Développement d'Application Client-Serveur

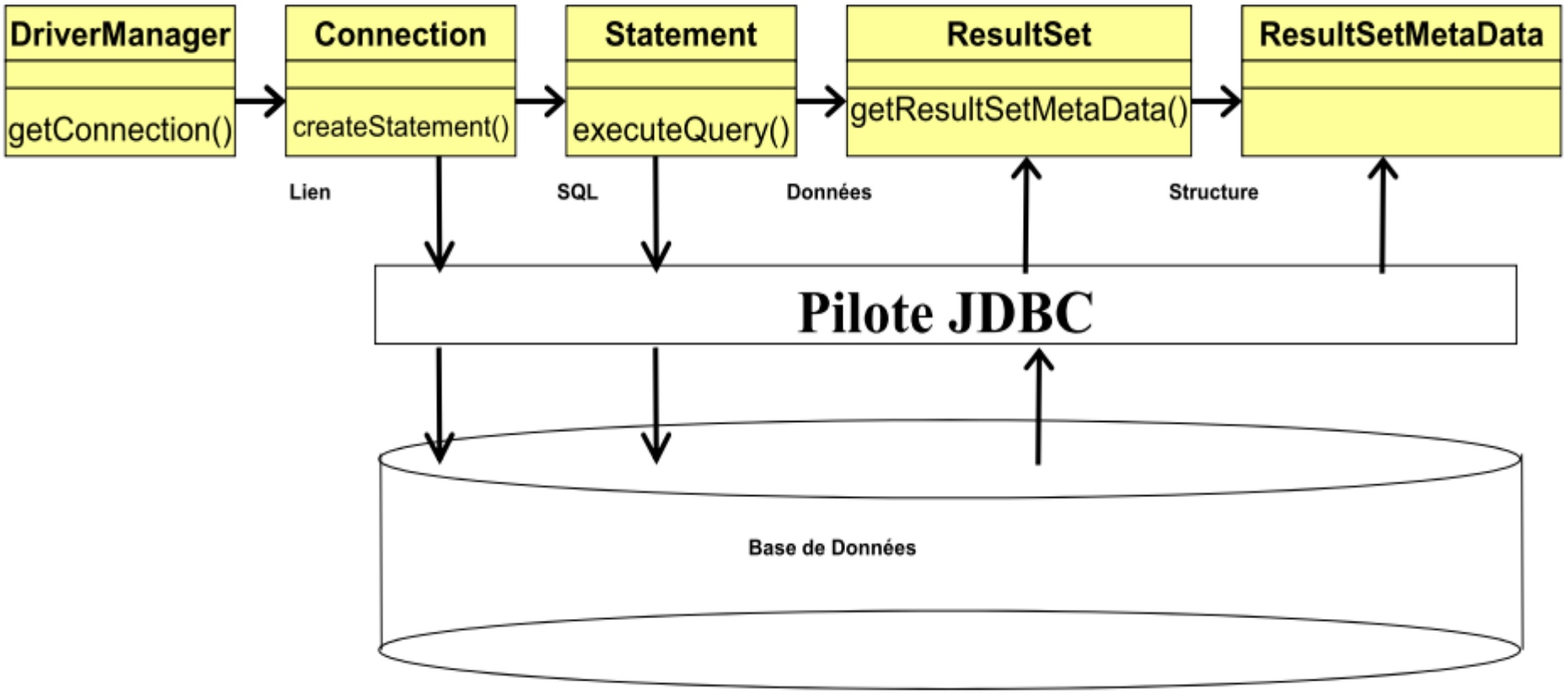
Chapitre 5 : Accès aux bases de données via JDBC

Créer une application JDBC

Pour créer une application élémentaire de manipulation d'une base de données il faut suivre les étapes suivantes :

- Chargement du Pilote JDBC ;
- Identification de la source de données ;
- Allocation d'un objet **Connection**
- Allocation d'un objet Instruction **Statement** (ou **PreparedStatement**);
- Exécution d'une requête à l'aide de l'objet Statement ;
- Récupération de données à partir de l'objet renvoyé **ResultSet** ;
- Fermeture de l'objet ResultSet ;
- Fermeture de l'objet Statement ;
- Fermeture de l'objet Connection.

