



Java DataBase Connectivity

dernière modification : 10/02/2025 16:53

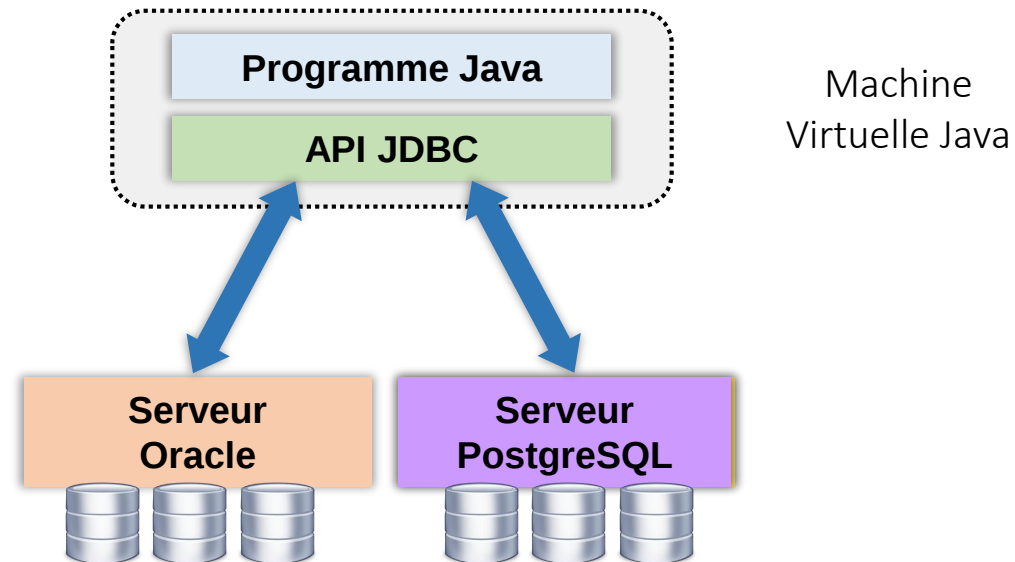
Philippe Genoud



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

- JDBC Java Data Base Connectivity

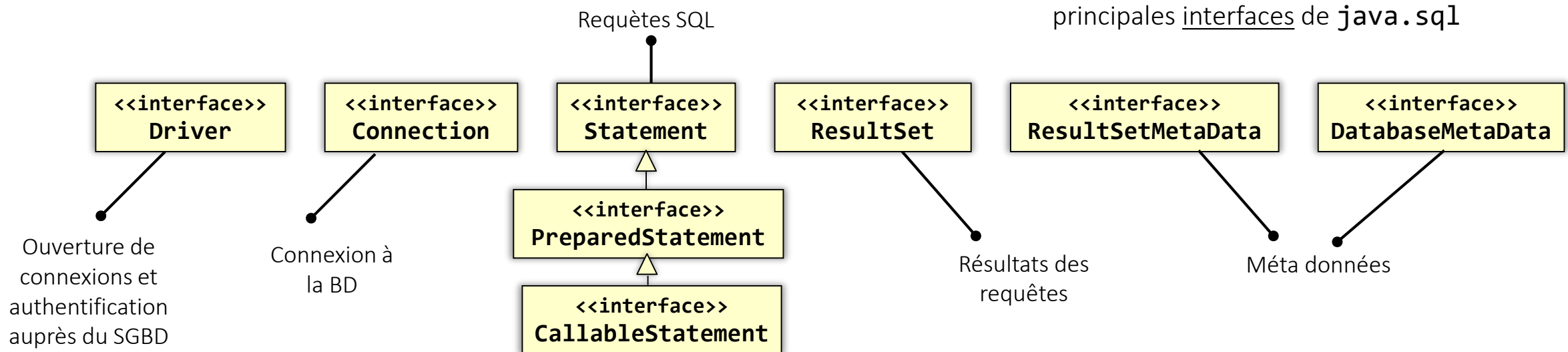
- API java standard (fait partie de Java SE) qui permet un accès homogène à des bases de données depuis un programme Java au travers du langage SQL.



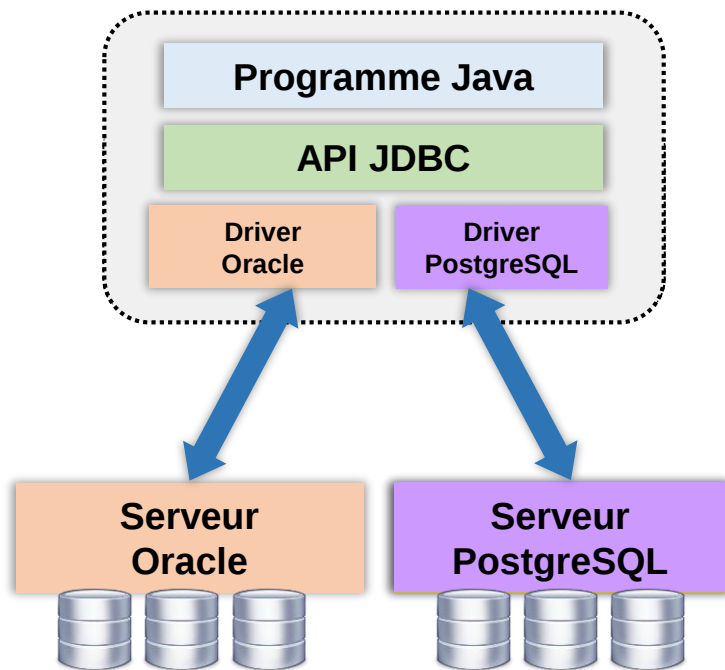
- L'API JDBC est **indépendante** des SGBD.
 - Un changement de SGBD ne doit pas impacter le code applicatif.

<https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>

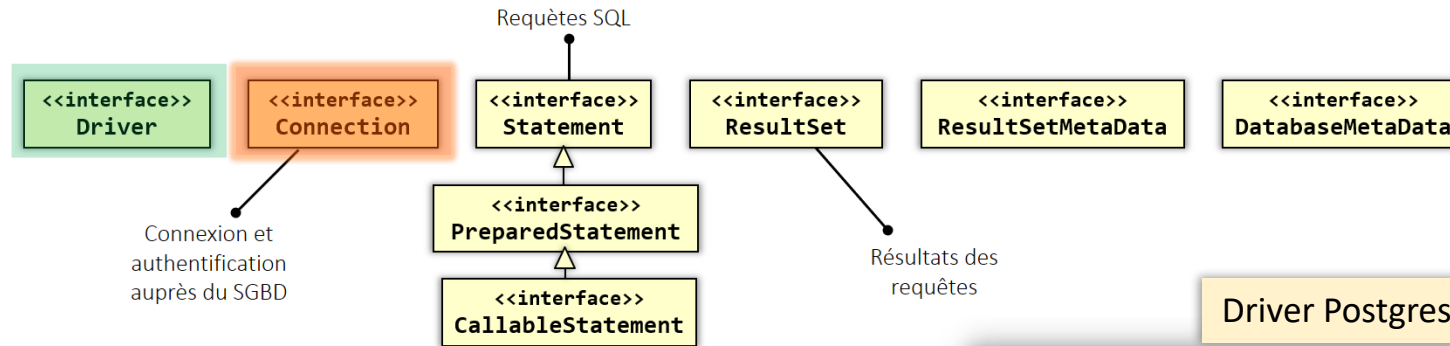
- L'API JDBC est basée sur un ensemble **d'interfaces** (package `java.sql`) qui définissent un protocole de communication entre le programme java client et le serveur de base de données pour
 - ouverture/fermeture de connexions à une base de données
 - exécution de requêtes SQL
 - exploitation des résultats
 - *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, SQLite...)
- 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

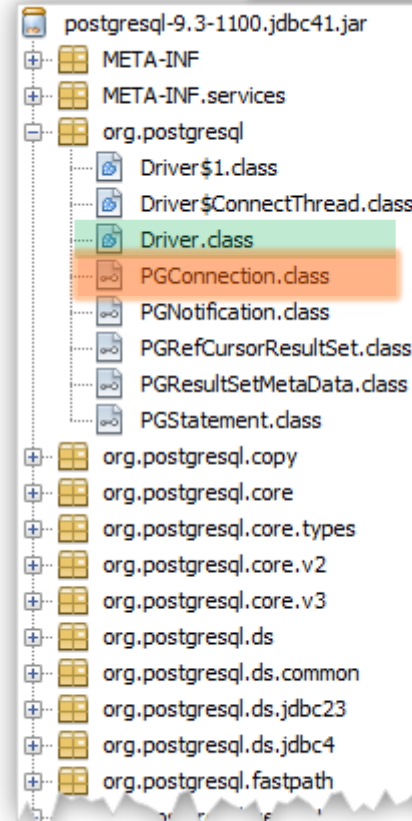


Les interfaces définissent une **abstraction** du pilote (driver) de la base de données. Chaque fournisseur propose **sa propre implémentation** de ces interfaces.

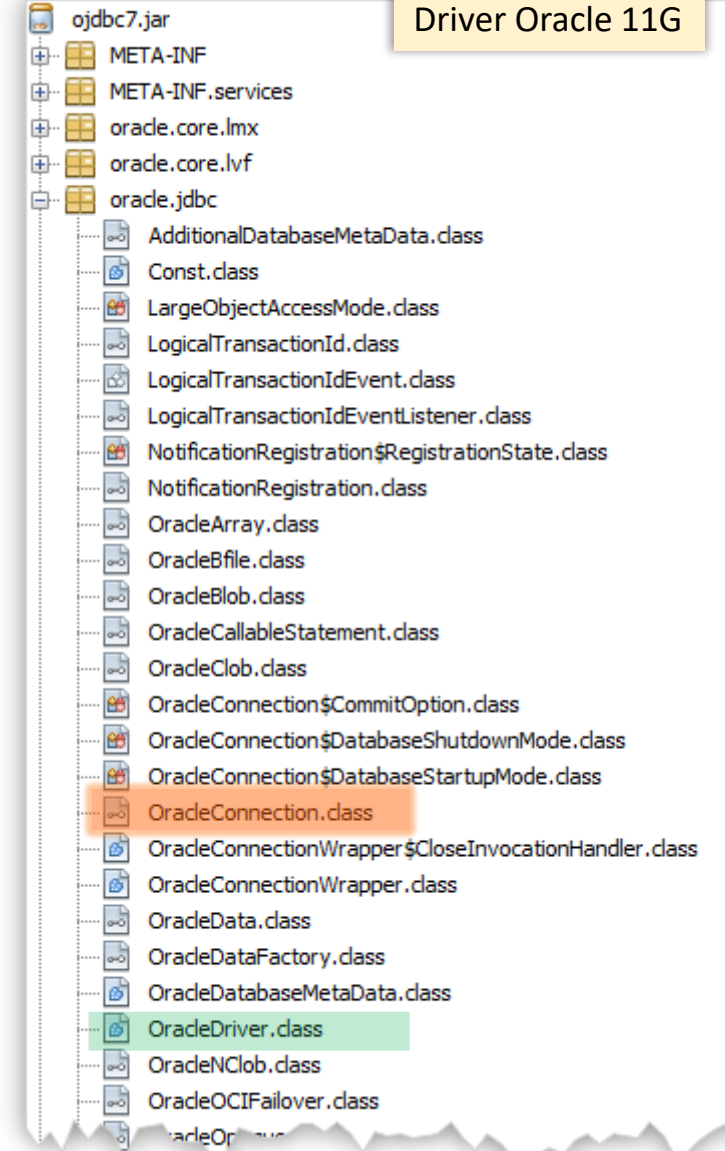
Les classes d'implémentation du driver jdbc sont dans une archive (fichier **.jar**) qu'il faut intégrer (sans la décompresser) au niveau du **classpath** de l'application au moment de l'exécution.

