



ECOLE MAROCAINE DES
SCIENCES DE L'INGENIEUR
Membre de
HONORIS UNITED UNIVERSITIES

ACCÈS AUX BASES DE DONNÉES

Ecole Marocaine des Sciences de l'ingénieur
-EMSI-

Mme OUHMIDA Asmae

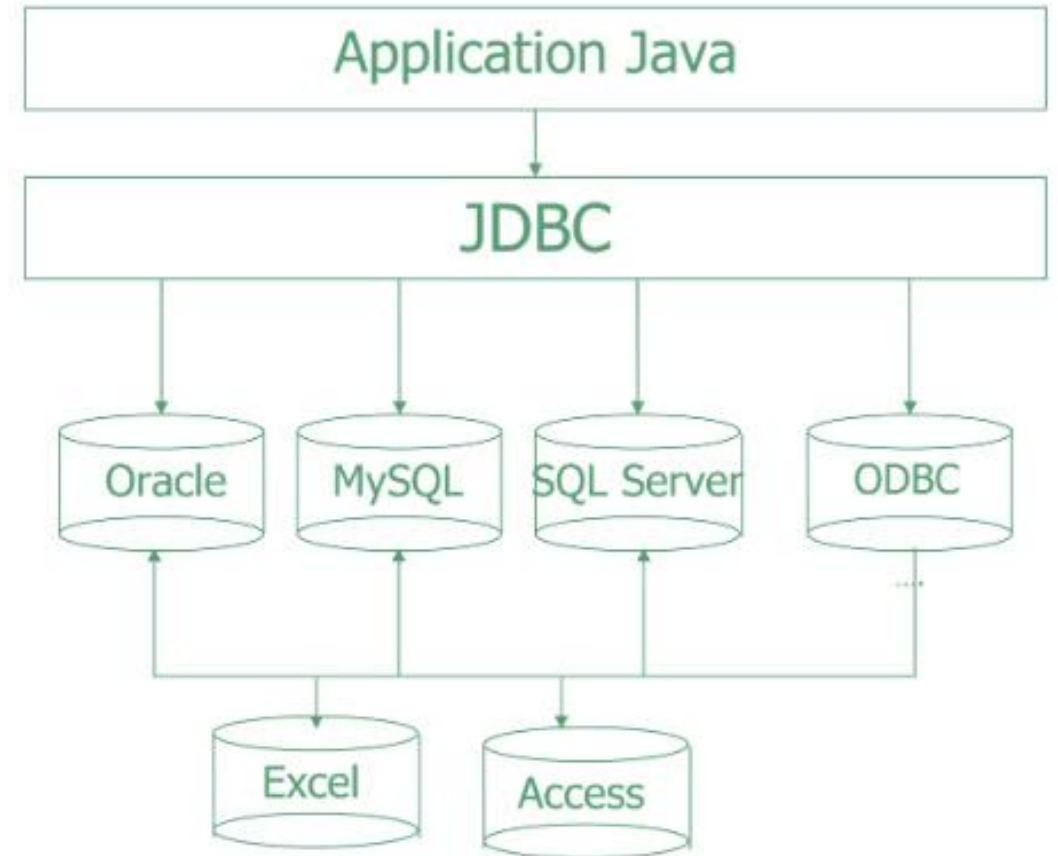
Année universitaire 2022/2023

PLAN

1. Pilotes JDBC
2. Créer une application JDBC
3. Démarche JDBC
4. Créer une connexion
5. Créer une application JDBC
6. Objet PreparedStatement
7. Connexion à la base de données
8. Architecture d'une application
9. Le mapping objet relationnel