Créer une application JDBC



Démarche JDBC

- **Charger les pilotes JDBC :**
  - **Utiliser la méthode forName de la classe Class, en précisant le nom de la classe pilote.**
  - **Exemples:**
  - **Pour charger le pilote JdbcOdbcDriver:**

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver" );
```

  - **Pour charger le pilote jdbc de MySQL:**

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

Créer une connexion

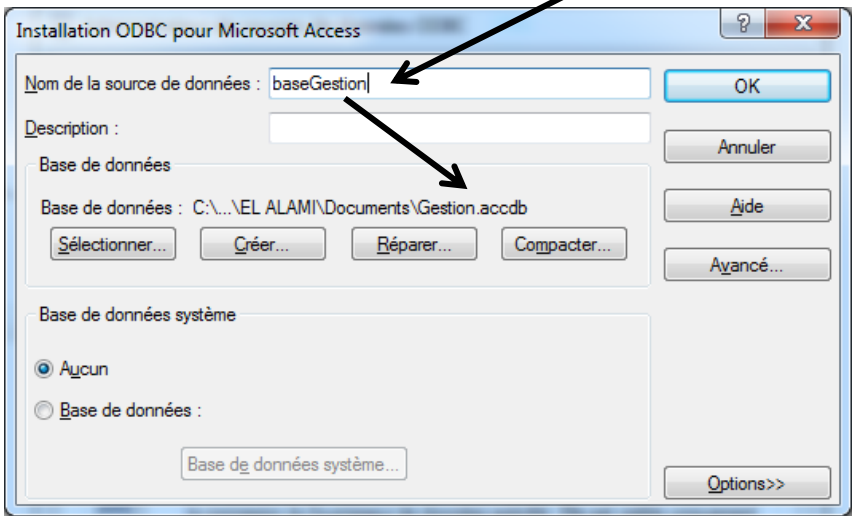
- Pour créer une connexion à une base de données, il faut utiliser la méthode statique getConnection() de la classe DriverManager. Cette méthode fait appel aux pilotes JDBC pour établir une connexion avec le SGBDR, en utilisant les sockets.

✓ **Pour un pilote `com.mysql.jdbc.Driver`:**

```
Connection conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/DB","user","pass");
```

✓ **Pour un pilote `sun.jdbc.odbc.JdbcOdbcDriver`:**

```
Connection conn=DriverManager.getConnection("jdbc:odbc:baseGestion", "user", "pass" );
```



Objets Statement, ResultSet et ResultSetMetaData

- Pour exécuter une requête SQL, on peut créer l'objet **Statement** en utilisant la méthode **createStatement()** de l'objet **Connection**.
- Syntaxe de création de l'objet **Statement**

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

- Exécution d'une requête SQL avec l'objet **Statement** :
  - Pour exécuter une requête SQL de type select, on peut utiliser la méthode **executeQuery()** de l'objet **Statement**. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet **ResultSet**:

```
ResultSet rs=st.executeQuery("select * from  PRODUITS");
```

  - Pour exécuter une requête SQL de type **insert**, **update** et **delete** on peut utiliser la méthode **executeUpdate()** de l'objet **Statement**:

```
st.executeUpdate("insert into  PRODUITS (…) values(…)");
```

- Pour récupérer la structure d'une table, il faut créer l'objet **ResultSetMetaData** en utilisant la méthode **getMetaData()** de l'objet **Resultset**.

```
ResultSetMetaData rsmd  = rs.getMetaData();
```

Objet PreparedStatement

- Pour exécuter une requête SQL, on peut également créer l'objet **PreparedStatement** en utilisant la méthode **prepareStatement()** de l'objet **Connection**.
- Syntaxte de création de l'objet **PreparedStatement**

```
PreparedStatement  ps=conn.prepareStatement("select *
from  PRODUITS where  NOM_PROD like  ? AND PRIX<?");
```

- Définir les valeurs des paramètres de la requête:

```
ps.setString(1,"%"+motCle+"%");
ps.setString(2, p);
```

- Exécution d'une requête SQL avec l'objet **PreparedStatement** :
  - Pour exécuter une requête SQL de type select, on peut utiliser la méthode **executeQuery()** de l'objet **PreparedStatement**. Cette méthode exécute la requête et stocke le résultat de la requête dans l'objet **ResultSet**:

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

  - Pour exécuter une requête SQL de type **insert**, **update** et **delete** on peut utiliser la méthode **executeUpdate**() de l'objet **PreparedStatement** :

