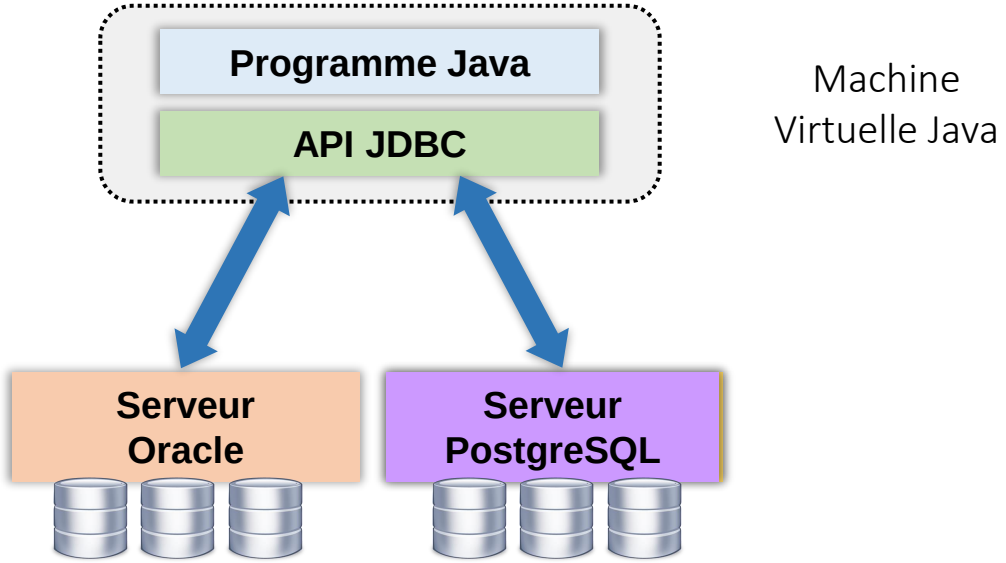
Java DataBase Connectivity

dernière modification : 21/02/2024 00:41

Philippe Genoud

- 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.

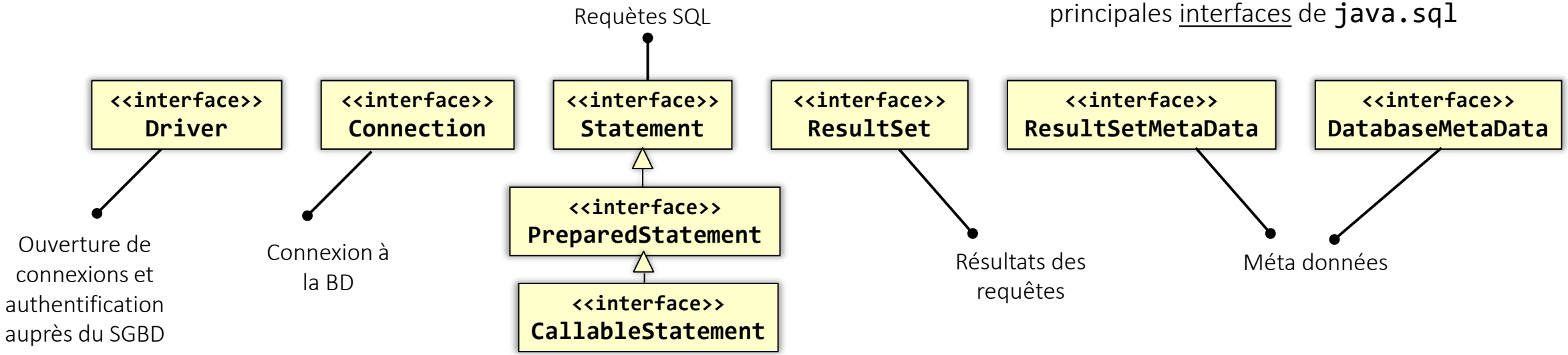


- L'API JDBC est **indépendante** des SGBD.
  - Un changement de SGBD ne doit pas impacter le code applicatif.

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

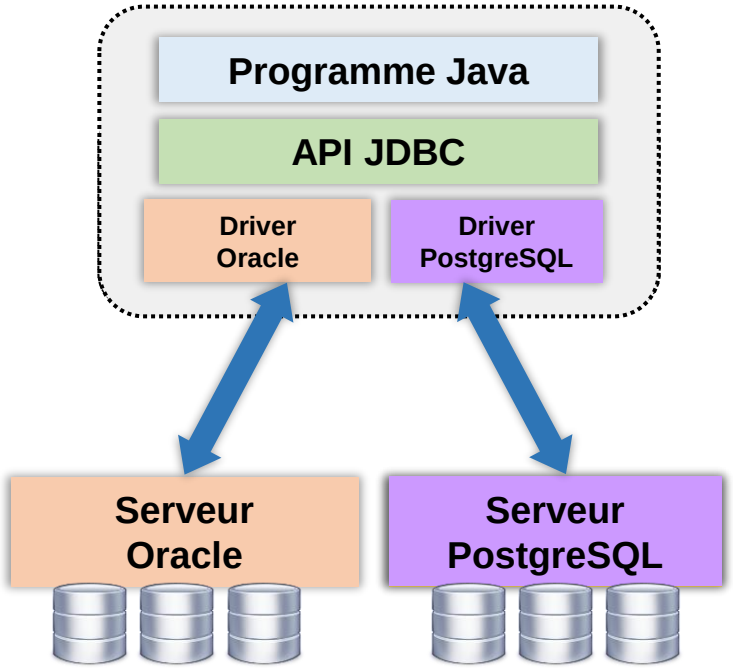
interfaces de l'API JDBC

- 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*



Pilotes JDBC

- 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**

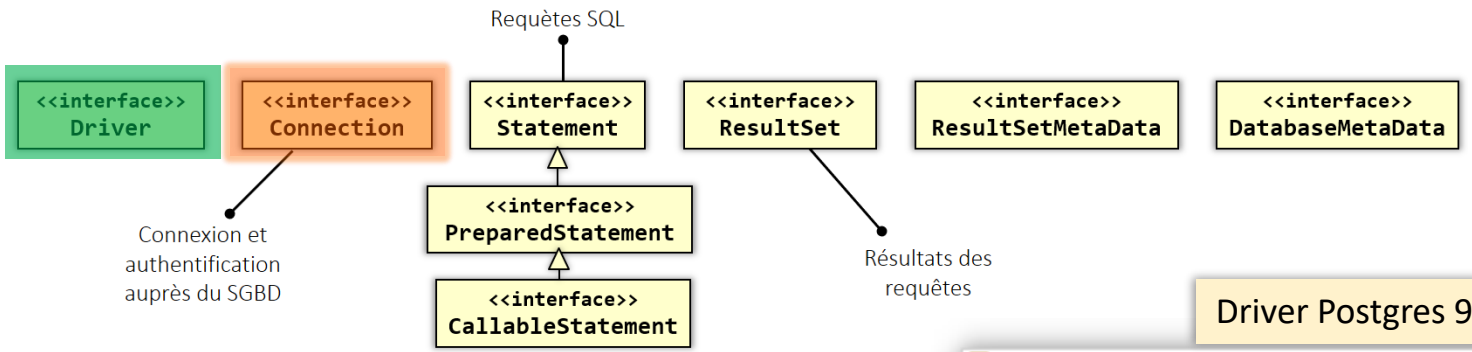


• Il existe des *pilotes* pour tous les SGBD importants du marché (Oracle, MySQL, PostgreSQL, DB2, SQLite…)

• 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

Pilotes JDBC

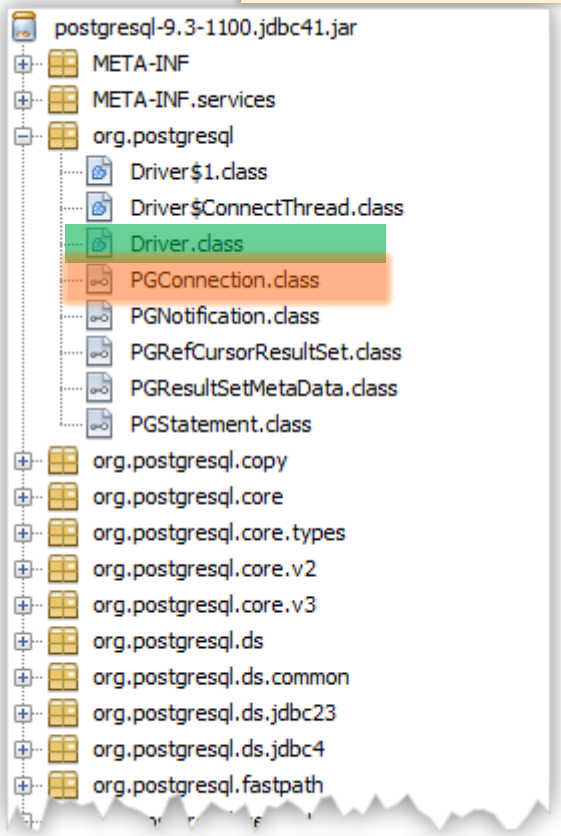


Les interfaces définissent une **abstraction** du pilote (driver) de la base de données. Chaque fournisseur propose **sa propre implémentation** de ces interfaces.

