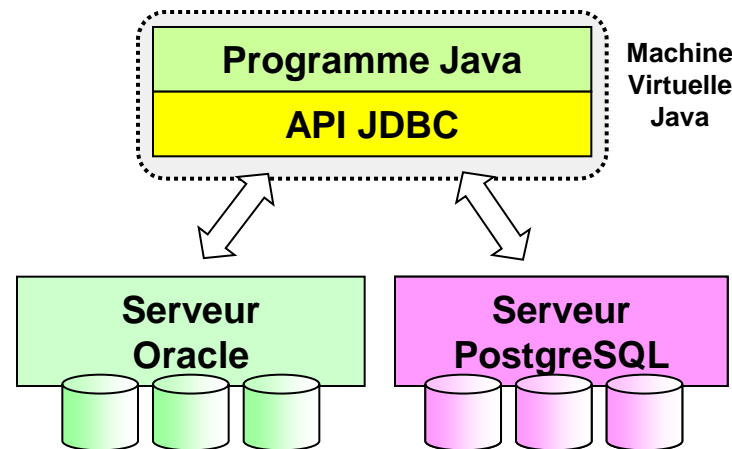




# Java DataBase Connectivity

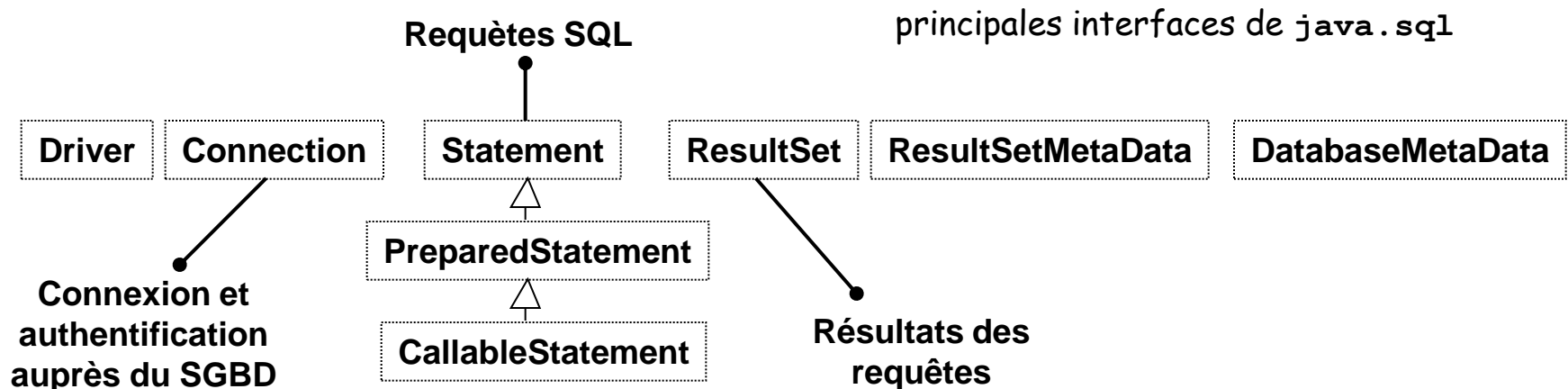
- **JDBC Java Data Base Connectivity**

- *API java standard qui permet un accès homogène à des bases de données depuis un programme Java au travers du langage SQL.*

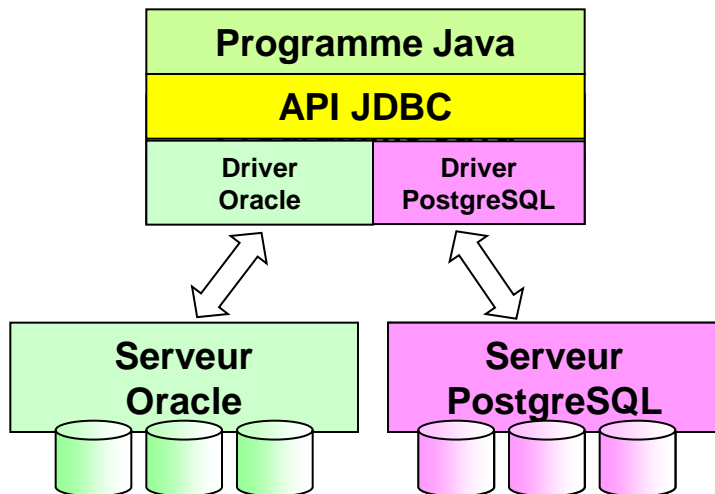


- *L'API JDBC est **indépendante** des SGBD.*
  - Un changement de SGBD ne doit pas impacter le code applicatif.

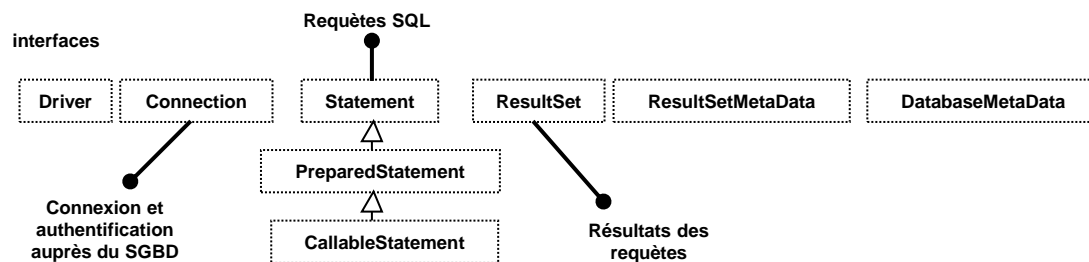
- L'API JDBC définit un ensemble d'interfaces (package `java.sql`) qui définissent un protocole de communication entre le programme java client et le serveur de base de données pour
  - ouverture/fermeture de connexions à une base de données
  - exécution de requêtes SQL
  - exploitation des résultats
    - correspondance types SQL-types JAVA
  - accès au méta-modèle
    - description des objets du SGBD



- Le code applicatif est basé sur les interfaces du JDBC
- Pour accéder à un SGBD il est nécessaire de disposer de classes **implémentant** ces interfaces.
  - Elles **dépendent** du SGBD adressé.
  - L'ensemble de ces classes pour un SGBD donné est appelé **pilote (driver) JDBC**



- Il existe des *pilotes* pour tous les SGBD importants du marché (Oracle, MySQL, PostgreSQL, DB2, ...)
- JDBC spécifie uniquement l'API que ces *pilotes* doivent respecter. Ils sont réalisés par une tierce partie (fournisseur du SGBD, «éditeur de logiciel...)
- l'implémentation des drivers est totalement libre

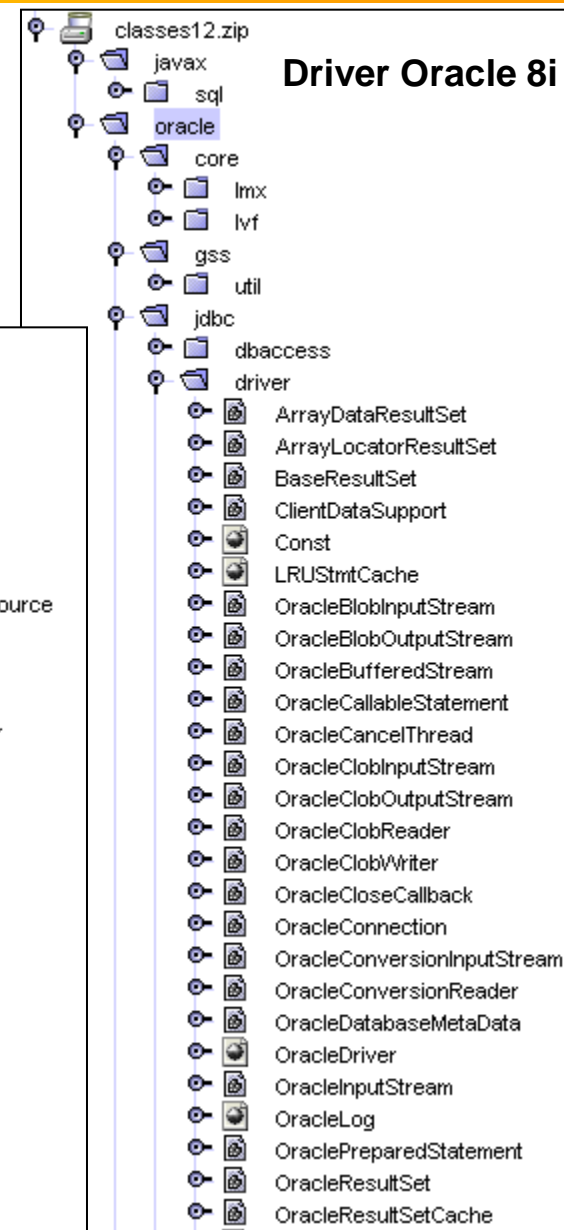
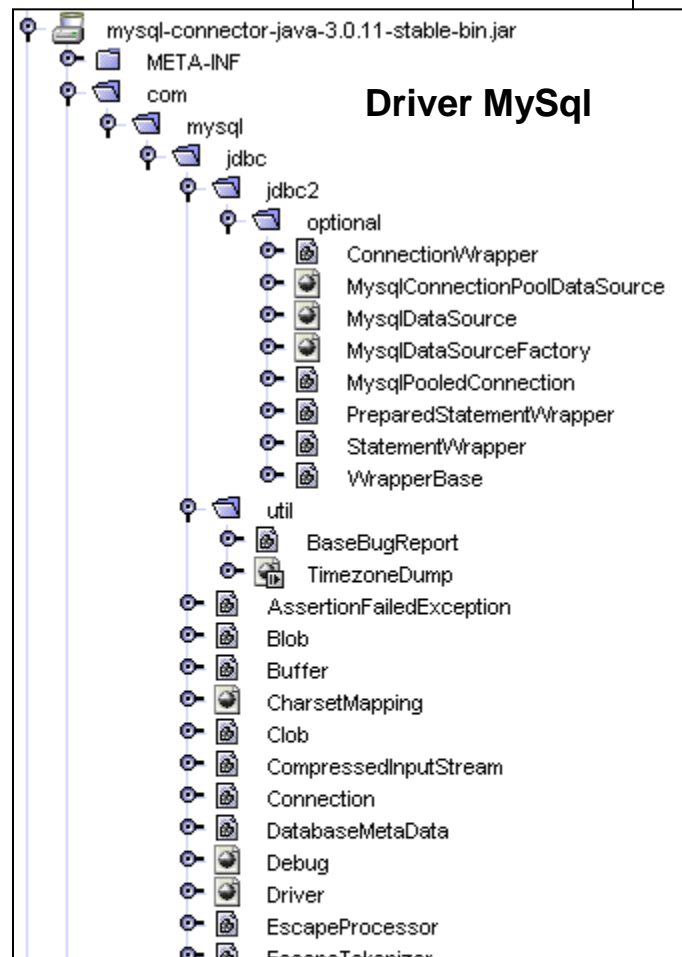


Les interfaces définissent une **abstraction** du pilote (driver) de la base de données.

