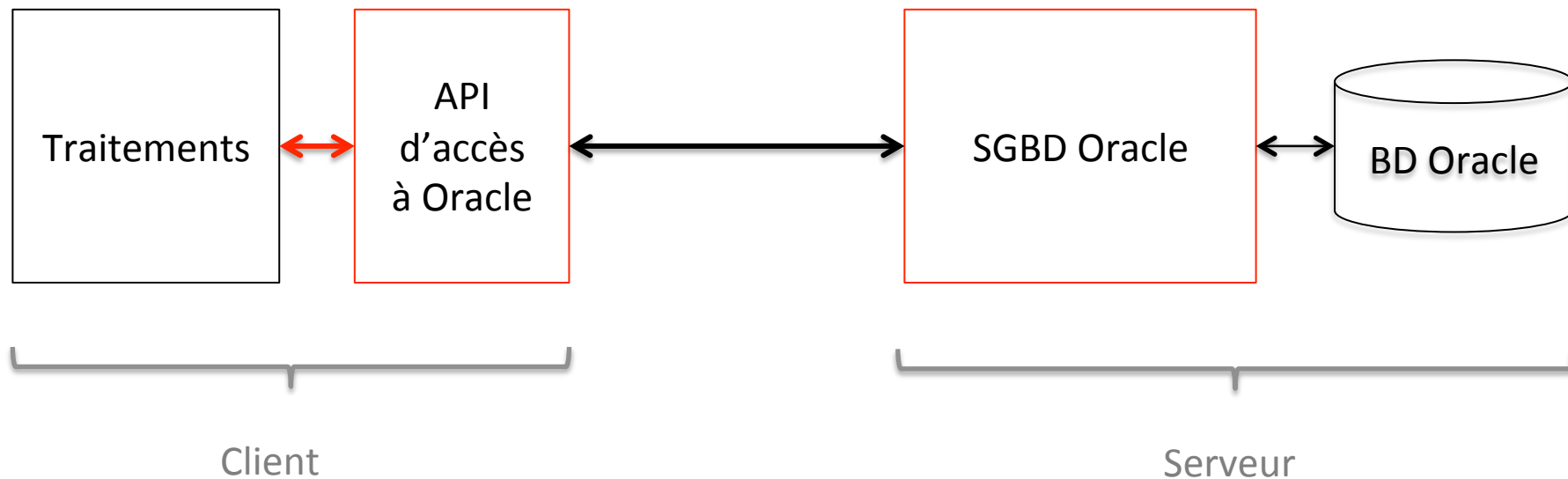
(Java DataBase Connectivity)

Introduction

- Problématique d'accès aux bases de données :
 - Multitude de bases de données : fournisseurs (Oracle, MySQL, SQLServer, etc.), protocole d'accès (local, distant, directives), formats des messages, langages d'implémentation, etc.
 - Maintenabilité : changement de bases de données

Introduction

- Dépendance du SGBD utilisé : problème de maintenabilité !