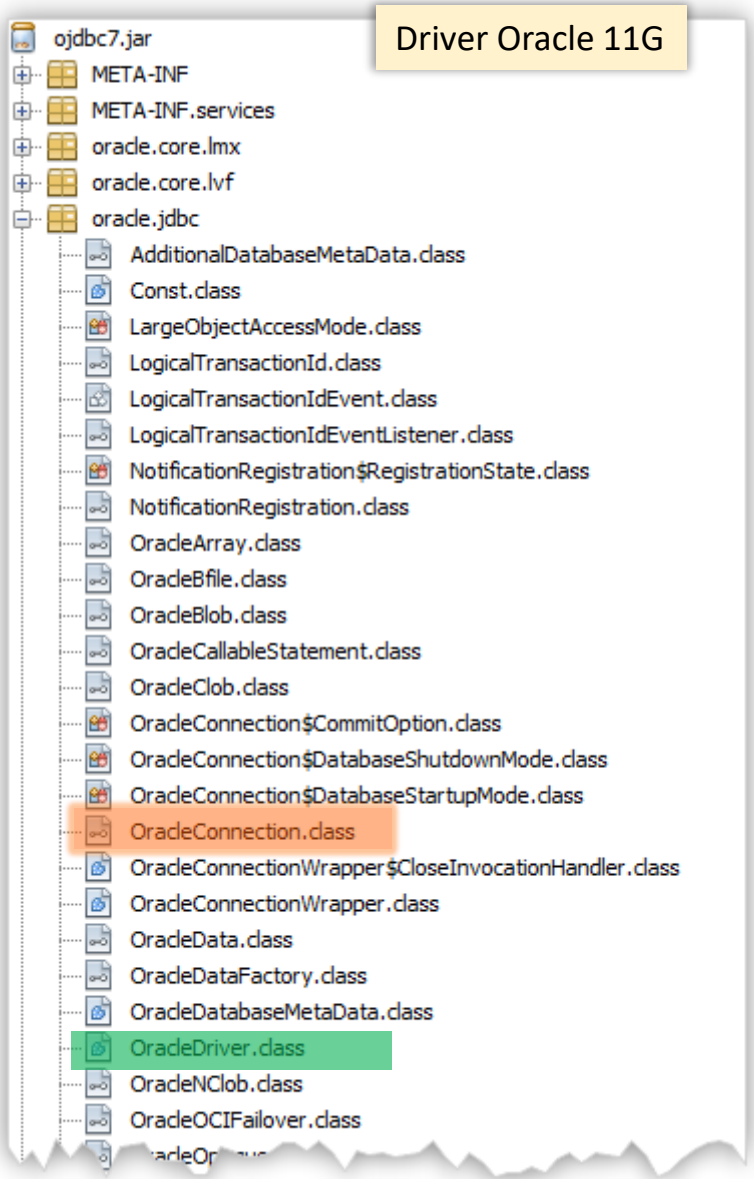
Les classes d'implémentation du driver jdbc sont dans une archive (fichier `.jar`) qu'il faut intégrer (sans la décompresser) au niveau du `classpath` de l'application au moment de l'exécution.

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

Driver Postgres 9.2

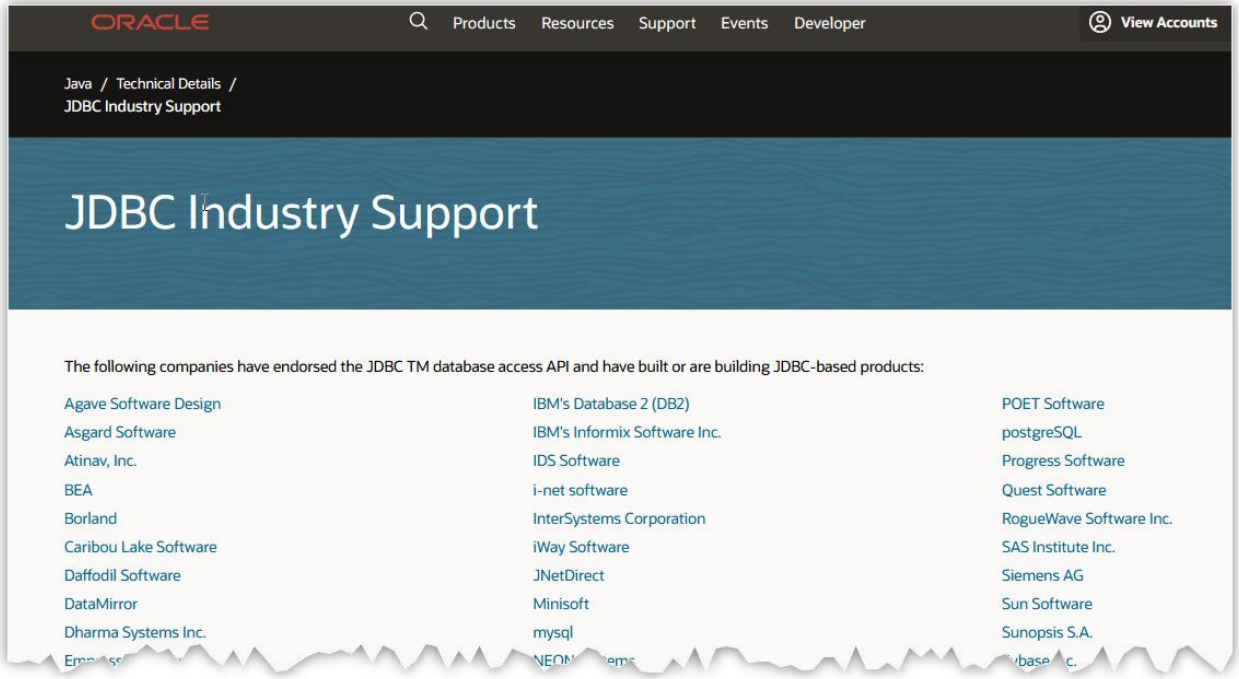


Driver Oracle 11G



Trouver un driver JDBC

- Liste des drivers disponibles à : https://www.oracle.com/java/technologies/industry-support.html

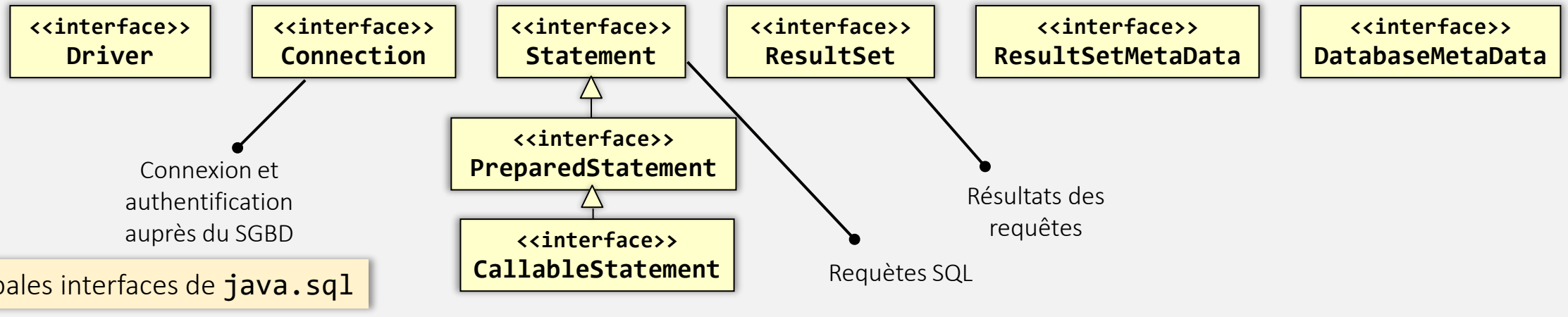


Capture du 19/02/2024
Pas nécessairement à jour

- 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
  - …

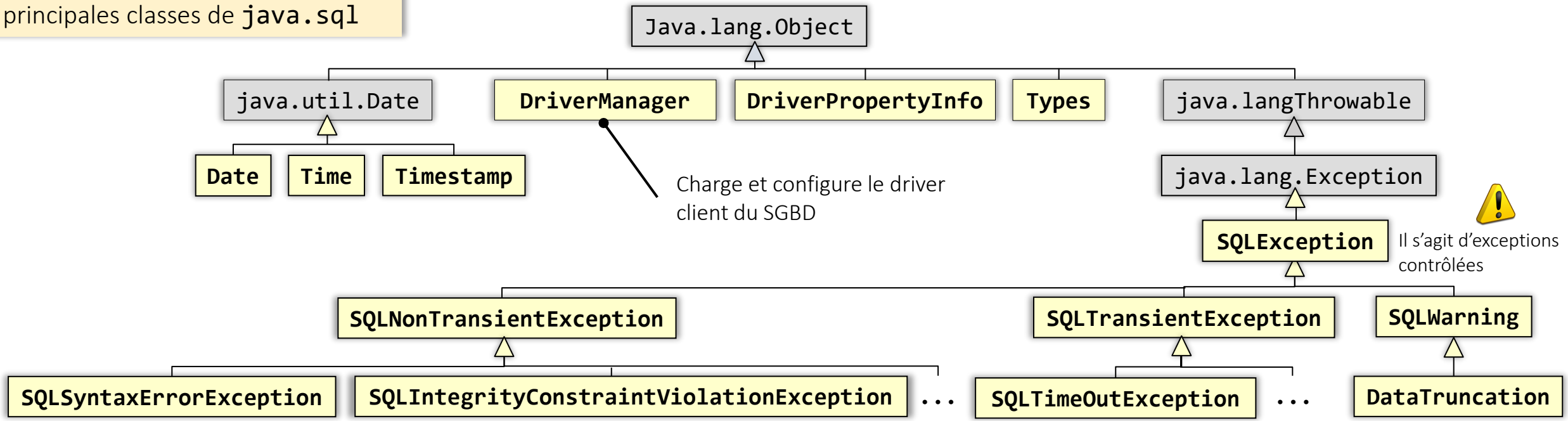
- Différentes versions
  - JDBC 1.0 Core API (JDK 1.1) : package `java.sql`
    - *supporte le standard SQL-2 entry level*
  - JDBC 2.0 (Java 2 JDK 1.2) : packages `java.sql javax.sql`
    - *support de certaines fonctionnalité de SQL-3*
  - JDBC 3.0 (JDK 1.4) : packages `java.sql javax.sql javax.sql.rowset`
    - *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`

