JDBC
Java DataBase Connectivity

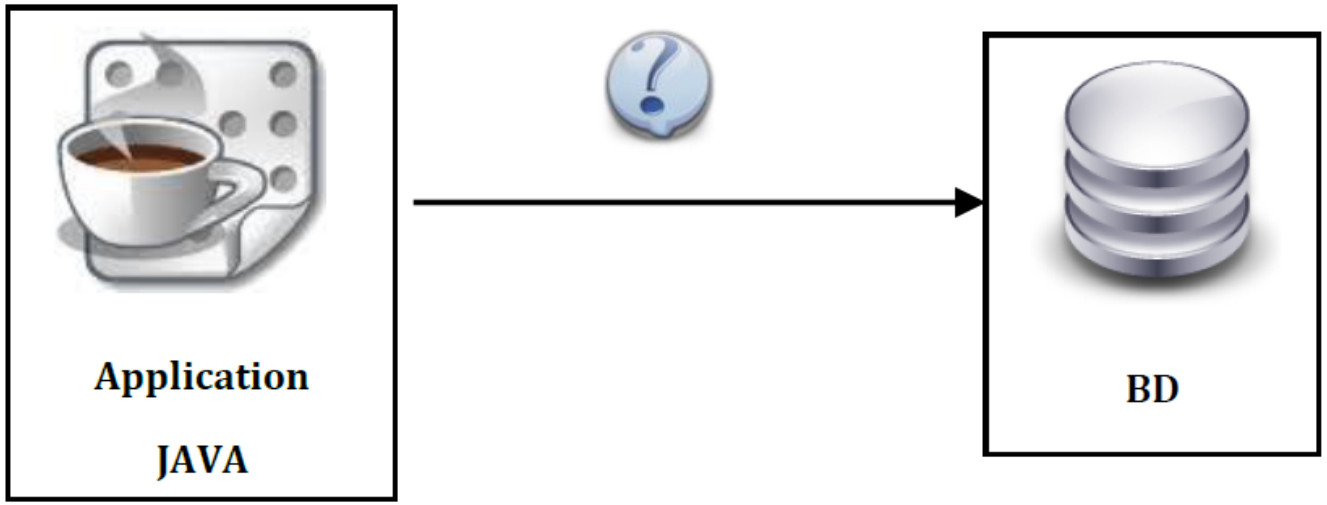
Anissa CHALOUAH

ISET Bizerte

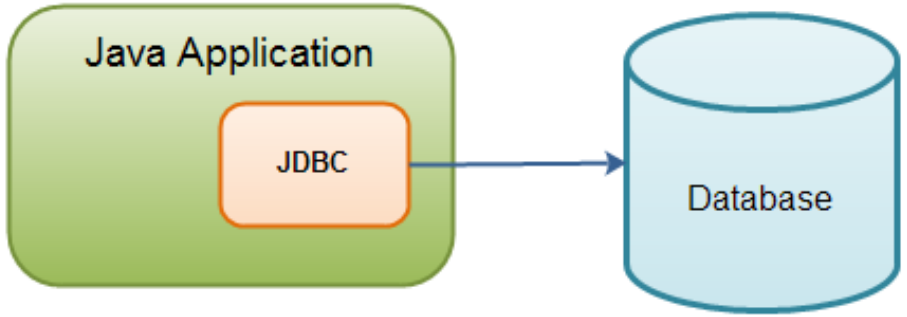
Introduction à JDBC

Problème

- Comment accéder à une base de données depuis une application Java.



