Conception par Objet et Programmation Java

Chapitre 11 : Connexion base de données

Equipe Java

esprit
Ecole Supérieure Privée
d'Ingénierie et de Technologies

Plan

- Introduction
- Classe et objet
- Encapsulation
- Héritage
- Polymorphisme
- **Exceptions**
- Connexion Base de donnée
- Interfaces
- Lambda Expression
- Collections
- Stream

Objectifs

- ✓ Fournir un accès à un SGBDR .
- ✓ Abstraire l'accès direct à une base donnée
- ✓ Manipuler les requêtes SQL

Introduction

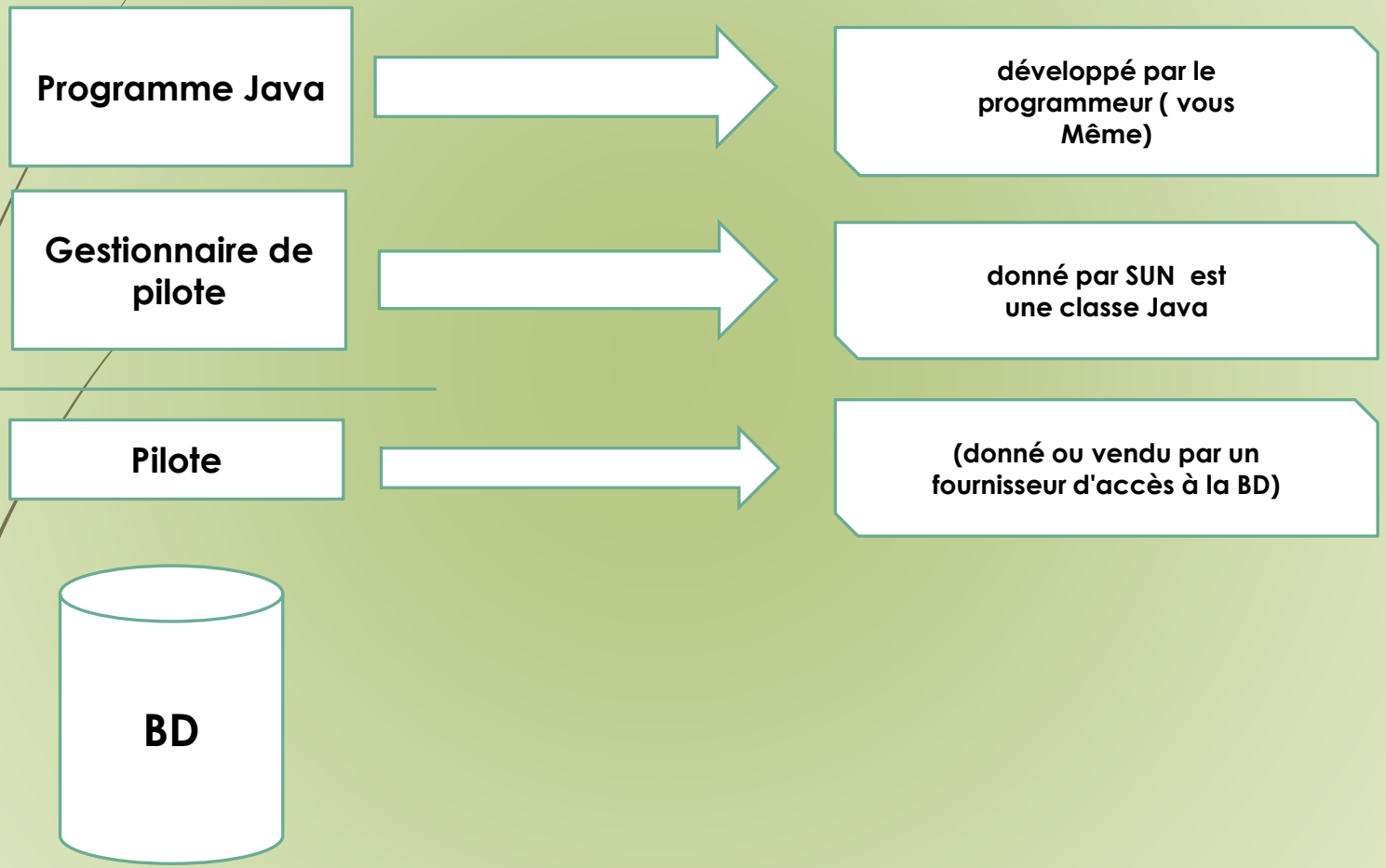
A. JDBC: Java database connectivity
   - Framework JAVA (ensemble de classes et d'interfaces défini par SUN)
   - Il a été développé par SUN pour permettre à des applications Java d'accéder à des bases de données relationnelles quelconques.