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

Driver Postgres 9.2

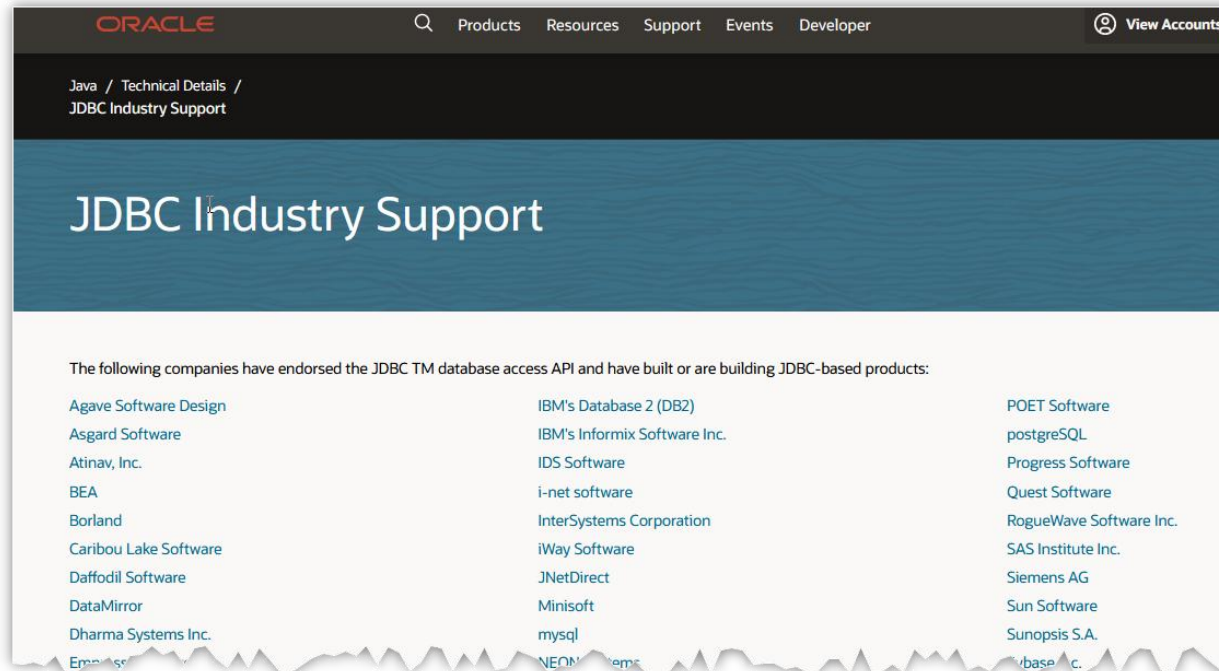


Driver Oracle 11G



- Liste des drivers disponibles à :

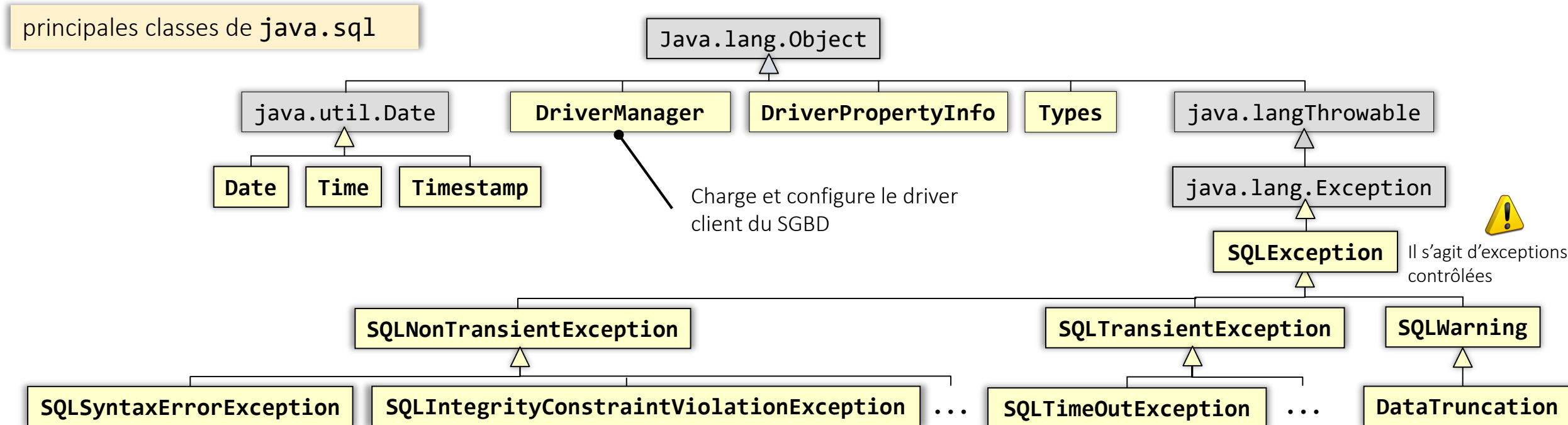
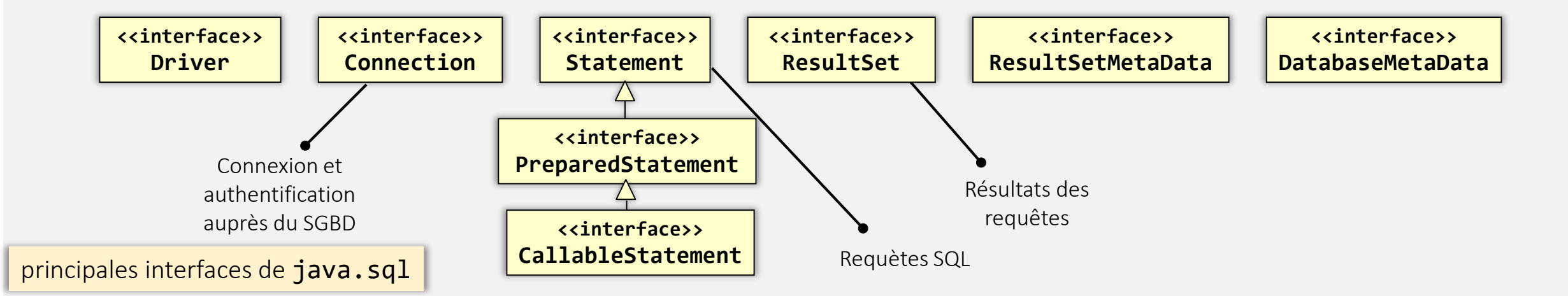
<https://www.oracle.com/java/technologies/industry-support.html>



Capture du 19/02/2024
Pas nécessairement à jour

- sur les sites des fournisseurs de BD
 - SQLite : <https://bitbucket.org/xerial/sqlite-jdbc/downloads/>
 - Oracle : <https://www.oracle.com/database/technologies/appdev/jdbc-downloads.html>
 - Postgres : <https://jdbc.postgresql.org>
 - ...

- 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*
 - JDBC 4.1 (JDK 1.7)
 - *Try avec ressources pour fermeture des ressources (Connexions, requêtes, résultats), RowSet 1.1*
 - JDBC 4.2 (JDK 1.8)
 - *nouveaux support pour les types, changements dans les interfaces existantes, RowSet 1.2*



□ Fonctionnement général de JDBC

1) Etablir une connexion à la base de données

DriverManager permet de créer des objets **Connection**
(objets instance d'une classe qui implémente l'interface **Connection**)

```
Connection c = DriverManager.getConnection(...);
```

2) Créer un objet de transport qui permettra d'envoyer des requêtes SQL à la BD

l'objet **Connection** permet de créer des objets **Statement**
(objets instance d'une classe qui implémente l'interface **Statement**)

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

3) Exécuter la requête sur la BD et récupérer ses résultats

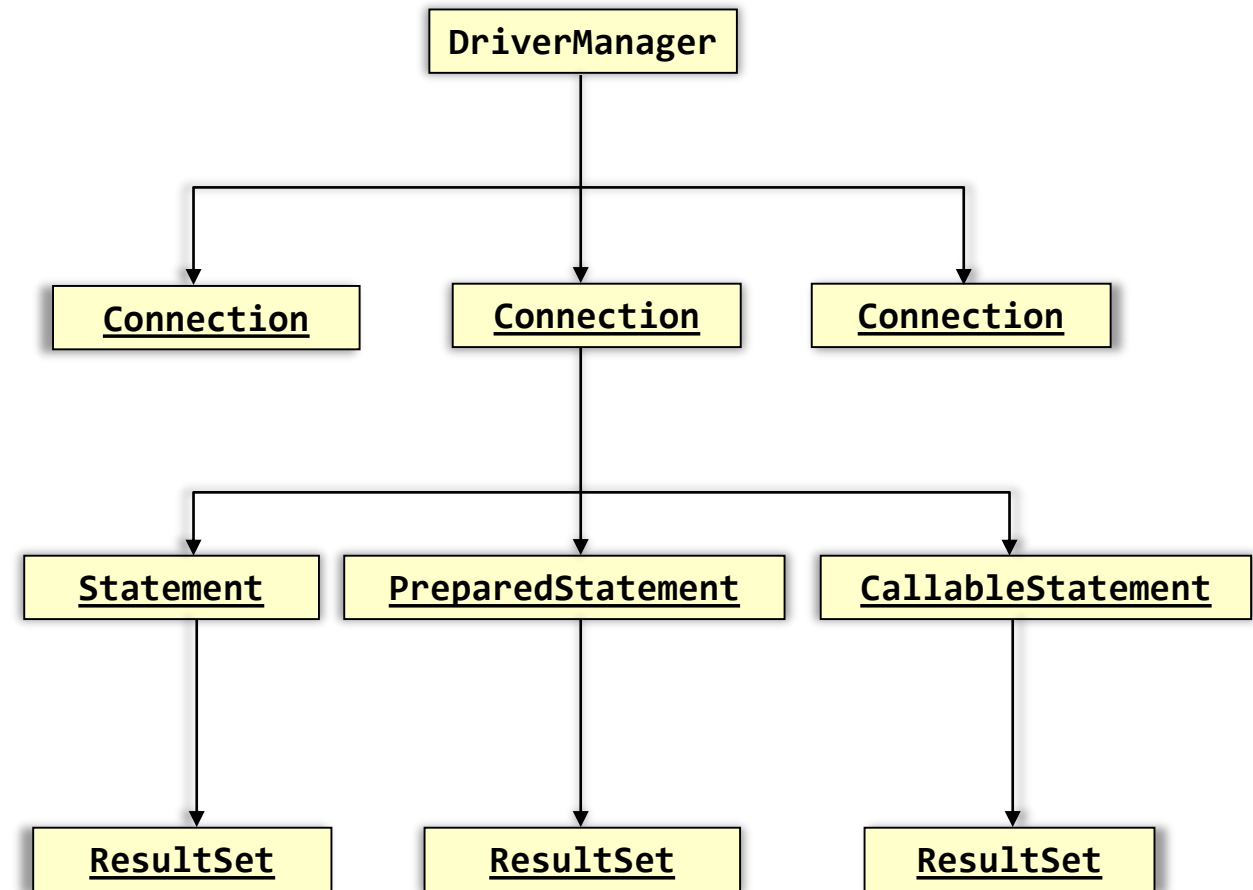
l'objet **Statement** permet d'envoyer requêtes SQL et de créer des objets **ResultSet** encapsulant le résultat des requêtes