JDBC : Définition

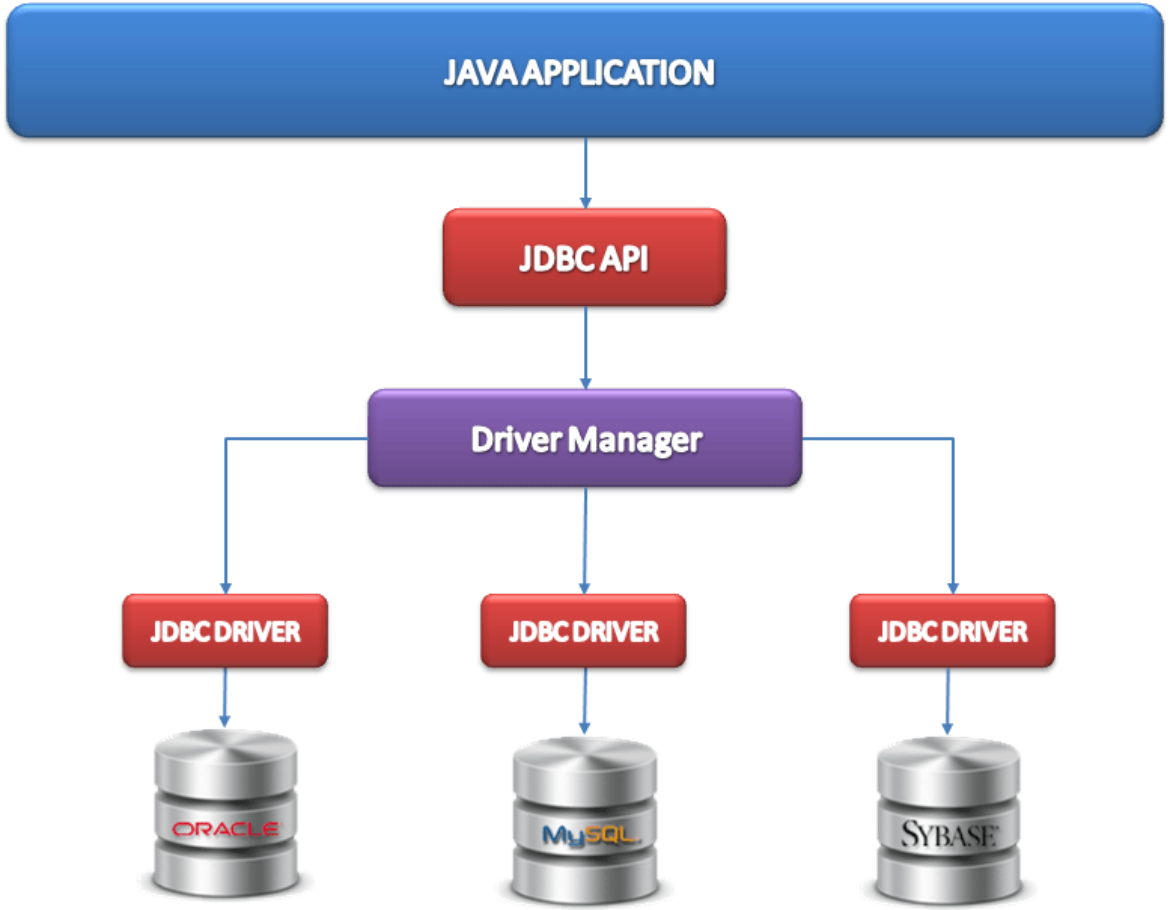


- JDBC est une **API** (bibliothèque d'interfaces et de classes) java standard qui permet un accès homogène à des bases de données depuis un programme Java au travers du langage SQL.
- C'est de plus une tentative de standardiser l'accès aux bases de données car l'API est indépendante du SGBD choisi, pourvu que le **Driver JDBC** existe pour ce SGBD, et qu'il implémente les classes et interfaces de l'API JDBC.

JDBC : Driver

- Drivers
  - chaque SGBD utilise un pilote (driver) qui lui est propre et qui permet de convertir les requêtes JDBC dans le langage natif du SGBD
  - le driver est un ensemble de classes qui implantent les interfaces de JDBC
  - les drivers sont le lien entre le programme Java et le SGBD
  - ces drivers dits JDBC existent pour tous les principaux SGBD: Oracle,Sybase, Informix, DB2, MySQL,...

Architecture JDBC



JDBC : Avantages

- Comme JDBC repose sur SQL, cette bibliothèque bénéficie de toutes ses fonctionnalités éprouvées : création et mise à jour de tables, sélections avec jointure, transactions, appels à des procédures stockées.
- La durée d'apprentissage de JDBC est réduite pour les personnes utilisant déjà un SGBDR avec SQL.
- Pour permettre d'exploiter leur produit en Java, les éditeurs de SGBDR du marché (Oracle, Sybase…) n'ont qu'à développer un driver JDBC, ensemble de classes qui implémentent les interfaces du paquetage java.sql.
- Une application peut se connecter à plusieurs SGBDR en même temps.
- Si votre programme utilise la version standard de SQL, il suffit de changer le driver approprié en cas de changement de SGBDR.

JDBC : Inconvénients

- Les instructions SQL étant construites sous forme de chaînes de caractères, leur exactitude ne peut être vérifiée ni par le compilateur Java ni par JDBC, mais uniquement par le SGBDR au moment de leur traitement.
- JDBC n'oblige pas un éditeur de SGBDR à implémenter tout SQL dans son driver, ce qui peut gêner la portabilité d'une application Java en cas de changement de SGBDR.
- Basée sur SQL, JDBC est limité à l'utilisation de bases de données relationnelles
- JDBC peut difficilement évoluer indépendamment de SQL.

