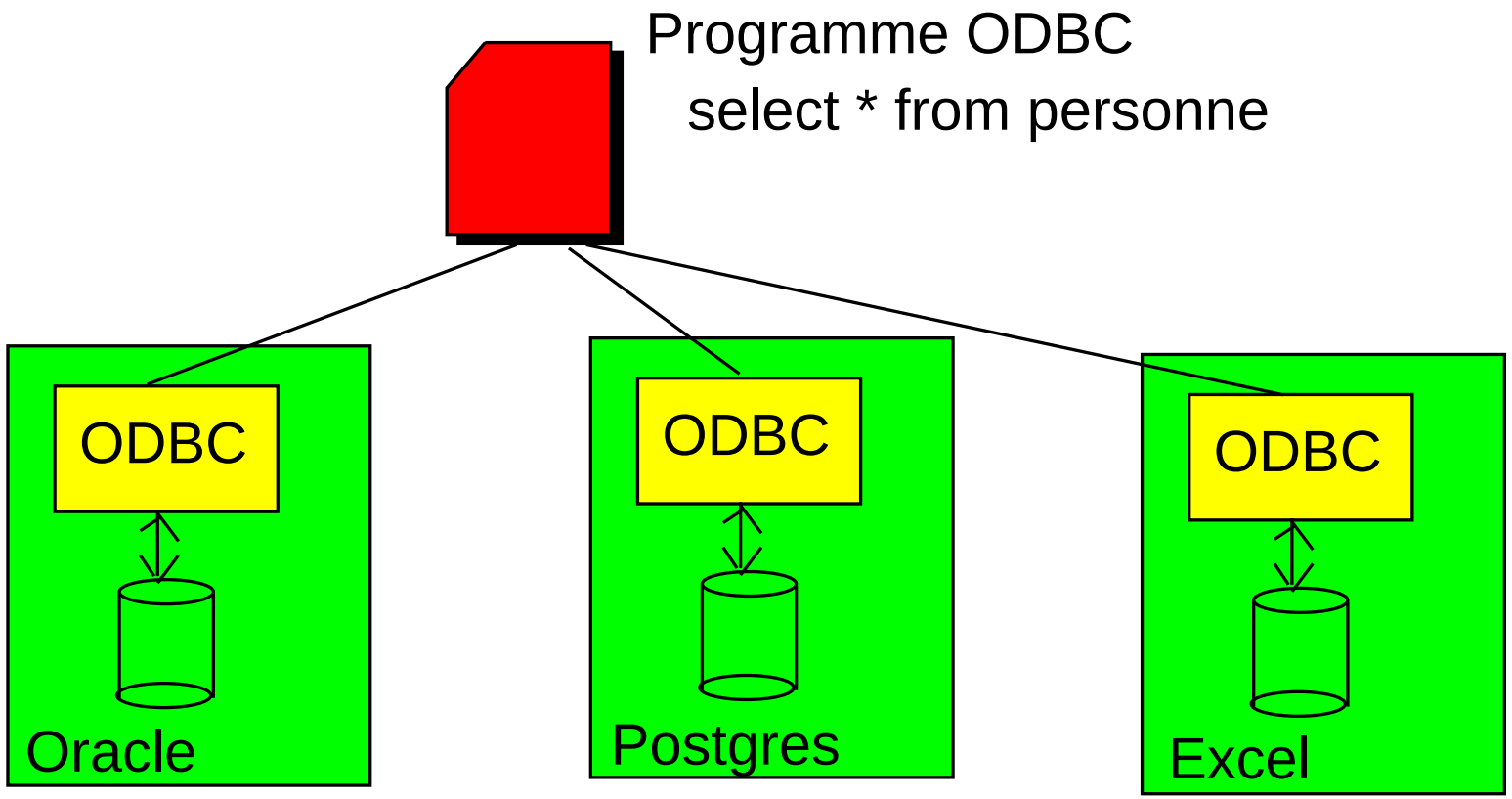
Le protocole JDBC (Java DataBase Connectivity)

Olivier Caron

Polytech Lille
Avenue Paul Langevin Cité Scientifique
Université de Lille
59655 Villeneuve d'Ascq cedex

http://ocaron.polytech-lille.net
Olivier.Caron@polytech-lille.fr

L'ancêtre ODBC



JDBC

- Langage Java (multi-plateforme :-)

[1] Un fichier `First.java` contenant les exemples JDBC du cours est disponible sur mes pages webs

JDBC

- Langage Java (multi-plateforme :-)
- Interface de programmation (API)[1]

[1] Un fichier `First.java` contenant les exemples JDBC du cours est disponible sur mes pages webs

JDBC

- Langage Java (multi-plateforme :-)
- Interface de programmation (API)[1]
- Adopté et implémenté par presque tous les constructeurs