Chaque fournisseur propose sa **propre implémentation** de ces interfaces.

Les classes d'implémentation du driver jdbc sont dans une archive (fichier jar ou zip) qu'il faut intégrer (sans la décompresser) au niveau du classpath de l'application au moment de l'exécution

Au niveau du programme d'application **on ne travaille qu'avec les abstractions** (interfaces) sans se soucier des classes effectives d'implémentation



- il existe 4 types de pilotes JDBC
  - *type 1 : pont JDBC – ODBC*
  - *type 2 : pilote qui fait appel à des fonctions natives (code non Java, le plus souvent en C ou C++) de l'API du SGBD*
  - *type 3 : pilote qui permet l'utilisation d'un serveur middleware*
  - *type 4 : pilote entièrement en Java qui utilise directement le protocole réseau du SGBD*

- Liste des drivers disponibles à :  
<http://www.oracle.com/technetwork/java/index-136695.html>
- **sélection d'un driver :**
  - *choix entre vitesse, fiabilité et portabilité.*
  - *Programme « standalone », avec une interface graphique qui s'exécute toujours sur un système Windows peut tirer bénéfice de performances d'un driver type 2 (driver code-natif).*
  - *Une applet peut nécessiter un driver de type 3 (pour passer un firewall).*
  - *Une servlet déployée sur de multiples plateformes peut nécessiter la souplesse offerte par des drivers de type 4.*
  - ...

- 4 catégories de drivers JDBC
- type 1 : Pont JDBC-ODBC (Open Data Base Connectivity)

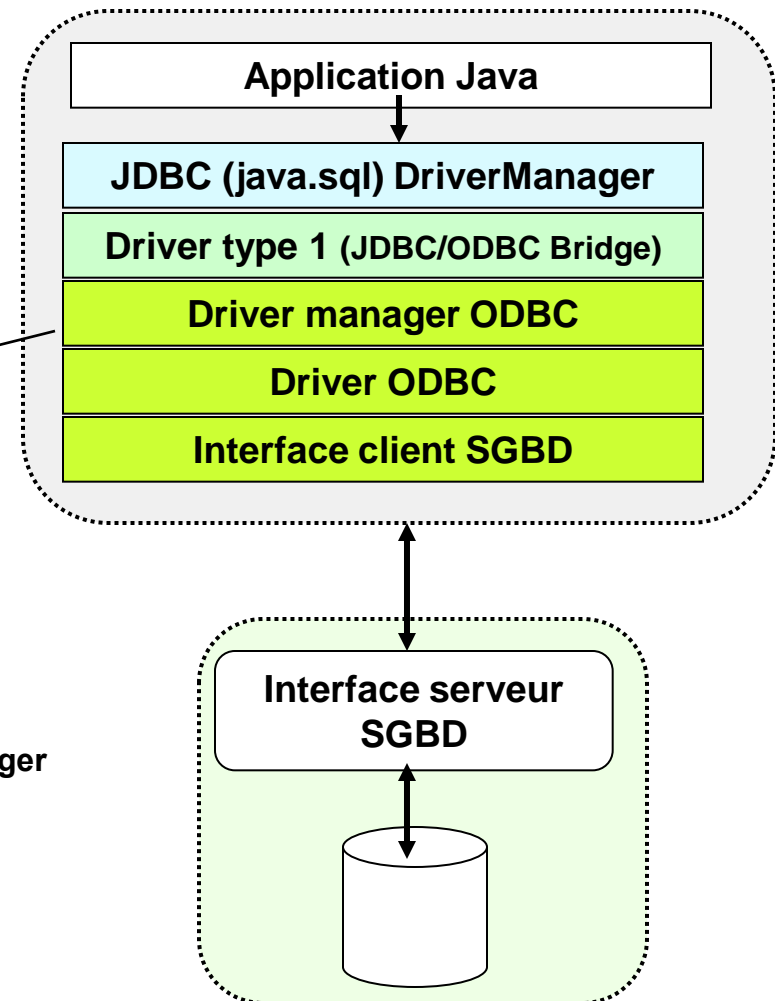
## ODBC

- interface d'accès (C) aux SGBD définie par Microsoft
- standard de fait, très grand nombre de SGBD accessibles

impose chargement dans la mémoire vive de la plateforme d'exécution de bibliothèques dynamiques

## code binaire ODBC sur le client

- alourdit processus d'installation et de maintenance
- problème de sécurité pour les applets
  - applets « untrusted » n'ont pas l'autorisation de charger en mémoire du code natif





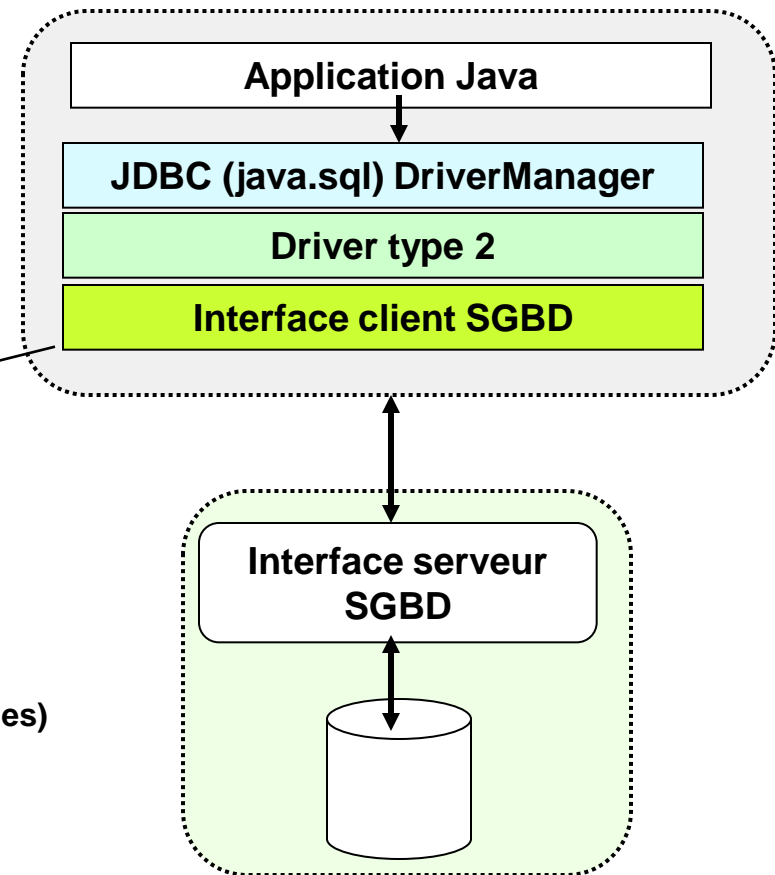
- Type 2 : API native

interface d'accès entre le driver manager  
JDBC et l'interface cliente du SGBD

impose chargement dans la mémoire vive de la  
plateforme d'exécution de bibliothèques dynamiques  
(code binaire de l'interface client spécifique au SGBD  
par exemple bibliothèques OCI, Oracle Call Interface, conçues  
initialement pour programmeurs C/C++ )

## Driver dédié à un SGBD particulier

- moins ouvert que pont JDBC/ODBC
  - potentiellement plus performant (moins de couches logicielles)
- mêmes problèmes qu'avec pont JDBC-ODBC
- code natif sur plateforme d'exécution



- Type 3 : JDBC-Net

traduit appels JDBC suivant un protocole réseau à vocation universelle indépendant des fournisseurs de SGBD (Sql\*net, NET8)

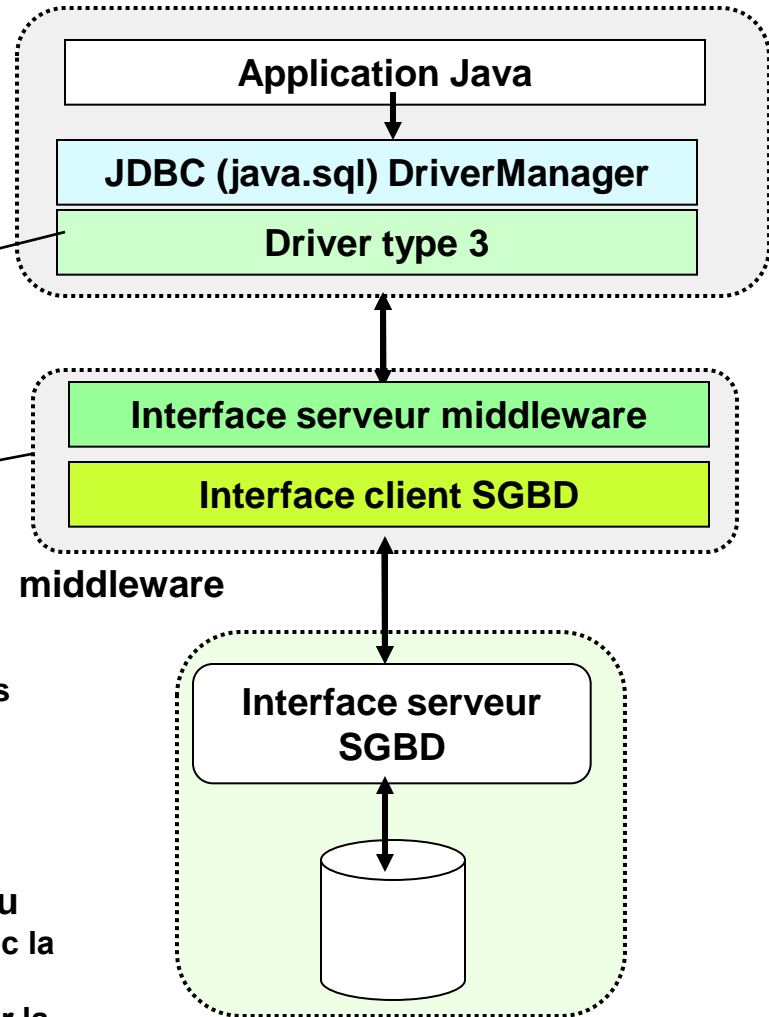
requêtes réseau doivent être ensuite traduites par un serveur dédié aux requêtes spécifiques à un SGBD (par exemple WebLogic de BEA)