Fonctionnement de JDBC

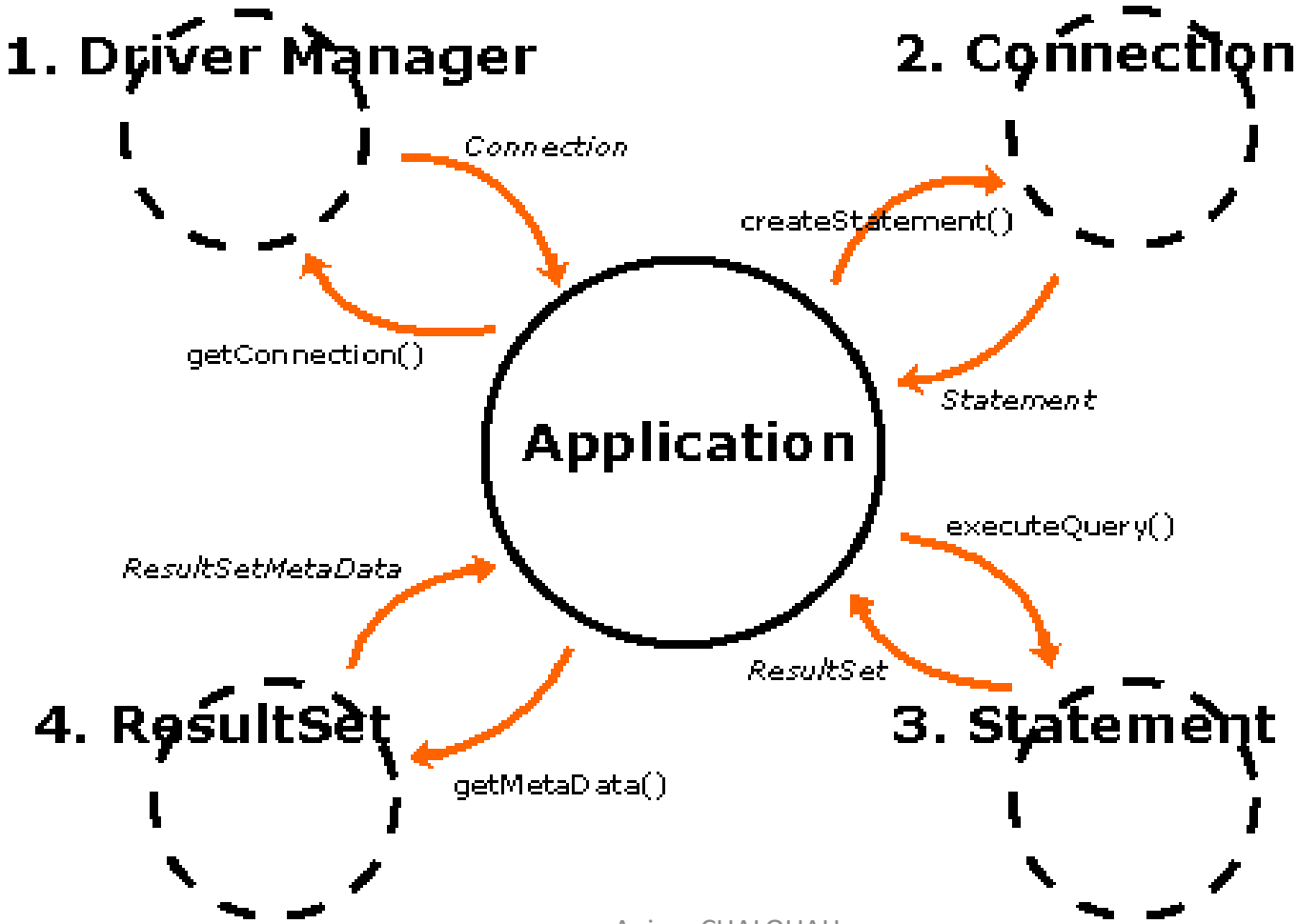
JDBC : les étapes

JDBC : Les étapes

Avant toute chose, n'oubliez pas d'importer le package **java.sql**

1) **Chargement du driver JDBC** : Disposer du driver spécifique ou utiliser ODBC sinon, car le pilote ODBC est dans l'API java.
2) **Etablir la connexion au SGBD** : Connaître l'URL de la base et y s'assurer d'avoir un compte d'accès username/password.
3) **Créer une requête** (ou instruction SQL)
4) **Exécuter la requête** : Mise à feu de la transaction et retour des résultats
5) **Traiter les données retournées**
6) **Fermer la connexion** : Prendre soin de fermer les curseurs de dialogue liés à la connexion

Mise en œuvre de JDBC



Etape 1 : Chargement du Driver

Syntaxe

- Pour se connecter à une base de données il est essentiel de charger dans un premier temps le pilote de la base de données à laquelle on désire se connecter grâce à un appel au DriverManager (gestionnaire de pilotes)

```
Class.forName(" NomDuDriver ");
```

- La méthode static forName() de la classe Class peut lever l'exception java.lang.ClassNotFoundException.