B. Les étapes principales:
   - Se connecter à une base de données.
   - Envoyer une requête SQL.
   - Lire et manipuler le résultat.

C. L'ensemble des classes qui implémentent les interfaces spécifiées par JDBC pour un gestionnaire de bases de données particulier est appelé un pilote JDBC.

D. Packages: java.sql et javax.sql

Architecture JDBC



Types de pilotes JDBC(1/2)

- Il existe 4 types de pilotes :

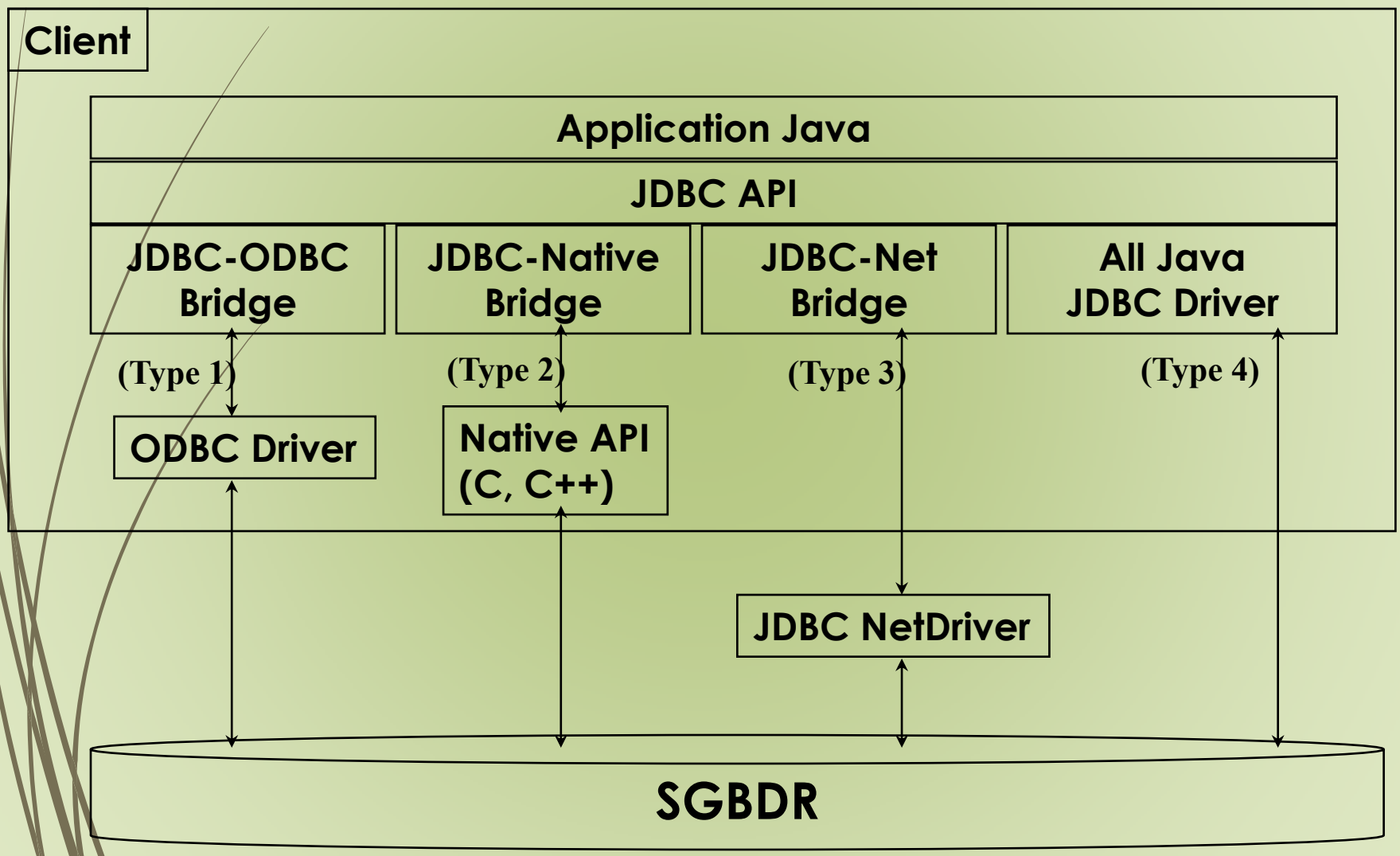
Bridge ODBC (fourni avec JDBC) :Pilote agissant comme passerelle en permettant l'accès à une base de données grâce à une autre technologie (JDBC-ODBC).