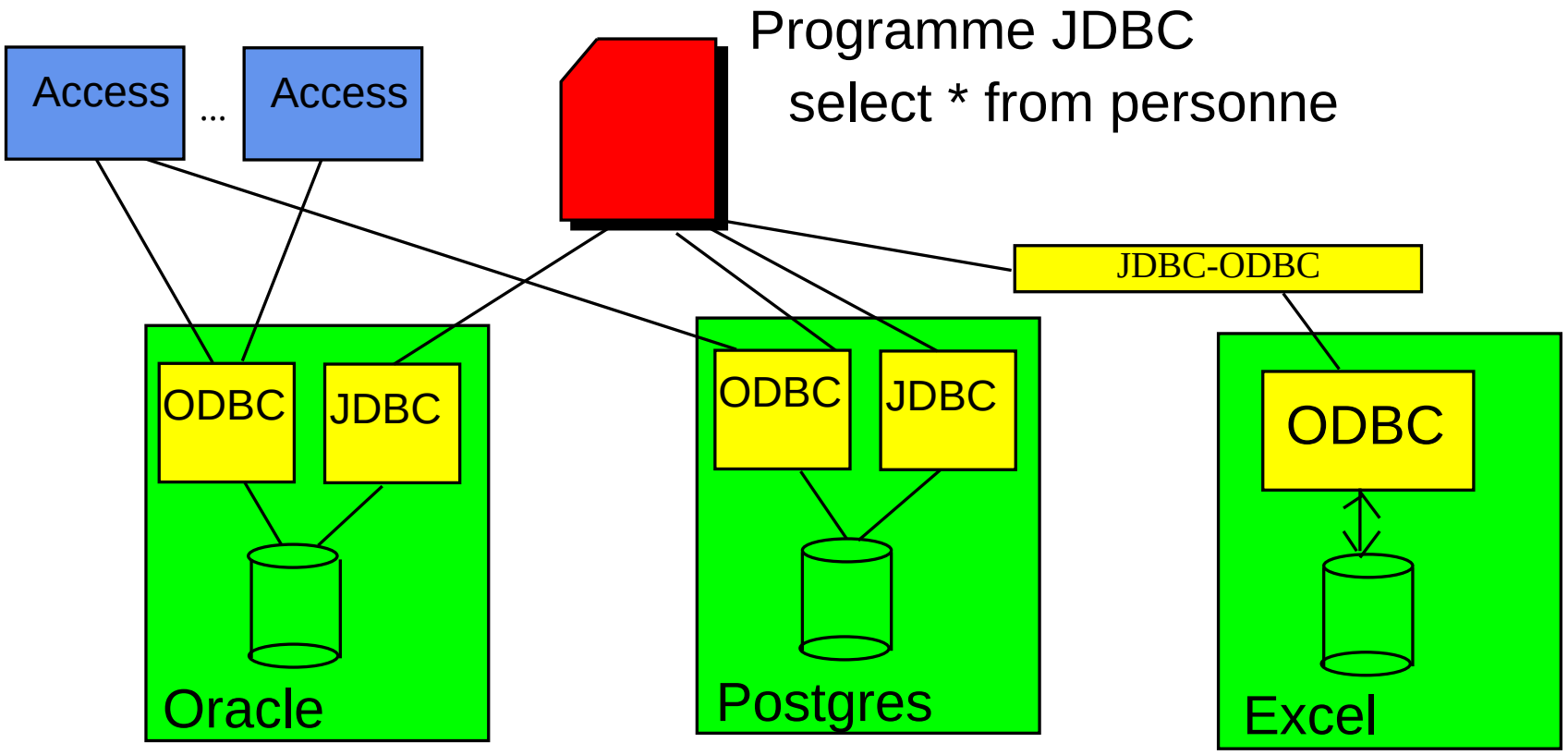
[1] Un fichier `First.java` contenant les exemples JDBC du cours est disponible sur mes pages webs

JDBC

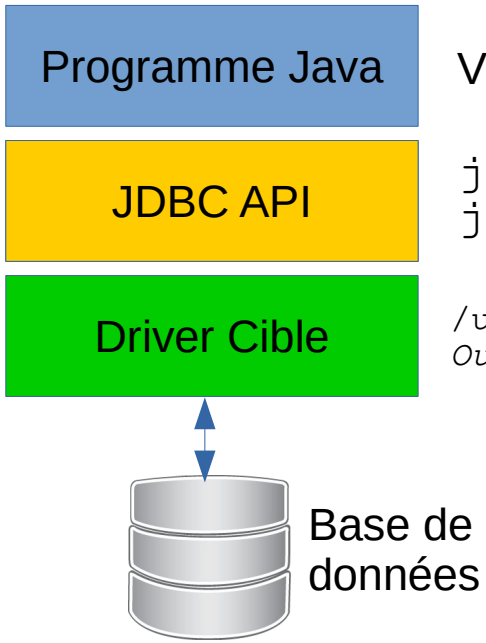
- Langage Java (multi-plateforme :-)
- Interface de programmation (API)[1]
- Adopté et implémenté par presque tous les constructeurs
- Passerelle ODBC-JDBC

[1] Un fichier `First.java` contenant les exemples JDBC du cours est disponible sur mes pages webs

Des utilisations possibles des protocoles



Structuration d'un programme JDBC



Votre code Java

`java.sql.*` et/ou
`javax.sql.*`

`/usr/share/java/postgresql-jdbc4.jar`
Ou `/usr/share/java/postgresql.jar`

Compilation, exécution d'un programme Java-JDBC :

```
javac progs/First.java
```

```
java -cp /usr/share/java/postgresql.jar:. progs.First
```

Déroulement d'un programme JDBC

- Déroulement classique :

Déroulement d'un programme JDBC

- Déroulement classique :
  1. Connexion à une base

Déroulement d'un programme JDBC

- Déroulement classique :
  1. Connexion à une base
  2. Exécution de requêtes et récupération des données

Déroulement d'un programme JDBC

- Déroulement classique :
  1. Connexion à une base
  2. Exécution de requêtes et récupération des données
  3. Déconnexion

Déroulement d'un programme JDBC

- Déroulement classique :
  1. Connexion à une base
  2. Exécution de requêtes et récupération des données
  3. Déconnexion
- Plein d'autres variantes (ex : accès multi-bases)

Établissement d'une connexion

- Pour se connecter à une base, il faut fournir les éléments suivants :

Établissement d'une connexion

- Pour se connecter à une base, il faut fournir les éléments suivants :
  1. L'adresse du serveur de base de données (son nom de machine)

Établissement d'une connexion

- Pour se connecter à une base, il faut fournir les éléments suivants :
  1. L'adresse du serveur de base de données (son nom de machine)
  2. Le type du serveur de base de données (postgresql, mysql, oracle,. . .)