Classes et interfaces de JDBC



principales interfaces de `java.sql`

principales classes de `java.sql`



Classes et interfaces de JDBC

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

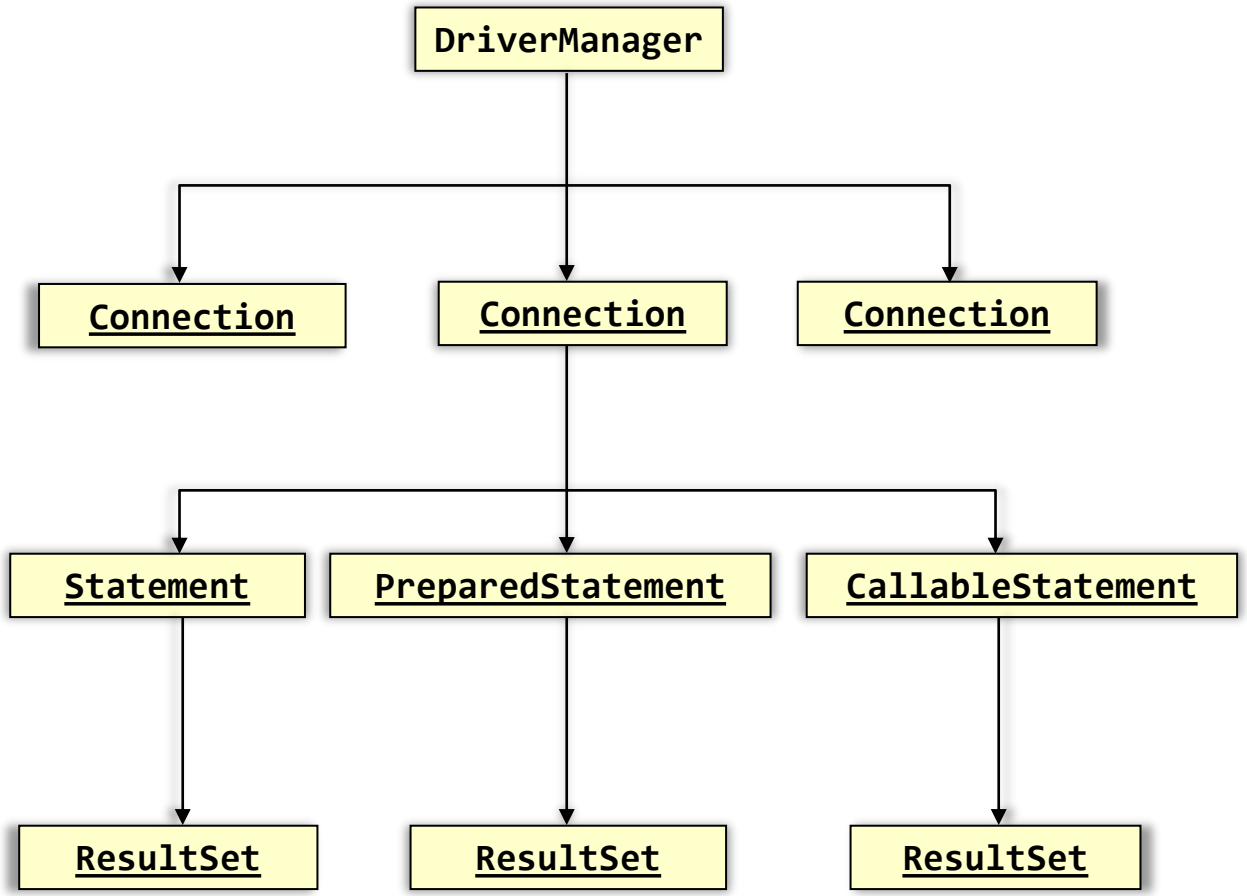
l'objet `Statement` permet d'envoyer requêtes SQL et de créer des objets `ResultSet` encapsulant le résultat des requêtes

```
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

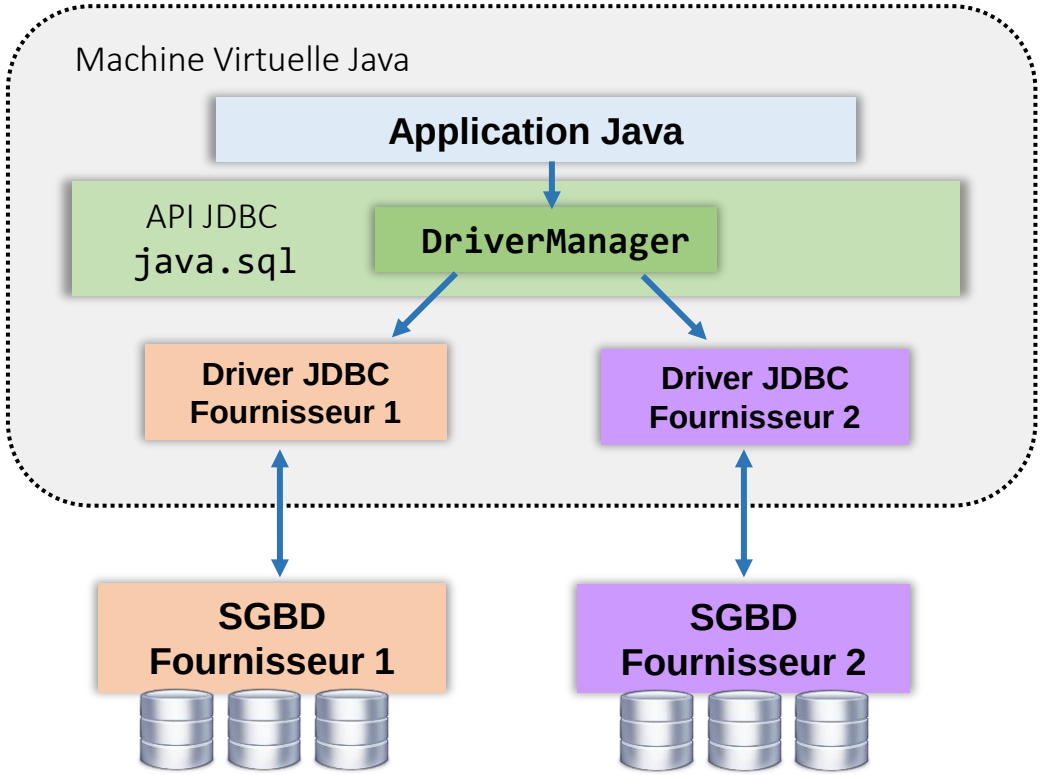
```
while (rs.next()) {
  String nom = rs.getString("NOM");
  ...
}
```



- 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

0) Chargement du driver

- `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

0) Chargement du driver

- 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

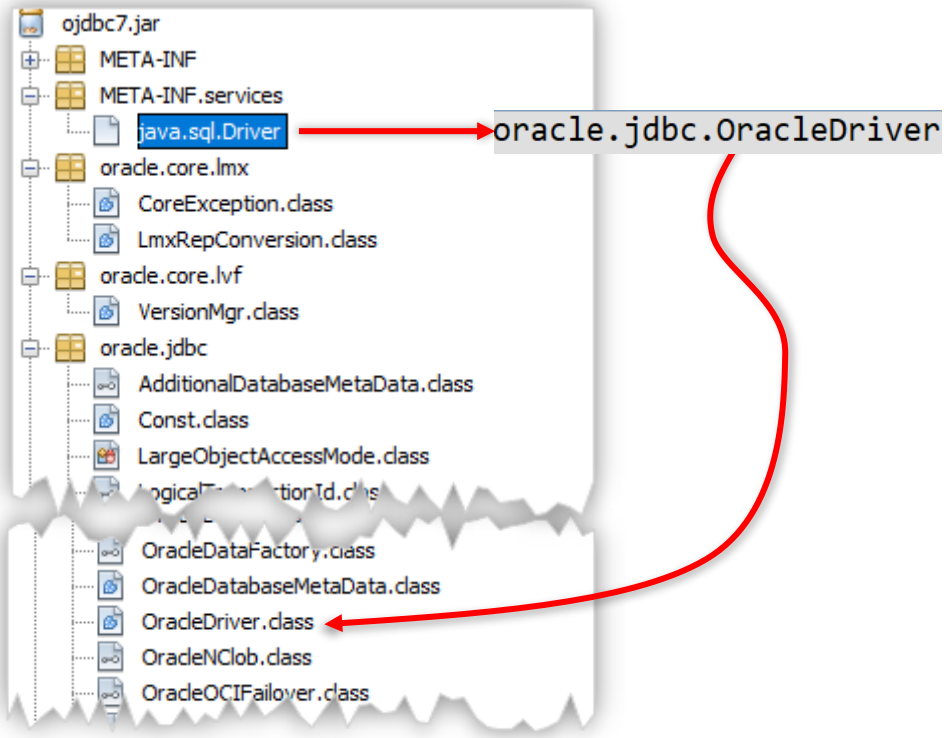
Chargement d'un pilote Oracle

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

0) Chargement du driver

- 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+ .

Yann D'Isanto : Développer des services en Java
http://ydisanto.developpez.com/tutoriels/java/services/

1) Connexion à la BD

- Ouverture de la connexion :

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

- Identification de la BD via un URL (Uniform Ressource 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`.

1) Connexion à la BD – exemple

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

```
# fichier de proprietés pour une connexion à la base Oracle 11G de l'ufr
# 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=xxxxxxxxxxx
```

bdUFRIM2AG.properties

```
# fichier de proprietés 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=xxxxxxxxxxx
```

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

1) Connexion à la BD : fermeture

- 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) {
   ...
}
```

Erreur SQL lors du dialogue avec la BD

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

Comment garantir la fermeture des connexions ?

1) Connexion à la BD : fermeture

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

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

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

```
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`**

1) Connexion à la BD : fermeture

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

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

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

```
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**

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

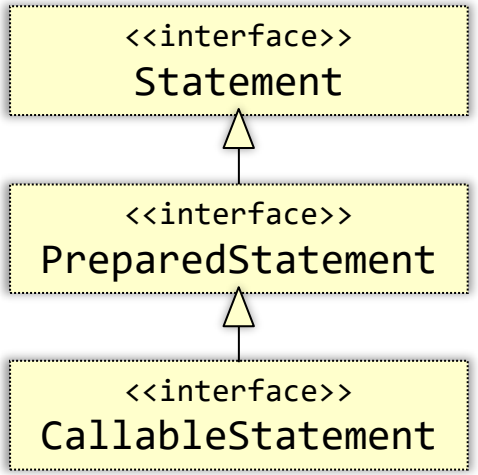
```
try (Connection conn =
        DriverManager.getConnection(url, user, passwd)) {
   ...
   // utilisation de la connexion pour dialoguer
   // avec la BD
   ...
}
catch (SQLException e) {
   ...
}
```

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

2) Préparer/ 3)exécuter une requête

- 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 connait 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*

2) Préparer / 3) exécuter une requête simple

- 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 :

la requête est une chaîne SQL qui doit être valide

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

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

espaces nécessaires si concaténation de chaînes

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

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

2) Préparer / 3) exécuter une requête simple

- 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 :

la requête est une chaîne SQL qui doit être valide

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

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

espaces nécessaires si concaténation de chaînes

```
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 **"""**

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

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

4) Lecture 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
  - Curseur de lecture placé avant la première ligne à la création du `ResultSet`
  - `boolean next()` permet d'avancer à la ligne suivante, → `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
}
```