Etape 1 : Chargement du Driver

JDBC Driver Name

Database	JDBC Driver Name
SQLServer	com.microsoft.sqlserver.jdbc.SQLServerDriver
Sybase	com.sybase.jdbc2.jdbc.SybDriver
MySQL <= 5.1	com.mysql.jdbc.Driver
MySQL>=8	com.mysql.cj.jdbc.Driver
Oracle	oracle.jdbc.driver.OracleDriver
DB2	com.ibm.db2.jcc.DB2Driver
PostgreSQL	org.postgresql.Driver
MongoDB	mongodb.jdbc.MongoDriver

Etape 1 : Chargement du Driver

Exemple

```
 try {
    Class.forName("oracle.jdbc.driver.OracleDriver");
 }
catch(ClassNotFoundException ex) {
    System.err.println("Problème de chargement du Driver!");
    System.exit(1);
}
```

Etape 2 : Connexion à la base

- Après avoir chargé le pilote, vous pouvez établir une connexion à l'aide de la classe java.sql.DriverManager. Son rôle est de créer des connexions en utilisant le driver préalablement chargé.

- Cette classe dispose d'une méthode statique getConnection() prenant en paramètre l'**URL** de connexion, le **nom d'utilisateur** et le **mot de passe**.

- La méthode getConnection() peut lever une exception de la classe java.sql.SQLException.

Etape 2 : Connexion à la base

Syntaxe

```
Connection conn = DriverManager.getConnection(url, "Login", "Password");
```

- Une URL est une adresse qui pointe vers votre base de données de la forme

  jdbc:<sous-protocole>:<nom-BD>;param=valeur, ...

- L' URL spécifie :
  - l'utilisation de **JDBC** (**protocole**)
  - le **driver** ou le **type du SGBDR** (**sous-protocole**)
  - le **nom de la base** locale ou distante avec des paramètres de configuration éventuels : **nom utilisateur, mot de passe**, ...

Etape 2 : Connexion à la base

Database URL format

Database	Database URL format	Exemple URL
SQLServer	jdbc:sqlserver://{hostname}:{port};databaseName={database_name}	jdbc:sqlserver://localhost:1433/maBase
Sybase	dbc:jtds:sybase://{hostname}:{port}/{database_name}	jdbc:jtds:sybase://127.0.0.1:5000/maBase
MySQL	jdbc:mysql://{hostname}:{port}/{database_name}	jdbc:mysql://localhost:3306/maBase
Oracle	jdbc:oracle:thin:@{hostname}:{port}/{database_name}	jdbc:oracle:thin:@localhost:1521:maBase
DB2	jdbc:db2://{hostname}:{port}/{database_name}	jdbc:db2://127.0.0.1:446/maBase
PostgreSQL	jdbc:postgresql://{hostname}:{port}/{database_name}	jdbc:postgresql://localhost:5432/maBase
MongoDB	jdbc:mongo://{hostname}:{port}/{database_name}	Jdbc:mango://localhost:27017/maBase

Etape 2 : Connexion à la base

Exemple

```
String url = "jdbc:mysql//localhost:3306/maBase";
try {
   Connection conn = DriverManager.getConnection(url, "root", "");
   }
catch (SQLException e) {
   System.err.println("Error opening SQL connection:"+ e.getMessage());
}
```

Etape 2 : Connexion à la base

Fermeture de la connexion

- À la fin de votre programme JDBC, vous devez explicitement fermer toutes les connexions à la base de données pour terminer chaque session de base de données.
- Cependant, si vous l'oubliez, le **garbage collector** de Java fermera la connexion lorsqu'il nettoiera les objets périmés.
- Pour fermer la connexion, vous devez appeler la méthode close () comme suit