### drivers 100% Java

peuvent être utilisés depuis une applet (les drivers ne sont plus du code natif et peuvent être chargés comme n'importe quel composant Java)

si l'application est une applet, le *modèle classique de sécurité* peut poser des problèmes de connexion réseau

- une applet « untrusted » ne peut ouvrir une connexion qu'avec la machine sur laquelle elle est hébergée
- il suffit d'installer le serveur Web et le serveur middleware sur la même plateforme. Possibilité d'accéder alors à un SGBD situé n'importe où sur le réseau.



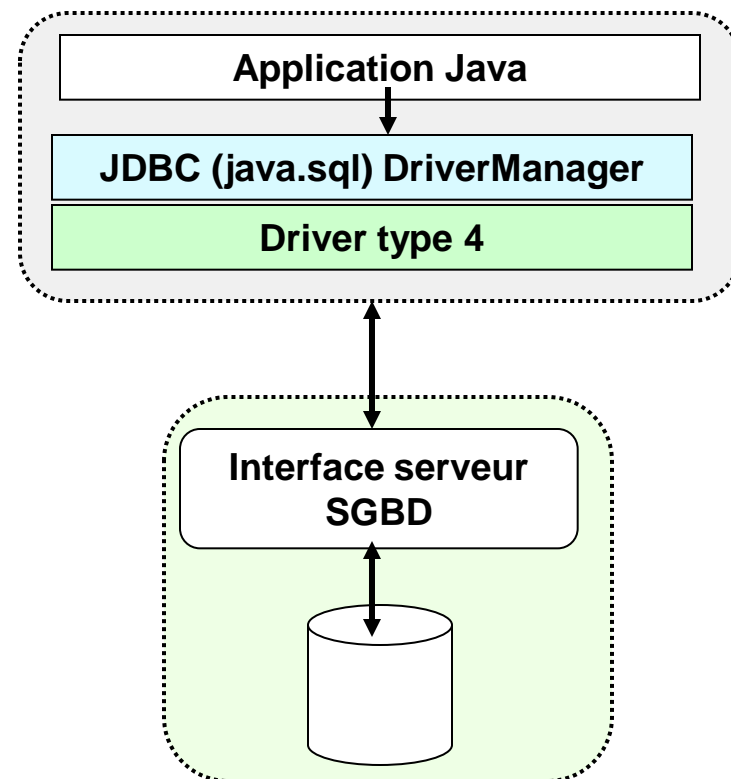
- Type 4 : Thin (protocole natif)

le driver interagit directement avec le gestionnaire du SGBD  
utilise directement protocole réseau du SGBD  
(spécifique à un fournisseur de SGBD)

Driver 100% Java (connexion via sockets Java)  
Solution la plus la plus élégante et la plus souple

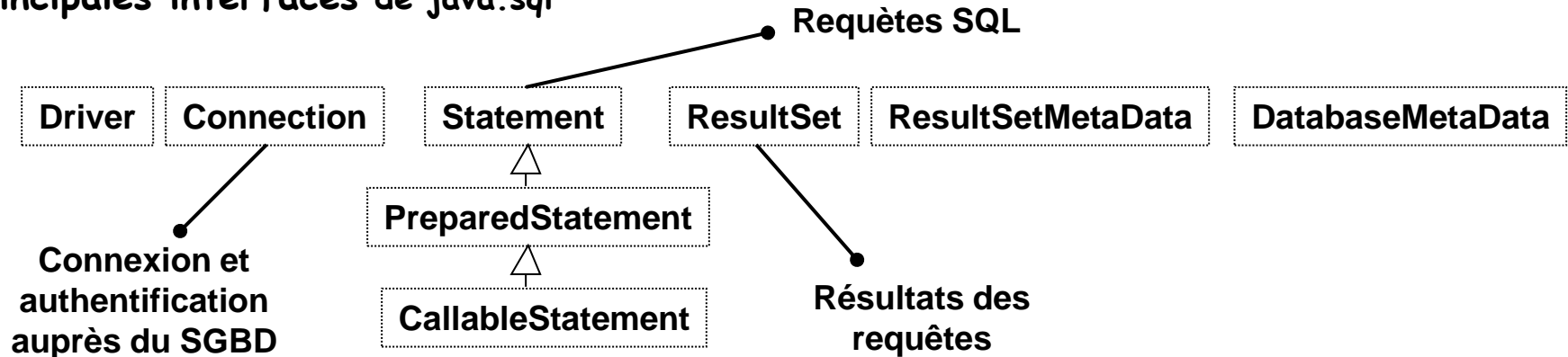
si l'application est une applet, le *modèle classique de sécurité* des applets impose que le SGBD soit hébergé sur le serveur Web

Durant le projet le driver utilisé pour accéder à Oracle sera de ce type

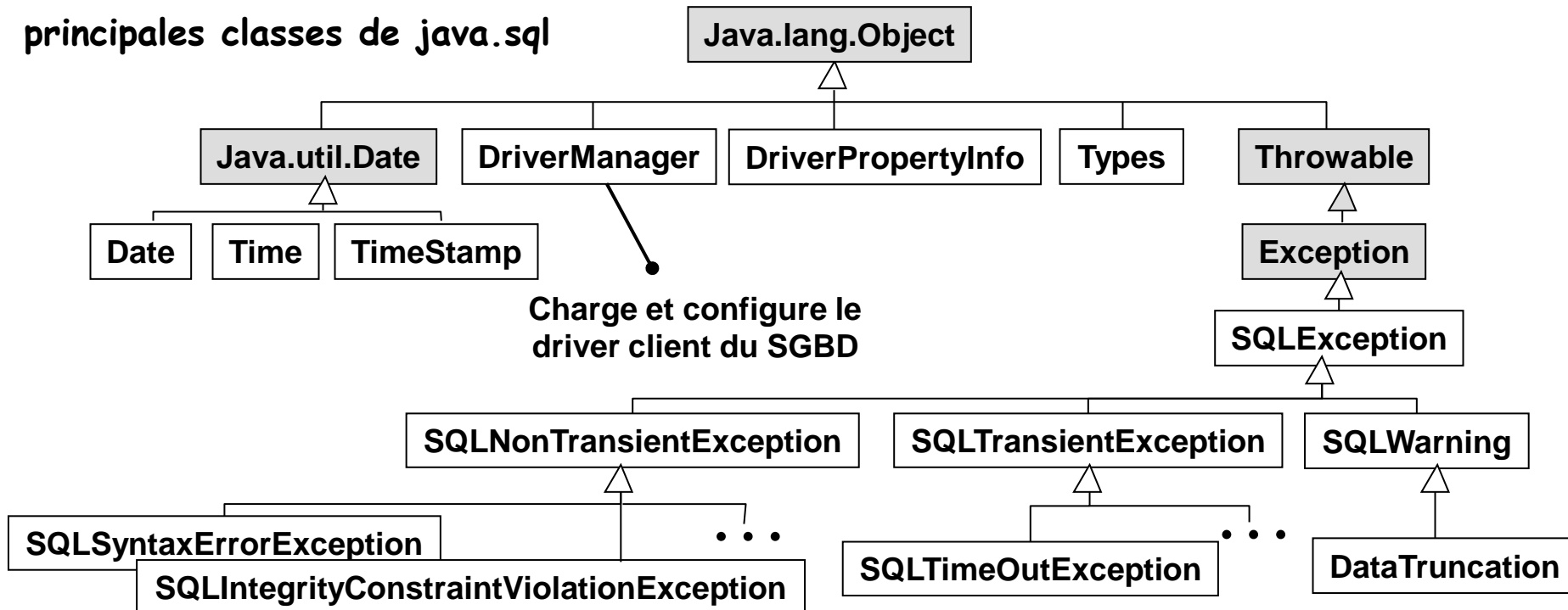


- Différentes versions
  - JDBC 1.0 Core API (JDK 1.1) : package `java.sql`
    - supporte le standard SQL-2 entry level
  - JDBC 2.0 (Java 2 JDK 1.2) : packages `java.sql` `javax.sql`
    - support de certaines fonctionnalités de SQL-3
  - JDBC 3.0 (JDK 1.4) : packages `java.sql` `javax.sql`  
`javax.sql.rowset`
    - support d'autres fonctionnalités de SQL-3
    - nouveau support pour la gestion des `ResultSet`
  - JDBC 4.0 (JDK 1.6)
    - facilité d'écriture au travers d'annotations