4) Lecture des résultats

- 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

4) Lecture des résultats

Equivalences des types Java-SQL

- Pour chaque méthode **get*XXX*** 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 *Boolean*
TINYINT	getByte	byte *Integer*
SMALLINT	getShort	short *Integer*
INTEGER	getInt	int *Integer*
BIGINT	getLong	long *Long*
REAL	getFloat	float *Float*
FLOAT	getDouble	double *Double*
DOUBLE	getDouble	double *Double*
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

4) Lecture des résultats

Traitement des valeurs nulles

- 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()`**

4) Lecture des résultats

Traitement des valeurs nulles

- 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");
}
```

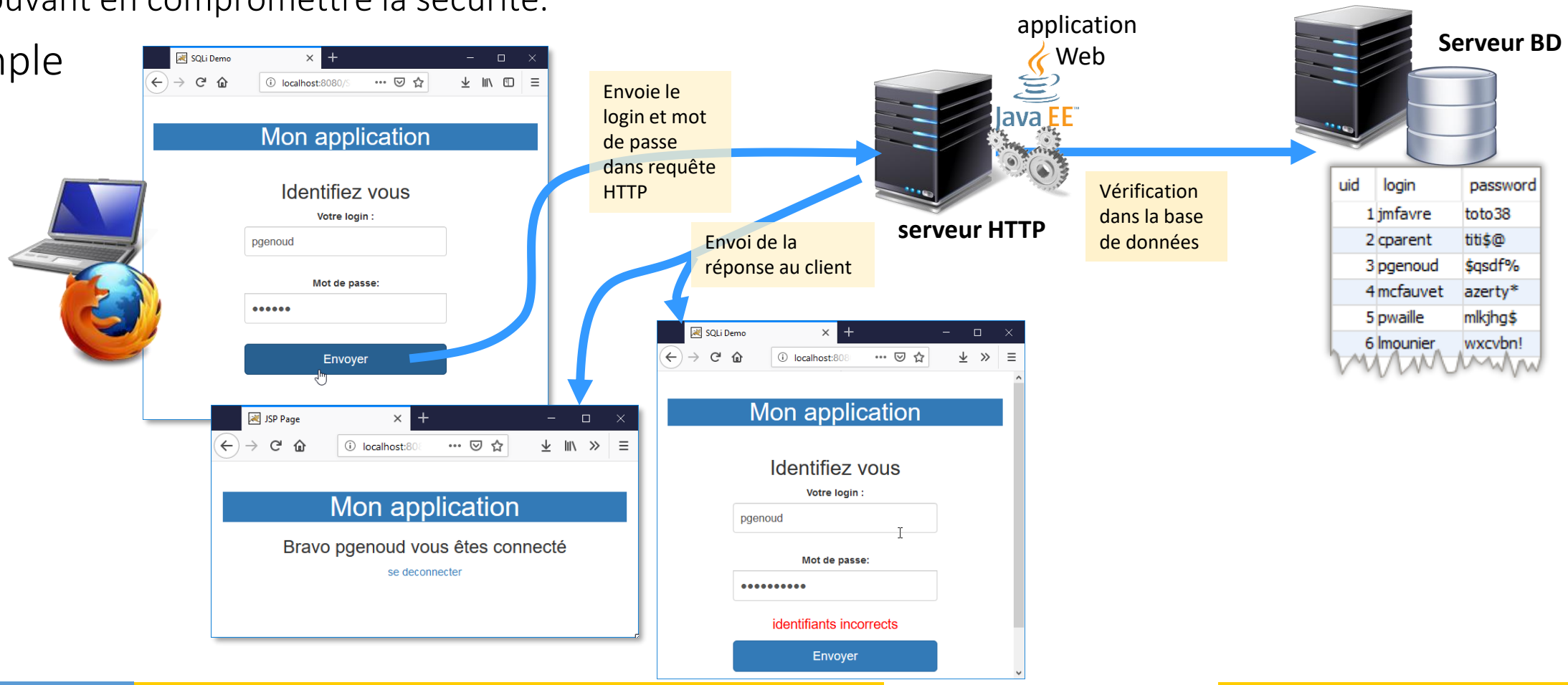
```
TITI Fifi Marié
TOTO Riri Non Marié
TUTU Mimi Non Marié
...
```

- 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

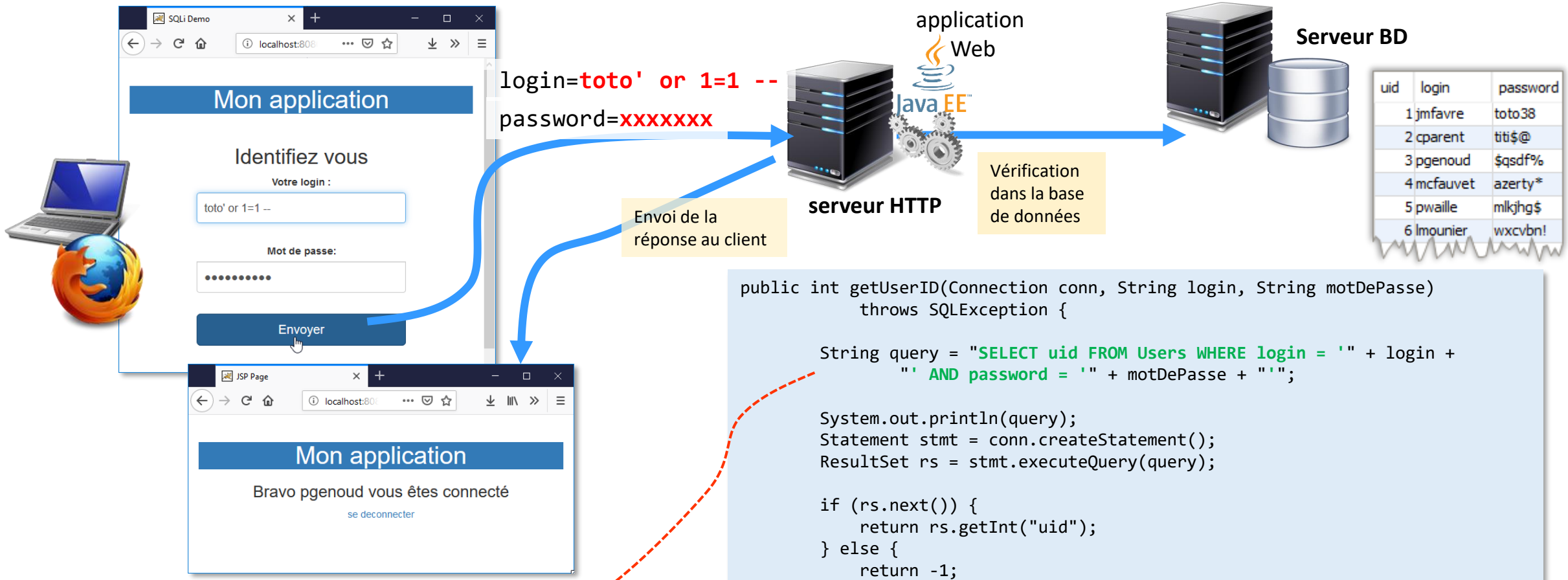
```
/**
 * @param conn la connexion JDBC
 * @param login l'identifiant de l'utilisateur
 * @param motDePasse le mot de passe propose
 *
 * @return l'identifiant interne (uid) si le mot de passe est correct
 *         -1 si le mot de passe est incorrect
 *
 * @throws SQLException si pb avec la BD
 */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

Statements et injection SQL



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

```
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;
        }
}
```

La requête exécutée

```
SELECT uid FROM Users WHERE login = 'toto' or 1=1 --' AND password = 'XXXXXX'
```

⇔ `SELECT uid FROM Users`

Le `ResultSet` contient tous les `uid`, la fonction retourne le premier de ceux-ci !

ne pas utiliser la concaténation de chaînes pour construire des requêtes

Préparer/exécuter une requête précompilée

- 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)

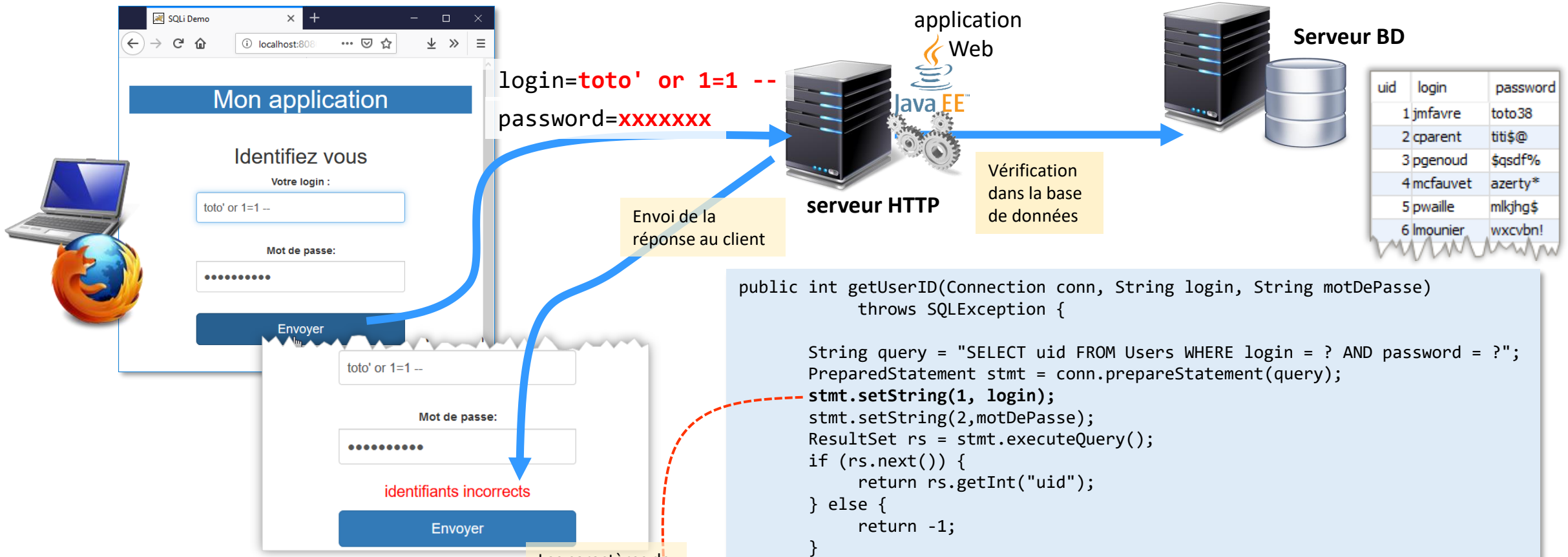
```
String[] names = ...

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

PreparedStatements et injection SQL