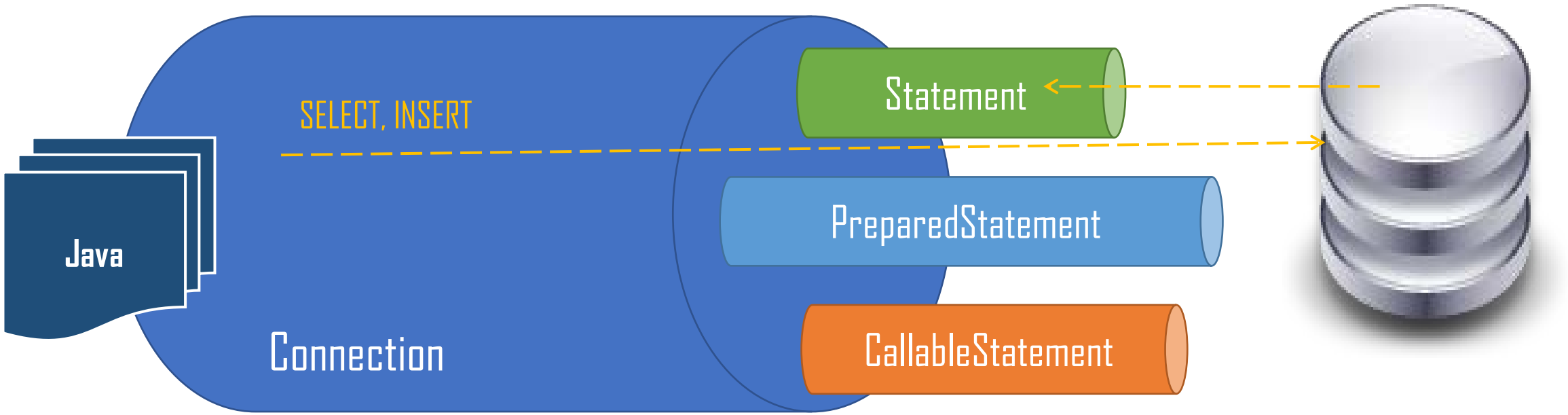
```
conn.close();
```

3 types de requêtes

Statement	PreparedStatement	CallableStatement
• Ordre SQL **Simple** • Ces états sont construits par la méthode createStatement appliquée à la connexion.	• Ordre SQL **paramétrés** • Ces états sont construits par la méthode prepareStatement appliquée à la connexion.	• **Procédures et fonctions catalogués** • Ces états sont construits par la méthode prepareCall appliquée à la connexion.

▶ S'il ne doit plus être utilisé dans la suite du code Java, chaque objet de type Statement, PreparedStatement ou CallableStatement devra être fermé à l'aide de la méthode close.

Étape 3 : Établir une requête SQL



Étape 3 : Établir une requête SQL
Statement

Syntaxe d'un Statement

```
try {
      Statement stmt = conn.createStatement();
    }
catch (SQLException e) {
 System.err.println("Error creating SQL statement: "
+e.getMessage());
                          }
```

- Il n'est pas nécessaire de définir un objet Statement pour chaque ordre SQL : il est possible d'en définir un et de le réutiliser.

Étape 4 : Exécution d'une requête de type Statement

- Il y a 3 types d'exécutions :

executeQuery

- Requête SELECT
- Méthode : Statement.executeQuery().
- résultat : java.sql.ResultSet contenant les lignes sélectionnées

executeUpdate

- Requêtes INSERT, UPDATE, DELETE, CREATE TABLE et DROP TABLE
- Méthode : Statement.executeUpdate()
- résultat : (int) nombre de lignes affectées par la requête.

execute

- pour quelques cas rares (procédures stockées)

Étape 4 : Exécution d'une requête de type Statement avec executeQuery

```
String query = "SELECT CIN, name,email FROM users;";
try {
  ResultSet rs = stmt.executeQuery(query);
}
catch (SQLException e) {
  System.err.println("Error executing query: " +e.getMessage());
}
```

Étape 4 : Exécution d'une requête de type Statement avec executeUpdate

```
String query = "UPDATE users SET
email='john_smith@aol.com' WHERE nom='SMITH';";
try {
  int result = stmt.executeUpdate(query);
}
catch (SQLException e) {
  System.err.println("Error executing query: " +e.getMessage());
}
```

```
int count = stmt.executeUpdate("DELETE FROM ENTREPRISES WHERE CODEPOST
LIKE '77%'");
System.out.println("Il y a eu " + count + " lignes supprimées.");
```

Etape 5 : Traitement des résultats

- L'objet ResultSet permet d'avoir un accès aux données résultantes de notre requête en mode **ligne par ligne**.
- La méthode ResultSet.next() permet de **passer d'une ligne à la suivante**. Cette méthode renvoie false dans le cas où il n'y a pas de ligne suivante.
- Il est nécessaire d'appeler au moins une fois cette méthode, le curseur est placé au départ avant la première ligne (si elle existe).

Etape 5 : Traitement des résultats

- La classe ResultSet dispose aussi d'un certain nombres **d'accesseurs** (**ResultSet.getXXX()**) qui permettent de récupérer le résultat contenu dans une colonne sélectionnée.
- On peut utiliser soit **le numéro de la colonne** désirée, soit **son nom** avec l'accesseur. La numérotation des colonnes commence à 1.
- XXX correspond au **type de la colonne**. Le tableau suivant précise les relations entre type SQL, type JDBC et méthode à appeler sur l'objet ResultSet.

Etape 5 : Traitement des résultats

Type SQL	Type JDBC	Méthode d'accès
char	String	getString()
varchar	String	getString()
integer	Integer	getInt()
double	Double	getDouble()
float	Float	getDouble()
Date	Date	getDate()
Blob	Blob	getBlob()

La méthode getString() permet d'obtenir la valeur d'un champ de n'importe quel type.