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

4) Exploiter les résultats de la requête

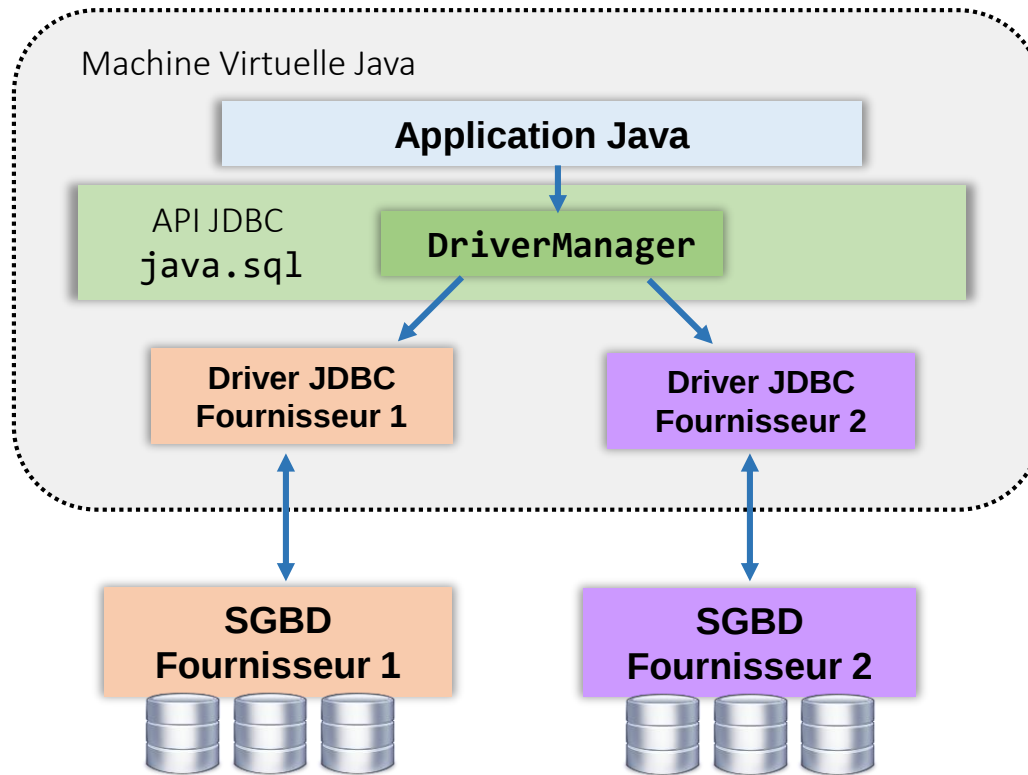
l'objet **ResultSet** permet de parcourir le résultat de la requête et de récupérer les données

```
while (rs.next()) {  
    String nom = rs.getString("NOM");  
    ...  
}
```



- A la compilation on n'utilise que les types définis dans l'API JDBC (`java.sql`)
- A l'exécution les objets sont instanciés à des classes d'implémentation fournies par le pilote (driver) JDBC

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



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

- 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("org.sqlite.JDBC");  
    Class.forName("oracle.jdbc.OracleDriver").newInstance();  
}  
catch (ClassNotFoundException e) {  
    ...  
}
```

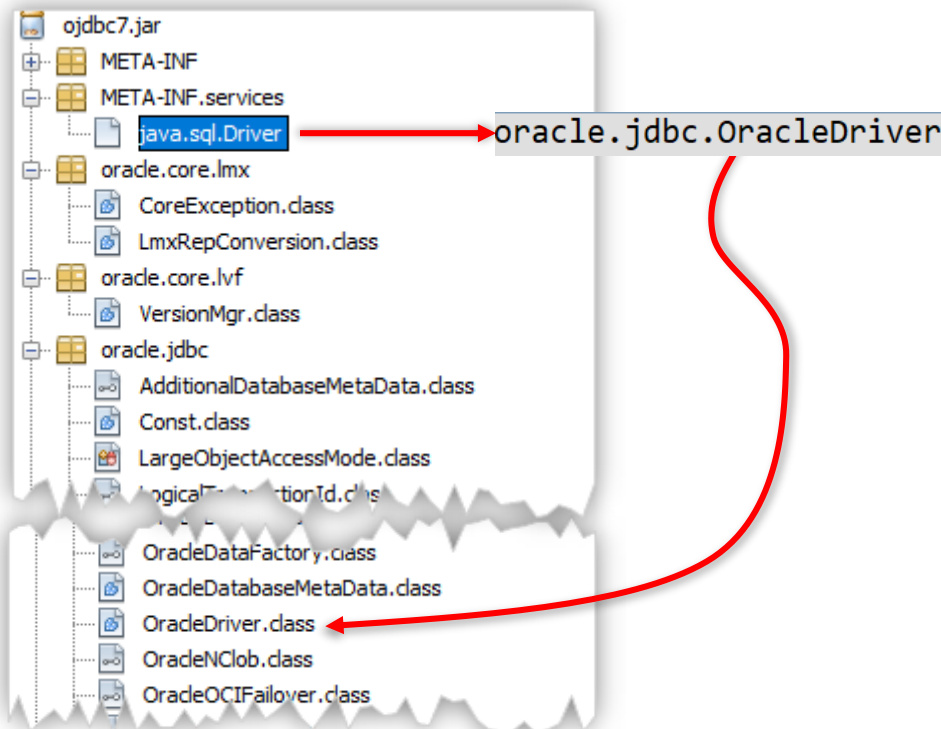
Chargement d'un pilote SQLite

Chargement d'un pilote Oracle

- Permet de changer 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

fichier jar du pilote JDBC de Oracle



- Utilisation du mécanisme Java (Java6+) pour les fournisseurs de services
 - service défini par une interface par exemple pour un pilote JDBC : `java.sql.Driver`
 - si un fichier `.jar` contient un code d'implémentation du service, son répertoire **META-INF/services** contient un fichier de nom identique au service et dont le contenu est le nom de la classe d'implémentation du service
- Le **DriverManager** parcourt tous les `.jar` présents dans le **classpath** et peut charger automatiquement (en utilisant les services de la classe `java.util.ServiceLoader`) tous les drivers compatibles JDBC 4.0+ .

- 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.

Base Oracle avec Driver Oracle Thin (100% java)

jdbc:**oracle**:**thin**:**@serveur:port:base**
nom IP du serveur numéro de port socket à utiliser nom de la base

Ex: base Oracle sur le serveur de l'UFR-IM²AG

```
Connection conn = DriverManager.getConnection(  
    "jdbc:oracle:thin:@im2ag-oracle.e.ujf-grenoble.fr:1521:im2ag",  
    user, password  
);
```

Base locale SQLite

jdbc:**sqlite**:**data_base_file_path**
chemin relatif ou absolu au fichier .db

Ex: base SQLite **test.db** dans le répertoire **C:/sqlite/db**

```
Connection conn = DriverManager.getConnection(  
    "jdbc:sqlite:C:/sqlite/db/test.db",  
    user, password  
);
```

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

Paramètres de connexion définis dans un fichier propriétés (Properties file) : fichier texte de couples **nomprop=valeur**

```
# fichier de propriétés pour une connexion à la base Oracle 11G de l'ufv
# im2ag s'exécutant sur le serveur im2ag-oracle.e.ujf-grenoble.fr
jdbcDriver=oracle.jdbc.OracleDriver
dataBaseUrl=jdbc:oracle:thin:@im2ag-oracle.e.ujf-grenoble.fr:1521:im2ag
userName=genoudph
passwd=XXXXXXXXXX
```

bdUFRIM2AG.properties

```
# fichier de propriétés pour une connexion à une base postgres locale
# (c.a.d. sur la même machine que, celle qui exécute le programme java)
jdbcDriver=org.postgresql.Driver
dataBaseUrl=jdbc:postgresql://localhost:5433/cafe
userName=genoudph
passwd=XXXXXXXXXX
```

bdPostgres.properties

Le fichier de propriétés est passé comme argument du programme au lancement de l'exécution

```
C:> java -classpath .;lib/ojdbc7.jar MonAppliJDBC bdUFRIM2AG.properties
```



Il faut que les classes du driver soient dans le **classpath** à l'exécution

```
Properties options = new Properties();
options.load(new FileInputStream(new File(args[0])));
String driverClass = options.getProperty("jdbcDriver");
String dbURL = options.getProperty("dataBaseUrl");
String userId = options.getProperty("userName");
String passwd = options.getProperty("passwd");

...
Connection conn = ouvrirConnexion(driverClass, dbURL, userId, passwd);