```
public int getUserID(Connection conn, String login, String motDePasse)
            throws SQLException {

        String query = "SELECT uid FROM Users WHERE login = ? AND password = ?";
        PreparedStatement stmt = conn.prepareStatement(query);
        stmt.setString(1, login);
        stmt.setString(2,motDePasse);
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
             return rs.getInt("uid");
        } else {
             return -1;
        }
}
```

Les caractères de fin de chaîne sont échappés

La requête exécutée

```
SELECT uid FROM Users WHERE login = 'toto\' or 1=1 --' AND password = 'XXXXXX'
```

Le `ResultSet` est vide la fonction retourne `-1`

CallableStatements : Procédures stockées

- 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 est 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

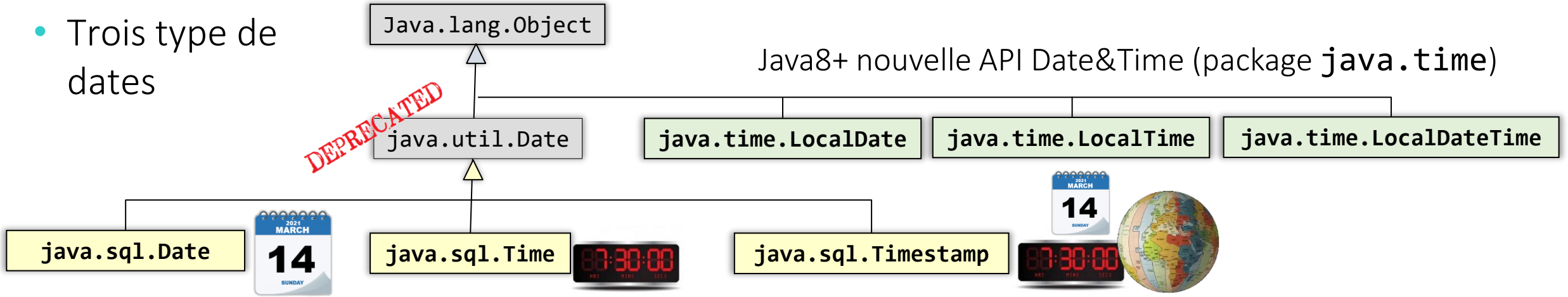
```
ResultSet rs = proc.executeQuery();
```

- Exploitation du ResultSet (idem que pour Statement et PreparedStatement)
- Récupération des paramètres OUT

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

Traitement des dates

- Trois type de dates



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

jour dans une année

heure dans une journée

jour dans année + heure + fuseau horaire (time zone)

```
ResulSet 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

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.toLocaleDate();
LocalTime lt = t.toLocalTime();
LocalDateTime ldt = ts.toLocaleDateTime();
```

Fermeture des Statements et RésultSet

- 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`.

Accès aux méta-données

- 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)`
    - *…*

Accès aux méta-données

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

Renvoie `true` si requête de type `Query`, `false` sinon (`Update`)

String contenant une requête quelconque

```
if  stmt.execute(cmd)  {
    ResultSet rs = stmt.getResultSet();
    ...
    //Exploitation du ResultSet
    ...

    ...
    rs.close();
}
else
    System.out.println("nombre de lignes modifiées " + stmt.getUpdateCount() );
}
```

Accès au `ResultSet` produit par la requête

Besoin d'accès aux méta-données du `ResultSet`

```
ResultSetMetaData rsmd = rs.getMetaData();
int numberOfColumns = rsmd.getColumnCount();
//affichage du resutlat de la requête
System.out.println("Résultats de la requête\n");
while (rs.next()) {
    System.out.pritln("------------------------------------");
    for (int i = 1; i <= numberOfColumns; i++) {
            System.out.print(rsmd.getColumnName(i) + " : "  +rs.getString(i) + " ");
     }
 }
```

entier donnant le nombre d'enregistrements modifiés

Gestion des transactions

- 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)

Gestion des transactions

- 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
}
```

- 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

Gestion des transactions

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

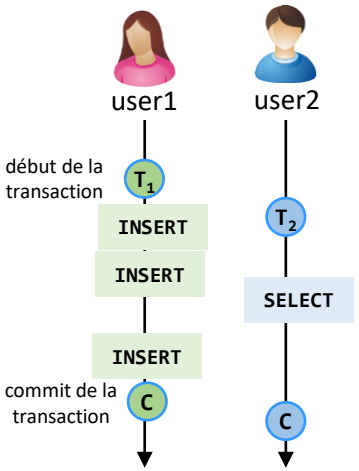
java.sql.Connection

Modifier and Type	Constant Field	Value
public static final int	TRANSACTION_NONE	0
public static final int	TRANSACTION_READ_UNCOMMITTED	1
public static final int	TRANSACTION_READ_COMMITTED	2
public static final int	TRANSACTION_REPEATABLE_READ	4
public static final int	TRANSACTION_SERIALIZABLE	8

	Lecture 'sale' (Dirty read)	Lecture 'non-répétable' (Nonrepeatable read)	Lecture 'fantôme' (Phantom read)
TRANSACTION_READ_UNCOMMITTED	possible	possible	possible
TRANSACTION_READ_COMMITTED	-	possible	possible
TRANSACTION_REPEATABLE_READ	-	-	possible
TRANSACTION_SERIALIZABLE	-	-	-

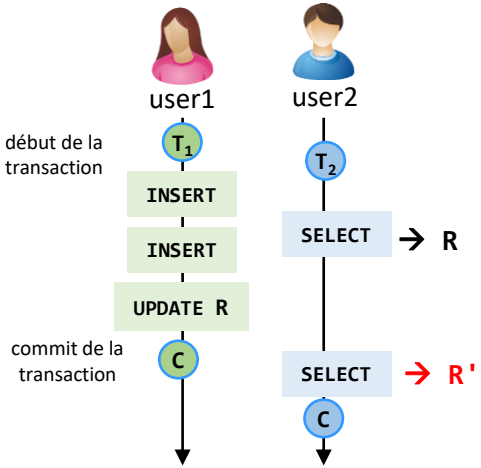
plus le niveau d'isolation est fort plus c'est couteux en performances

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



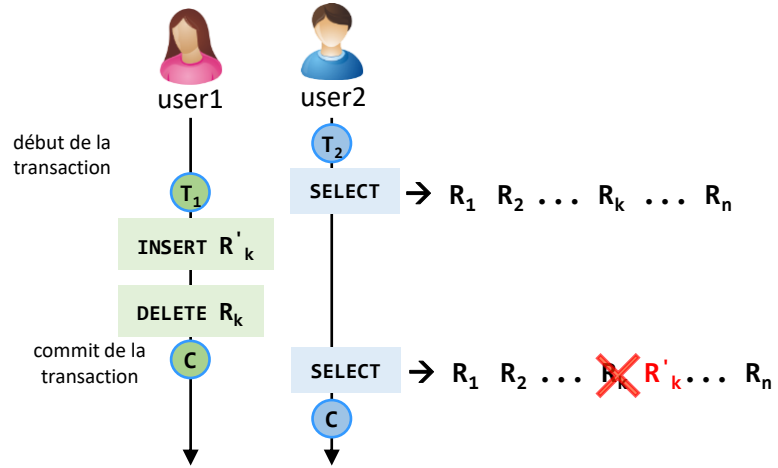
Dirty Read

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…)



Non Repeatable Read

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

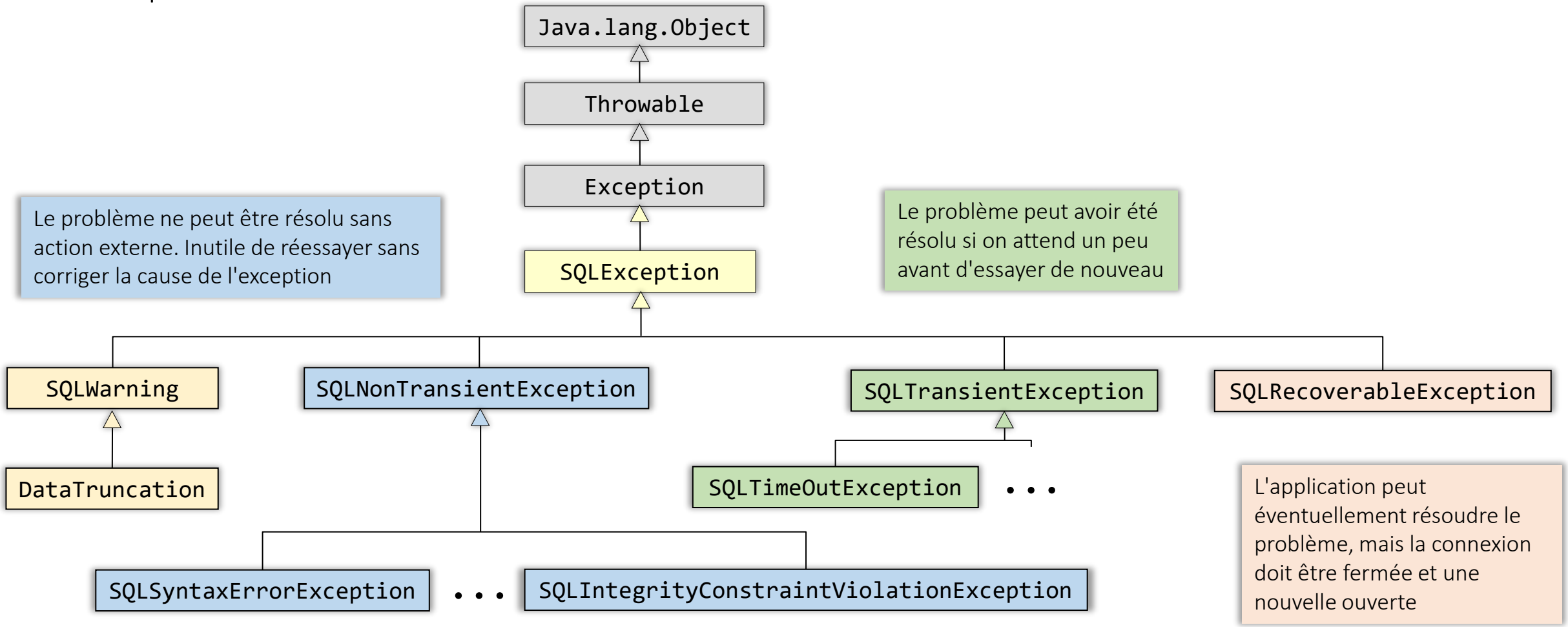


Phantom Read

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

A propos de SQLException

Les exceptions dans API JDBC 4.0



A propos de SQLException

- `SQLException` définit les méthodes suivantes :
  - `getSQLState()` --> un code d'état de la norme SQL ANSI-92
  - `getErrorCode()` --> un code d 'erreur specifique (« 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();
  }
}
```

SQLWarning

- 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`, `PreparedSatement`, `CallableStatement`