Etape 5 : Traitement des résultats

```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
try {
  while (rs.next()) {
        System.out.println(rs.getInt("CIN")+" - " +
         rs.getString("NOM") +" - " +
        rs.getString("EMAIL"));
              }
}
catch (SQLException e) {
  System.err.println("Error browsing query results: " +  e.getMessage());
}
```

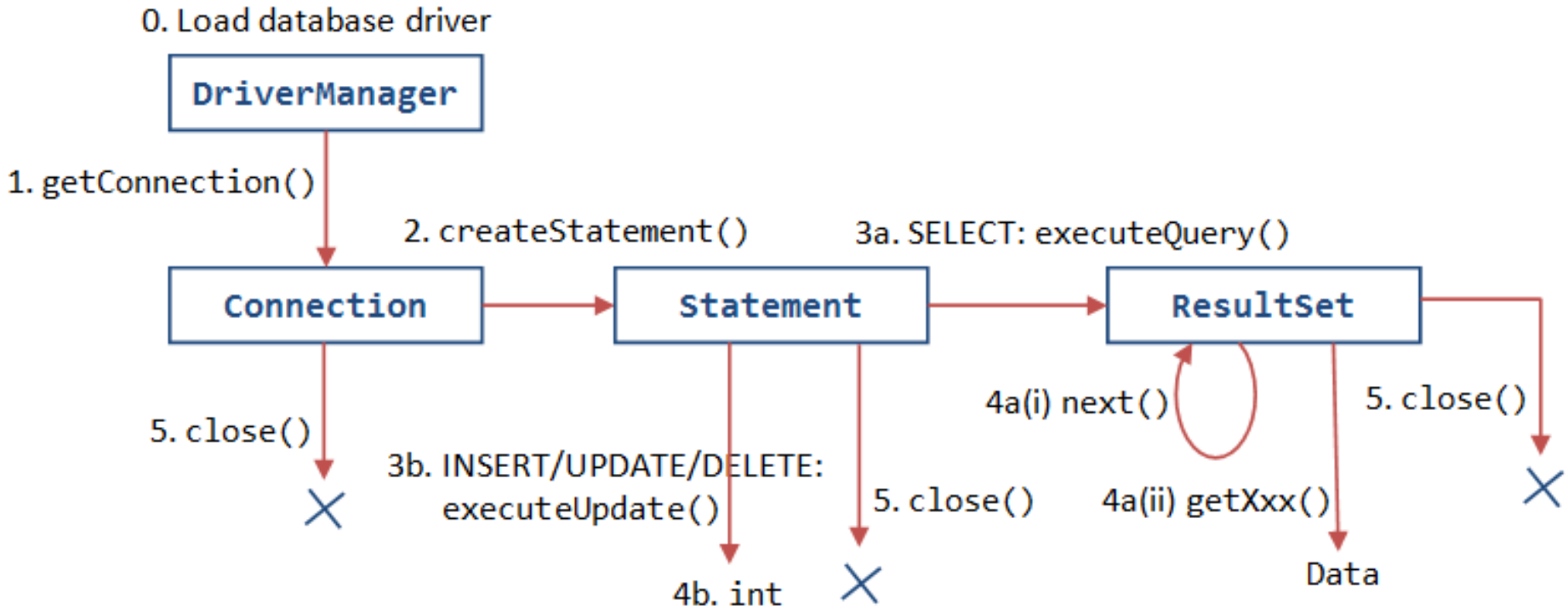
Etape 5 : Traitement des résultats

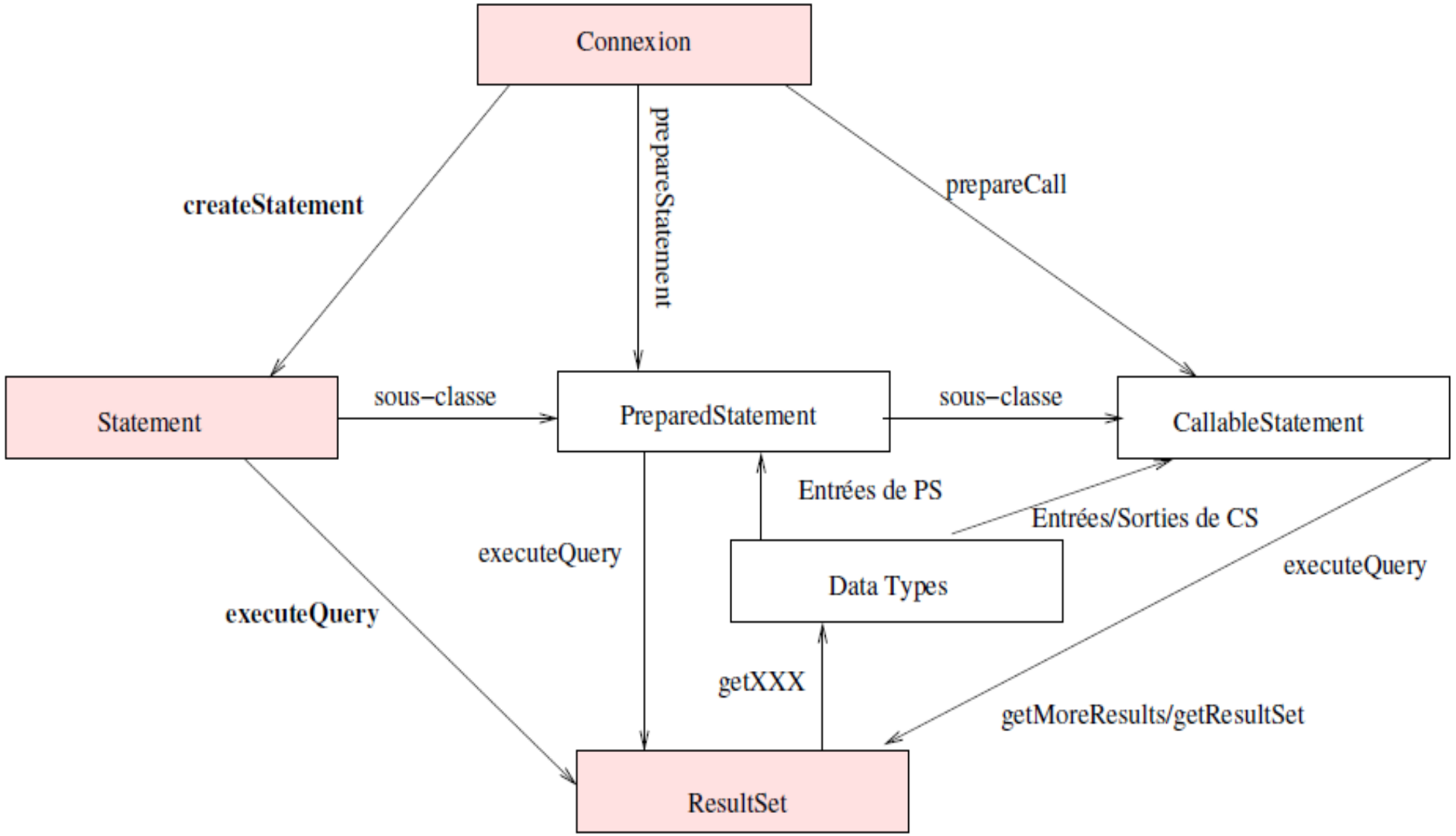
```
ResultSet rs = stmt.executeQuery("SELECT * FROM users");
//Pour accéder à chacun des tuples du résultat de la requête :
…
while (rs.next()) {
  int cin=rs.getInt(1) ;
  String nom = rs.getString(2);
  String prenom = rs.getString(3);
  java.sql.Date date_nais = rs.getDate(4);
...
}
```

Etape 6 : Fermeture des différents espaces

- Pour terminer proprement un traitement, il faut fermer les différents espaces ouverts sinon le garbage collector s'en occupera mais moins efficace
- Chaque objet possède une méthode close() :
  - **resultset.close();**
  - **statement.close();**
  - **connection.close();**

Récapitulatif JDBC




Connexion
createStatement
prepareStatement
prepareCall
Statement
sous−classe
PreparedStatement
sous−classe
CallableStatement
Entrées de PS
Entrées/Sorties de CS
executeQuery
executeQuery
Data Types
executeQuery
getXXX
getMoreResults/getResultSet
ResultSet

Accès aux méta-données

Accès aux méta-données

- JDBC permet de récupérer des informations :
  - sur **le type de données** que l'on vient de récupérer par un SELECT (interface ResultSetMetaData)
  - mais aussi sur **la base elle-même** (interface DatabaseMetaData)