... // utilisation de la connexion pour accéder à la base
conn.close();
```

Chargement des propriétés

Si la classe du driver n'est pas trouvée

Si la connexion ne peut être obtenue

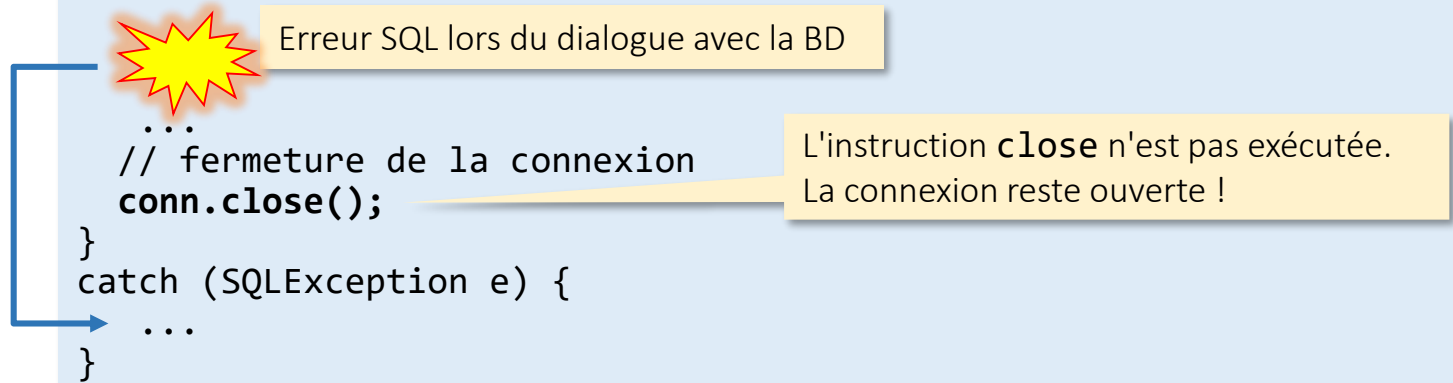
```
public static Connection ouvrirConnexion(String driverClass, String dbURL,
String userId, String passwd) throws ClassNotFoundException, SQLException

Class.forName(driverClass);
return DriverManager.getConnection(dbURL, userId, passwd);
}
```

inutile avec Pilote JDBC4.0+. Le chargement des pilotes est automatique

- 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 connexion 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

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



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(url, 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` car sinon message d'erreur : `conn` peut ne pas avoir été initialisée

`close` peut provoquer une `SQLException`

- 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(url, 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();  
    }  
}
```

`close` peut provoquer une `SQLException`

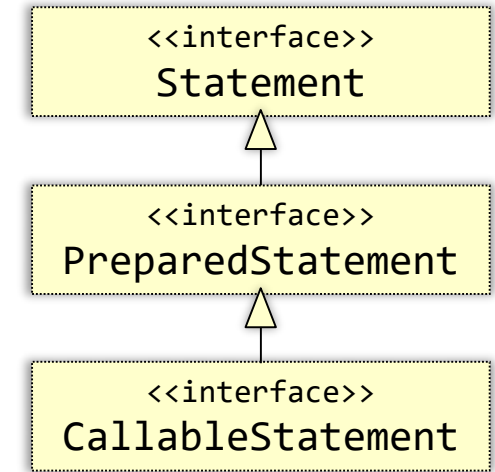
Le compilateur impose d'initialiser `conn` car sinon message d'erreur : `conn` peut ne pas avoir été initialisée

utilisation d'un **try** avec ressources (Java7+) au lieu de la clause **finally**

```
try (Connection conn =  
        DriverManager.getConnection(url, 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 et la **sécurité**
 - **CallableStatement** : encapsule procédures SQL stockées dans le SGBD
- 3 formes (méthodes) d'exécutions selon le type de requête :
 - requêtes de consultation (SELECT)
 - *ResultSet executeQuery(String sql)* → un objet *ResultSet* pour accéder à la table résultat
 - requêtes de mise à jour (INSERT, UPDATE, DELETE, CREATE TABLE et DROP TABLE)
 - *int executeUpdate(String sql)* → un entier (ex nombre d'enregistrements affectés par la requête)
 - requêtes ouvertes (on ne connaît pas a priori la nature de la requête)
 - *boolean execute(String sql)* → *true* si requête de type *SELECT*, *false* si requête de mise à jour



- Création d'un *Statement* :

```
Connection conn = DriverManager.getConnection(...);  
...  
Statement stmt = conn.createStatement() ;
```

référence vers un objet **Statement** qui utilisera cette connexion

référence vers un objet **Connection** vers la BD cible

- Exécution de la requête :

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

la requête est une chaîne SQL qui doit être valide

⚠ espaces nécessaires si concaténation de chaînes

```
ResultSet rs = stmt.executeQuery(myQuery) ;
```

référence vers un objet **ResultSet** qui permettra d'exploiter les résultats

appel de méthode **executeQuery** car requête de type **SELECT**

- Création d'un *Statement* :

```
Connection conn = DriverManager.getConnection(...);  
...  
Statement stmt = conn.createStatement() ;
```

référence vers un objet **Statement** qui utilisera cette connexion

référence vers un objet **Connection** vers la BD cible

- Exécution de la requête :

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

la requête est une chaîne SQL qui doit être valide

⚠ espaces nécessaires si concaténation de chaînes

```
ResultSet rs = stmt.executeQuery(myQuery) ;
```

référence vers un objet **ResultSet** qui permettra d'exploiter les résultats

appel de méthode **executeQuery** car requête de type **SELECT**

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

Chaînes multilignes avec Java15+ délimitées par ""

- `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
 - Curseur de lecture placé avant la première ligne à la création du `ResultSet`
 - `boolean next()` permet d'avancer à la ligne suivante, $\rightarrow$ `false` si pas de ligne suivante
 - Il faut appeler une fois `next()` avant d'accéder à la première ligne de résultat
 - Parcours complet du résultat

```
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 colA, colB, colC FROM Table1");

while (rs.next()) {
    int entier = rs.getInt("colA");           // rs.getInt(1);
    String chaine = rs.getString("colB");     // rs.getString(2);
    byte bytes[] = rs.getBytes("colC");      // rs.getBytes(3);
    System.out.println("ROW = " + entier + " " + chaine + " " + bytes[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
TIMESTAMP	getTimestamp	java.sql.Timestamp

getString() 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 exemple **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.next()
...
... 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**

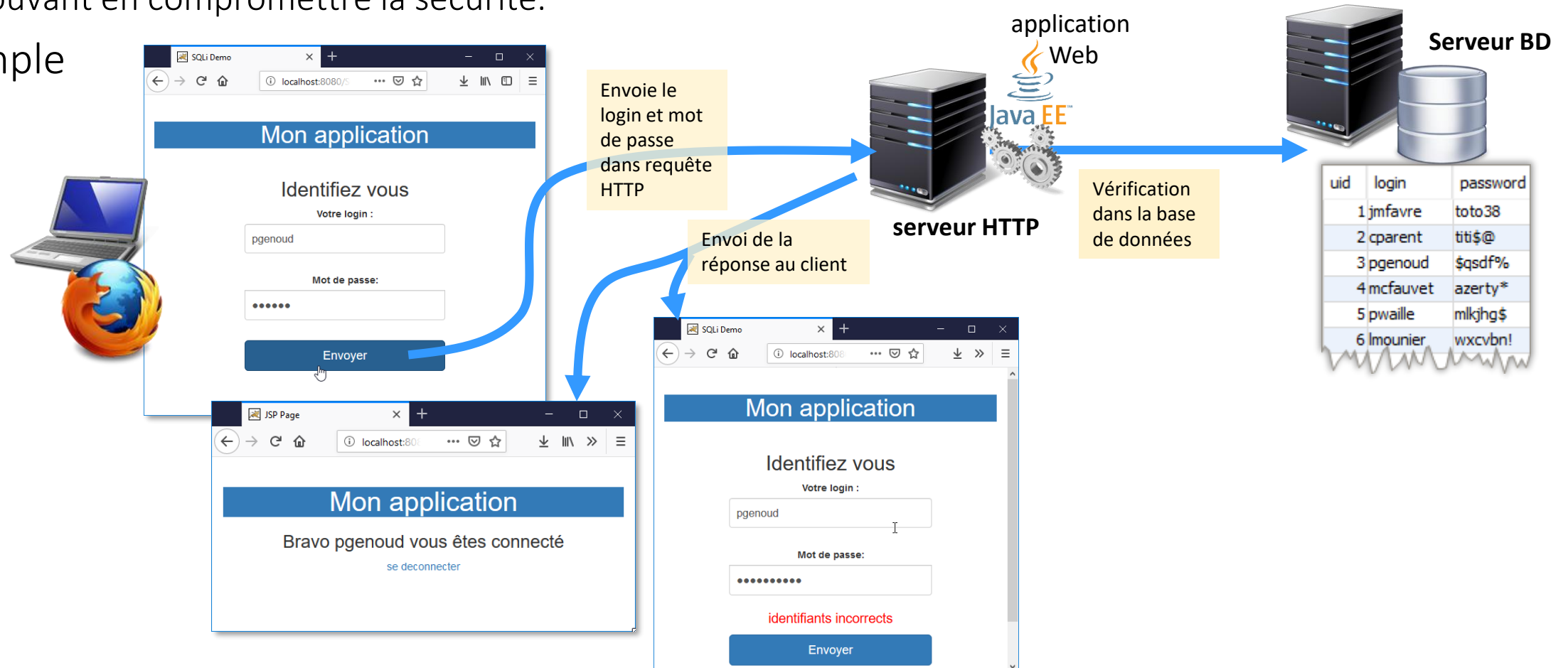
- Renvoie **true** si on vient de lire une valeur **NULL**, **false** sinon

```
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
...

- injection SQL:
 - méthodes exploitant des failles de sécurité d'une application interagissant avec une base de données
 - principe : insérer dans la requête SQL en cours un morceau de requête non prévu par le système et pouvant en compromettre la sécurité.

- exemple



- mise en œuvre avec JDBC

uid	login	password
1	jmfavre	toto38
2	cparent	titi\$@
3	pgenoud	\$qsdf%
4	mcfauvet	azerty*
5	pwaille	mlkjhg\$
6	lmounier	wxcvbn!



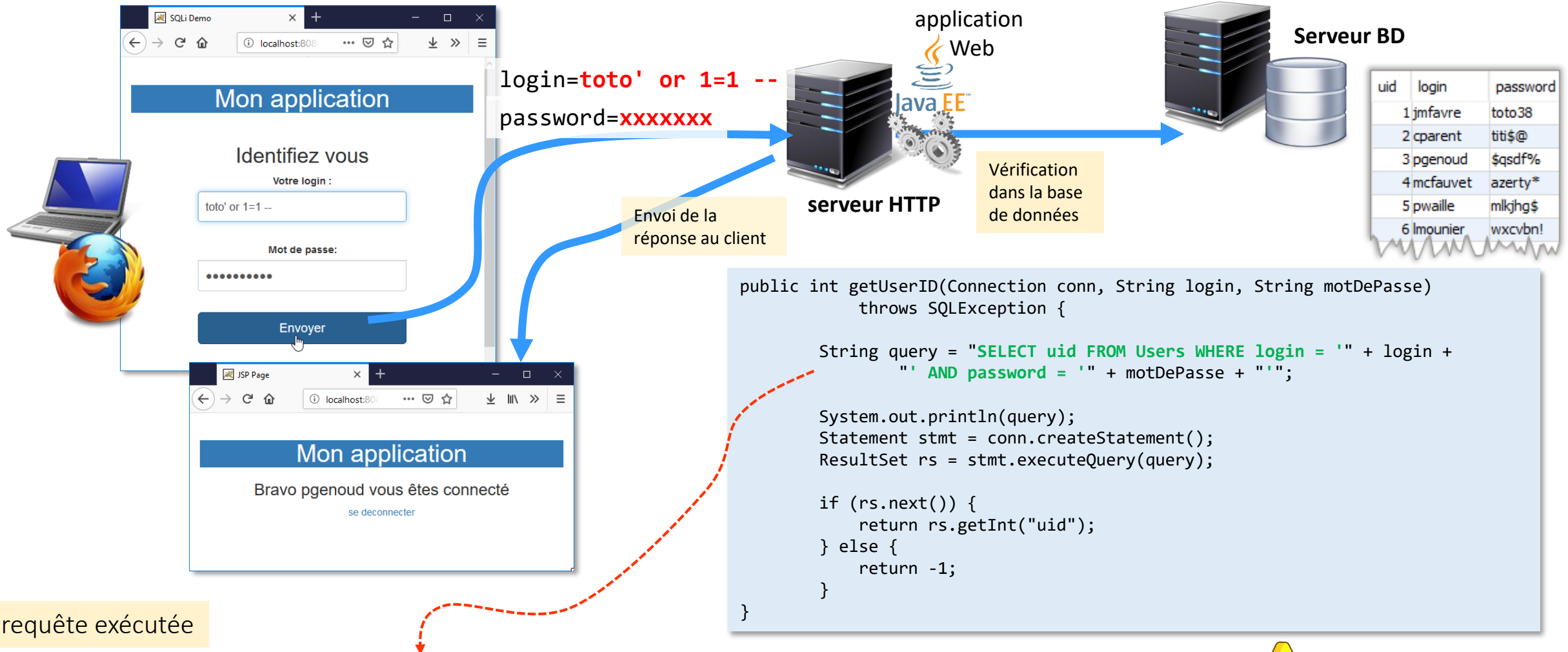
Possibilité de se connecter sans connaître le mot de passe d'un utilisateur