```
void printWarninsg(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 2.0 Batch updates

- 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

JDBC 2.0 Batch updates

- 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 Entreprise Edition)
  - Inclus en standard dans JSE (Java Standard Edition) depuis version 1.4

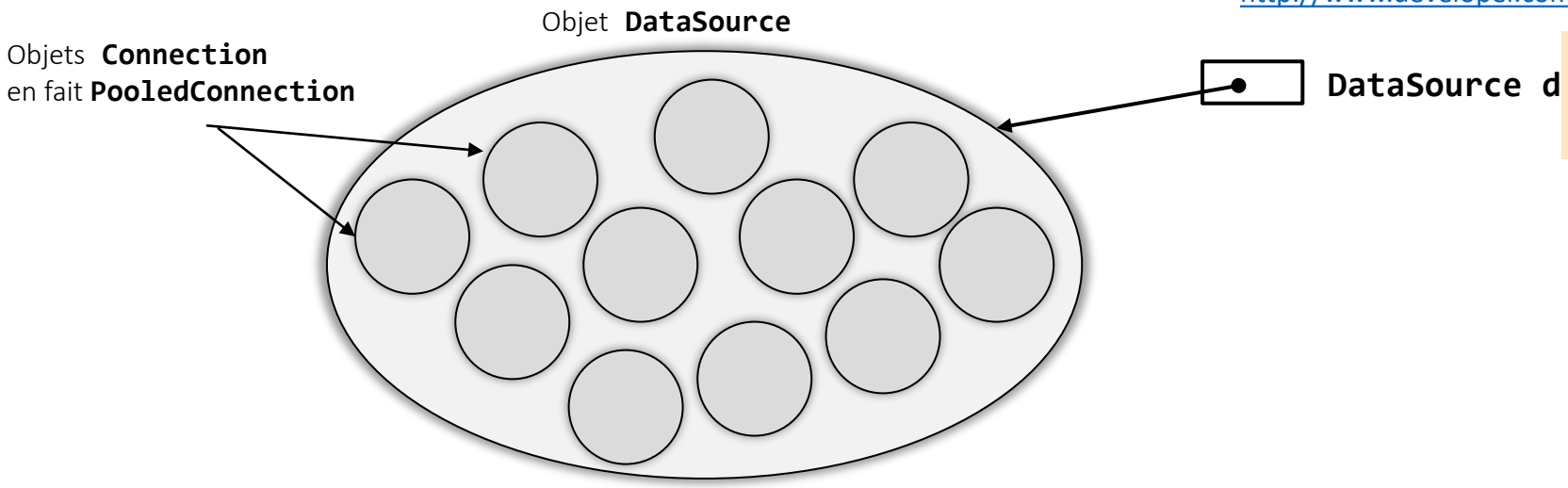
  - **`DataSource`** : Obtention du nom de la base à partir de serveurs de noms plutôt que d'avoir le nom de la base de données codé « en dur » dans l'application.
    - *Utilisation de JNDI (Java Naming and Directory Interface) pour connexion à une base de donnée*
  - **`PooledConnection`** : support 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*)**.

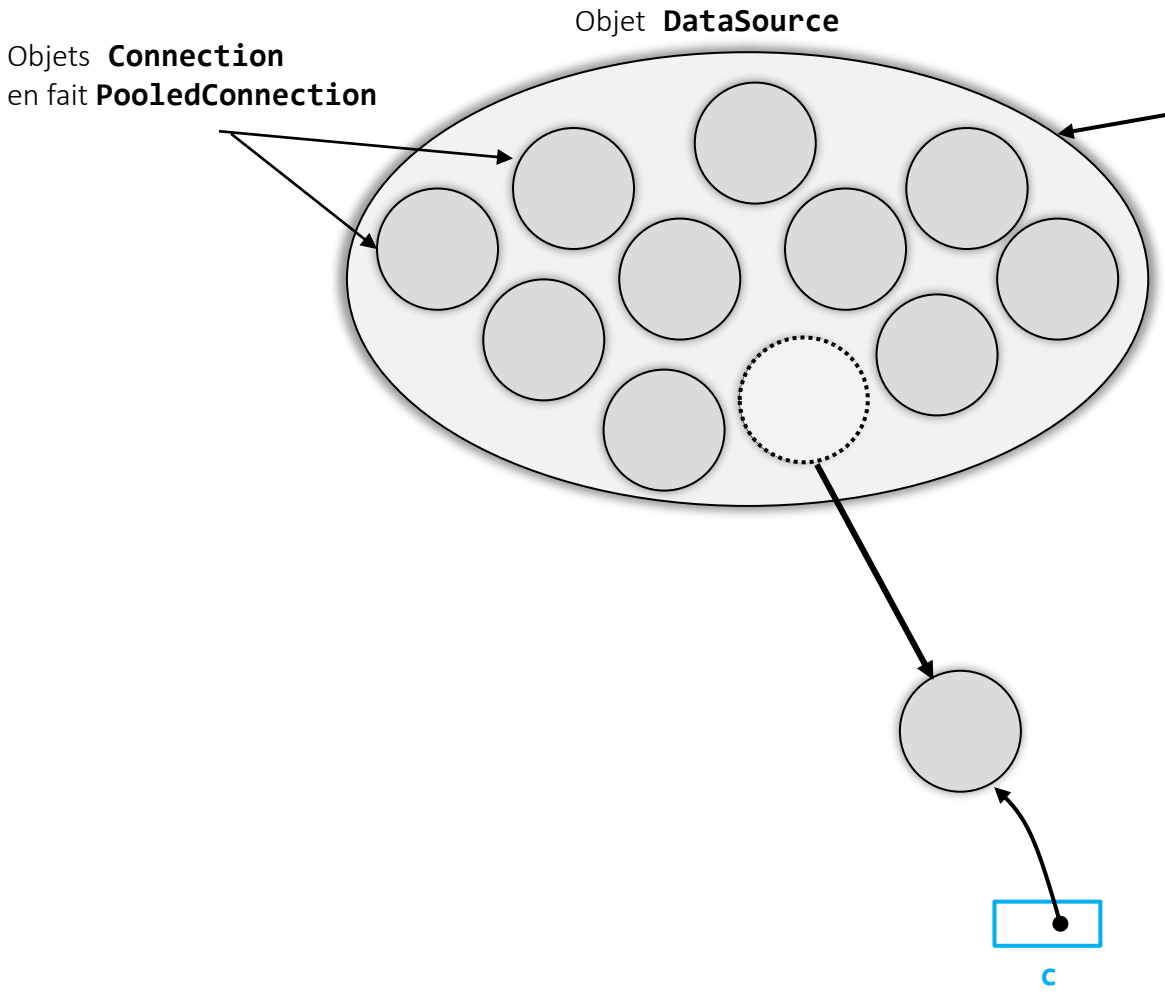
* Understanding JDBC Connection Pooling, Manoj Debnath, May 19, 2017
http://www.developer.com/java/data/understanding-jdbc-connection-pooling.html



cette variable est créée par le serveur d'application (Tomcat) et peut être partagée par des threads différents

- 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



cette variable est créée par le serveur d'application (Tomcat) et peut être partagée par des threads différents

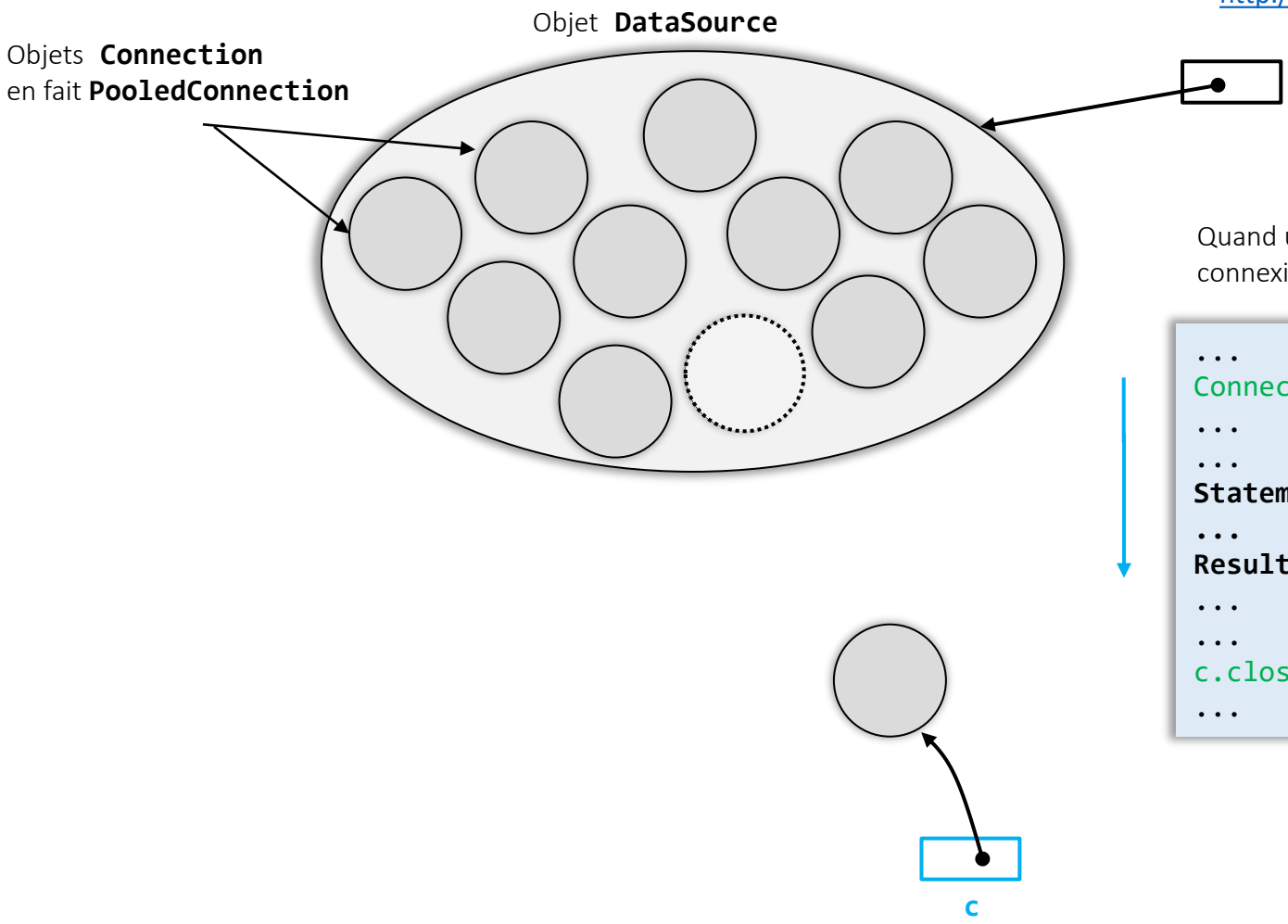
Quand un code applicatif a besoin d'accéder à la base de données, il obtient une connexion en s'adressant à la **DataSource**

```
...
Connection c = ds.getConnection();
...
...
Statement s = c.createStatement();
...
ResultSet = s.executeQuery(sql);
...
...
c.close();
...
```