Établissement d'une connexion

- Pour se connecter à une base, il faut fournir les éléments suivants :
  1. L'adresse du serveur de base de données (son nom de machine)
  2. Le type du serveur de base de données (postgresql, mysql, oracle,. . .)
  3. Le nom de la base

Établissement d'une connexion

- Pour se connecter à une base, il faut fournir les éléments suivants :
  1. L'adresse du serveur de base de données (son nom de machine)
  2. Le type du serveur de base de données (postgresql, mysql, oracle,. . .)
  3. Le nom de la base
  4. Les nom et mot de passe de l'utilisateur référencé du serveur

Établissement d'une connexion

- Pour se connecter à une base, il faut fournir les éléments suivants :
  1. L'adresse du serveur de base de données (son nom de machine)
  2. Le type du serveur de base de données (postgresql, mysql, oracle,. . .)
  3. Le nom de la base
  4. Les nom et mot de passe de l'utilisateur référencé du serveur
- Les éléments 1 à 3 seront fournis via une URL de la forme :

  `jdbc:typeServeur://adresseServeur/nomBaseDonnees`

  Exemple :

  `jdbc:postgresql://serveur-etu.polytech-lille.fr/videoclub`

Établissement d'une connexion

- Pour se connecter à une base, il faut fournir les éléments suivants :
  1. L'adresse du serveur de base de données (son nom de machine)
  2. Le type du serveur de base de données (postgresql, mysql, oracle,. . .)
  3. Le nom de la base
  4. Les nom et mot de passe de l'utilisateur référencé du serveur
- Les éléments 1 à 3 seront fournis via une URL de la forme :

  `jdbc:typeServeur://adresseServeur/nomBaseDonnees`

  Exemple :

  `jdbc:postgresql://serveur-etu.polytech-lille.fr/videoclub`
- Ces données seront idéalement externalisées du code source Java dans un fichier de propriétés.

Fichier de propriétés

- Un fichier de propriétés permet d'avoir un code évolutif sans recompilation
- Soit le fichier de propriétés de nom "`database.properties`":

```
jdbc.serverType=postgresql
jdbc.serverHost=serveur-etu.polytech-lille.fr
jdbc.dbName=test
jdbc.user=admin
jdbc.password=secret
```

Exemple d'une connexion JDBC (1/2)

```
package progs ;

import java.sql.* ;
import java.io.* ;
import java.util.Properties ;
import javax.sql.* ;
import javax.sql.rowset.* ;

public class First {
 public static void main(String args[]) {
  try {
   Properties props= new Properties() ;
   FileInputStream in=new FileInputStream("database.properties") ;
   props.load(in) ;
   String url = "jdbc:"+props.getProperty("jdbc.serverType") +"://"+
                props.getProperty("jdbc.serverHost")+"/"+
                props.getProperty("jdbc.dbName") ;
```

Exemple d'une connexion JDBC (2/2)

```
1       String user=props.getProperty("jdbc.user") ;
2       String password=props.getProperty("jdbc.password") ;
3
4       Connection db = DriverManager.getConnection(url,user,password);
5       System.out.println("Connexion établie") ;
6       ...
7       } catch(FileNotFoundException e1) {
8        System.err.println("Fichier de propriétés non trouvé") ;
9       } catch(IOException e2) {
10       System.err.println("Erreur lecture fichier propriété") ;
11      } catch (SQLException e3) {
12       System.err.println("Erreur SQL") ;
13       e3.printStackTrace() ;
14      }
15  }
```

L'interface `PreparedStatement`

- Le programme précédent a permis d'obtenir une variable `db` de type `Connection` (ligne 4, diapositive précédente).

L'interface `PreparedStatement`

- Le programme précédent a permis d'obtenir une variable `db` de type `Connection` (ligne 4, diapositive précédente).
- Un objet de type `PreparedStatement` est nécessaire pour définir des requêtes SQL, fournir des paramètres puis exécuter et recevoir éventuellement des données issues de l'exécution de ces requêtes.

L'interface `PreparedStatement`

- Le programme précédent a permis d'obtenir une variable `db` de type `Connection` (ligne 4, diapositive précédente).
- Un objet de type `PreparedStatement` est nécessaire pour définir des requêtes SQL, fournir des paramètres puis exécuter et recevoir éventuellement des données issues de l'exécution de ces requêtes.
- Les données issues de requêtes `select` sont stockées dans des objets de type `ResultSet`.

L'interface `PreparedStatement`

- Le programme précédent a permis d'obtenir une variable `db` de type `Connection` (ligne 4, diapositive précédente).
- Un objet de type `PreparedStatement` est nécessaire pour définir des requêtes SQL, fournir des paramètres puis exécuter et recevoir éventuellement des données issues de l'exécution de ces requêtes.
- Les données issues de requêtes `select` sont stockées dans des objets de type `ResultSet`.
- La fermeture d'un `PreparedStatement` engendre la fermeture automatique de tous les `ResultSet` associés.

`PreparedStatement` : requêtes de modification

- Toutes les méthodes doivent prendre en compte `SQLException`
- La méthode `prepareStatement` permet d'obtenir un objet de type `PreparedStatement` à partir d'un objet de type `Connection` (ligne 3)
- La méthode `int executeUpdate(String requeteSQL)` exécute toute requête SQL (hors `select`) et retourne le nombre de **lignes** concernés par la commande (parfois 0 pour certaines commandes).

```
1  String SQLRequestCreate="create table tester "+
2                          "(num   integer primary key, ch text)";
3  stmt = db.prepareStatement(SQLRequestCreate) ;
4  stmt.executeUpdate() ; // cette exécution retourne 0
5  stmt.close();
```

PreparedStatement : passage de paramètres

- L'expression de la requête SQL permet de spécifier des paramètres (caractère '?' - cf ligne 1)
- Les méthodes `setType(int numParam, Type valeur)` permettent de fixer les valeurs.
  Le premier paramètre a le numéro 1,... (lignes 3 et 6)
- Permet d'exécuter efficacement un flot de requêtes similaires

```
1  String SQLRequestInsert="insert into tester values (?,?)" ;
2  stmt = db.prepareStatement(SQLRequestInsert) ;
3  stmt.setInt(1,1) ; stmt.setString(2,"durant") ;
4  int rowsAffected = stmt.executeUpdate() ;
5  System.out.println (rowsAffected+" ligne insérée") ;
6  stmt.setInt(1,2) ; stmt.setString(2,"dubois") ;
7  rowsAffected= stmt.executeUpdate() ;
8  System.out.println (rowsAffected+" ligne insérée") ;
9  stmt.close();
```

Consultation de la base

- Le processus de consultation (et même de modification) est le suivant :

Consultation de la base

- Le processus de consultation (et même de modification) est le suivant :
  1. Création d'un objet de type `PreparedStatement`. Selon les paramètres fournis lors de cette étape, les possibilités de lecture des données reçues ainsi que d'écriture diffèrent.