```
ps.executeUpdate();
```

Récupérer les données d'un ResultSet

- Pour parcourir un **ResultSet**, on utilise sa méthode **next()** qui permet de passer d'une ligne à l'autre. Si la ligne suivante existe, la méthode **next()** retourne **true**. Si non elle retourne **false**.
- Pour récupérer la valeur d'une colonne de la ligne courante du **ResultSet**, on peut utiliser les méthodes **getInt(colonne)**, **getString(colonne)**, **getFloat(colonne)**, **getDouble(colonne)**, **getDate(colonne)**, etc… colonne représente le numéro ou le nom de la colonne de la ligne courante.
- **Syntaxte:**

```
while(rs.next()){
        System.out.ptintln(rs.getInt(1));
        System.out.ptintln(rs.getString("NOM_PROD"));
        System.out.ptintln(rs.getDouble("PRIX"));
}
```

Objet Statement (Exemple)

```
import java.sql.Connection;import java.sql.DriverManager;
import java.sql.ResultSet;import java.sql.Statement;import java.util.Scanner;
public class Application {
public Application() {
try {Class.forName("com.mysql.jdbc.Driver");
Connection conn=
DriverManager.getConnection("jdbc:mysql://localhost:3306/gs_prod","root","");
System.out.println("Entrer la quantité minimale:");
Scanner clavier=new Scanner(System.in); int qte=clavier.nextInt();
Statement st=conn.createStatement();
String req="select * from produits where quantite>="+qte;
ResultSet rs=st.executeQuery(req);
while(rs.next()){System.out.print(rs.getString("Ref_Produit")+"\t");
System.out.print(rs.getString("Nom_Produit")+"\t");
System.out.print(rs.getDouble("Prix")+"\t");
System.out.print(rs.getInt("Quantite")+"\t");
System.out.print(rs.getInt("id_cat")+"\n");}
} catch (Exception e) {e.printStackTrace();}
}
public static void main(String[] args) {
new Application();
}
}
```

Objet PreparedStatement (Exemple)

```
public class App {
public App() {
try {Class.forName("com.mysql.jdbc.Driver");
Connection con=
DriverManager.getConnection("jdbc:mysql://localhost:3306/gs_prod","root","");
Scanner sc=new Scanner(System.in);
System.out.print("Donner la quantité minimale:"); int qte=sc.nextInt();
sc.close();
String req="select * from produit where quantite>=?";
PreparedStatement ps=con.prepareStatement(req); ps.setInt(1, qte);
ResultSet rs=ps.executeQuery();
while(rs.next()){
System.out.print(rs.getString("ref_prod")+"\t");
System.out.print(rs.getString(2)+"\t");
System.out.print(rs.getDouble("prix")+"\t");
System.out.print(rs.getInt("quantite")+"\t");
System.out.print(rs.getInt(5)+"\t");
System.out.println();}
rs.close();ps.close();con.close();
} catch (Exception e) {e.printStackTrace();}
}
public static void main(String[] args) {
new App();}
}
```

Exploitation de l'objet ResultSetMetaData

- L'objet **ResultSetMetaData** est très utilsé quand on ne connait pas la structure d'un **ResultSet**. Avec L'objet **ResultSetMetaData**, on peut connaitre le nombre de colonnes du **ResultSet**, le nom, le type et la taille de chaque colonne.
- Pour afficher, par exemple, le nom, le type et la taille de toutes les colonnes d'un **ResultSet** rs, on peut écrire le code suivant:

```
ResultSetMetaData  rsmd=rs.getMetaData();
for(int  i=1;i<=rsmd.getColumnCount();i++){// Parcourir toutes les colonnes
   // afficher le nom de la colonne numéro i
   System.out.println(rsmd.getColumnName(i));
   // afficher le type de la colonne numéro i
   System.out.println(rsmd.getColumnTypeName(i));
   // afficher la taille de la colonne numéro i
   System.out.println(rsmd.getColumnDisplaySize(i));
}
while  (rs.next()){// Afficher tous les enregistrements du ResultSet  rs
for(int  i=1;i<=rsmd.getColumnCount();i++){
System.out.println(rs.getString(i));
}
}
```

DatabaseMetaData

❑ **DatabaseMetaData**: est utilisé pour récupérer le "méta-données" d'une base de données: nom du SGBD et sa version, nom du driver et sa version, les noms des tables, des vues,…