Native-API partly-Java driver : Pilotes d'API natifs. C'est un mélange de pilotes natifs et de pilotes Java. Les appels JDBC sont convertis en appels natifs pour le serveur de bases de données (Oracle, Sybase, ou autres) généralement en C ou en C++ .

JDBC-Net all-Java driver : Pilotes convertissant les appels JDBC en un protocole indépendant de la base de données. Un serveur convertit ensuite ceux-ci dans le protocole requis (modèle à 3 couches) .

Native-protocol all-Java driver : Pilotes convertissant les appels JDBC directement en un protocole réseau exploité par la base de données.

Types de pilotes JDBC(2/2)



Etapes générales d'accès à une Base de données

1ère Etape: Charger le pilote

Préciser le type de driver que l'on veut utiliser: Le Driver permet de gérer l'accès à un type particulier de SGBD.

2ème Etape: Connexion à la base

Récupérer un objet « Connection » en s'identifiant auprès du SGBD et en précisant la base utilisée .

3ème Etape :Création d'un Statement

4ème Etape : Exécution de la requête et lecture des résultats.

Chargement du pilote et connexion

- Charger le pilote : se fait à travers le gestionnaire de pilotes JDBC qui permet de connecter les programmes Java au bon pilote JDBC.

```
Class.forName("com.mysql.jdbc.Driver");
```

  - La classe: java.sql.DriverManager
  - L'interface: java.sql.Connection
  - Le Driver est dépendant du SGBD utilisé.
  - La méthode principale getConnection.

L'appel à la méthode forName déclenche un chargement dynamique du pilote

- Ouvrir la connexion:

```
static Connection getConnection( String url, String user, String password)
```

  - url : identification de la base considérée sur le SGBD
  - user : nom de l'utilisateur qui se connecte à la base
  - password : mot de passe de l'utilisateur

- NB: Le format de l'URL dépend du SGGB utilisé
- A partir de la version 4du JDBC , le chargement explicite du Driver est initule.

Connexion à la base

- Accès via un URL qui spécifie :
  - l'utilisation de JDBC
  - le driver ou le type du SGBDR
  - l'identification de la base
- Exemple :

```
String url=" jdbc:mysql://localhost:3306/esprit";
String username="root";
password="root";
```

- Ouverture de la connexion :

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

Exemple

```
public class connexionBaseD {
// JDBC driver name and database URL
static final String jdbcDriver = "com.mysql.jdbc.Driver";
static final String databaseUrl = "jdbc:mysql://localhost:3306/esprit";
static final String user = "user";
static final String pass = "user";
try {
// Charger le pilote
Class.forName(jdbcDriver );
// Etablir la connection
connection = DriverManager.getConnection(databaseUrl , user, pass);
} // end try
catch ( ClassNotFoundException classNotFound ) {
System.err.prinln("Impossible de charger le pilote");
classNotFound.printStackTrace();
System.exit( 1 );
} // end catch
catch ( SQLException sqlException ){
System.err.prinln("Connetion Impossible");
sqlException.printStackTrace();
System.exit( 1 );
} // end catch
```