connexion retirée du pool et mise à disposition de l'application

- 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



cette variable est créée par le serveur d'application (Tomcat) et peut être partagée par des threads différents

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

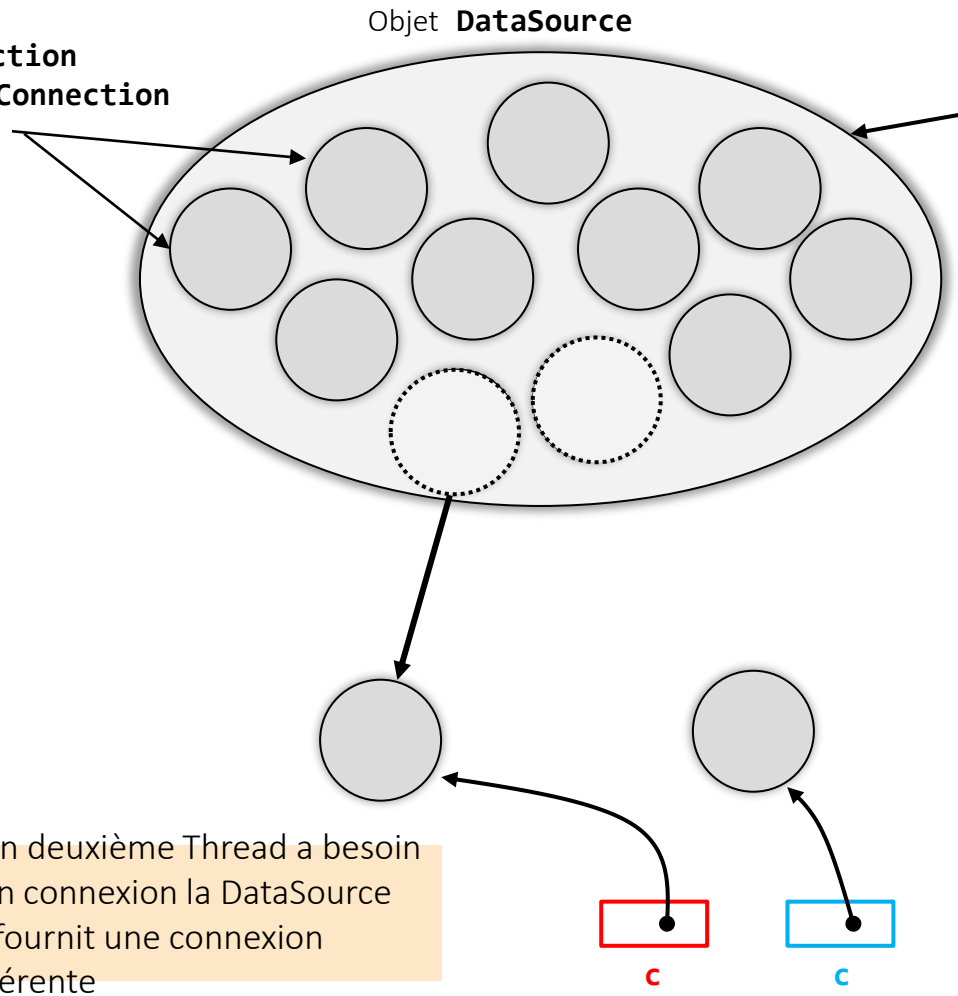
```
...
Connection c = ds.getConnnection();
...
...
Statement s = c.createStatement();
...
ResultSet = s.executeQuery(sql);
...
...
c.close();
...
```

connexion retirée du pool et mise à disposition de l'application

Il peut ensuite l'utiliser comme n'importe quel objet **Connection**

- 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



cette variable est créée par le serveur d'application (Tomcat) et peut être partagée par des threads différents

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

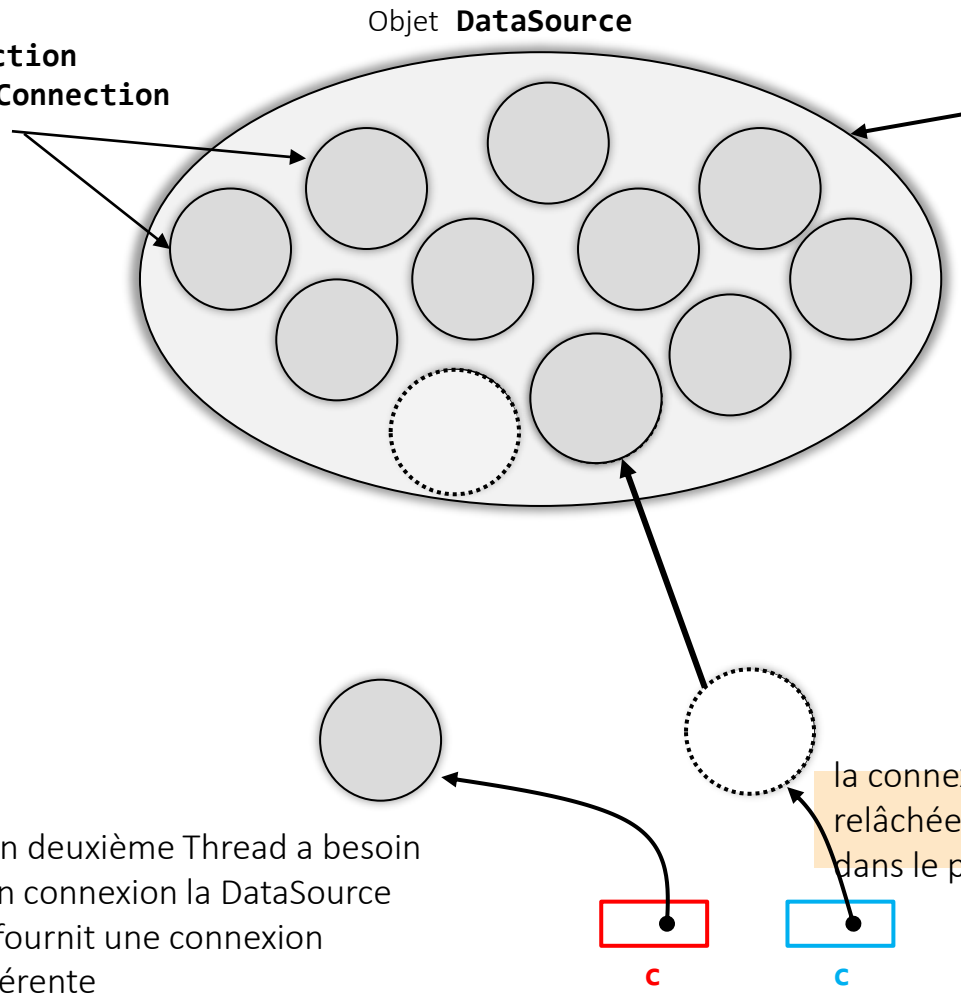
```
...
Connection c = ds.getConnnection();
...
...
Statement s = c.createStatement();
...
ResultSet = s.executeQuery(sql);
...
...
c.close();
...
```

connexion retirée du pool et mise à disposition de l'application
Il peut ensuite l'utiliser comme n'importe quel objet **Connection**

si un deuxième Thread a besoin d'un connexion la DataSource lui fournit une connexion différente

- 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



cette variable est créée par le serveur d'application (Tomcat) et peut être partagée par des threads différents

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

```
...
Connection c = ds.getConnnection();
...
...
Statement s = c.createStatement();
...
ResultSet = s.executeQuery(sql);
...
...
c.close();
...
```

connexion retirée du pool et mise à disposition de l'application
Il peut ensuite l'utiliser comme n'importe quel objet **Connection**

Il conserve la connexion jusqu'à sa "fermeture" explicite.

la connexion est relâchée et remise dans le pool.

si un deuxième Thread a besoin d'un connexion la DataSource lui fournit une connexion différente

- 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/*
  - Fameworks de persistance ou de mapping O/R (ORM)
    - *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*

- documentation : https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#jdbc

3. Data Access with JDBC

The value provided by the Spring Framework JDBC abstraction is perhaps best shown by the sequence of actions outlined in the following table below. The table shows which actions Spring takes care of and which actions are your responsibility.

Table 4. Spring JDBC - who does what?