```
/**
 * @param conn la connexion JDBC
 * @param login l'identifiant de l'utilisateur
 * @param motDePasse le mot de passe propose
 *
 * @return l'identifiant interne (uid) si le mot de passe est correct
 *         -1 si le mot de passe est incorrect
 *
 * @throws SQLException si pb avec la BD
 */
public int getUserID(Connection conn, String login, String motDePasse)
    throws SQLException {

    String query = "SELECT uid FROM Users WHERE login = '" + login +
        "' AND password = '" + motDePasse + "'";

    System.out.println(query);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(query);

    if (rs.next()) {
        return rs.getInt("uid");
    } else {
        return -1;
    }
}
```



! ne pas utiliser la concaténation de chaînes pour construire des requêtes

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

- paramètres formels spécifiés à l'aide de ?

```
PreparedStatement ps = conn.prepareStatement("SELECT POPULATION FROM COMMUNES  
WHERE NAME = ? AND ANNEE = ? ");
```

- Passage des paramètres effectifs

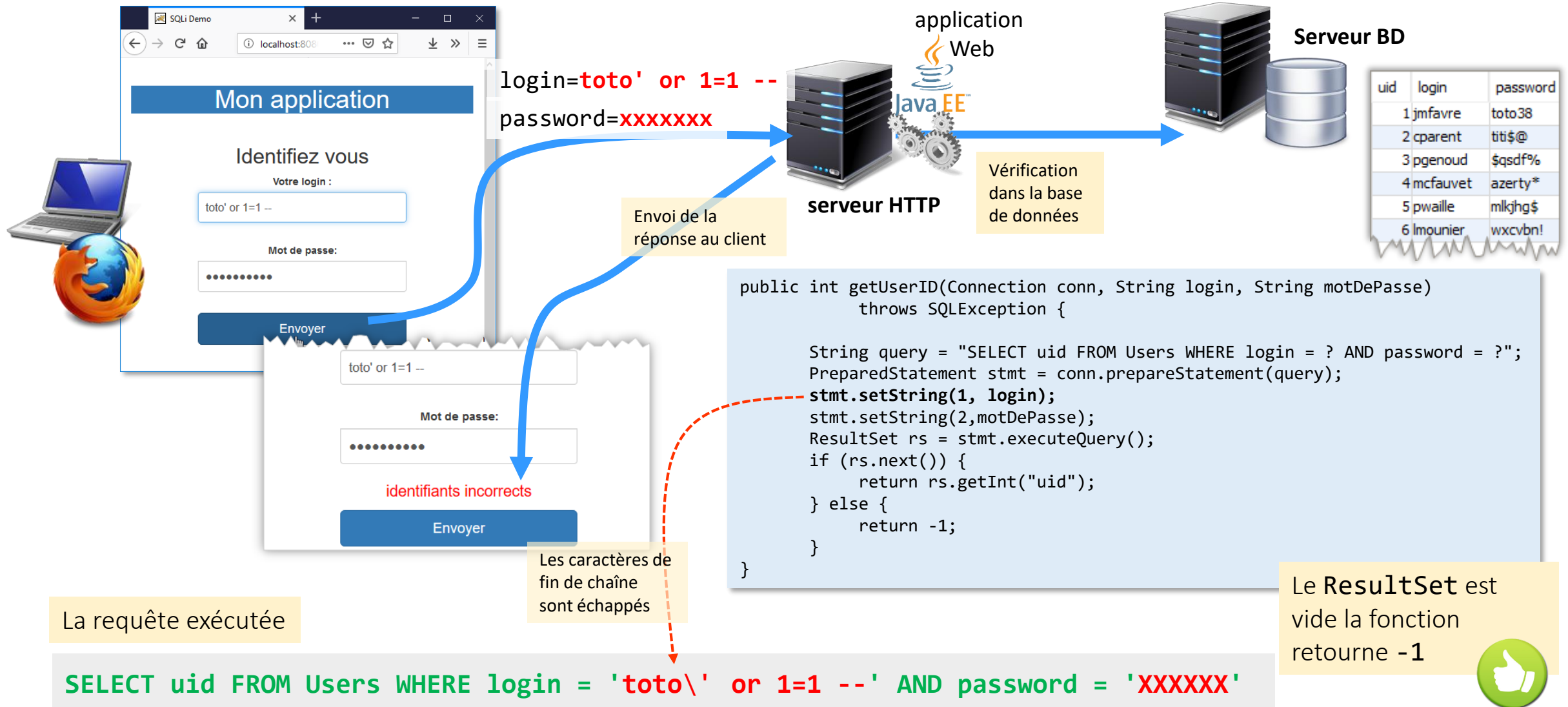
- à l'aide de méthodes au format **setXXX(indice, valeur)** où XXX représente le type du paramètre

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

- Invocation par **executeQuery()** (sans paramètres)

```
String[] names = ...  
  
for (int annee=2010; i < 2019; i++) {  
    ps.setInt(2, annee);  
    ResultSet rs = ps.executeQuery();  
    // ... Exploitation des résultats  
}
```

phase identique à celle utilisée pour SQL statique



- 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 maProcedure(?,?)}");
```

Appel sans valeur de retour et avec paramètres

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

❑ 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

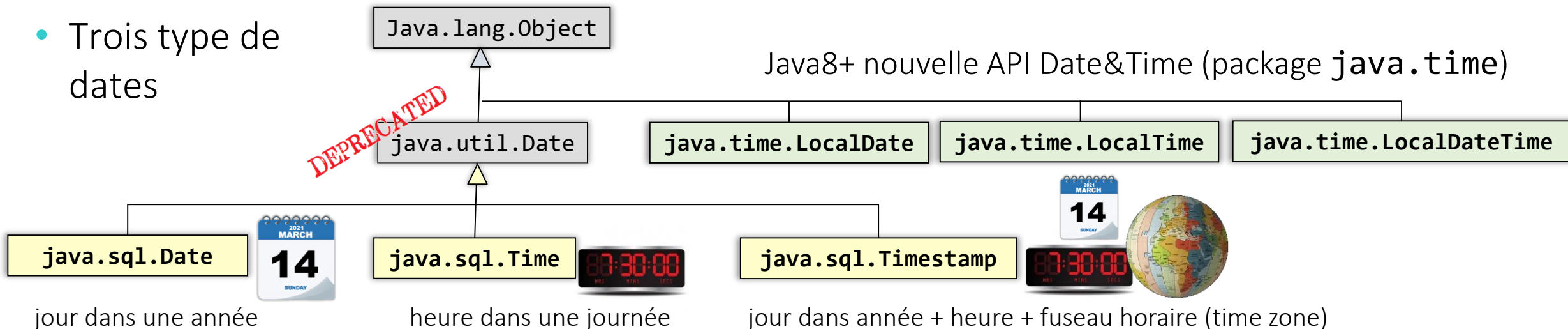
```
ResultSet rs = proc.executeQuery();
```

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

❑ Récupération des paramètres OUT

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

- Trois type de dates



```
ResultSet rs = ...;
```

```
Date d = rs.getDate("...");
Time t = rs.getTime("...");
Timestamp ts = rs.getTimestamp("...");
```

```
PreparedStatement pstmt = ...;
```

```
pstmt.setDate(index1,aDate);
pstmt.setTime(index2,aTime);
pstmt.setTimestamp(index3,aTimeStamp);
```

DEPRECATED

Objets peu pratiques à manipuler,
en particulier à construire

Java8+ nouvelle API Date&Time (package `java.time`)

Méthodes statiques de construction (Factory methods) d'objets `LocalDate`, `LocalTime`, `LocalDateTime`

```

LocalDate ld = LocalDate.of(2021, Month.MARCH, 14);
LocalTime lt = LocalTime.of(7, 30, 0);
LocalDateTime ldt = LocalDateTime.of(2021, Month.MARCH, 14, 7, 30, 0);
  
```

Conversion de `java.time` vers `java.sql`

```

Date d = Date.valueOf(ld);
Time t = Time.valueOf(lt);
Timestamp ts = Timestamp.valueOf(ldt);
  
```

Conversion de `java.sql` vers `java.time`

```

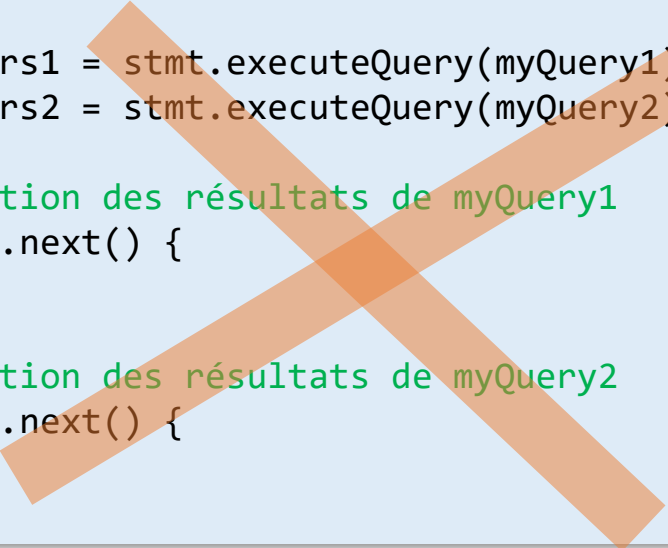
LocalDate ld = d.toLocalDate();
LocalTime lt = t.toLocalTime();
LocalDateTime ldt = ts.toLocalDateTime();
  