Consultation de la base

- Le processus de consultation (et même de modification) est le suivant :
  1. Création d'un objet de type `PreparedStatement`. Selon les paramètres fournis lors de cette étape, les possibilités de lecture des données reçues ainsi que d'écriture diffèrent.
  2. Exécution d'une requête SQL de type `Select`, données stockées dans un objet de type `ResultSet`.

Consultation de la base

- Le processus de consultation (et même de modification) est le suivant :
  1. Création d'un objet de type `PreparedStatement`. Selon les paramètres fournis lors de cette étape, les possibilités de lecture des données reçues ainsi que d'écriture diffèrent.
  2. Exécution d'une requête SQL de type `Select`, données stockées dans un objet de type `ResultSet`.
  3. Consultation/modification des données d'un `ResultSet`

Paramétrage de `PreparedStatement` (1/2)

- La méthode générale de création d'un `PreparedStatement` est :

```
PreparedStatement prepareStatement(String sql,
int resultSetType, int resultSetConcurrency)
```

Paramétrage de `PreparedStatement` (1/2)

- La méthode générale de création d'un `PreparedStatement` est :

```
PreparedStatement prepareStatement(String sql,
int resultSetType, int resultSetConcurrency)
```

- Les valeurs possibles de `resultSetType` :

Paramétrage de `PreparedStatement` (1/2)

- La méthode générale de création d'un `PreparedStatement` est :

  ```
  PreparedStatement prepareStatement(String sql,
  int resultSetType, int resultSetConcurrency)
  ```

- Les valeurs possibles de `resultSetType` :
  - **`ResultSet.Type_FORWARD_ONLY` (par défaut)** la lecture des données issues de la requête ne pourra se faire que de la première ligne vers la dernière. Les données sont celles de l'état de la base au moment de l'exécution de la requête (pas de rafraîchissement possible).

Paramétrage de `PreparedStatement` (1/2)

- La méthode générale de création d'un `PreparedStatement` est :

  ```
  PreparedStatement prepareStatement(String sql,
  int resultSetType, int resultSetConcurrency)
  ```

- Les valeurs possibles de `resultSetType` :
  - **`ResultSet.Type_FORWARD_ONLY` (par défaut)** la lecture des données issues de la requête ne pourra se faire que de la première ligne vers la dernière. Les données sont celles de l'état de la base au moment de l'exécution de la requête (pas de rafraîchissement possible).
  - **`ResultSet.Type_SCROLL_INSENSITIVE`** la lecture peut se faire en avant et/ou arrière. Pas de rafraîchissement possible des données.

Paramétrage de `PreparedStatement` (1/2)

- La méthode générale de création d'un `PreparedStatement` est :

```
PreparedStatement prepareStatement(String sql,
int resultSetType, int resultSetConcurrency)
```

- Les valeurs possibles de `resultSetType` :
  - **`ResultSet.Type_FORWARD_ONLY` (par défaut)** la lecture des données issues de la requête ne pourra se faire que de la première ligne vers la dernière. Les données sont celles de l'état de la base au moment de l'exécution de la requête (pas de rafraîchissement possible).
  - **`ResultSet.Type_SCROLL_INSENSITIVE`** la lecture peut se faire en avant et/ou arrière. Pas de rafraîchissement possible des données.
  - **`ResultSet.Type_SCROLL_SENSITIVE`** la lecture peut se faire en avant et/ou arrière. Rafraîchissement possible des données.

Paramétrage de `PreparedStatement` (2/2)

- Les valeurs possibles de `resultSetConcurrency` :

Paramétrage de `PreparedStatement` (2/2)

- Les valeurs possibles de `resultSetConcurrency` :
  - **`ResultSet.CONCUR_READ_ONLY` (par défaut)** les données du `ResultSet` ne sont pas modifiables

Paramétrage de `PreparedStatement` (2/2)

- Les valeurs possibles de `resultSetConcurrency` :
  - **`ResultSet.CONCUR_READ_ONLY` (par défaut)** les données du `ResultSet` ne sont pas modifiables
  - **`ResultSet.CONCUR_UPDATABLE`** les données du `ResultSet` sont modifiables et synchronisables avec la base.

Paramétrage de `PreparedStatement` (2/2)

- Les valeurs possibles de `resultSetConcurrency` :
  - **`ResultSet.CONCUR_READ_ONLY` (par défaut)** les données du `ResultSet` ne sont pas modifiables
  - **`ResultSet.CONCUR_UPDATABLE`** les données du `ResultSet` sont modifiables et synchronisables avec la base.
- La méthode `stmt = db.prepareStatement(sql)` est un raccourci pour :