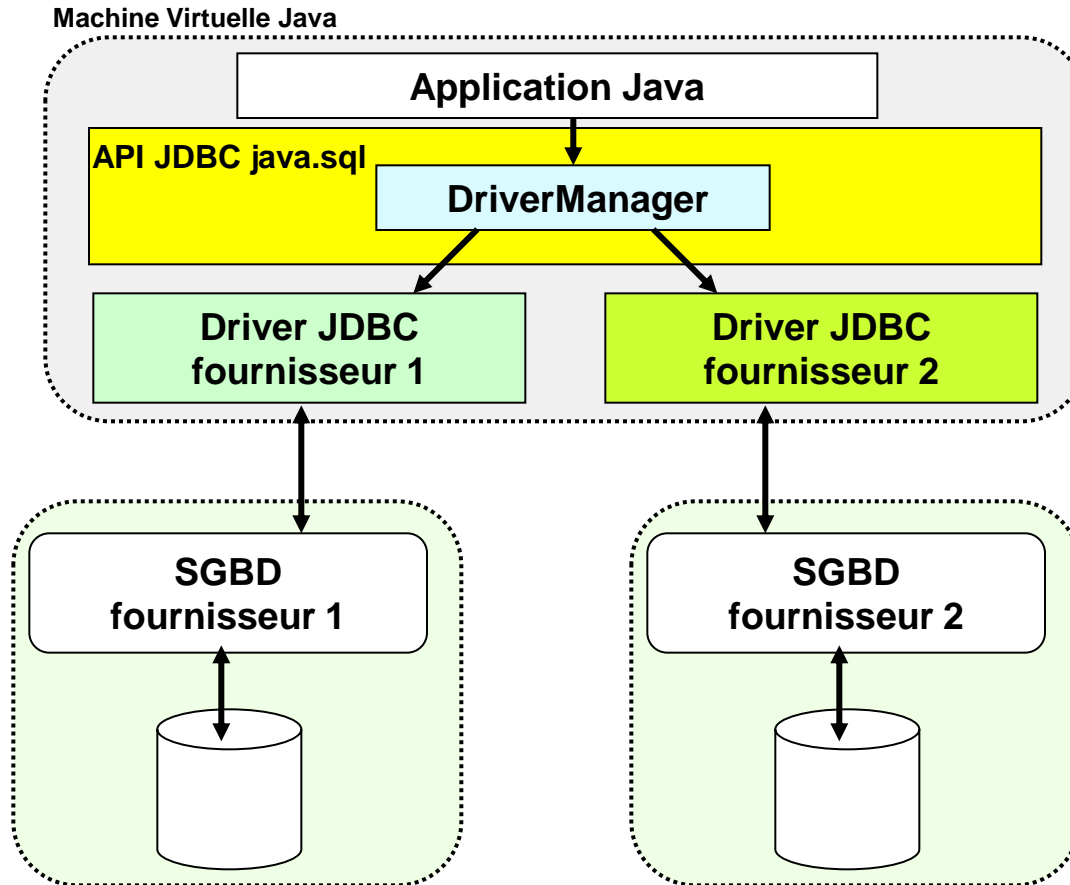
## principales interfaces de java.sql



## principales classes de java.sql



- DriverManager : classe java à laquelle s'adresse le code de l'application cliente.



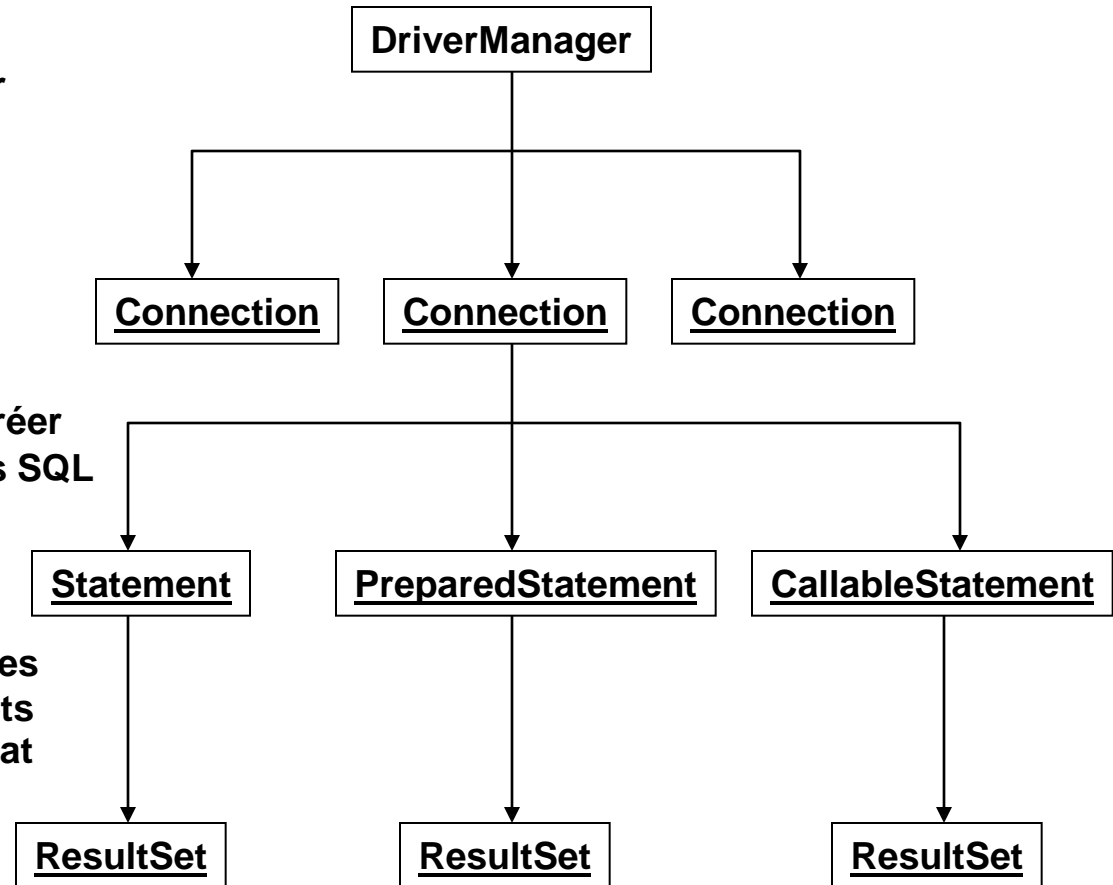
Le driver manager permet de charger et configurer les pilotes JDBC nécessaires à l'application

- Objets instanciés à partir des types Java définis dans `java.sql`

**DriverManager** permet de créer des objets **Connection**

Un objet **Connection** permet de créer des objets encapsulant des requêtes SQL

Les objets encapsulant les requêtes SQL permettent de créer des objets **ResultSet** encapsulant le résultat d'une requête



- Avant de pouvoir être utilisé, le driver doit être enregistré auprès du **DriverManager** de jdbc.

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

- Mais si on regarde mieux la doc de JDBC...

When a Driver class is loaded, it should create an instance of itself and register it with the DriverManager.

- Il est donc préférable d'exploiter les possibilités de chargement dynamique de classes de JAVA
  - *Utiliser la méthode **forName** de la classe **Class** avec en paramètre le nom complet de la classe du driver.*

```
try {  
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");  
    Class.forName("oracle.jdbc.OracleDriver").newInstance();  
}  
catch (ClassNotFoundException e) {  
    ...  
}
```

Chargement d'un pilote ODBC

Chargement d'un pilote Oracle

- *Permet de paramétrer le driver sans modifier l'application (par exemple nom du driver stocké dans un fichier de configuration (properties file))*



- à partir de JDBC 4.0 plus besoin de charger explicitement le pilote JDBC
  - *tous les drivers compatibles JDBC 4.0 présent dans le classpath de l'application sont automatiquement chargés.*

- Ouverture de la connexion :

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

- Identification de la BD via un URL (Uniform Resource Locator)  
de la forme générale

**jdbc:driver:base**

l'utilisation de JDBC	le driver ou le type du SGBDR	identification de la base
--------------------------	----------------------------------	------------------------------

La forme exacte dépend de la BD, chaque BD nécessitant des informations spécifiques pour établir la connexion. Par exemple pour le driver Oracle JDBC-Thin :

**jdbc:oracle:thin:@serveur:port:base**

nom IP du serveur	numéro de port socket à utiliser	nom de la base
----------------------	-------------------------------------	-------------------

- Exemple :

```
Connection conn = DriverManager.getConnection(  
    "jdbc:oracle:thin:@nqecnjquv:1521<ziud", user, password);
```

- Trouver un driver JDBC :

- sur les sites des fournisseurs de BD

- Liste des drivers disponibles à :

<http://www.oracle.com/technetwork/java/index-136695.html>

**Oracle 11g sur im2ag-oracle.e.ujf-grenoble.fr :**

```
try {
    Class.forName("oracle.jdbc.OracleDriver"),
    Connection connect = DriverManager.getConnection(
        "jdbc:oracle:thin:@nqecnjquv:1521:ziud",
        user, password);
    ... // utilisation connexion pour accéder à la base
}
catch (ClassNotFoundException e) {
    ...
}
catch (SQLException sqle) {
    ...
}
```

ojdbc6.jar

Quand getConnection est invoquée le DriverManager interroge chaque driver enregistré, si un driver reconnaît l'url il crée et retourne un objet Connection.

**une base MySQL 4.1 locale :**

**mysql-connector-java-3.1.6-bin.jar**

```
try {
    Class.forName("com.mysql.jdbc.Driver");
    Connection connect = DriverManager.getConnection(
        "jdbc:mysql://localhost/test",
        user, password);
    ...
}
catch (ClassNotFoundException e) {
    ...
}
catch (SQLException e) {
    ...
}
```

- Une application peut maintenir des connexions multiples
  - *le nombre limite de connexions est fixé par le SGBD lui même (de quelques dizaines à des milliers).*
- Quand une `Connection` n'a plus d'utilité prendre soin de la **fermer explicitement**.
  - *Libération de mémoire et surtout des ressources de la base de données détenues par la connexion*

Erreur SQL  
lors du  
dialogue  
avec la BD

```
try {
    Connection conn = DriverManager.getConnection("jdbc:odbc:companydb",
        user, passwd);
    ...
    // utilisation de la connexion pour dialoguer avec la BD
    ...
    // fermeture de la connexion
    conn.close();
}
catch (SQLException e) {
    ...
}
```

L'instruction `close` n'est pas exécutée.  
La connexion reste ouverte !