Interface ResultSetMetaData

- La méthode getMetaData() permet d'obtenir des informations **sur les types de données** du ResultSet
- Elle renvoie des instances de ResultSetMetaData
- on peut connaître entre autres :
  - le **nombre de colonne** : getColumnCount()
  - le **nom d'une colonne** : getColumnName(int col)
  - le **nom de la table** : getTableName(int col)
  - le **type de donnée SQL de la colonne** : int getColumnType(int)
  - si **un NULL** SQL **peut être stocké dans une colonne** : isNullable()

Interface ResultSetMetaData

```
ResultSet rs = st.executeQuery("SELECT * FROM users");
ResultSetMetaData rsmd = rs.getMetaData();
int nbColonnes = rsmd.getColumnCount();
for (int i = 1; i <= nbColonnes; i++) {
  String typeColonne = rsmd.getColumnType(i);
  String nomColonne= rsmd.getColumnName(i);
  System.out.println("Colonne " + i+ " de nom " + nomColonne+ " de type " +
typeColonne);
}
```

Interface DatabaseMetaData

- Elle permet de donner des informations **sur la base de données**
- Méthode getMetaData() de l'objet Connection
- Dépend du SGBD avec lequel on travaille
- Elle renvoie des instances de DatabaseMetaData
- on peut connaître entre autres :
  - **les tables de la base** : getTables()
  - **le nom de l'utilisateur** : getUserName()

Requêtes précompilés

PreparedStatement

- L'interface PreparedStatement définit les méthodes pour un objet qui va encapsuler **une requête précompilée**.
- Ce type de requête est particulièrement adapté pour une exécution répétée d'une même requête avec des **paramètres différents**.
- Cette interface hérite de l'interface Statement.