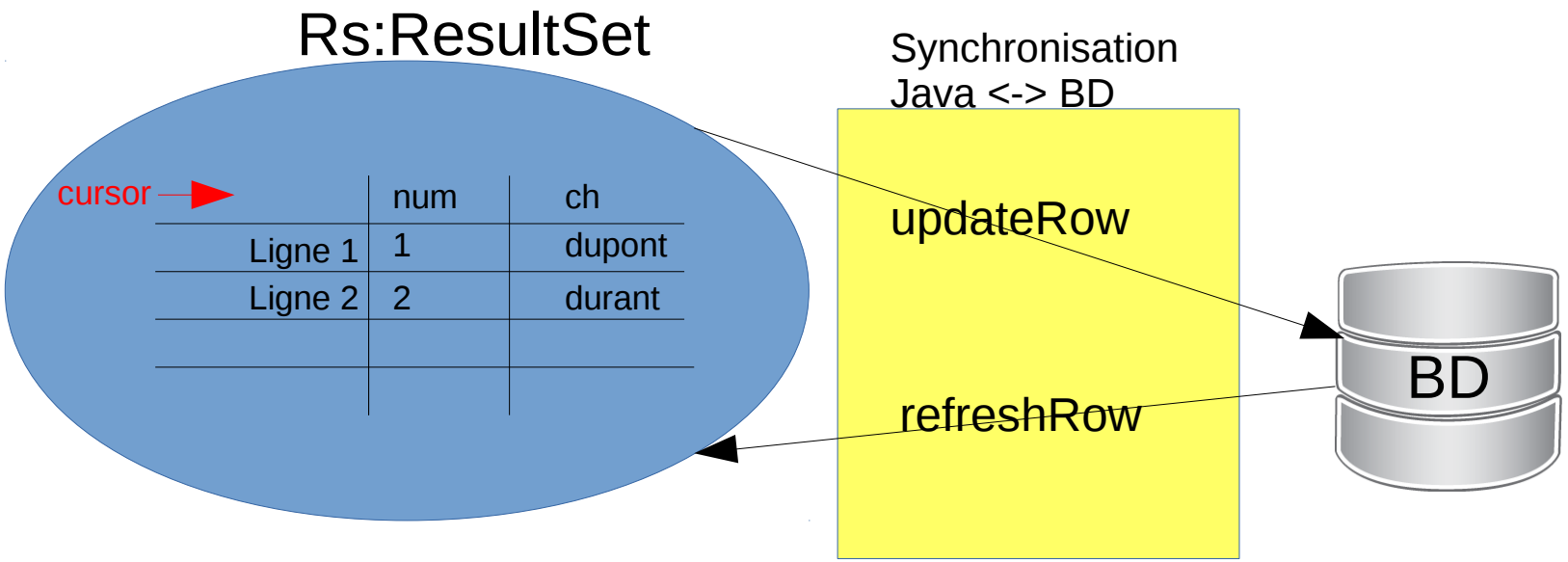
```
stmt = db.prepareStatement(sql, ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_READ_ONLY)
```

Exécution d'une requête `select`

- Utilisation de la méthode `executeQuery` de la classe `PreparedStatement`.

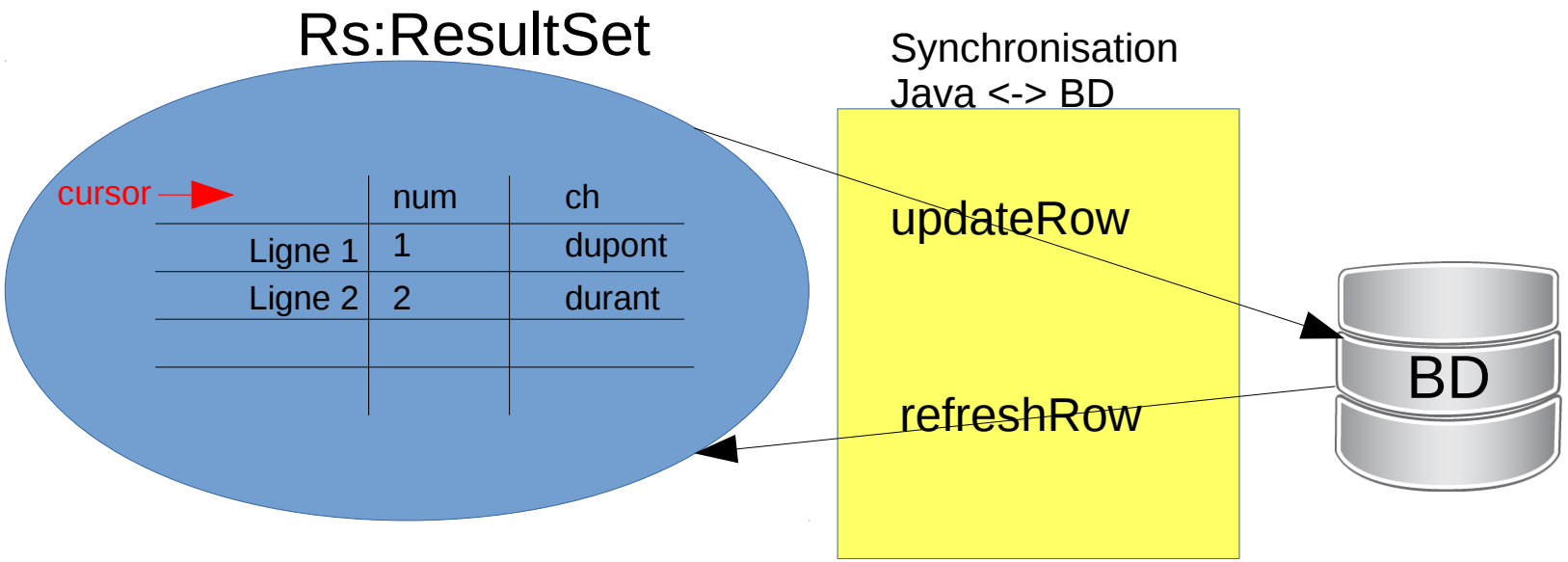
```
String SQLrequestSelect = "select num, ch from tester";
stmt = db.prepareStatement(SQLrequestSelect,
                           ResultSet.TYPE_SCROLL_SENSITIVE,
                           ResultSet.CONCUR_UPDATABLE) ;
ResultSet rs=stmt.executeQuery() ;
```

L'objet de type `ResultSet`



- Gestion d'un curseur pour parcourir les lignes résultats de la requête.

L'objet de type `ResultSet`



- Gestion d'un curseur pour parcourir les lignes résultats de la requête.
- Après initialisation de l'objet, le curseur se trouve **avant** la première ligne

`ResultSet` : gestion du curseur (1/2)

- Plusieurs méthodes booléennes pour **déplacer** le curseur.