**Comment garantir la fermeture des connexions ?**

- Pour garantir fermeture de la connexion : Utilisation d'une clause **finally**

Pour que `conn` soit connue dans le bloc **finally**

```
Connection conn = null;
try {
    conn = DriverManager.getConnection("jdbc:odbc:companydb",
        user, passwd);
    ...
    // utilisation de la connexion pour dialoguer avec la BD
    ...
}
catch (SQLException e) {
    ...
}
finally {
    try {
        if (conn != null)
            conn.close();
    }
    catch (SQLException e) {
        e.printStackTrace();
    }
}
```

Le compilateur impose d'initialiser `conn`

`conn` peut ne pas avoir été initialisée

`close` peut provoquer une `SQLException`

- Pour garantir fermeture de la connexion : utilisation d'un try avec ressources (Java7) au lieu de la clause **finally**

```
Connection conn = null;
try {
    conn = DriverManager.getConnection("jdbc:odbc:companydb",
        user, passwd);
    ...
    // utilisation de la connexion pour dialoguer avec la BD
    ...
}
catch (SQLException e) {
    ...
}

finally {
    try {
        if (conn != null)
            conn.close();
    }
    catch (SQLException e){
        e.printStackTrace();
    }
}
```

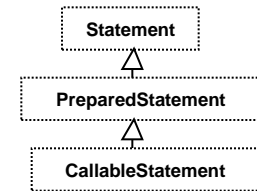
Nouveau !



```
try ( Connection conn =
    DriverManager.getConnection("jdbc:odbc:companydb",
        user, passwd) ) {
    ...
    // utilisation de la connexion pour dialoguer avec la BD
    ...
}
catch (SQLException e) {
    ...
}
```

**Les ressources sont automatiquement fermées à la fin de l'instruction try**

- Une fois une **Connection** créée on peut l'utiliser pour créer et exécuter des requêtes (*statements*) SQL.
- **3 types (interfaces) d'objets *statement* :**
  - *Statement* : requêtes simples (SQL statique)
  - *PreparedStatement* : requêtes précompilées (SQL dynamique si supporté par SGBD) qui peuvent améliorer les performances
  - *CallableStatement* : encapsule procédures SQL stockées dans le SGBD
- **3 formes (méthodes) d'exécutions :**
  - ***executeQuery*** : pour les requêtes qui retournent un résultat (*SELECT*)
    - résultat accessible au travers d'un objet *ResultSet*
  - ***executeUpdate*** : pour les requêtes qui ne retournent pas de résultat (*INSERT, UPDATE, DELETE, CREATE TABLE et DROP TABLE*)
  - ***execute*** : quand on ne sait pas si la requête retourne ou non un résultat, procédures stockées



- Création d'un *statement* :

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

- Exécution de la requête :

```
String myQuery = "SELECT prenom, nom, email " +  
                 "FROM employe " +  
                 "WHERE (nom='Dupont') AND (email IS NOT NULL) " +  
                 "ORDER BY nom";  
  
ResultSet rs = stmt.executeQuery(myQuery);
```

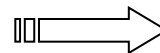
- **executeQuery(String q)** renvoie un objet de type **ResultSet**
  - *permet de décrire la table des résultats*



- **executeQuery()** renvoie un objet de classe **ResultSet**
  - *permet de décrire la table des résultats*

```
java.sql.Statement stmt = conn.createStatement();  
ResultSet rs = stmt.executeQuery("SELECT nom, code_client FROM Clients");
```

Nom	Prénom	Code_client	Adresse
DUPONT	Jean	12345	135 rue du Lac
DUROND	Louise	12545	13 avenue de la Mer
...			
...			
ZORG	Albert	45677	8 Blvd De la Montagne



Nom	Code_client
DUPONT	12345
DUROND	12545
...	
...	
ZORG	45677



- Les rangées du **ResultSet** se parcourent itérativement ligne (row) par ligne
  - **boolean next()** permet d'avancer à la ligne suivante, → **false** si pas de ligne suivante
  - Placé avant la première ligne à la création du **ResultSet**

```
while (rs.next())  
{  
    ... Exploiter les données  
}
```

- Les colonnes sont référencées par leur numéro ou par leur nom
- L'accès aux valeurs des colonnes se fait par des méthodes `getxxx(String nomCol)` ou `getxxx(int numCol)` où **xxx** dépend du type de la colonne dans la table SQL
  - *Pour les très gros row, on peut utiliser des streams.*

```
java.sql.Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM Table1");
while (rs.next())
{
    int i = rs.getInt("a");          // rs.getInt(1);
    String s = rs.getString("b");    // rs.getString(2);
    byte b[] = rs.getBytes("c");    // rs.getBytes(3);
    System.out.println("ROW = " + i + " " + s + " " + b[0]);
}
```

Attention ! En SQL les  
numéros de colonnes  
débutent à 1

- Pour chaque méthode **getXXX** le driver JDBC doit effectuer une conversion entre le type de données de la base de données et le type Java correspondant

Type SQL	Méthode	Type Java
CHAR	getString	String
VARCHAR	getString	String
NUMERIC	getBigDecimal	java.Math.BigDecimal
DECIMAL	getBigDecimal	java.Math.BigDecimal
BIT	getBoolean	boolean <i>Boolean</i>
TINYINT	getByte	byte <i>Integer</i>
SMALLINT	getShort	short <i>Integer</i>
INTEGER	getInt	int <i>Integer</i>
BIGINT	getLong	long <i>Long</i>
REAL	getFloat	float <i>Float</i>
FLOAT	getDouble	double <i>Double</i>
DOUBLE	getDouble	double <i>Double</i>
DATE	getDate	java.sql.Date
TIME	getTime	java.sql.Time
TIME STAMP	getTimestamp	java.sql.Timestamp

Peut être appelée sur n'importe quel type de valeur

getObject peut retourner n'importe quel type de donnée « packagé » dans un objet java (wrapper object )

Si une conversion de données invalide est effectuée (par ex DATE -> int), une SQLException est lancée

- Que se passe-t-il si une méthode `getXXX()` de `ResultSet` est appliquée à une valeur `NULL` SQL ?

**PERSONNES**

Column Name	Datatype	NOT NULL	AUTO INC	Comment
NOM	VARCHAR(32)	✓		
PRENOM	VARCHAR(32)	✓		
ADRESSE	VARCHAR(32)			
CODE_POSTAL	INTEGER			
DATE_NAISS	DATE			date de naissance de la personne
MARIE	TINYINT(1)			vrai si la personne est mariée

Valeurs nulles acceptées

NOM	PRENOM	ADRESSE	CODE_POSTAL	DATE_NAISS	MARIE
TOTO	Riri	NULL	38920	NULL	NULL
TITI	Fifi	NULL	73550	1961-03-14	1
TUTU	Mimi	Rue Chose	73350	1957-06-10	0

```
ResultSet rs = stmt.executeQuery("SELECT * FROM PERSONNES");
```

```
...
```

```
... rs.getString("ADRESSE")
```

```
... rs.getDate("DATE_NAISS")
```

→ ?

- Conversion automatique vers une valeur "acceptable" selon le type retourné par `getXXX()`
  - `null` si `getXXX()` retourne un type objet (ex : `getString()`, `getDate()`, ...)
  - `0` si `getXXX()` retourne un type numérique (ex : `getInt()`, `getDouble()`, ...)
  - `false` pour `getBoolean()`

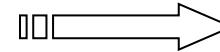
- Comment distinguer valeurs NULL des autres ?

## PERSONNES

NOM	PRENOM	ADRESSE	CODE_POSTAL	DATE_NAISS	MARIE
TOTO	Riri	NULL	38920	NULL	NULL
TITI	Fifi	NULL	73550	1961-03-14	1
TUTU	Mimi	Rue Chose	73350	1957-06-10	0

```
ResultSet rs = stmt.executeQuery("SELECT NOM,PRENOM,MARIE FROM PERSONNES ORDER BY NOM");
```

```
while (rs.next()) {  
    System.out.print(rs.getString("NOM"));  
    System.out.print(" " + rs.getString("PRENOM") + " ");  
    System.out.println(rs.getBoolean("MARIE")?"Marié":"Non Marié");  
}
```

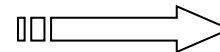


TITI Fifi Marié  
TOTO Riri Non Marié  
TUTU Mimi Non Marié  
...

- Méthode `wasNull()` de `ResultSet`

- Renvoie *true* si on vient de lire une valeur NULL, *false* sinon

```
while (rs.next()) {  
    System.out.print(rs.getString("NOM"));  
    System.out.print(" " + rs.getString("PRENOM") + " ");  
    boolean marié = rs.getBoolean("MARIE");  
    if (rs.wasNull())  
        System.out.println("?");  
    else  
        System.out.println(marié?"Marié":"Non Marié");  
}
```



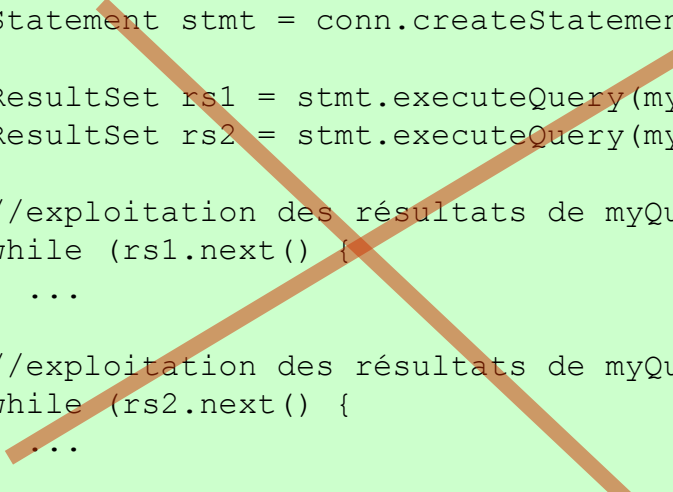
TITI Fifi Marié  
TOTO Riri ?  
TUTU Mimi Non Marié  
...