```
DatabaseMetaData dbmd=con.getMetaData();
System.out.println("Nom du driver:"+dbmd.getDriverName());
System.out.println("Version du
driver:"+dbmd.getDriverVersion());
System.out.println("User name:"+dbmd.getUserName());
System.out.println("SGBD:"+dbmd.getDatabaseProductName());
System.out.println("Version du
SGBD:"+dbmd.getDatabaseProductVersion());
String table[]={"TABLE"};
ResultSet rsdb=dbmd.getTables(null, null, null, table);
while(rsdb.next()){
System.out.println(rsdb.getString(3));
}
```

Mapping objet relationnel

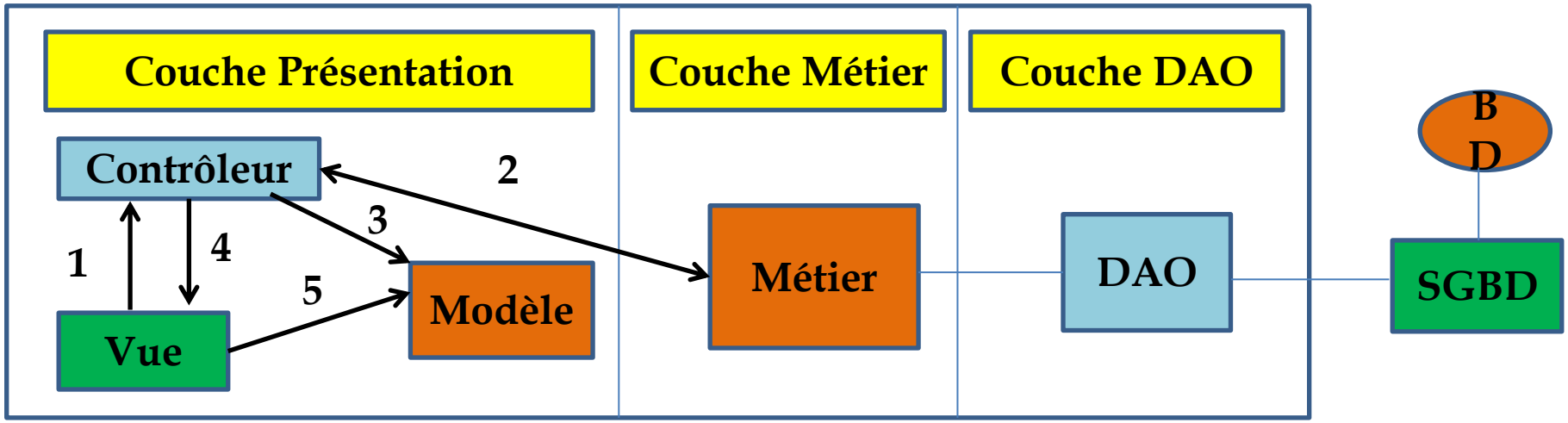
- Dans la pratique, on cherche toujours à séparer la logique de métier de la logique de présentation.
- On peut dire qu'on peut diviser une application en 3 couches:
  - La couche d'accès aux données: DAO
    - Partie de l'application qui permet d'accéder aux données de l'applications . Ces données sont souvent stockées dans des bases de données relationnelles .
  - La couche Métier:
    - Regroupe l'ensemble des traitements que l'application doit effectuer.
  - La couche présentation:
    - S'occupe de la saisie des données et de l'affichage des résultats;

Architecture d'une application

- Une application se compose de plusieurs couches:
  - La couche **DAO** qui s'occupe de l'accès aux bases de données.
  - La couche **métier** qui s'occupe des traitements.
  - La couche **présentation** qui s'occupe de la saisie, le contrôle et l'affichage des résultats. Généralement la couche présentation respecte le **pattern MVC** qui fonctionne comme suit:

1.La vue permet de saisir les données, envoie ces données au contrôleur

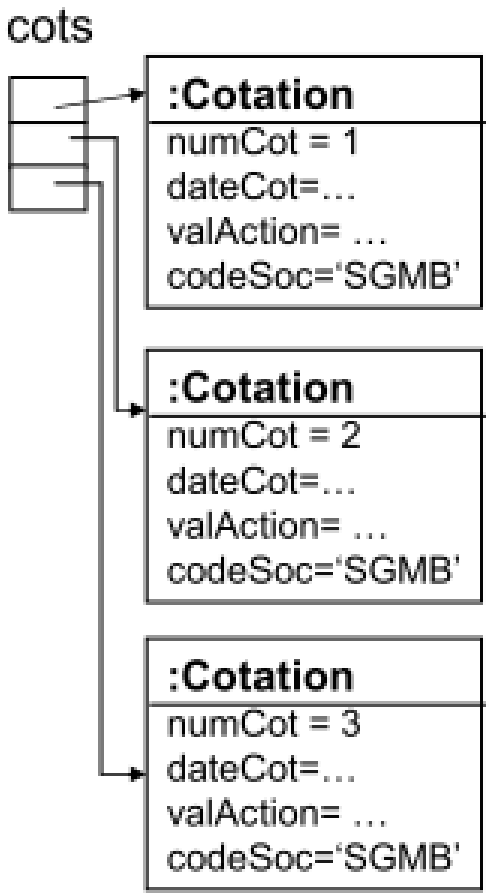
2. **Le contrôleur récupère les données saisies. Après la validation de ces données, il fait appel à la couche métier pour exécuter des traitements.**
3. **Le contrôleur stocke le résultat du modèle.**
4. **Le contrôleur fait appel à la vue pour afficher les résultats.**
5. **La vue récupère les résultats à partir du modèle et les affiche.**



Mapping objet relationnel

- D'une manière générale les applications sont orientée objet :
  - Manipulation des objet et des classes
  - Utilisation de l'héritage et de l'encapsulation
  - Utilisation du polymorphisme
- D'autres part les données persistantes sont souvent stockées dans des bases de données relationnelles.
- Le mapping Objet relationnel consiste à faire correspondre un enregistrement d'une table de la base de données à un objet d'une classe correspondante.
- Dans ce cas on parle d'une classe persistante.
- Une classe persistante est une classe dont l'état de ses objets sont stockés dans une unité de sauvegarde (Base de données, Fichier, etc..)

Couche Métier : Mapping objet relationnel



Application orientée objet