Création d'un Statement

- 3 types de *statement* :
  - `statement` : requêtes simples
  - `prepared statement` : requêtes précompilées
  - `callable statement` : procédures stockées
- Création d'un *statement* :

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

Exécution d'une requête (1/2)

- 3 types d'executions :
  - `executeQuery` : pour les requêtes qui retournent un `ResultSet`
  - `executeUpdate` : pour les requêtes `INSERT`, `UPDATE`, `DELETE`, `CREATE TABLE` et `DROP TABLE`
  - `execute` : pour quelques cas rares (procédures stockées)

Exécution d'une requête (2/2)

- Execution de la requête :

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

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

Lecture des résultats (1/2)

- `La Classe: java.sql.ResultSet`
- Contient les résultats d'une requête SELECT
- Plusieurs lignes contenant plusieurs colonnes
- On y accède ligne par ligne puis valeur par valeur dans la ligne
- `executeQuery()` renvoit un `ResultSet`
- Le RS se parcourt itérativement *row(ligne)* par *row*
- Les colonnes sont référencées par leur numéro ou par leur nom.
- L'accès aux valeurs des colonnes se fait par les méthodes `getXXX()` où `XXX` représente le type de l'objet
- Pour les très gros *row*, on peut utiliser des *streams*.

Lecture des résultats (2/2)

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

PreparedStatement(instruction paramétré)

L'envoi d'une requête à la BD pour exécution passe par les étapes suivantes :

- Analyse de la requête
- Compilation de la requête
- Optimisation de la requête
- Exécution de la requête

- Même si cette requête est la même que la précédente !! Or les 3 premières étapes ont déjà été effectuées dans ce cas.

- Les bases de données définissent la notion de requête préparée, requête où les 3 premières étapes ne sont effectuées qu'une seule fois.

JDBC propose l'interface **PreparedStatement** pour modéliser cette notion.

Syntaxe de PreparedStatement (1/2)

Au lieu :

```
Statement smt = conX.createStatement();
ResultSet rs = smt.executeQuery( "SELECT * FROM Livres" );
```

On écrit :

```
PreparedStatement pSmt = conX.prepareStatement("SELECT * FROM Livres" );
ResultSet rs = pSmt.executeQuery();
```

la requête est décrite au moment de la "construction" pas lors de l'exécution qui est lancée par executeQuery() sans argument.

Avantages: compilation unique et paramètres binaires plus faciles à passer

Syntaxe de PreparedStatement (2/2)

Requêtes de la forme :
SELECT nom FROM Personnes WHERE age > ? AND adresse = ?

On écrit :
PreparedStatement pSmt = conX.prepareStatement("SELECT * FROM Livres");
ResultSet rs = pSmt.executeQuery();

la requête est décrite au moment de la "construction" pas lors de l'exécution qui est lancée par executeQuery() sans argument

Exemple:

```
PreparedStatement pSmt = conX.prepareStatement("SELECT nom FROM Personnes
WHERE age > ? AND adresse = ?" );
pSmt .setInt(1, 22);
pSmt .setString(2, "Ali");
ResultSet rs = pSmt.executeQuery();
```

Bonnes Pratiques

- Évitez d'utiliser SELECT * FROM ... (coûteux en transfert).
- Ne pas **disperser** les noms des colonnes SQL dans votre code Java.
- Faites le **maximum** de travail en SQL et le **minimum** en Java.
- Utilisez un *pool* de connexions si possible au lieu d'utiliser une seule connexion .
- Vous pouvez **fermer** (close) un résultat de requête (ResultSet).
- Vous pouvez **fermer** (close) une instruction (Statement) ce qui provoque la fermeture des résultats liés à cette instruction.