`ResultSet` : gestion du curseur (1/2)

- Plusieurs méthodes booléennes pour **déplacer** le curseur.
- En mode `ResultSet.TYPE_FORWARD_ONLY`, seule la méthode `next()` est possible

`ResultSet` : gestion du curseur (1/2)

- Plusieurs méthodes booléennes pour **déplacer** le curseur.
- En mode `ResultSet.TYPE_FORWARD_ONLY`, seule la méthode `next()` est possible
- Ces méthodes retournent `false` si opération impossible (ex : fin de table), la première ligne est la ligne 1.

méthode	**action**
next()	déplacement vers la ligne suivante
previous()	déplacement vers la ligne précédente
absolute(i)	$i > 0$, aller à la i$^{\text{ème}}$ ligne
absolute(-i)	$i > 0$, aller à la i$^{\text{ème}}$ ligne en partant de la dernière
relative(i)	$i > 0$, descendre de i lignes
relative(-i)	$i > 0$, remonter de i lignes

`ResultSet` : gestion du curseur (2/2)

méthode	action
afterlast()	aller en fin de la table (**après** la dernière ligne)
beforeFirst()	aller en début de table (**avant** la première ligne)
last()	aller à la dernière ligne
first()	aller à la première ligne

Les méthodes booléennes suivantes ne déplacent pas le curseur mais testent sa position :

méthode	action
isLast()	vrai si curseur en dernière ligne
isFirst()	vrai si curseur en première ligne
isAfterLast()	vrai si curseur est après la dernière ligne
isBeforeFirst()	vrai si curseur est avant la première ligne

`ResultSet` : lecture des lignes

- Plusieurs méthodes pour lire les données de la ligne courante (celle pointée par le curseur)

`ResultSet` : lecture des lignes

- Plusieurs méthodes pour lire les données de la ligne courante (celle pointée par le curseur)
- Deux modes : n° de colonne (indicée à partir de 1) ou nom de colonne

méthode	**action**
Date getDate(int i)	retourne l'objet Date de la colonne i
Date getDate(String col)	retourne l'objet Date de la colonne col
int getInt(int i)	retourne un entier de la colonne i
int getInt(String col)	retourne un entier de la colonne col
String getString(int i)	retourne l'objet String de la colonne i
String getString(String col)	retourne l'objet String de la colonne col
...	...

Exemple de consultation

```
int lineNumber = 1;
rs.beforeFirst() ;
while (rs.next()) {
   int num = rs.getInt("num");
   String ch = rs.getString("ch");
   System.out.println("ligne " + lineNumber + ": "
                      + num + ", " + ch);
   lineNumber++;
}
```

- `next()` est obligatoire même pour lire une seule ligne (ex : count)

ResultSet : modification de lignes

- Des nouvelles méthodes pour `ResultSet` :

```
void updateType(String nomColonne, type
nouvelleValeur)
void updateRow() ,void refreshRow()
void cancelRowUpdates()
```

- Un exemple :

```
rs.first() ; // retour première ligne
rs.refreshRow() ; // synchronisation BD -> Java
rs.updateString("ch","dupont") ;
rs.updateRow() ; // synchronisation Java -> BD
rs.updateString("ch","Caron") ;
rs.cancelRowUpdates(); // précédente ligne invalidée
```

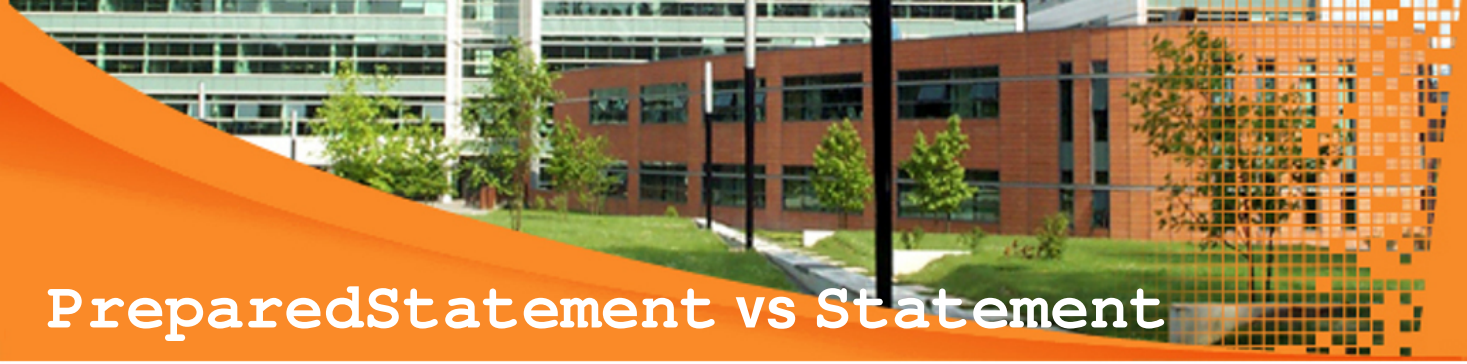
`ResultSet` : insertion de lignes

- Des nouvelles méthodes pour `ResultSet` :

```
void moveToInsertRow()
void moveToCurrentRow()
void insertRow()
```

- Un exemple :

```
// phase d'insertion
rs.moveToInsertRow();
rs.updateInt("num",3) ;
rs.updateString("ch", "garcia");
rs.insertRow(); // synchronisation Java -> BD
rs.moveToCurrentRow(); // retour au curseur avant insertion
```