```
public List<Cotation> getCotations(String codeSoc){
List<Cotation> cotations=new ArrayList<Cotation>();
try {
Class.forName("com.mysql.jdbc.Driver");
Connection conn=DriverManager.getConnection
("jdbc:mysql://localhost:3306/bourse_ws","root","");
PreparedStatement ps=conn.prepareStatement
("select * from cotations where CODE_SOCIETE=?");
ps.setString(1, codeSoc);
ResultSet rs=ps.executeQuery();
while(rs.next()){
Cotation cot=new Cotation();
cot.setNumCotation(rs.getLong("NUM_COTATION"));
cot.setDateCotation(rs.getDate("DATE_COTATION"));
cot.setCodeSociete(rs.getString("CODE_SOCIETE"));
cot.setValAction(rs.getDouble("VAL_ACTION"));
cotations.add(cot);
}
} catch (Exception e) { e.printStackTrace();}
return(cotations);
}
```

NUM_COTATION	DATE_COTATION	VAL_ACTION	CODE_SOCIETE
1	2008-08-30 15:57:50	2093.17199826538	SGMB
2	2008-08-30 15:57:52	258.769396752267	SGMB
3	2008-08-30 15:57:52	1050.71222698514	SGMB

Base de données relationnelle

Objet CallableStatement

- L'interface **CallableStatement**, qui étends PreparedStatement, permet de faire appel aux **procédures stockées** et aux **fonctions** de manière standard pour tous les SGBD.
- La principale différence avec les PreparedStatement se situe au niveau des paramètres. Ceux-ci sont toujours définis par des points d'interrogation, mais en plus des **paramètres d'entrée (IN)**, CallableStatement peux avoir des **paramètres de sortie (OUT)**. Ces paramètres définissent le résultats de la procédure. On peux aussi **combiner ces deux types (INOUT)**.
- Une instance de **CallableStatement** s'obtient grâce aux méthodes **prepareCall de Connection**. Le premier argument de ces méthodes est une chaîne de caractères définissant l'instruction SQL.

Objet CallableStatement

- Pour les **procédures stockées** :

```
String sql = "{call nomDeLaProcedure[(?, ?, ...)]}";
//[(?, ?, ...)] sont les éventuels arguments de la procédure
//ces arguments peuvent être de type IN, OUT ou INOUT
CallableStatement statement = connection.prepareCall(sql);
```

- Pour les **fonctions** (procédures stockées renvoyant un résultat) :

```
String sql = "{? = call nomDeLaProcedure[(?, ?, ...)]}";
//le premier ? est le résultat de la procédure
//[(?, ?, ...)] sont les éventuels arguments de la procédure
CallableStatement statement = connection.prepareCall(sql);
```

- Les autres arguments de la méthode prepareCall servent à déterminer les types de ResultSet obtenus à partir de la procédure. Par exemple :

```
String sql = "{? = call max(?, ?)}";
CallableStatement statement = connection.prepareCall(sql,
ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
```

Objet CallableStatement (Exemple)