```

- 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 **même** 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();

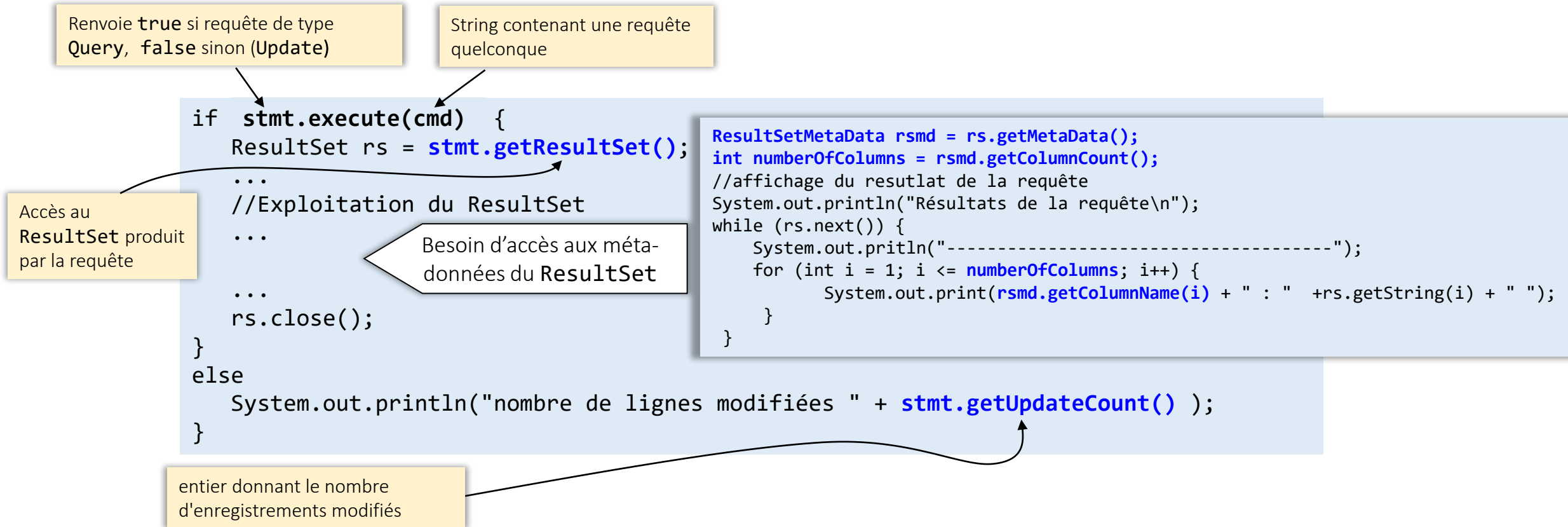
ResultSet rs1 = stmt.executeQuery(myQuery1);
//exploitation des résultats de myQuery1
while (rs1.next()) {
    ...
}

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**.

- Permet de découvrir dynamiquement (au moment de l'exécution) des propriétés concernant la base de données ou les résultats de requêtes
- La méthode **getMetaData()** du type **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()** du type **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)*
 - ...

- Exemple : lors de l'exécution d'une requête par `execute()`.



- 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

Atomicité

on ne peut pas diviser une transaction : toutes les opérations ont eu lieu, ou aucune n'a eu lieu

Cohérence

toutes les contraintes d'intégrité de la BD (clés primaires, clés étrangères...) doivent être vérifiées quand on sort de la transaction

Isolation

assure que l'exécution simultanée de transactions produit le même état que celui qui serait obtenu par l'exécution en série des transactions (Toute transaction doit s'exécuter comme si elle était la seule sur le système).

quand et comment les modifications effectuées au cours d'une transaction sont vues par les autres utilisateurs de la BD

Durabilité

assure que lorsqu'une transaction a été confirmée, les résultats sont enregistré de façon permanente dans la BD (même en cas de panne)

- L'interface **Connection** offre des services de gestion des transactions.
- Deux modes possibles :
 - **auto-commit** :
 - *chaque mise à jour de la base de données est immédiatement rendue permanente après l'exécution de l'instruction SQL.*
 - *activé par défaut par les pilotes JDBC pour chaque nouvelle connexion à une base de données*
 - **gestion manuelle de transactions** :
 - *lorsque le mode auto-commit est désactivé le programmeur doit gérer lui-même les transactions à l'aide de méthodes de l'objet **java.sql.Connection***
 - **setAutoCommit(boolean autoCommit)** permet d'activer/désactiver le mode auto-commit
 - **commit()** déclenche validation de la transaction
 - *toutes les modifications apportées à la base de données depuis le dernier **commit()** ou **rollback()** sont effectuées et deviendront visibles (si il n'y a pas d'erreur).*
 - **rollback()** annule la transaction
 - *toutes les modifications apportées à la base de données depuis le dernier **commit()** ou **rollback()** sont annulées. Cela signifie que la base de données revient à l'état dans lequel elle se trouvait avant le début de la transaction.*

- L'interface **Connection** offre des services de gestion des transactions
 - par défaut les connections sont en mode auto-commit
 - `setAutoCommit(boolean autoCommit)` définit le mode de la connexion
 - `setAutoCommit(false)` : *il va falloir gérer 'manuellement' les transactions*
 - `commit()` déclenche validation de la transaction
 - `rollback()` annule la transaction

```
try {
    conn.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
    conn.commit();
}
catch (SQLException e) {
    conn.rollback(); // annule les opérations de la transaction
}
```

- exemple

Stock

produit_id	quantite
101	50
102	30
103	20

Commandes

commande_id	produit_id	quantite	date_commande
1	101	5	2025-02-13
2	102	10	2025-02-14
3	103	15	2025-02-15

```
CREATE TABLE stock (
    produit_id INT PRIMARY KEY,
    quantite INT
);
```

```
INSERT INTO stock (produit_id, quantite) VALUES (101, 50);
INSERT INTO stock (produit_id, quantite) VALUES (102, 30);
INSERT INTO stock (produit_id, quantite) VALUES (103, 20);
```

```
CREATE TABLE commandes (
    commande_id INT PRIMARY KEY,
    produit_id INT,
    quantite INT,
    date_commande DATE,
    FOREIGN KEY (produit_id) REFERENCES stock(produit_id)
);
```

```
INSERT INTO commandes (commande_id, produit_id, quantite, date_commande)
VALUES (1, 101, 5, '2025-02-13');
INSERT INTO commandes (commande_id, produit_id, quantite, date_commande)
VALUES (2, 102, 10, '2025-02-14');
INSERT INTO commandes (commande_id, produit_id, quantite, date_commande)
VALUES (3, 103, 15, '2025-02-15');
```

Ajout d'une nouvelle commande 2 opérations :

- Insérer une nouvelle commande,
- Mettre à jour le stock en conséquence.

Transaction

Si une erreur se produit, nous utiliserons `rollback()` pour annuler les modifications.

```
connection.setAutoCommit(false);  
  
try (Statement stmt = connection.createStatement()) {  
    // Ajouter une nouvelle commande  
    String ajouterCommande =  
        "INSERT INTO commandes (commande_id, produit_id, quantite, date_commande) "  
        + "VALUES (1, 101, 5, '2025-02-13')";  
    stmt.executeUpdate(ajouterCommande);  
  
    // Mettre à jour le stock  
    String mettreAJourStock =  
        "UPDATE stock SET quantite = quantite - 5 WHERE produit_id = 101";  
    stmt.executeUpdate(mettreAJourStock);  
  
    // Valider la transaction  
    connection.commit();  
    System.out.println("Transaction réussie !");  
} catch (SQLException ex) {  
    // Annuler la transaction en cas d'erreur  
    connection.rollback();  
    System.out.println("Transaction annulée en raison d'une erreur.");  
    System.out.println(ex.getMessage());  
}
```

Début de la transaction

Ajout de la commande

Mise à jour du stock

Toutes les instructions SQL se sont exécutées sans erreur : validation de la transaction

Annulation en cas d'erreur



Après avoir appelé `commit()`, la connexion reste dans le mode de transaction manuel si le mode auto-commit a été désactivé précédemment.
→ vous devez continuer à gérer les transactions manuellement jusqu'à ce que vous réactiviez explicitement le mode auto-commit.

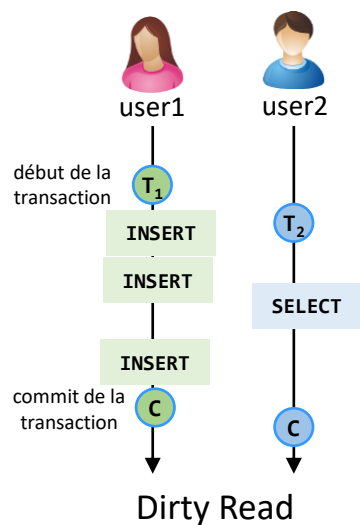
- `int getTransactionIsolation()` (de l'interface `Connection`) pour savoir quel support le SGBD et le pilote JDBC offrent pour les transactions

java.sql.Connection

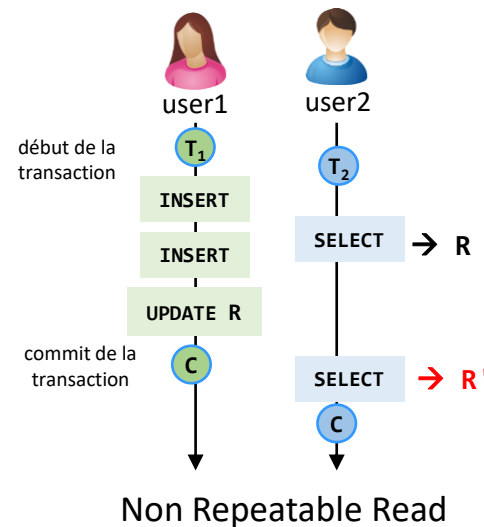
Modifier and Type	Constant Field	Value	Lecture 'sale' (Dirty read)	Lecture 'non-répétable' (Nonrepeatable read)	Lecture 'fantôme' (Phantom read)
public static final int	<code>TRANSACTION_NONE</code>	0			
public static final int	<code>TRANSACTION_READ_UNCOMMITTED</code>	1	possible	possible	possible
public static final int	<code>TRANSACTION_READ_COMMITTED</code>	2	-	possible	possible
public static final int	<code>TRANSACTION_REPEATABLE_READ</code>	4	-	-	possible
public static final int	<code>TRANSACTION_SERIALIZABLE</code>	8	-	-	-

plus le niveau d'isolation est fort plus c'est couteux en performances

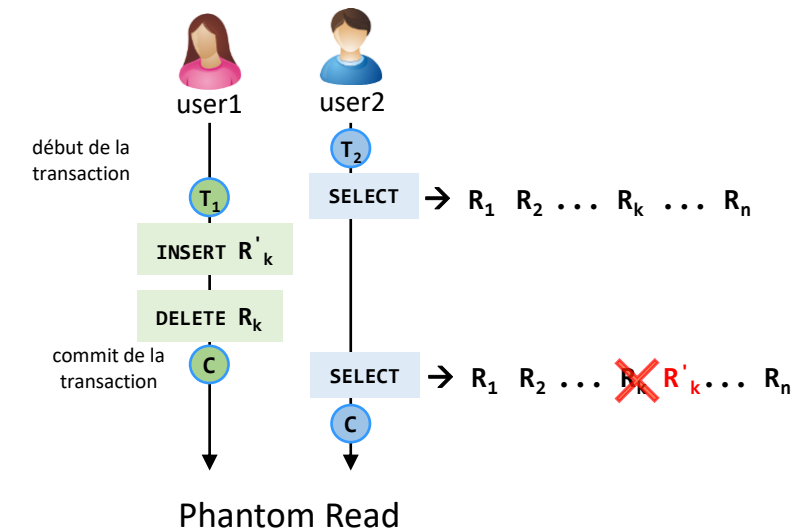
possibilité d'ajuster le niveau de transaction avec `setTransactionIsolation(int level)`



quand dans une transaction on peut lire des données qui n'ont pas encore été comitées. L'intégrité des données peut être compromise, (clés étrangères violées, contraintes d'unicité ignorées...)