PreparedStatement vs Statement

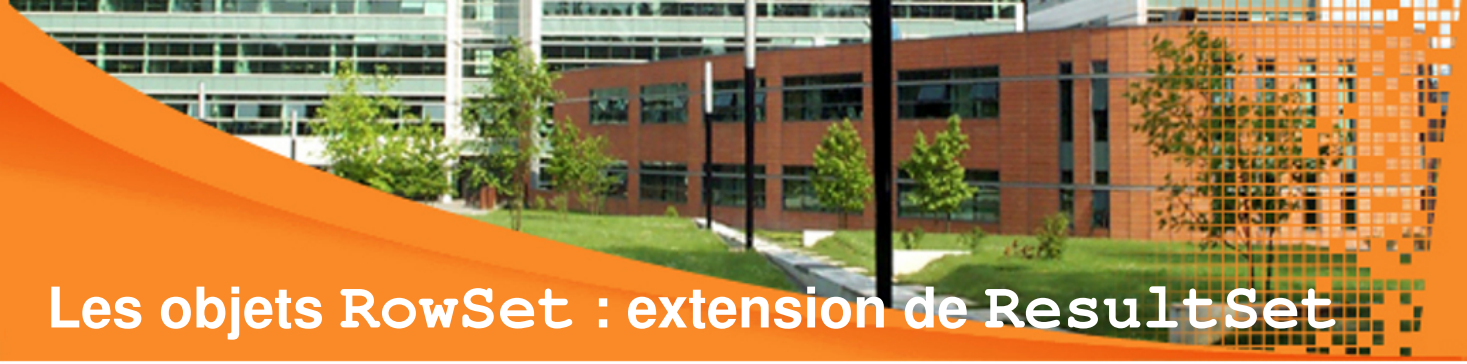
- `PreparedStatement` est une spécialisation de `Statement`
- `PreparedStatement` est plus performant, notamment lorsque la requête est exécutée plusieurs fois
- `PreparedStatement` est plus sécurisé, contrôle des valeurs de paramètres
- un exemple de `Statement` :

```
Statement stmt = db.createStatement();
stmt.executeUpdate("create table tester "+
                   " (num   integer primary key, ch text)") ;
int nb= stmt.executeUpdate("insert into tester values "+
                           "(1,'dupont'), (2,'durant')") ;
System.out.println (nb+" lignes insérées");
```

Fermeture de la connexion

- Fermer le dialogue SQL
- Fermer la session avec la base

```
stmt.close() ;
db.close() ;
```

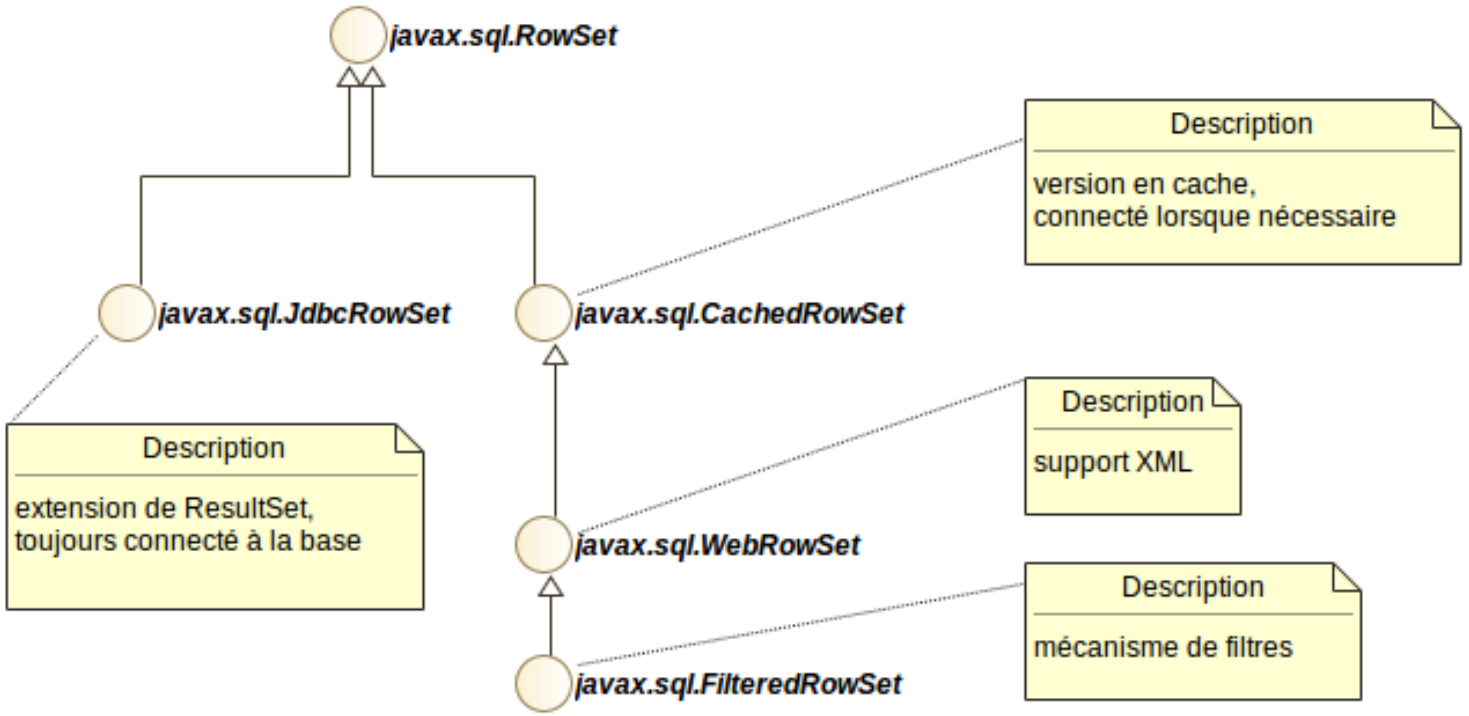


Les objets `RowSet` : extension de `ResultSet`

- JDBC 4 : fusion de `Connection`, `PreparedStatement` et `ResultSet`

Les objets `RowSet` : extension de `ResultSet`

- JDBC 4 : fusion de `Connection`, `PreparedStatement` et `ResultSet`
- Intègre mécanisme d'observation des données modifiées.