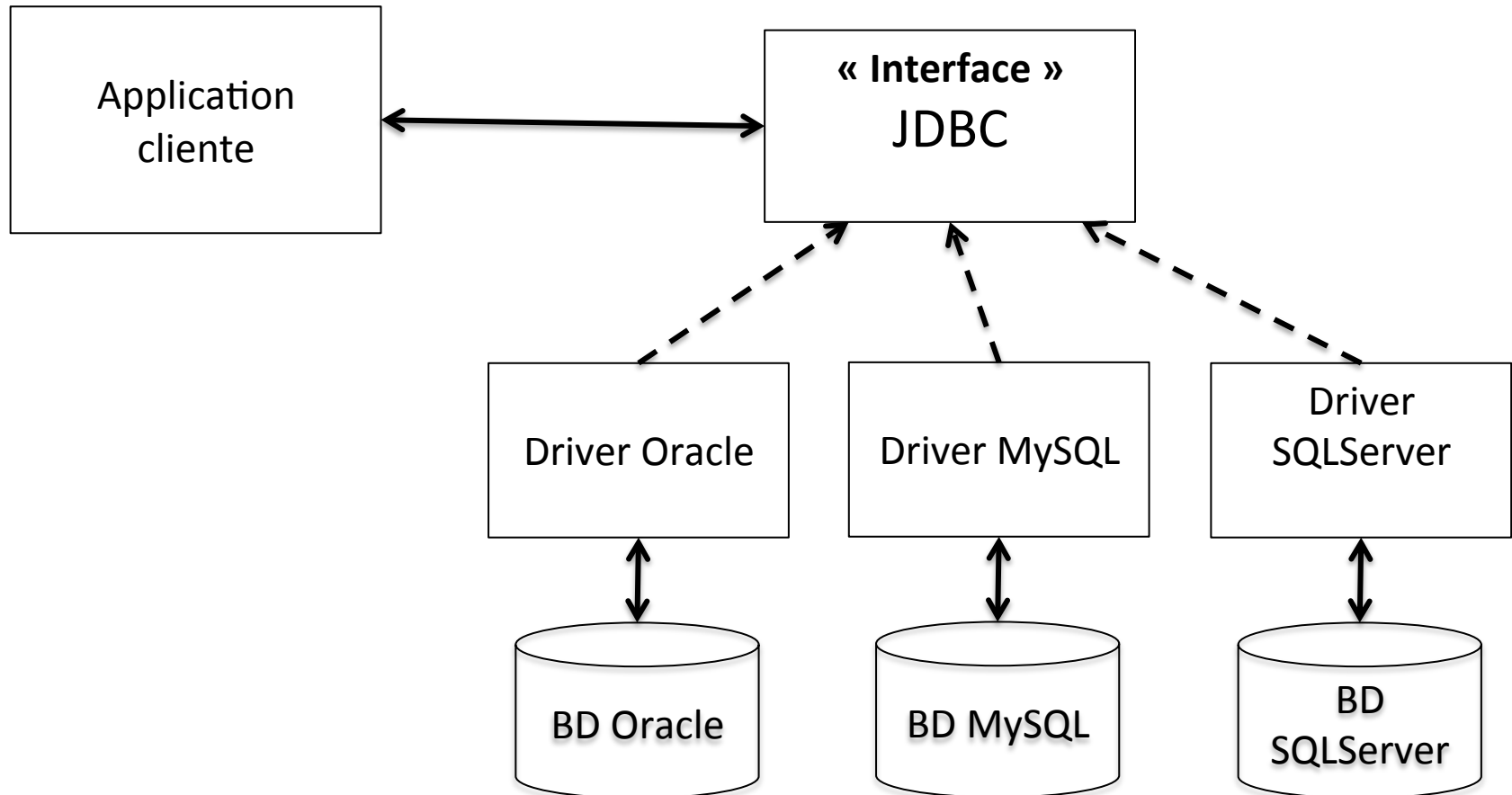


JDBC

- JDBC : spécification pour l'accès aux bases de données à partir de programmes Java
- Objectif : écrire des applications Java **indépendantes** de la base de données (fournisseur, protocole, formats de données, *etc.*)
- JDBC : définit un ensemble d'interfaces faisant abstraction des SGBD des fournisseurs