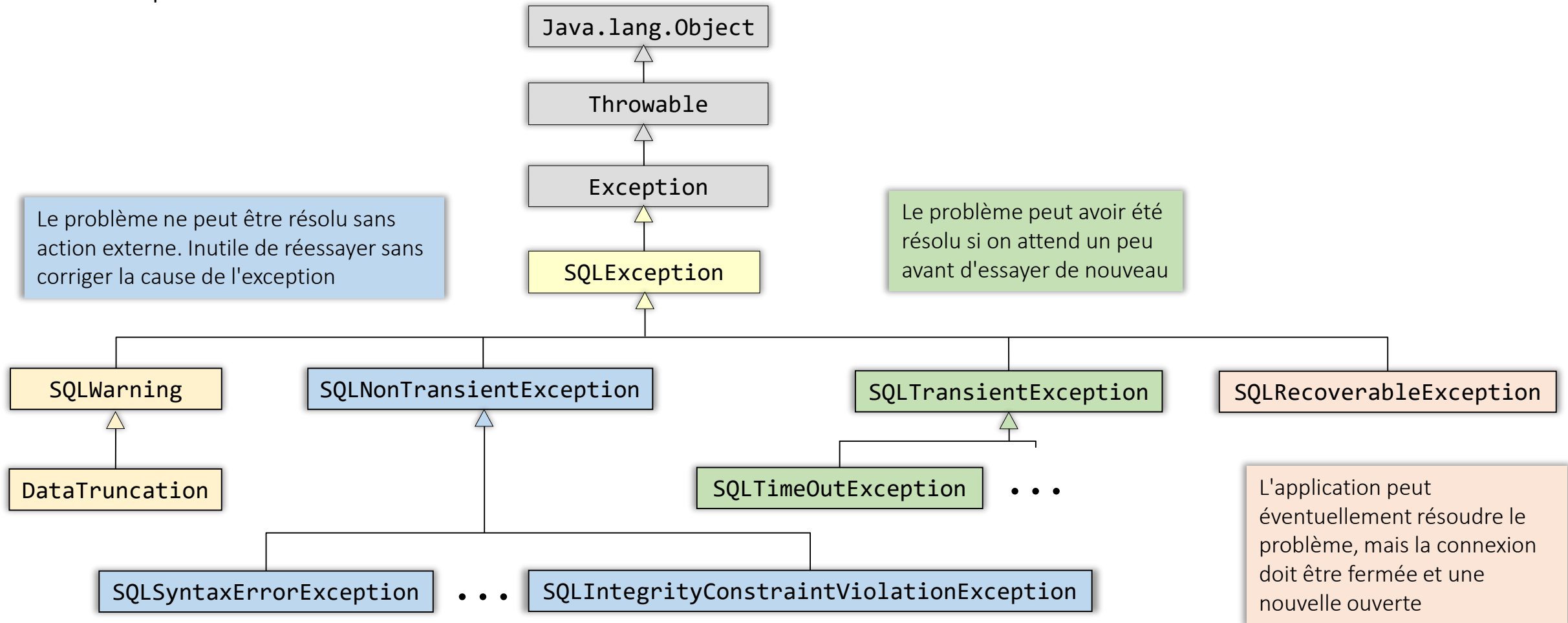


quand dans une transaction on lit plusieurs fois un même enregistrement (row), et que l'on peut obtenir des valeurs différentes.



quand dans une transaction une même requête effectuée plusieurs fois peut donner des ensembles d'enregistrements différents (apparitions de nouveaux enregistrements, disparition d'autres)