Exemple RowSet

```
JdbcRowSet jrs = RowSetProvider.newFactory().createJdbcRowSet() ;
jrs.setType(ResultSet.TYPE_SCROLL_SENSITIVE) ;
jrs.setUrl(url);
jrs.setUsername(user);
jrs.setPassword(password);
jrs.setCommand("SELECT * FROM tester WHERE ch= ?");
jrs.setString(1, "garcia");
jrs.execute();
```

Gestion des transactions

- Le mode par défaut est "auto-commit" : chaque requête SQL forme une transaction
- Initier une transaction, choix du mode d'isolation standard :

```
db.setAutoCommit(false) ;
db.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE) ;
```

- Valider une transaction :

```
stmt=db.prepareStatement("delete from emprunter where numl=?") ;
stmt.setInt(1,2) ; stmt.executeUpdate() ;
stmt.setInt(1,34) ; stmt.executeUpdate() ;
stmt.setInt(1,27) ; stmt.executeUpdate() ;

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

Annuler une transaction

- méthode `rollback()`

```
try {
 db.setAutoCommit(false) ; // début transaction
 db.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED) ;
 stmt=db.prepareStatement("insert into emprunter values (?,?)") ;
 stmt.setInt(1,numl) ; stmt.setInt(2,numu) ; stmt.executeUpdate() ;
 if (nbEmprunts >= 3) db.rollback() ;  // fin transaction, insert annulé
 else db.commit() ;  //fin transaction OK
 db.setAutoCommit(true) ;
} catch(SQLException e) {
 if (db !=null) {
   try { db.rollback() ; db.setAutoCommit(true) ;
   } catch(SQLException ex) {
   System.err.println("on est mal !") ;
 }
}
```

Prise en compte de SQL 3

- SQL 3 : unification de concepts orienté-objet et le monde relationnel

Prise en compte de SQL 3

- SQL 3 : unification de concepts orienté-objet et le monde relationnel
- Des colonnes de type non simple :

Prise en compte de SQL 3

- SQL 3 : unification de concepts orienté-objet et le monde relationnel
- Des colonnes de type non simple :
  - Des tableaux (`rs.getArray("nomColonne")`)

Prise en compte de SQL 3

- SQL 3 : unification de concepts orienté-objet et le monde relationnel
- Des colonnes de type non simple :
  - Des tableaux (`rs.getArray("nomColonne")`)
  - Des objets (`rs.getBlob("nomColonne")`)

Le niveau Meta

Une meta-donnée est une donnée qui décrit une donnée

- Des Exemples :

Le niveau Meta

Une meta-donnée est une donnée qui décrit une donnée

- Des Exemples :
  - Modèle Relationnel : table(colonne1 type1, . . .)

Le niveau Meta

Une meta-donnée est une donnée qui décrit une donnée

- Des Exemples :
  - Modèle Relationnel : table(colonne1 type1, . . .)
  - Postgres pg_database, pg_user, pg_class,. . .

Le niveau Meta

Une meta-donnée est une donnée qui décrit une donnée

- Des Exemples :
  - Modèle Relationnel : table(colonne1 type1, . . .)
  - Postgres pg_database, pg_user, pg_class,. . .
  - Java : getClass() , getMethods, getFields(), . . .

Le niveau Meta

Une meta-donnée est une donnée qui décrit une donnée

- Des Exemples :
  - Modèle Relationnel : table(colonne1 type1, . . .)
  - Postgres pg_database, pg_user, pg_class,. . .
  - Java : getClass() , getMethods, getFields(), . . .
- JDBC propose une API pour analyser une base (introspection)

JDBC et le niveau Meta

- Analyser dynamiquement la structure d'une table résultat

l'interface ResultSetMetaData

int getColumnCount()	le nombre de colonnes
int getColumnDisplaySize(int column)	taille d'affichage d'une col
String getColumnLabel(int column)	nom suggéré d'une colonne
String getColumnName(int column)	nom de colonne
int getColumnType(int column)	type (cste) de la colonne
String getColumnTypeName(int column)	nom du type de la colonne

Un exemple d'analyse dynamique de table

```
stmt=db.prepareStatement(uneRequeteSQL);
rs = stmt.executeQuery();
rs.last(); int nbLignes = rs.getRow(); rs.beforeFirst();
ResultSetMetaData rsmd = rs.getMetaData() ;
String colName[] = new String[rsmd.getColumnCount()] ;
for (int i=0;i<rsmd.getColumnCount();i++) { // gestion des noms de col
   colName[i]= rsmd.getColumnLabel(i+1);
Object resultat[][]= new Object[rsmd.getColumnCount()][nbLignes] ;
int lig=0 ;
while (rs.next()) {
  for (int i=0;i<colName.length;i++)
  switch (rsmd.getColumnType(i+1)) {
   case Types.INTEGER: resultat[i][lig]=new Integer(rs.getInt(colName[i]));
    break ;
   case Types.VARCHAR: resultat[i][lig]=new String(rs.getString(colName[i]));
    break ;
    ...
  }
}
```

Analyse dynamique de la base

- L'interface DatabaseMetaData

Analyse dynamique de la base

- L'interface DatabaseMetaData
  - API très riche (et donc complexe)

Analyse dynamique de la base

- L'interface DatabaseMetaData
  - API très riche (et donc complexe)
  - Permet de connaître les possibilités du SGBD

Analyse dynamique de la base

- L'interface DatabaseMetaData
  - API très riche (et donc complexe)
  - Permet de connaître les possibilités du SGBD
  - Portable (de nombreux outils génériques)

Analyse dynamique de la base

- L'interface DatabaseMetaData
  - API très riche (et donc complexe)
  - Permet de connaître les possibilités du SGBD
  - Portable (de nombreux outils génériques)
- Permet de développer des outils génériques (aka PhpPgAdmin)

Conclusion

- API très riche, une petite partie de JDBC est présentée dans ce cours.

Conclusion

- API très riche, une petite partie de JDBC est présentée dans ce cours.
- Une API unique quelque soit le SGBD

Conclusion

- API très riche, une petite partie de JDBC est présentée dans ce cours.
- Une API unique quelque soit le SGBD
- En constante évolution : gestion de ”row sets”, gestion de blobs, support de SQL XML, programmation par annotations Java, . . . ,