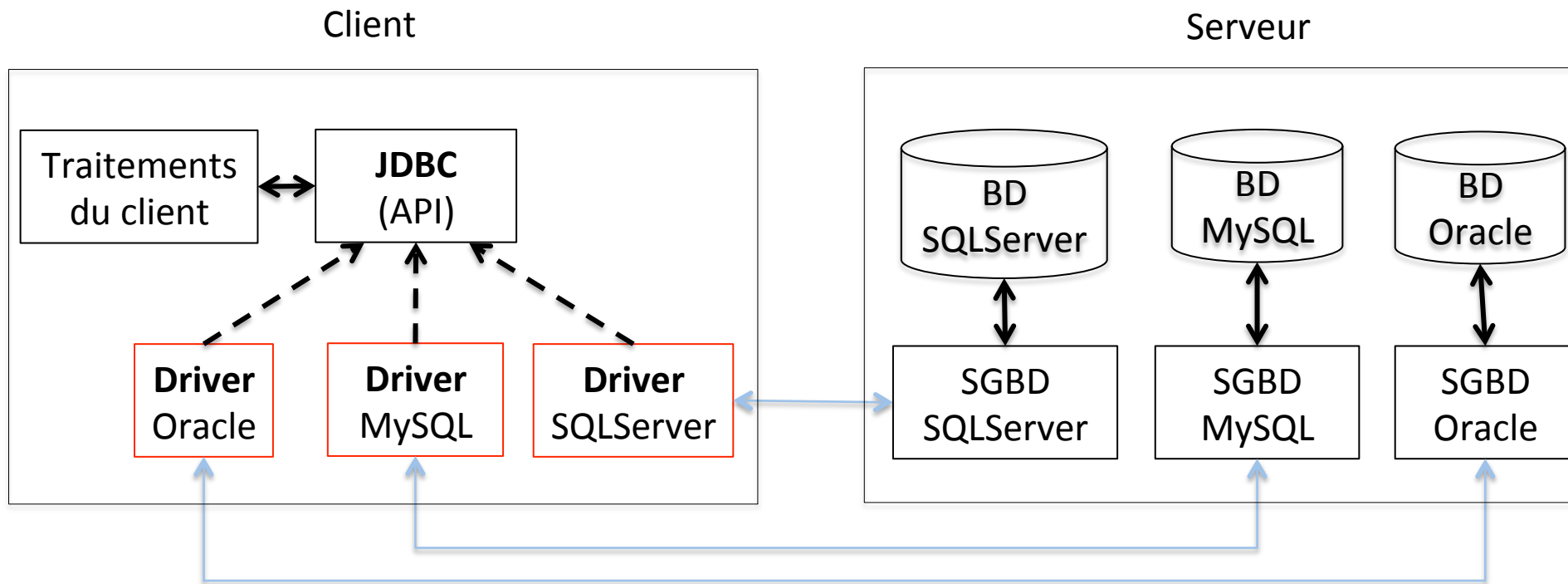
Introduction

- Chaque fournisseur de BD fournit son implémentation des interfaces : c'est le « *pilote* » (ou « *driver* »)



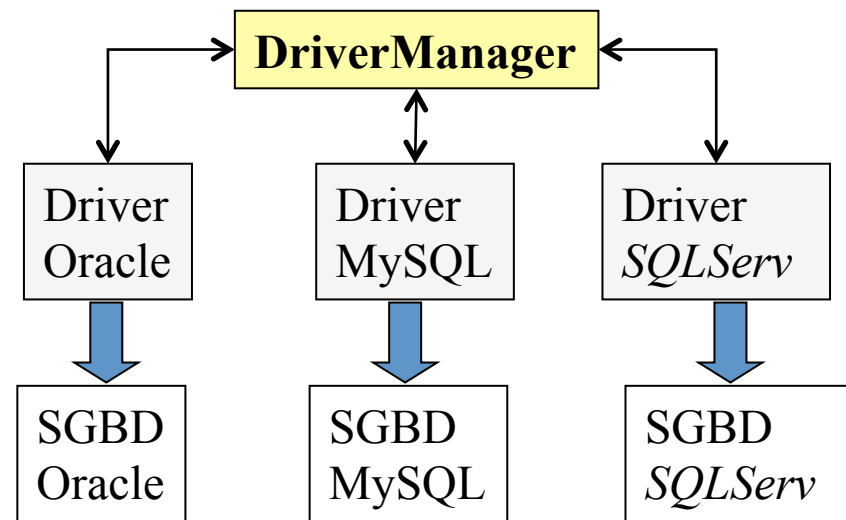
Introduction

- Solution : architecture en couplage faible, JDBC



Interfaces JDBC

- Huit interfaces JDBC principales :
 - **Driver** : établir des connexions à la base de données
 - **Connection** : représente une connexion à la base de données
 - **Statement, PreparedStatement, CallableStatement** : exécution de requêtes
 - **ResultSet** : structurer et consulter le résultat d'une requête
 - **DatabaseMetaData, ResultSetMetaData** : accéder au schéma de la base de données
- Une classe JDBC :
 - **DriverManager** : gestionnaire des pilotes de bases de données