Les exceptions dans API JDBC 4.0



- *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() );  
...
```

- Fonctionnalités ajoutées à l'interface **Statement** pour permettre de regrouper des traitements qui seront envoyés en une seule fois au SGBD
 - → 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 et 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 rangs affectés
- **EXECUTE_FAILED** si la commande a échoué

En cas d'echec 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)

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

- **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)*

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

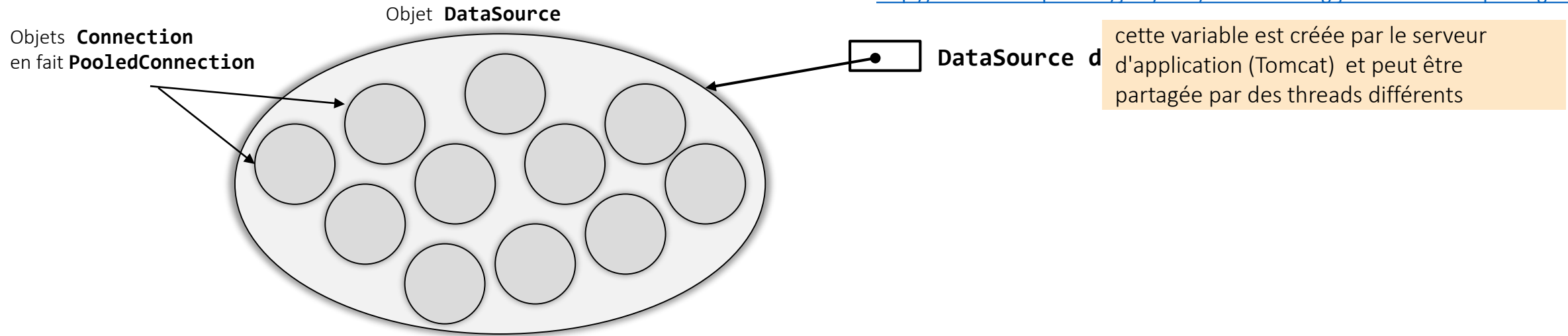
* Understanding JDBC Connection Pooling, Manoj Debnath, May 19, 2017

<http://www.developer.com/java/data/understanding-jdbc-connection-pooling.html>

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

* Understanding JDBC Connection Pooling, Manoj Debnath, May 19, 2017

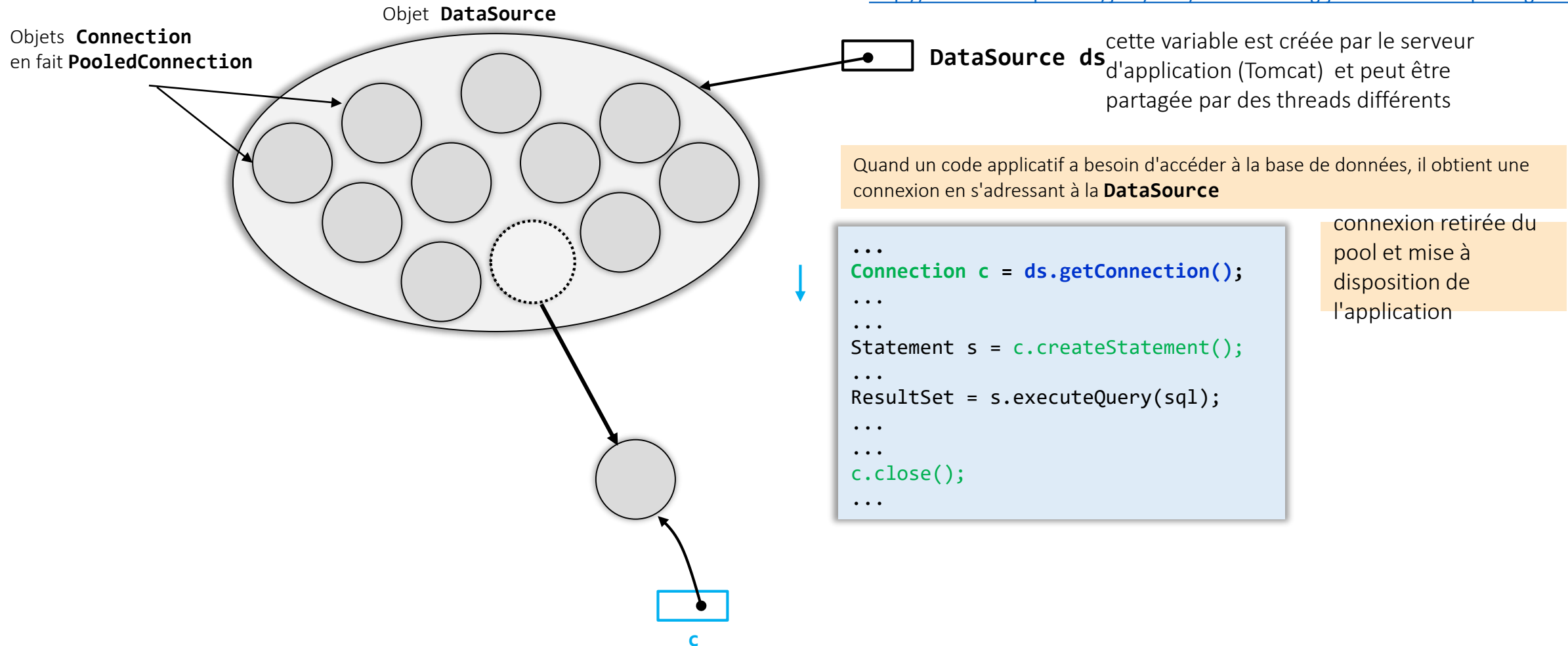
<http://www.developer.com/java/data/understanding-jdbc-connection-pooling.html>



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

* Understanding JDBC Connection Pooling, Manoj Debnath, May 19, 2017

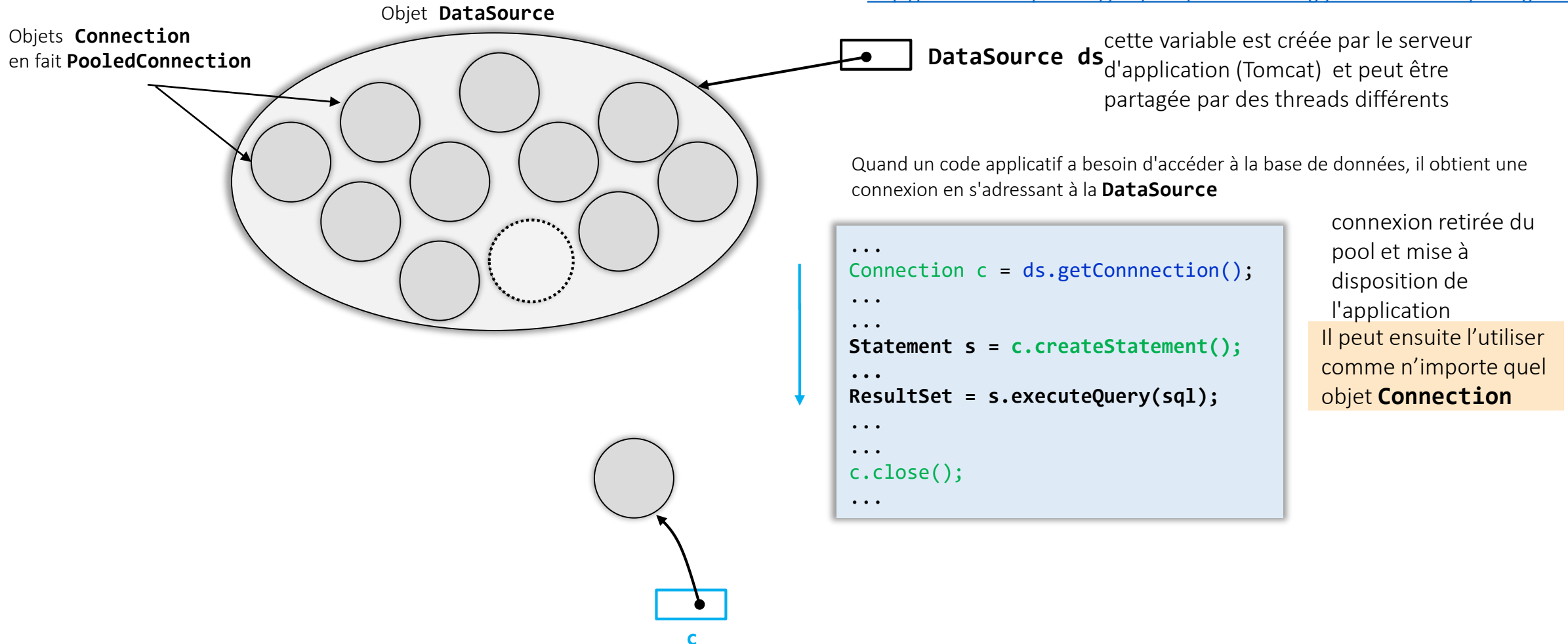
<http://www.developer.com/java/data/understanding-jdbc-connection-pooling.html>



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

* Understanding JDBC Connection Pooling, Manoj Debnath, May 19, 2017

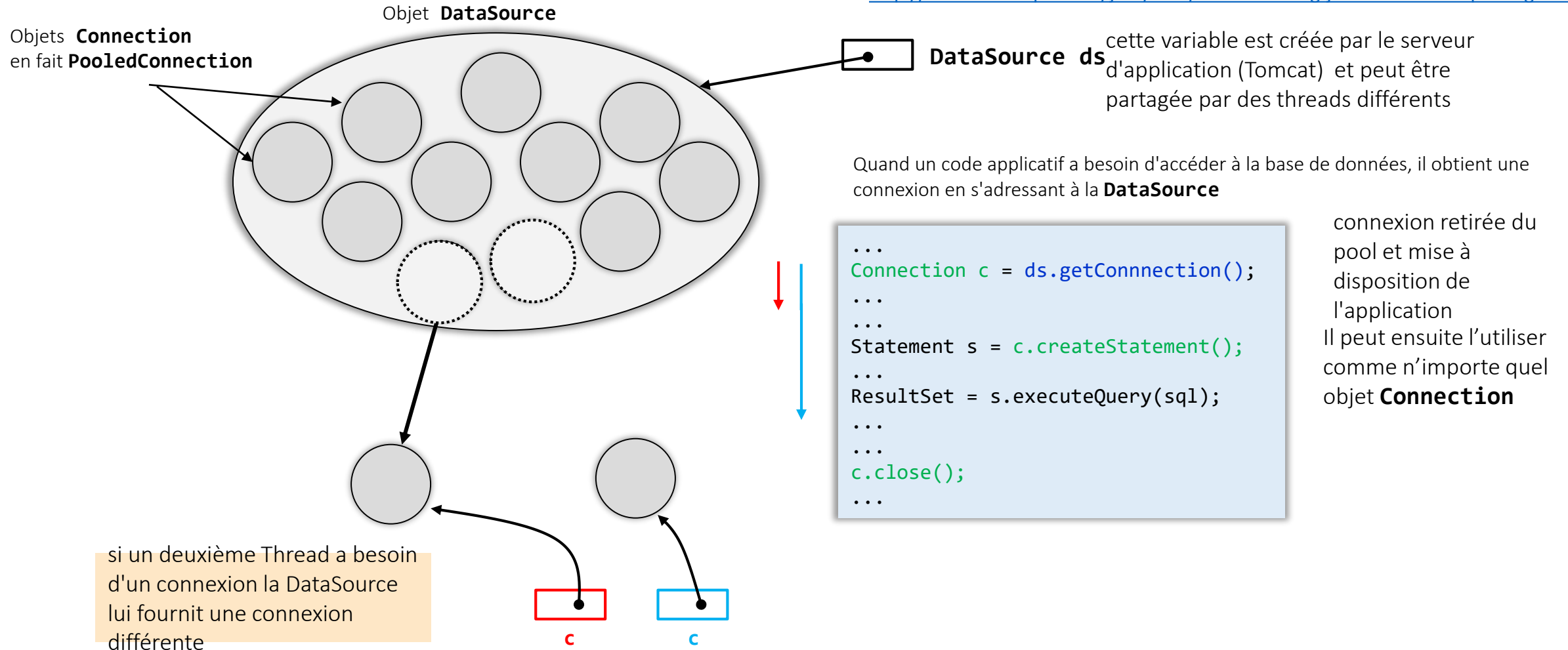
<http://www.developer.com/java/data/understanding-jdbc-connection-pooling.html>



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

* Understanding JDBC Connection Pooling, Manoj Debnath, May 19, 2017

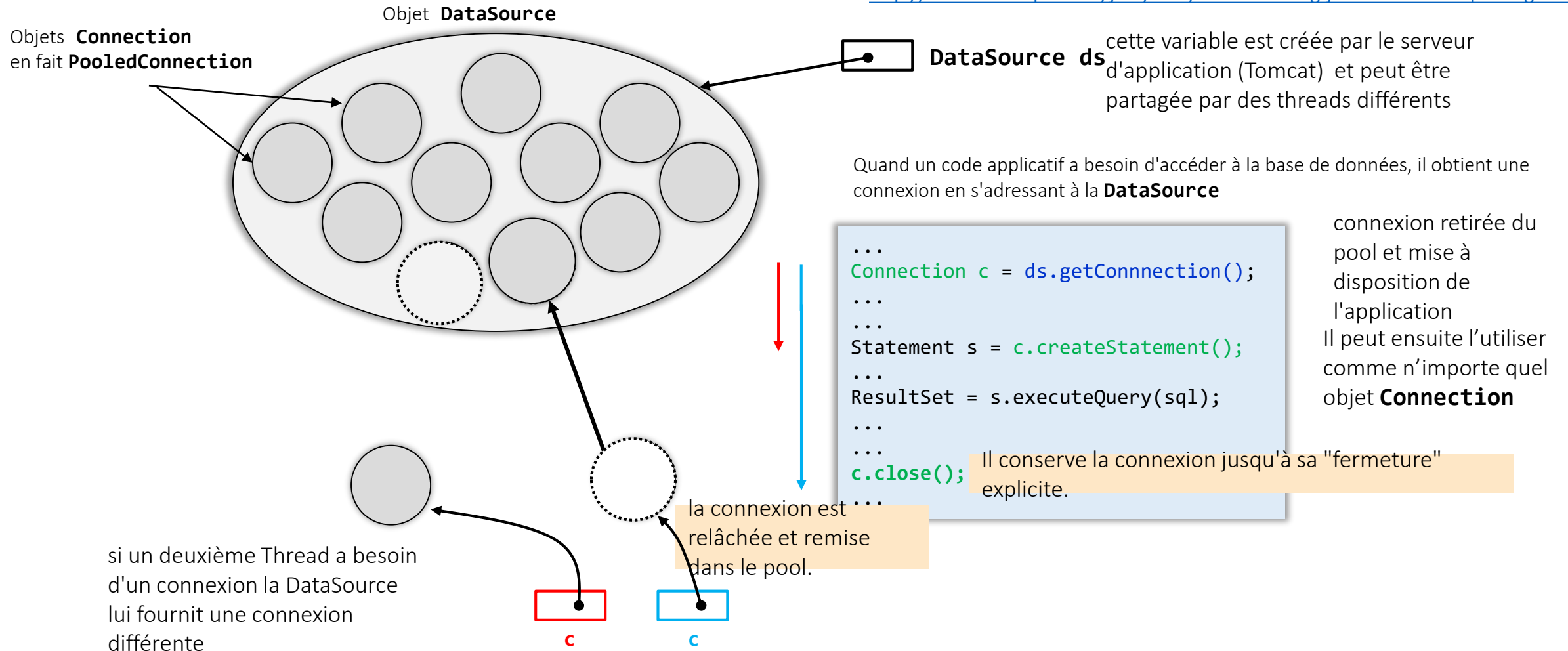
<http://www.developer.com/java/data/understanding-jdbc-connection-pooling.html>



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

* Understanding JDBC Connection Pooling, Manoj Debnath, May 19, 2017

<http://www.developer.com/java/data/understanding-jdbc-connection-pooling.html>



- JDBC API de bas niveau
 - Parfois difficile à pendre en main
 - Demande des connaissances "pointues" en BD
 - Dépendances par rapport au SGBD cible (particularités du dialecte SQL)
- Nombreuses API construites au dessus de JDBC
 - Spring JDBC
 - <https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#jdbc>
 - Jakarta Commons DbUtils (simplifie utilisation de JDBC)
 - <http://jakarta.apache.org/commons/dbutils/>
 - Frameworks de persistance ou de mapping (*ORM Object Relational Mapping*)
 - Hibernate, <http://www.hibernate.org>
 - MyBatis, <http://blog.mybatis.org/>
 - JPA – Java Persistence API, JEE5+
 - modèle de persistance EJB (Entreprise Java Beans) 3.0 issu d'Hibernate