- Un objet *Statement* représente une simple (seule) requête SQL.
  - *Un appel à `executeQuery()`, `executeUpdate()` ou `execute()` ferme implicitement tout `ResultSet` actif associé avec l'objet `Statement`.*
  - *Avant d'exécuter une autre requête avec un objet `Statement` il faut être sûr d'avoir exploité les résultats de la requête précédente.*

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

ResultSet rs1 = stmt.executeQuery(myQuery1);
ResultSet rs2 = stmt.executeQuery(myQuery2);

//exploitation des résultats de myQuery1
while (rs1.next()) {
    ...
}
//exploitation des résultats de myQuery2
while (rs2.next()) {
    ...
}
```



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

ResultSet rs1 = stmt.executeQuery(myQuery1);
//exploitation des résultats de myQuery1
while (rs1.next()) {
    ...
}

ResultSet rs2 = stmt.executeQuery(myQuery2);
//exploitation des résultats de myQuery2
while (rs2.next()) {
    ...
}
```

- *Si application nécessite d'effectuer plus d'une requête simultanément, nécessaire de créer et utiliser autant d'objets `Statement`.*

```
import java.sql.*;

public class TestJDBC {
    public static void main(String[] args) throws Exception {
        Class.forName("postgres95.pgDriver");

        Connection conn = DriverManager.getConnection("jdbc:pg95:mabase",
                                                    "dedieu", "");

        Statement stmt = conn.createStatement();

        ResultSet rs = stmt.executeQuery("SELECT * from employe");
        while (rs.next()) {
            String nom = rs.getString("nom");
            String prenom = rs.getString("prenom");
            String email = rs.getString("email");
        }

        rs.close();
        stmt.close();
        conn.close();
    }
}
```

- Création d'un *PreparedStatement* (requête SQL dynamique):

- paramètres formels spécifiés à l'aide de ?

```
PreparedStatement ps = conn.prepareStatement(  
    "SELECT * FROM ? WHERE NAME = ? " );
```

Dès que l'objet est instancié, la procédure SQL est transmise au SGBD qui la pré-compile

- Passage des paramètres effectifs

- à l'aide de méthodes au format *setXXX(indice,valeur)* où XXX représente le type du paramètre

```
ps.setString(1, "Person" );
```

- Invocation et exploitation des résultats

- phase identique à celle utilisée pour SQL statique

```
for (int i=0; i < names.length; i++) {  
    ps.setString(2, names[i]) ;  
    ResultSet rs = ps.executeQuery();  
    // ... Exploitation des résultats  
}
```



- La plupart des SGBD incluent un langage de programmation interne (ex: PL/SQL d 'Oracle) permettant aux développeurs d'inclure du code procédural dans la BD, code pouvant être ensuite invoqué depuis d'autres applications.
  - *le code est écrit une seule fois et peut être utilisé par différentes applications.*
  - *permet de séparer le code des applications de la structure interne des tables. (cas idéal : en cas de modification de la structure des tables seul les procédures stockées ont besoin d'être modifiées)*
- Utilisation des procédures stockées depuis JDBC via interface **CallableStatement**
  - *Syntaxe unifiée indépendante de la manière dont celles-ci sont gérées par le SGBD (chaque SGBD a sa propre syntaxe)*
  - *Utilisation possible de la valeur de retour*
  - *Gestion des paramètres IN, OUT, INOUT*

- Préparation de l'appel

- Appel avec valeur de retour et paramètres

```
CallableStatement proc = conn.callableStatement(  
    "{? = call maProcédure(?,?)}" );
```

- Appel sans valeur de retour et avec paramètres

```
CallableStatement proc = conn.callableStatement(  
    "{call maProcédure(?,?)}" );
```

- Préparation des paramètres

```
proc.registerOUTParameter(2, Types.DECIMAL, 3);
```

2ème paramètre de type OUT

Nombre de chiffres après décimale

- Passage des paramètres IN

```
proc.setByte(1, 25);
```

1er paramètre (type IN)

valeur

- Appel

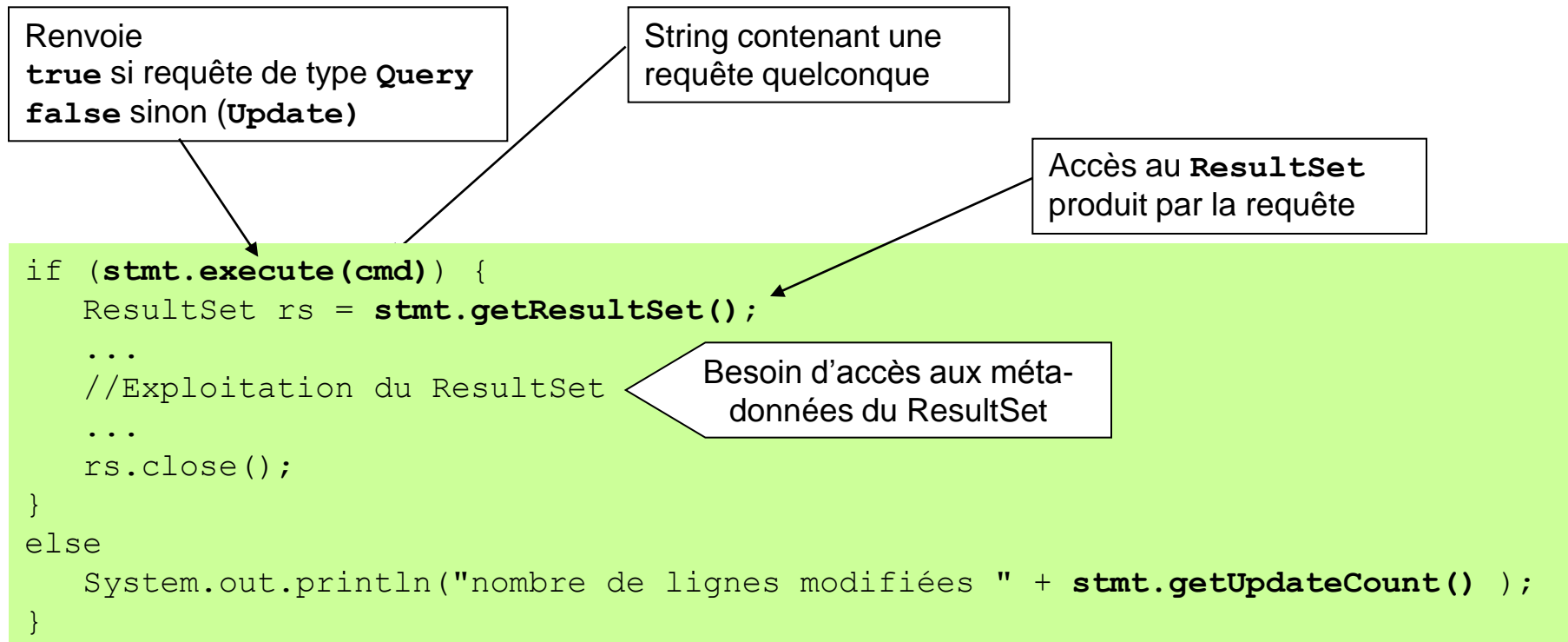
```
ResultSet rs = proc.executeQuery();
```

- Exploitation du ResultSet (idem que pour Statement et PreparedStatement)

- Récupération des paramètres OUT

```
java.Math.BigDecimal bigd = proc.getBigDecimal(2, 3);
```


- Permet de découvrir dynamiquement (au moment de l'exécution) des propriétés concernant la base de données ou les résultats de requêtes
- Exemple : lors de l'exécution d'une requête non connue à l'avance.



- Permet de découvrir dynamiquement (au moment de l'exécution) des propriétés concernant la base de données ou les résultats de requêtes
- La méthode `getMetaData()` de la classe `Connection` permet d'obtenir les méta-données concernant la base de donnée.
  - Elle renvoie un `DataBaseSetMetaData`.
  - On peut connaître :
    - les éléments SQL supportés par la base
    - la structure des données de celle-ci (`getCatalog`, `getTables...`)
- La méthode `getMetaData()` de la classe `ResultSet` permet d'obtenir les méta-données d'un `ResultSet`.
  - Elle renvoie un `ResultSetMetaData`.
  - On peut connaître :
    - Le nombre de colonnes : `getColumnCount()`
    - Le nom d'une colonne : `getColumnName(int col)`
    - Le type d'une colonne : `getColumnType(int col)`
    - ...

- Transaction : permet de ne valider un ensemble de traitements sur une BD que si ils se sont tous effectués correctement
  - *Exemple : transfert de fond = débiter un compte + créditer un autre compte*
- L'interface `Connection` offre des services de gestion des transactions
  - *`setAutoCommit(boolean autoCommit)` définit le mode de la connexion (auto-commit par défaut)*
  - *`commit()` déclenche validation de la transaction*
  - *`rollback()` annule la transaction*

```
try {
    con.setAutoCommit(false);
    // exécuter les instructions qui constituent la transaction
    stmt.executeUpdate("UPDATE INVENTORY SET ONHAND = 10 WHERE ID = 5");
    stmt.executeUpdate("INSERT INTO SHIPPING (QTY) VALUES (5)");
    ...
    // valide la transaction
    con.commit()
}
catch (SQLException e) {
    con.rollback(); // annule les opérations de la transaction
}
```

- `int getTransactionIsolation()` (de l'interface `Connection`) pour savoir quel support le SGBD et le pilote JDBC offrent pour les transactions
  - `Connection.TRANSACTION_NONE`
    - *pas de support*
  - `Connection.TRANSACTION_READ_UNCOMMITTED`
    - *"dirty-reads" un row modifié par une transaction peut être lu par une autre transaction avant que les modifications n'aient été validées par un commit*
    - *"non-repeatable reads" une transaction lit un row, une seconde transaction modifie le row, la première transaction relit le row et obtient des valeurs différentes*
    - *"phantom-reads" une transaction lit tous les row satisfaisant une condition (clause `WHERE`), une seconde transaction insère un row qui satisfait cette condition, la première transaction relit les row avec la même condition et elle obtient les row supplémentaires insérés par la seconde.*
  - `Connection.TRANSACTION_READ_COMMITTED`
    - *pas de dirty-reads*  **Niveau pour Oracle 9i sur hopper avec `ojdbc14`**
  - `Connection.TRANSACTION_REPEATABLE_READ`,
    - *pas de dirty-reads et non-repeatable-reads*
  - `Connection.TRANSACTION_SERIALIZABLE`
    - *pas de dirty-reads, non-repeatable-reads et phantom-reads*