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and run the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, the statement, and the resultset.	X	

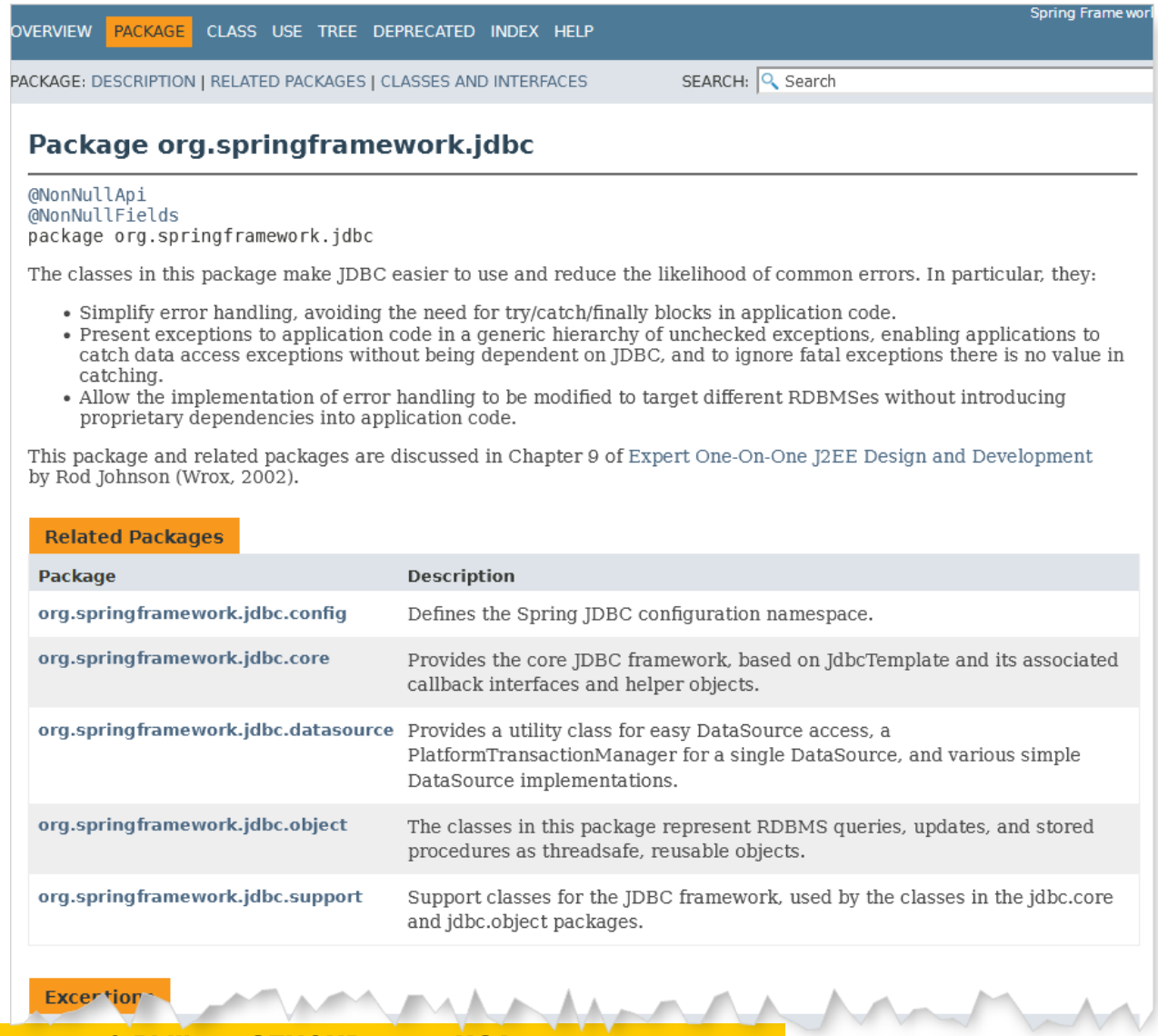
The Spring Framework takes care of all the low-level details that can make JDBC such a tedious API.

3.1. Choosing an Approach for JDBC Database Access

You can choose among several approaches to form the basis for your JDBC database access. In addition to three flavors of `JdbcTemplate`, a new `SimpleJdbcInsert` and `SimpleJdbcCall` approach optimizes database metadata, and the RDBMS Object style takes a more object-oriented approach similar to that of JDO Query design. Once you start using one of these approaches, you can still mix and match to include a feature from a different approach. All approaches require a JDBC 2.0-compliant driver, and some advanced features require a JDBC 3.0 driver.

- `JdbcTemplate` is the classic and most popular Spring JDBC approach. This "lowest-level" approach and all others use a JdbcTemplate under the covers.
- `NamedParameterJdbcTemplate` wraps a `JdbcTemplate` to provide named parameters instead of the traditional JDBC `?` placeholders. This approach provides better documentation and ease of use when you have multiple parameters for an SQL statement.
- `SimpleJdbcInsert` and `SimpleJdbcCall` optimize database metadata to limit the amount of necessary configuration. This approach simplifies coding so that you need to provide only the name of the table or procedure and provide a map of parameters matching the column names. This works only if the database provides adequate metadata. If the database does not provide this metadata, you have to provide explicit configuration of the parameters.
- RDBMS objects — including `MappingSqlQuery`, `SqlUpdate`, and `StoredProcedure` — require you to create reusable and thread-safe objects during initialization of your data-access layer. This approach is modeled after JDO Query, wherein you define your query string, declare parameters, and compile the query. Once you do that, `execute(…)`, `update(…)`, and `findObject(…)` methods can be called multiple times with various parameter values.

- Packages https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/package-summary.html



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP Spring Framework

PACKAGE: DESCRIPTION | RELATED PACKAGES | CLASSES AND INTERFACES SEARCH: Search

Package org.springframework.jdbc

```
@NonNullApi
@NonNullFields
package org.springframework.jdbc
```

The classes in this package make JDBC easier to use and reduce the likelihood of common errors. In particular, they:

- Simplify error handling, avoiding the need for try/catch/finally blocks in application code.
- Present exceptions to application code in a generic hierarchy of unchecked exceptions, enabling applications to catch data access exceptions without being dependent on JDBC, and to ignore fatal exceptions there is no value in catching.
- Allow the implementation of error handling to be modified to target different RDBMSes without introducing proprietary dependencies into application code.

This package and related packages are discussed in Chapter 9 of Expert One-On-One J2EE Design and Development by Rod Johnson (Wrox, 2002).

Related Packages

Package	Description
org.springframework.jdbc.config	Defines the Spring JDBC configuration namespace.
org.springframework.jdbc.core	Provides the core JDBC framework, based on JdbcTemplate and its associated callback interfaces and helper objects.
org.springframework.jdbc.datasource	Provides a utility class for easy DataSource access, a PlatformTransactionManager for a single DataSource, and various simple DataSource implementations.
org.springframework.jdbc.object	The classes in this package represent RDBMS queries, updates, and stored procedures as threadsafe, reusable objects.
org.springframework.jdbc.support	Support classes for the JDBC framework, used by the classes in the jdbc.core and jdbc.object packages.

Exceptions

- JDBC Template

https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jdbc/core/JdbcTemplate.html

OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP

Spring Framework

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

SEARCH: Search

Package org.springframework.jdbc.core

Class JdbcTemplate

java.lang.Object
  org.springframework.jdbc.support.JdbcAccessor
    org.springframework.jdbc.core.JdbcTemplate

All Implemented Interfaces:

InitializingBean, JdbcOperations

```
public class JdbcTemplate
extends JdbcAccessor
implements JdbcOperations
```

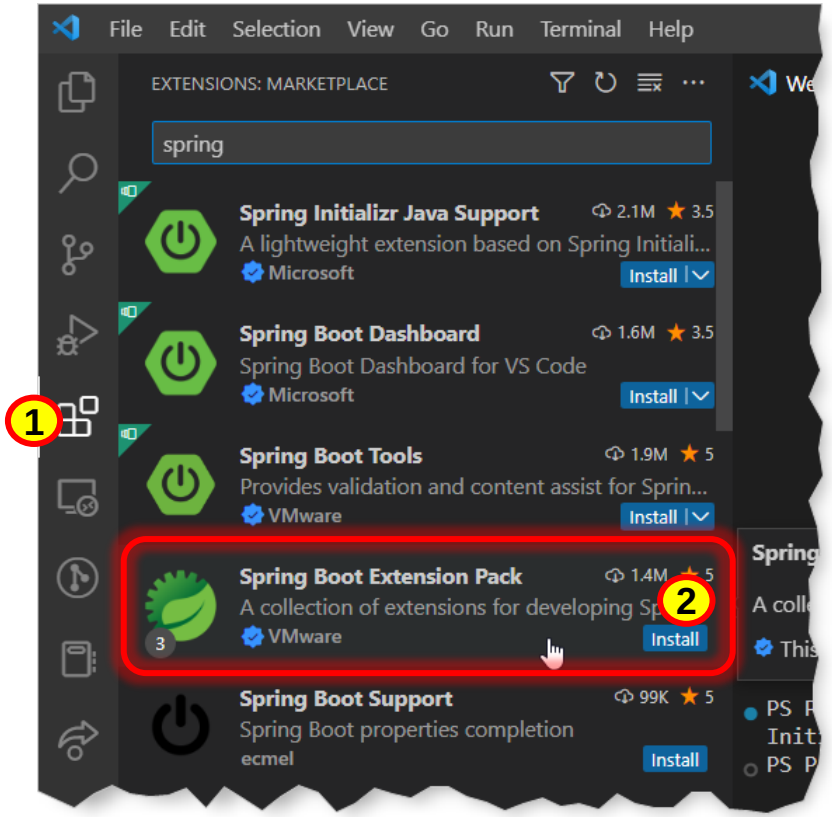
This is the central class in the JDBC core package. It simplifies the use of JDBC and helps to avoid common errors. It executes core JDBC workflow, leaving application code to provide SQL and extract results. This class executes SQL queries or updates, initiating iteration over ResultSets and catching JDBC exceptions and translating them to the generic, more informative exception hierarchy defined in the `org.springframework.dao` package.

Code using this class need only implement callback interfaces, giving them a clearly defined contract. The `PreparedStatementCreator` callback interface creates a prepared statement given a Connection, providing SQL and any necessary parameters. The `ResultSetExtractor` interface extracts values from a ResultSet. See also `PreparedStatementSetter` and `RowMapper` for two popular alternative callback interfaces.

Can be used within a service implementation via direct instantiation with a DataSource reference, or get prepared in an

- avec Spring Boot initializer
  - https://start.spring.io/

- dans VSCode : extension SpringBoot



JDBC Template exemple

- https://gricad-gitlab.univ-grenoble-alpes.fr/enseignement1/m2cci/awa/pi-poc