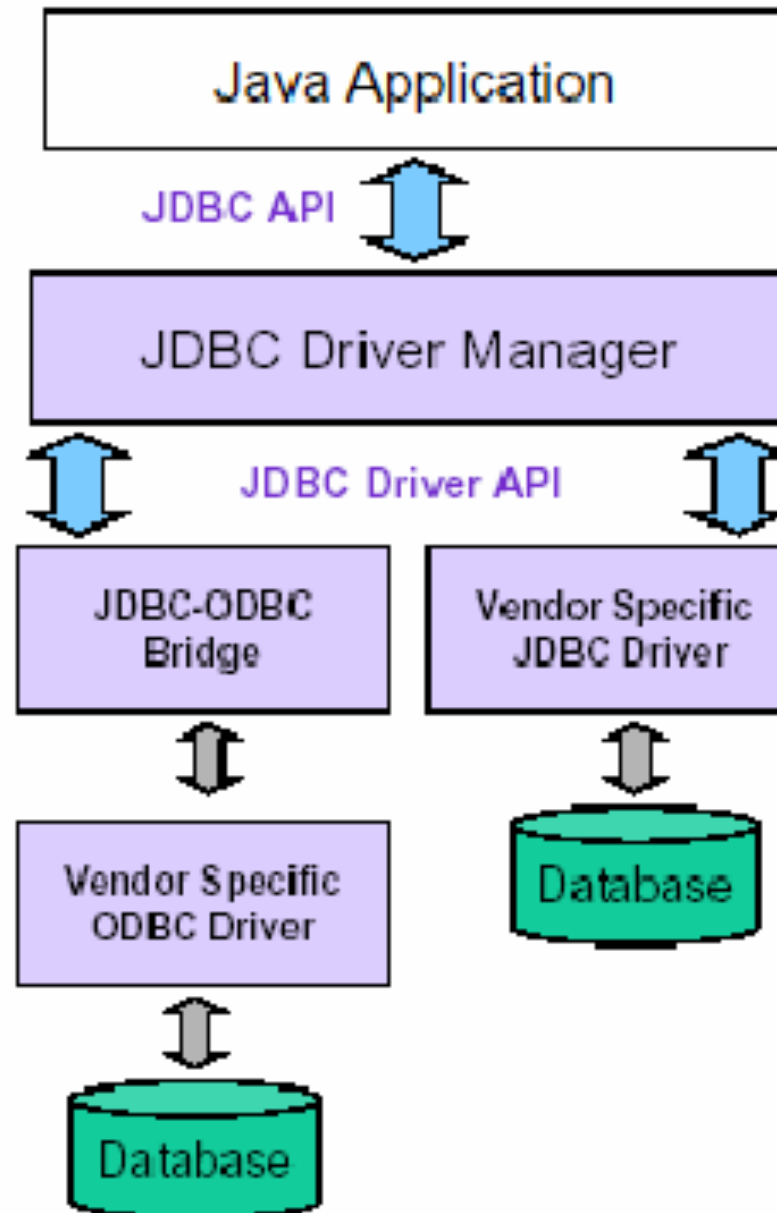


Interfaces JDBC

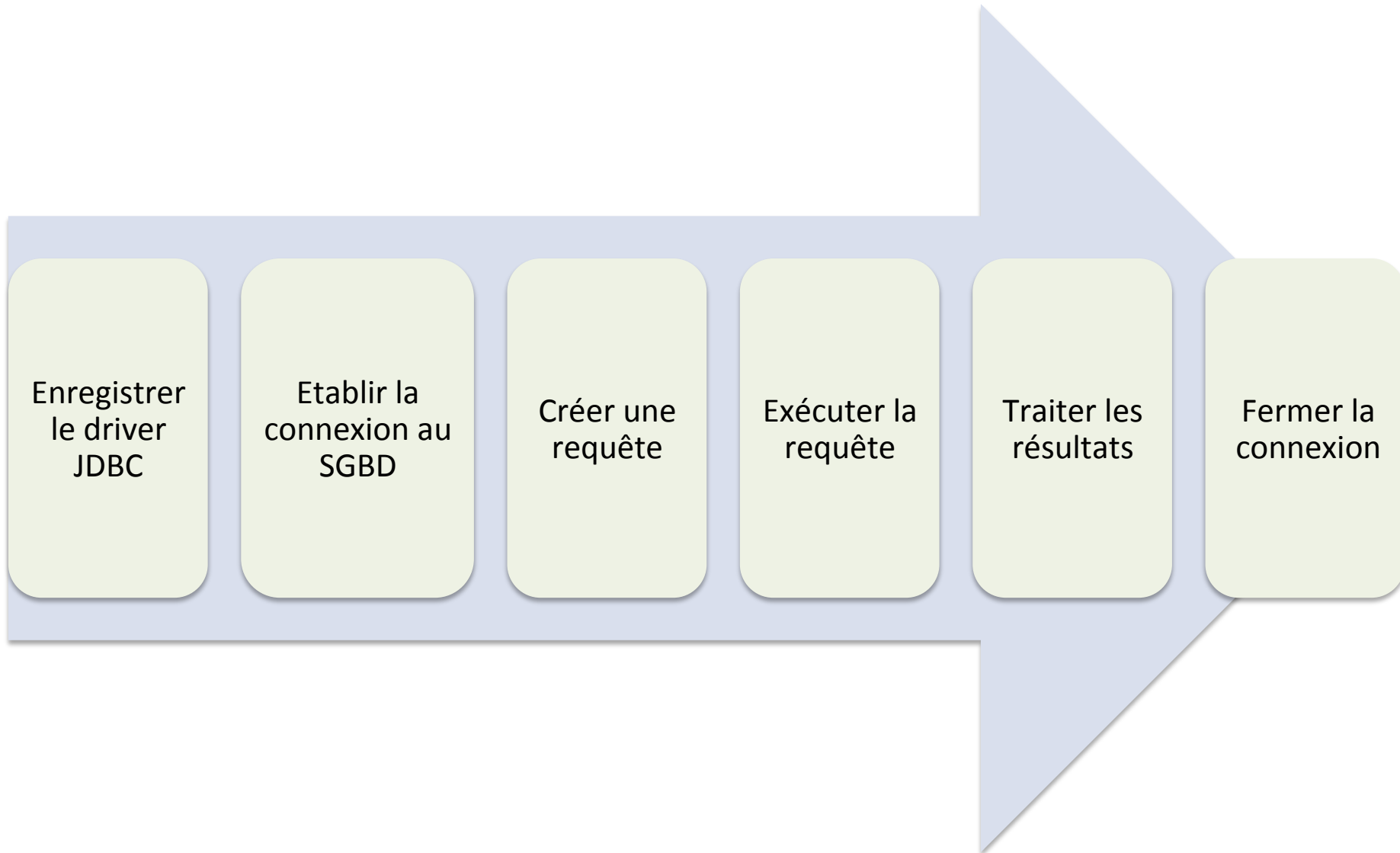
Les drivers

- Le pilote/driver Oracle
 - Package *oracle.jdbc*
 - *oracle.jdbc.**OracleDriver*** : implémente *Driver*
 - *oracle.jdbc.**OracleStatement***, *oracle.jdbc.**OracleResultSet***, ...
- Le pilote/driver MySQL
 - Package *com.mysql.jdbc*
 - *com.mysql.jdbc.**Driver*** : implémente *Driver*
 - *com.mysql.jdbc.**Statement***, *com.mysql.jdbc.**ResultSet***, ...

Architecture sous JDBC



Procédure d'accès à une base de données



Enregistrement d'un pilote

- Déclarer/enregistrer le driver à utiliser auprès du *DriverManager* de JDBC:

```
Class.forName("ClasseDuDriverDuSGBD")
```

```
Class.forName("oracle.jdbc.OracleDriver")
```

```
Class.forName("com.mysql.jdbc.Driver")
```

OU

```
DriverManager.registerDriver(instance de driver)
```

```
DriverManager.registerDriver(new oracle.jdbc.OracleDriver())
```

Enregistrement d'un pilote

- Pont ODBC-JDBC

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

- Mysql

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

- SQLServer

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

Connexion au SGBD

- C'est le *DriverManager* qui en est responsable : méthode *getConnection()*

```
Connection conn = DriverManager.getConnection(url, user, password);
```

- Trois arguments :
 - Chemin d'accès à la base de données : protocole d'accès + adresse du serveur + port de logique + nom de la base de données
 - Nom du compte de connexion
 - Mot de passe

Etablir une connexion au SGBD

L'URL (chemin) d'accès à la base de données : dépend du SGBD utilisé

- Oracle

URL="jdbc:oracle:thin:@128.123.65.9:1521:nomdelabase"

protocole adresse port nom de la base de données

- MySQL

URL="jdbc:mysql://localhost:3306/mysqlinstance"

- SQLServer

– URL= "jdbc:sqlserver://www.serveur.fr:1433;
databaseName=nomBD;user=username;password=pwd";