- API JDBC 3.0 a ajouté possibilité de définir des points de sauvegarde dans une transaction

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

conn.setAutoCommit(false);
...
int rows = stmt.executeUpdate("INSERT INTO TAB1 ... ");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("SAVEPOINT_1");
rows = stmt.executeUpdate("INSERT INTO TAB1 ... ");
...
conn.rollback(svpt1);  Annule toutes les opérations effectuées depuis le
                       point de sauvegarde
...
conn.commit();
```

**Pour retirer un point de sauvegarde**

```
conn.releaseSavePoint("SAVEPOINT_1")
```

**SAVEPOINT\_1 ne peut plus être utilisé dans un rollback par la suite**

- *SQLException* définit les méthodes suivantes :
  - *getSQLState()* : --> un code d'état de la norme SQL ANSI-92
  - *getErrorCode()* : --> un code d'erreur spécifique (« vendor-spécific »)
  - *getNextException()* : --> permet aux classes du JDBC de chaîner une suite de *SQLExceptions*

```
// du code très consciencieux
try {

    ...
}
catch (SQLException e) {
    while (e != null) {
        System.out.println("SQL Exception");
        System.out.println(e.getMessage());
        System.out.println("ANSI-92 SQL State : "+e.getSQLState());
        System.out.println("Vendor error code : "+e.getErrorCode());
        e = e.getNextException();
    }
}
```



- Les classes du JDBC ont la possibilité de générer sans les lancer des exceptions quand un problème est intervenu mais qu'il n'est pas suffisamment grave pour interrompre le programme
  - *Exemple : fixer une mode de transaction qui n'est pas supporté la base de données cible (un mode par défaut sera utilisé)*
- **SQLWarning** encapsule même information que **SQLException**
- Pour les récupérer pas de bloc **try catch** mais à l'aide de méthode **getWarnings** des interfaces **Connection**, **Statement**, **ResultSet**, **PreparedStatement**, **CallableStatement**

```
void printWarnings(SQLWarning warn) {  
    while (warn != null) {  
        System.out.println("\nSQL Warning");  
        System.out.println(warn.getMessage());  
        System.out.println("ANSI-92 SQL State : "+warn.getSQLState());  
        System.out.println("Vendor error code : "+warn.getErrorCode());  
        warn = warn.getNextException();  
    }  
}
```

```
...  
ResultSet rs = stmt.executeQuery("SELECT * FROM CLIENTS");  
printWarnings( stmt.getWarnings() );  
printWarnings( rs.getWarnings() );  
...
```

- JDBC 1.0
  - *package supplémentaire (add-on) pour JDK 1.0*
  - *intégré à l'API de base (core API) du JDK 1.1*
- JDBC 2.0
  - *spécification par SUN en mai 1998*
    - *extensions pour « meilleure » gestion des résultats (`ResultSet` « scrollables », « modifiables »)*
    - *misés à jour groupées (batch updates)*
    - *support pour BLOBs (Binary Large Objects) et CLOBs (Character Large Objects)*
    - *...*
  - *intégré à l'API de Java 2 (JDK 1.2)*
  - *compatibilité avec la version 1.0*
    - *code écrit pour JDBC 1.0 compile et fonctionne avec version 2.0 de l'API*

- Par défaut lorsque l'on crée un Statement les objets ResultSet sont en lecture seule (read only) et à accès séquentiel (forward only)

```
public Statement createStatement() throws SQLException
```

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

- Avec JDBC 2.0 possibilité de créer des ResultSet

- « Scrollable »

- *plus de limitation à un parcours séquentiel*

- « Updatable »

- *possibilité de modifier les données dans la BD*

```
public Statement createStatement(int resultSetType, int resultSetConcurrency)
```

ResultSet.TYPE\_FORWARD\_ONLY  
ResultSet.TYPE\_SCROLL\_INSENSITIVE  
ResultSet.TYPE\_SCROLL\_SENSITIVE

le ResultSet est sensible aux  
modifications des valeurs dans  
la base de données

ResultSet.CONCUR\_READ\_ONLY  
ResultSet.CONCUR\_UPDATABLE

On peut modifier les données  
de la base via le ResultSet

```
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE ,  
                                         ResultSet.CONCUR_UPDATABLE);
```

- Méthodes de parcours

<code>first()</code>	Positionne sur la première ligne (1er enregistrement)
<code>last()</code>	Positionne sur la dernière ligne (dernier enregistrement)
<code>next()</code>	Passe à la ligne suivante
<code>previous()</code>	Passe à la ligne précédente
<code>beforeFirst()</code>	Positionne avant la première ligne
<code>afterLast()</code>	Positionne après la dernière ligne
<code>absolute(int)</code>	Positionne à une ligne donnée
<code>relative(int)</code>	Déplacement d'un nombre de lignes donné par rapport à ligne courante

- Méthodes de test de la position du curseur

<code>boolean isFirst()</code>	True si curseur positionné sur la première ligne
<code>boolean isBeforeFirst()</code>	True si curseur positionné avant la première ligne
<code>boolean isLast()</code>	True si curseur positionné sur la dernière ligne
<code>boolean isAfterLast()</code>	True si curseur positionné après la dernière ligne

- Modification du ResultSet
  - *Se placer sur le rang concerné*
  - Méthodes **updateXXX (...)**
  - Puis **updateRow ()**
    - *le faire avant de déplacer le curseur sur une autre ligne*

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,  
                                     ResultSet.TYPE_CONCUR_UPDATABLE);  
ResultSet rs = stmt.executeQuery("SELECT NOM,ID_CLIENT FROM CLIENTS);  
rs.first();  
rs.updateInt(2,151970);  
rs.updateRow();
```

- Insertion d'une ligne
  - `moveToInsertRow()`
  - Méthodes *Méthodes* `updateXXX(...)`
  - *Puis* `insertRow()`

```
ResultSet rs = stmt.executeQuery("SELECT NOM, ID_CLIENT FROM CLIENTS");  
rs.moveToInsertRow();  
rs.updateString(1, "Jacques OUILLE");  
rs.updateInt(2, 151970);  
rs.updateRow();
```

- Si aucune valeur n'est spécifiée pour une colonne n'acceptant pas la valeur `null`, une *SQLException* est lancée.
  - `moveToCurrentRow()` permet de se repositionner sur la ligne courante avant l'appel à `moveToInsertRow()`
- Suppression d'une ligne
  - Se placer sur la ligne
  - `deleteRow()`

```
rs.last();  
rs.deleteRow();
```

- Tous les `ResultSet` ne sont pas nécessairement modifiables
  - *En général la requête ne doit référencer qu'une seule table sans jointure*
- Tous les drivers JDBC ne supportent pas nécessairement et entièrement les `ResultSet` « scrollable » et « updatable »
  - *l'objet `DataBaseMetaData` fournit de l'information quant au support proposé pour les `ResultSet`*
  - *Il faut être prudent si le logiciel que l'on écrit doit interagir avec une grande variété de drivers JDBC*

- Fonctionnalités ajoutées l'interface **Statement** pour permettre de regrouper des traitements qui seront envoyés en une seule fois au SGB
  - → *amélioration des performances si nombre de traitements important*
  - *Pas obligatoirement supportées par le pilote*
    - méthode **supportsBatchUpdates ()** de **DatabaseMetaData**

**void addBatch(String)**

Ajouter au "lot" une chaîne contenant une requête SQL.  
Requête de type INSERT, UPDATE, DELETE  
ou DDL (CREATE TABLE, DROP TABLE)

**int[] executeBatch()**

Exécute toutes les requêtes du lot.  
Renvoie un tableau d'entiers qui pour chaque requête contient soit :

- le nombre de mises à jour effectuées (entier >= 0)
- SUCCESS\_NO\_INFO si la commande a été exécutée mais on ne connaît pas le nombre de rang affectés
- EXECUTE\_FAILED si la commande a échoué

En cas d'échec sur l'une des requêtes une **BatchUpdateException** est lancée. Selon les pilotes les requêtes qui suivent dans le lot peuvent être ou ne pas être exécutées.

**void clearBatch()**

Supprime toutes les requêtes stockées



- Exemples (The JDBC Tutorial: Chapter 3 - Advanced Tutorial - Maydene Fisher  
<http://java.sun.com/developer/Books/JDBCTutorial/index.html>)

Batch update statique

```
con.setAutoCommit(false);

Statement stmt = con.createStatement();

stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Hazelnut', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Amaretto_decaf', 49,
                    10.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Hazelnut_decaf', 49,
                    10.99, 0, 0)");

int [] updateCounts = stmt.executeBatch();

con.commit();
con.setAutoCommit(true);
```