PreparedStatement

- Lors de l'utilisation d'un objet de type PreparedStatement, la requête est envoyée au moteur de la base de données pour que celui ci prépare son exécution.
- Un objet qui implémente l'interface PreparedStatement est obtenu en utilisant la méthode prepareStatement() d'un objet de type Connection.
- Cette méthode attend en paramètre une chaîne de caractères contenant la requête SQL. Dans cette chaine, **chaque paramètre** est représenté par un caractère **?**

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

PreparedStatement

- Un ensemble de méthode setXXX(int, valeur) (ou XXX représente un type primitif ou certains objets tel que String, Date, Object, …) permet de fournir les valeurs de chaque paramètre défini dans la requête.
  - Le premier paramètre précise **le numéro du paramètre** dont la méthode va fournir la valeur.
  - Le second paramètre précise cette **valeur**.

PreparedStatement

```
PreparedStatement pstmt = conn.prepareStatement( "UPDATE emp SET sal = ? " +
"WHERE grade = ?");
  pstmt.setDouble(1,5000);
  pstmt.setString(2, "directeur" );
  int count = pstmt.executeUpdate();
```

Procédure stockée (CallableStatement)

Procédure stockée

- Les procédures stockées permettent de fournir la même fonctionnalité à plusieurs utilisateurs
- Les procédures sont stockées dans la base côté serveur

CallableStatement

- JDBC offre une interface dédiée CallableStatement
  - dérive de l'interface PreparedStatement
  - gère le retour de valeur avec ResultSet
- Comment
  - création d'un objet CallableStatement
  - en utlisant Connection.prepareCall(…)
- La requête doit être formatée
  - encadrée par des accolades {…}
  - utilisation du préfixe call

CallableStatement

Trois modes pour l'appel

- la procédure **renvoie une valeur**

  { ? = call nomProcédure(?,?,...)}

- la procédure **ne renvoie aucune valeur**

  { call nomProcédure(?,?,...)}

- si on ne lui passe **aucun paramètre**

  { call nomProcédure }

Passage de paramètres possible comme pour PreparedStatement

CallableStatement

```
CallableStatement testCall;
testCall = conn.prepareCall("{ call setSalary(?,?) }");
testCall.setString(1, "€ EUR");
testCall.setLong(2, 2000);
testCall.execute();
```

CallableStatement

- Un PreparedStatement peut retourner des données grâce à un ResultSet
- Les procédures stockées étendent le modèle
  - appel de la procédure précédé du passage des paramètres in et out grâce aux méthodes setXXX()
  - type des paramètres out et in/out grâce à la méthode registerOutParameter()
  - exécution de la requête grâce à executeQuery(), executeUpdate() ou execute()
  - récupération des paramètres out et in/out grâce aux méthodes getXXX()

CallableStatement
Exemple

```
create or replace procedure augmentation(unDept in integer,
pourcentage in number,cout out number) is
begin
update emp
set sal = sal * (1 + pourcentage / 100)
where dept = unDept;
select sum(sal) * pourcentage / 100 into cout
from emp where dept = unDept;
end;
```

CallableStatement
Exemple

```
CallableStatement csmt = conn.prepareCall( "{ call augmentation(?, ?, ?) }");
// 2 chiffres après la virgule pour 3ème paramètre
csmt.registerOutParameter(3, Types.DECIMAL, 2);
// Augmentation de 2,5 % des salaires du dept 10
csmt.setInt(1, 10);
csmt.setDouble(2, 2.5);
csmt.executeQuery();
double cout = csmt.getDouble(3);
System.out.println("Cout total augmentation : " + cout);
```

CallableStatement

- Elles peuvent retourner n'importe quel type de données y compris des ResultSet
- utiliser executeQuery() au lieu de execute()
- récupération d'un ResultSet normal
  - utilisable comme d'habitude
  - permet de sauvegarder la requête SQL au niveau de la base

```
CallableStatement csmt;
csmt = myConn.prepareCall("{ call getDetails(?) }");
ResultSet rs;
csmt.setLong(1, 1000000);
rs = csmt.executeQuery();
```