Creation et exécution d'une requête

- Requêtes : syntaxe SQL

SELECT * FROM Personnel WHERE service = 'technique'

- *Statement* : permet d'exécuter des requêtes
- Trois types de requêtes (*Statement*) :
 - *Statement* : requêtes statiques simples
 - *PreparedStatement* : requêtes précompilées (souvent paramétrées)
 - *CallableStatement* : procédures stockées (pl/sql)

Creation et exécution de requêtes:

Requête statiques

- Requête de consultation de données

```
req1 = SELECT id, nom, fonction FROM Personnel
```

```
Connection conn = DriverManager.getConnection(url, user, password);
```

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

```
(ResultSet) st.executeQuery (req1);
```

- Requête de mise à jour (Update, Insert, Delete, Create Table, ...)

```
req2 = UPDATE Personnel SET fonction='ChefProjet' WHERE id=3;
```

```
Connection conn = DriverManager.getConnection(url, user, password);
```

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

```
(int) st.executeUpdate (req2);
```

Récupérer le résultat de requêtes

- Requêtes de consultation SELECT
 - `executeQuery()` renvoie les tuples du résultat dans un objet de la classe *ResultSet*
 - ***ResultSet*** (*Iterator*) : liste des lignes. Parcours des résultat avec un curseur (méthode *next*)
 - Méthodes de *ResultSet*
 - `next ()`: fait avancer le curseur
 - `getInt(...)` : retourne une cellule de type `int`
 - `getString (...)`: retourne une cellule de type `String`
 - `getObject (...)`: retourne une cellule de tout type

curseur 

1	Dupont
2	Durand
3	Alban

Récupérer le résultat de requêtes

- Méthode *next()* de *ResultSet*
 - au départ, le curseur est positionné avant le premier tuple
 - exécuter *next()* au moins une fois pour pointer le premier tuple
 - faire avancer le curseur sur le tuple suivant avec *next()*
 - *next()* retournera false après lecture du dernier tuple, true sinon

Récupérer le résultat de requêtes

- Récupération des résultats : méthodes de *ResultSet*
 - `getInt (int n)` ou `getInt (String colonne)`: retourne la valeur de la n^{ème} (portant le nom colonne donné) colonne de la ligne courante (pointée par le curseur). La colonne est de type **int**
 - `getString (int n)` ou `getString (String colonne)`: retourne la valeur de la n^{ème} (portant le nom colonne donné) colonne de la ligne courante (pointée par le curseur). La colonne est de type **String**

Correspondance de types Java-SQL

Types SQL	Types Java	Méthodes de <i>ResultSet</i>
BOOLEAN (TINYINT(1))	boolean	getBoolean()
INTEGER	int	getInt()
DOUBLE, FLOAT	double	getDouble()
CHAR, VARCHAR	String	getString()
REAL	float	getFloat()
DATE	Date	getDate()
NUMERIC, DECIMAL	BigDecimal	getBigDecimal()
TIME	Time	getTime()
...	...	

- Ces types sont définis par chaque fournisseur (implémentés dans le driver)

Exemple

//enregistrer le pilote

```
DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
```

//creer la connexion

```
Connection conn = DriverManager.getConnection(  
    "jdbc:oracle:thin:@serveur:1521:nomdb", user, pwd);
```

//creer un Statement pour une requête statique

```
String req = SELECT idpers, nompers FROM personnel
```

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

//execution de la requete

```
ResultSet res = stmt.executeQuery(req);
```

```
while (res.next()){
```

```
    int id = res.getInt ("idpers");                // ou res.getInt (1)
```

```
    String nom = res.getString("nompers");          // ou res.getString (2)
```

```
    System.out.println("Personnel : " + id + " " + nom);
```

```
}
```


Fermeture de la connexion

- `resultset.close()`
- `statement.close()`
- `connexion.close()`

Exemples de requêtes de mises à jour

Insertion de données :

```
String insertLehigh = "INSERT INTO Personnel values (1,'Dupont',100)" ;  
int nbadd = stmt.executeUpdate(insertLehigh);
```

Suppression de données :

```
Statement stmt = conn.createStatement();  
String reqDel = "DELETE FROM personnel WHERE nom = 'Harry' ";  
int nbdel = stmt.executeUpdate(reqDel);
```