```
import java.sql.CallableStatement;import java.sql.Connection;
import java.sql.DriverManager;import java.sql.ResultSet;
import java.sql.Statement;import java.util.Scanner;
public class Application2 {
public Application2() {
try {
Class.forName("com.mysql.jdbc.Driver");
Connection
conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/gs_prod","root","");
System.out.println("Entrer un mot clé:");
Scanner clavier=new Scanner(System.in);String mot=clavier.next();
Statement st=conn.createStatement();
st.executeUpdate("drop procedure if exists rechercherProduit");
st.executeUpdate(
"create procedure rechercherProduit(in mc varchar(20))\n"
+"begin\n"
+"select * from produits where Nom_Produit like mc;\n"
+"select count(*) from produits where Nom_Produit like mc;\n"
+"end\n"
);
String req="{call rechercherProduit(?)}";
CallableStatement call=conn.prepareCall(req);
call.setString(1, "%"+mot+"%");
```

Objet CallableStatement (Exemple)

```
if(call.execute()){
    //récupération des ResultSet
    ResultSet rs1 = call.getResultSet();
    call.getMoreResults(Statement.KEEP_CURRENT_RESULT);
    ResultSet rs2 = call.getResultSet();
    //traitement des informations
    while(rs1.next()){
        for(int i=0;i<rs1.getMetaData().getColumnCount();i++){
            System.out.print(rs1.getObject(i+1)+", ");
        }
        System.out.println("");
    }
    rs2.next();
    System.out.println("Nombre de lignes = "+rs2.getObject(1));
    rs1.close();
    rs2.close();
}
} catch (Exception e) {
e.printStackTrace();
}
}
public static void main(String[] args) {
new Application2();
}
}
```

Application

- On considère une base de données qui contient une table **ETUDIANTS** qui permet de stocker les étudiants d'une école. La structure de cette table est la suivante :

#	Colonne	Type	Interclassement	Attributs	Null	Défaut	Extra
1	**Id_Etud**	int(11)			Non	*Aucune*	AUTO_INCREMENT
2	**Nom**	varchar(25)	latin1_swedish_ci		Non	*Aucune*	
3	**Prenom**	varchar(25)	latin1_swedish_ci		Non	*Aucune*	
4	**Email**	varchar(80)	latin1_swedish_ci		Non	*Aucune*	
5	**Ville**	varchar(25)	latin1_swedish_ci		Non	*Aucune*	

Id_Etud	Nom	Prenom	Email	Ville
1	Alaoui	Rachid	alaoui@yahoo.fr	Fès
2	Slimani	Fatima	slimani@gmail.com	Meknès
3	Bennis	Abdelilah	bennis@gmail.com	Casablanca
4	Machhour	Hassan	machhour@yahoo.fr	Séfrou

- Nous souhaitons créer une application java qui permet de saisir au clavier un mot clé et d'afficher tous les étudiants dont le nom contient ce mot clé.
- Dans cette application devons séparer la couche métier de la couche présentation.

Application

- Pour cela, la couche métier est représentée par un modèle qui se compose de deux classes :
  - La classe Etudiant.java : c'est une classe persistante c'est-à-dire que chaque objet de cette classe correspond à un enregistrement de la table ETUDIANTS. Elle se compose des :
    - champs privés idEtudiant, nom, prenom, email et ville,
    - d'un constructeur par défaut,
    - des getters et setters.

    Ce genre de classe c'est ce qu'on appelle un java bean.
  - La classe Scolarite.java :
    - c'est une classe non persistante dont laquelle, on implémente les différentes méthodes métiers.
    - Dans cette classe, on fait le mapping objet relationnel qui consiste à convertir un enregistrement d'une table en objet correspondant.
    - Dans notre cas, une seule méthode nommée getEtudiants(String mc) qui permet de retourner une Liste qui contient tous les objets Etudiant dont le nom contient le mot clé « mc ».

Application

Travail à faire :

Couche données :

- Créer la base de données « SCOLARITE » de type MSAccess ou MySQL
- Pour la base de données Access, créer une source de données système nommée « dsnScolarite », associée à cette base de données.
- Saisir quelques enregistrements de test

Couche métier. (package metier) :

- Créer la classe persistante Etudiant.java
- Créer la classe des business méthodes Scolarite.java

Couche présentation (package pres):

- Créer une application de test qui permet de saisir au clavier le mot clé et qui affiche les étudiants dont le nom contient ce mot clé.