Batch update paramétré

```
con.setAutoCommit(false);
PreparedStatement pstmt = con.prepareStatement(
    "INSERT INTO COFFEES VALUES(
        ?, ?, ?, ?, ?)");

pstmt.setString(1, "Amaretto");
pstmt.setInt(2, 49);
pstmt.setFloat(3, 9.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

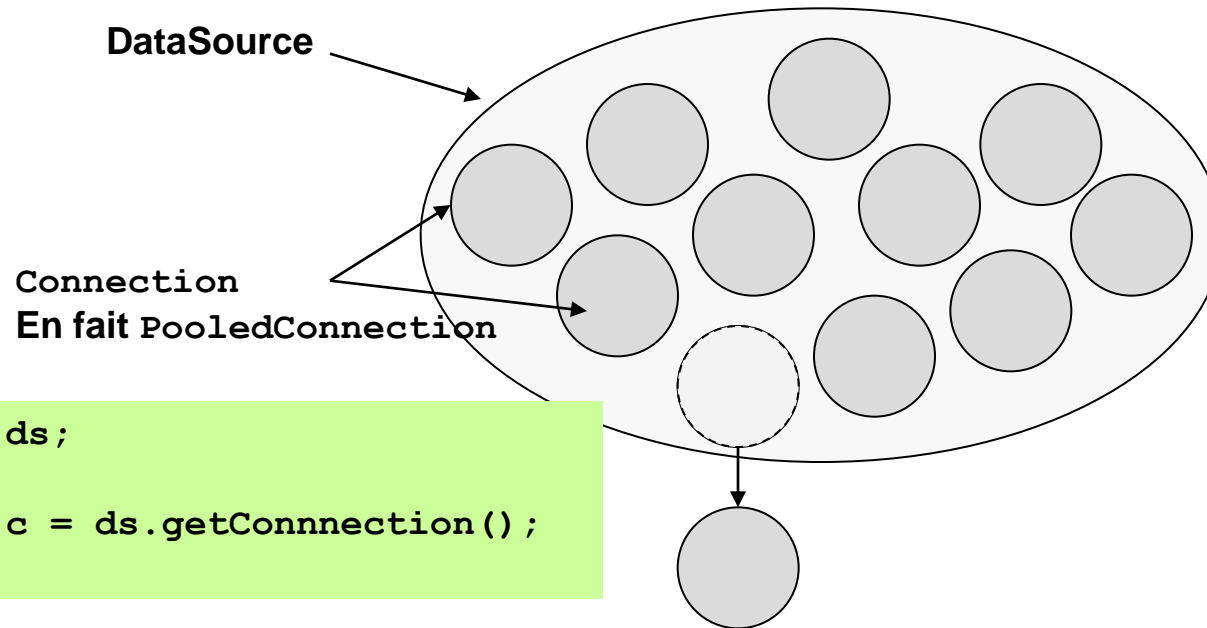
pstmt.setString(1, "Hazelnut");
pstmt.setInt(2, 49);
pstmt.setFloat(3, 9.99);
pstmt.setInt(4, 0);
pstmt.setInt(5, 0);
pstmt.addBatch();

// ... and so on for each new type of coffee

int [] updateCounts = pstmt.executeBatch();
con.commit();
```

- **javax.sql** package d'extension standard de JDBC
  - *Pour les applications JEE (Java Enterprise Edition)*
  - *Inclus en standard dans JSE (Java Standard Edition) depuis version 1.4*
- **DataSource** : Obtention du nom de la base à partir de serveurs de noms plutôt que d'avoir le nom de la base de données codé « en dur » dans l'application.
  - *Utilisation de JNDI (Java Naming and Directory Interface) pour connexion à une base de donnée*
- **PooledConnection** : support pour gestion d'un « pool » de connexion
  - *gestion d'un cache des connexion ouvertes*
  - *évite la création de nouvelles connexions (ce qui est coûteux)*
- **RowSet** : permet de traiter les résultats des requêtes comme des composants JavaBeans
- *Support pour les transactions distribuées*

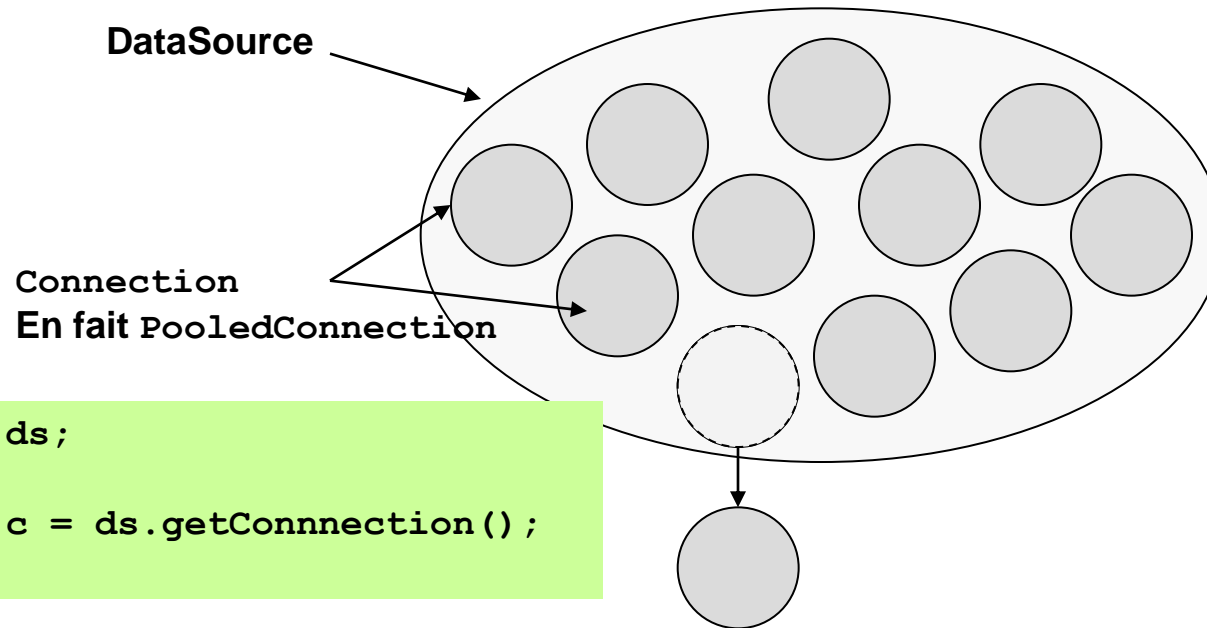
- L'objet **DataSource** avec pooling de connexions maintient un ensemble de connexions à la BD prêtes à l'emploi (**pool ou cache de connexions**).



```
DataSource ds;  
...  
Connection c = ds.getConnection();  
...
```

- Quand un code a besoin d'accéder à la base de données, il obtient une connexion en s'adressant à la **DataSource**

- L'objet **DataSource** avec pooling de connexions maintient un ensemble de connexions à la BD prêtes à l'emploi (**pool ou cache de connexions**).

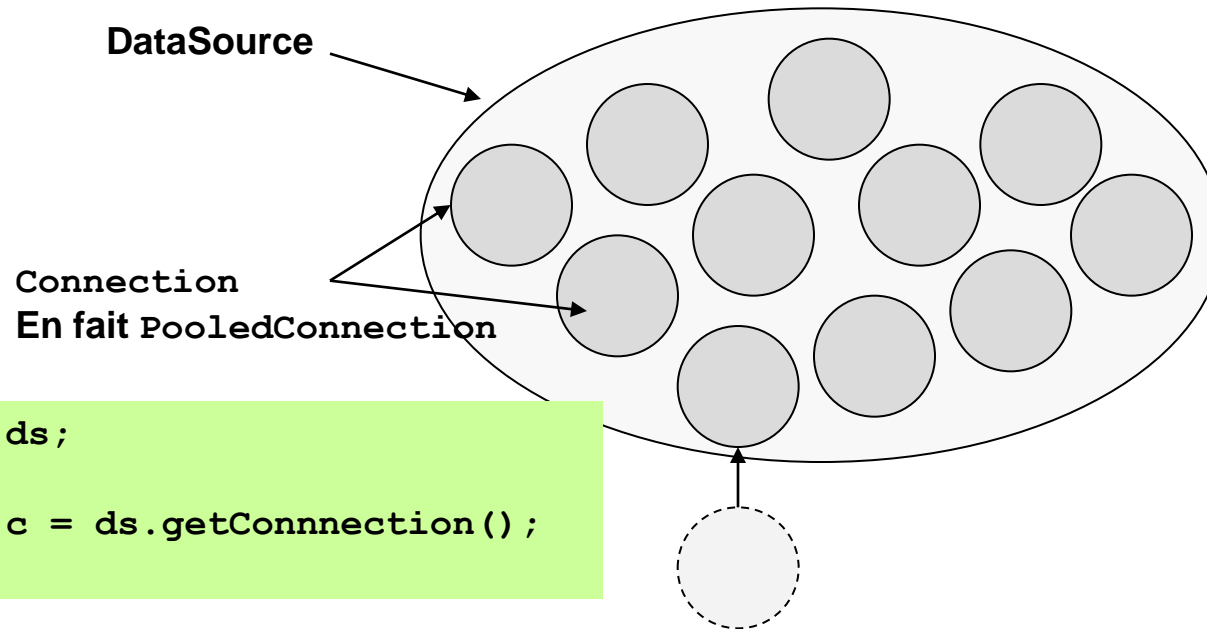


```
DataSource ds;  
...  
Connection c = ds.getConnection();  
...
```

```
c = c.close();
```

- Quand un code a besoin d'accéder à la base de données, il obtient une connexion en s'adressant à la **DataSource**.
- Il conserve la connexion jusqu'à sa fermeture explicite.

- L'objet **DataSource** avec pooling de connexions maintient un ensemble de connexions à la BD prêtes à l'emploi (**pool ou cache de connexions**).



```
DataSource ds;  
...  
Connection c = ds.getConnection();  
...
```

```
c = c.close();
```

- Quand un code a besoin d'accéder à la base de données, il obtient une connexion en s'adressant à la **DataSource**.
- Il conserve la connexion jusqu'à sa fermeture explicite.
- Lorsqu'une connexion est "fermée", elle est en fait relâchée et remise dans le pool.

- JDBC API de bas niveau
  - *Parfois difficile à pendre en main*
  - *Demande des connaissances "pointues" en BD*
  - *Dépendances par rapport au SGBD cible*
- Nombreuses API construites au dessus de JDBC
  - *Jakarta Commons DbUtils (simplifie utilisation de JDBC)*  
<http://jakarta.apache.org/commons/dbutils/>
  - *Frameworks de persistance ou de mapping O/R*
    - *Ibatis, <http://ibatis.apache.org/>*
    - *Hibernate, <http://www.hibernate.org>*
    - *JPA – Java Persistence APIE, JEE5*
      - *modèle de persistance EJB (Entreprise Java Beans) 3.0 issu d'Hibernate*
    - *JDO – Java Data Objects*