Création de table :

```
String reqldd = "CREATE TABLE Client (id INTEGER PRIMARY KEY, nom  
                VARCHAR2(30) " ;  
stmt.executeUpdate(reqldd);
```

Création et exécution de requêtes pré-compilées

- La méthode `prepareStatement()` de l'objet `Connection` crée un *PreparedStatement* :

```
String reqParam = "SELECT * FROM Clients WHERE name = ? AND age =?"  
PreparedStatement pstmt = conn.prepareStatement(reqParam);  
pstmt.setString (1, "Durand");  
pstmt.setInt (2, 30);
```

- Les paramètres sont spécifiés par un ?
- Les paramètres sont affectés les méthodes : `setInt(n°, val)`, `setString(n°, val)`, `setDate(n°, val)`, ...
- Ces méthodes nécessitent 2 arguments
 - le premier *n°* (int) indique le rang de l'argument dans la requête
 - le second *val* indique la valeur à affecter au paramètre

Création et exécution de requêtes pré-compilées

- Requête statique

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

```
stm.executeQuery("SELECT * FROM Personnel WHERE fonction="+ var);
```

- Requête (la même) paramétrée

```
PreparedStatement pstmt = conn.prepareStatement (
```

```
    "SELECT * FROM Personnel WHERE fonction=?" );
```

```
pstmt.setString (1, var);
```

```
pstmt.executeQuery();
```

- Quelle différence ? Performances...

- Privilégiez les requêtes paramétrées, notamment pour les applications web

Accès aux méta-données

- Méta-données de la base de données

`DataBaseMetaData md = conn.getMetaData ();`

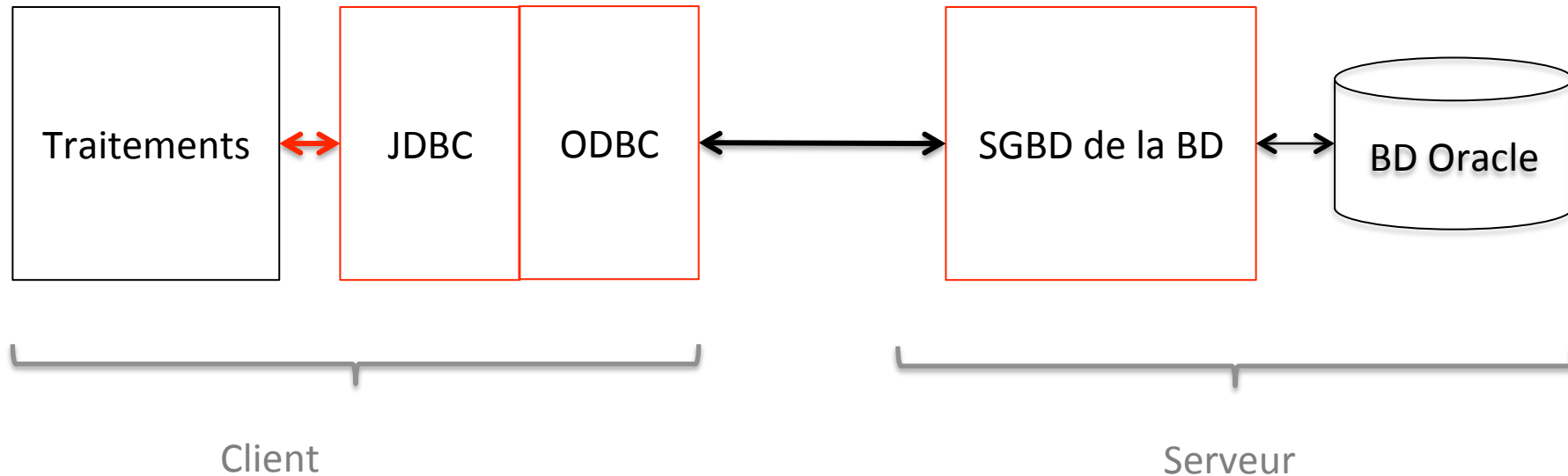
Instances de BD : `md.getShcemas ()`

Tables : `md.getTables ()`

Utilisateur connecté : `md.getUserName ()`

Le standard Microsoft ODBC

- ODBC (Open DataBase Connectivity) : standard d'accès aux bases de données sous Windows



- `Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")`
- N'est plus supporté à partir de JDK8 !