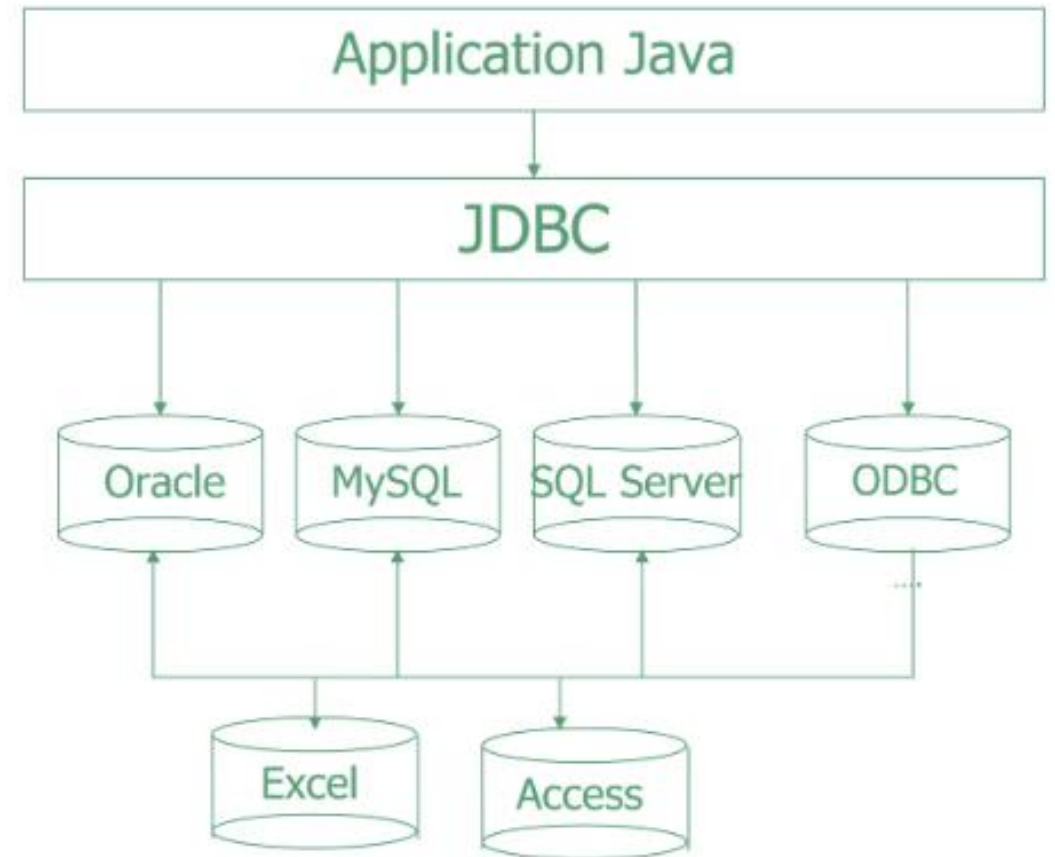
PILOTES JDBC

- Pour qu'une application puisse communiquer avec un serveur de base de données, elle a besoin d'utiliser les pilotes JDBC (Java Data Base Connectivity)
- Les pilotes JDBC est une bibliothèque de classes java qui permet, à une application java, de communiquer avec un SGBD via le réseau en utilisant le protocole TCP/IP
- Chaque SGBD possède ces propres pilotes JDBC.
- Il existe un pilote particulier « JdbcOdbcDriver » qui permet à une application java de communiquer avec n'importe qu'elle source de données via les pilotes ODBC (Open Data Base Connectivity)



PILOTES JDBC

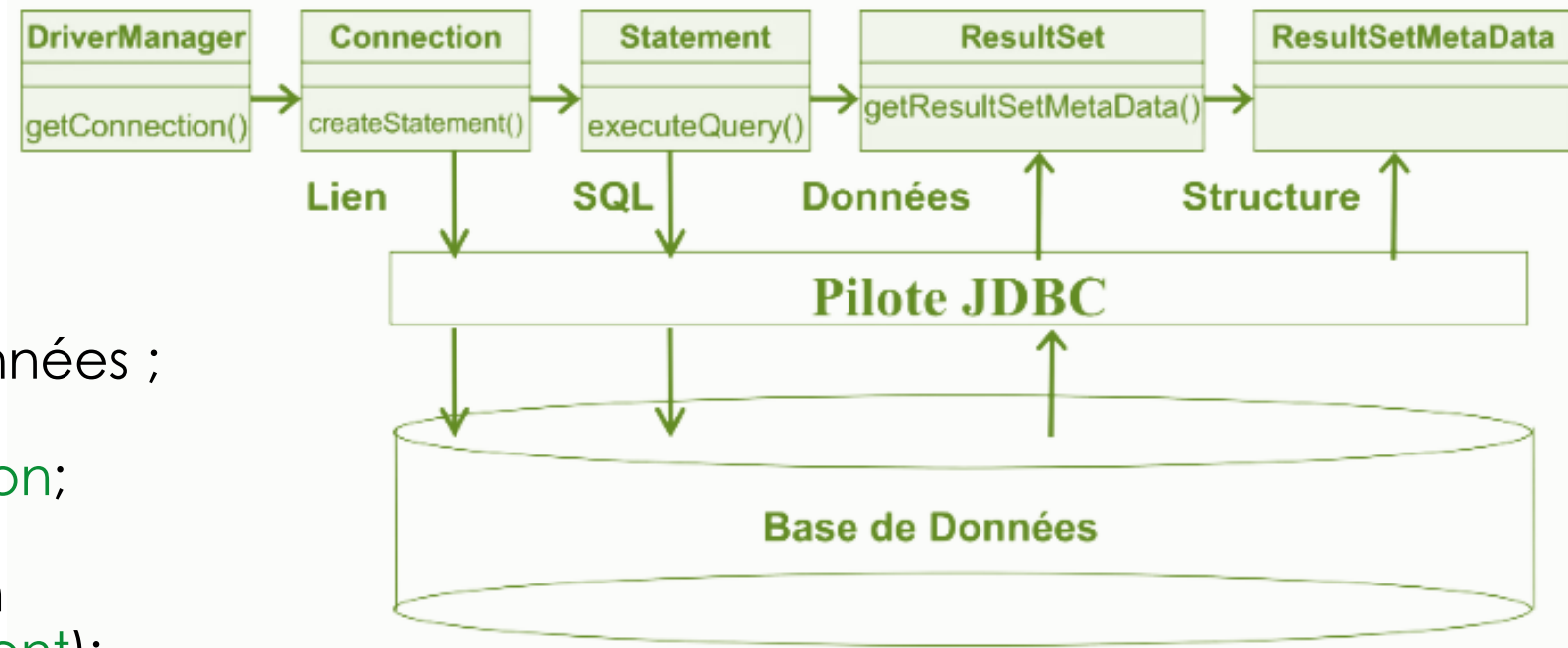
- Les pilotes ODBC permettent à une application Windows de communiquer avec une base de données quelconque (Acces, Excel, Mysql, Oracle, SQL Server etc...)
- La bibliothèque JDBC a été conçue comme interface pour l'exécution de requêtes SQL. Une application JDBC est isolée des caractéristiques particulières du systèmes de base de données utilisé.



CRÉER UNE APPLICATION JDBC

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

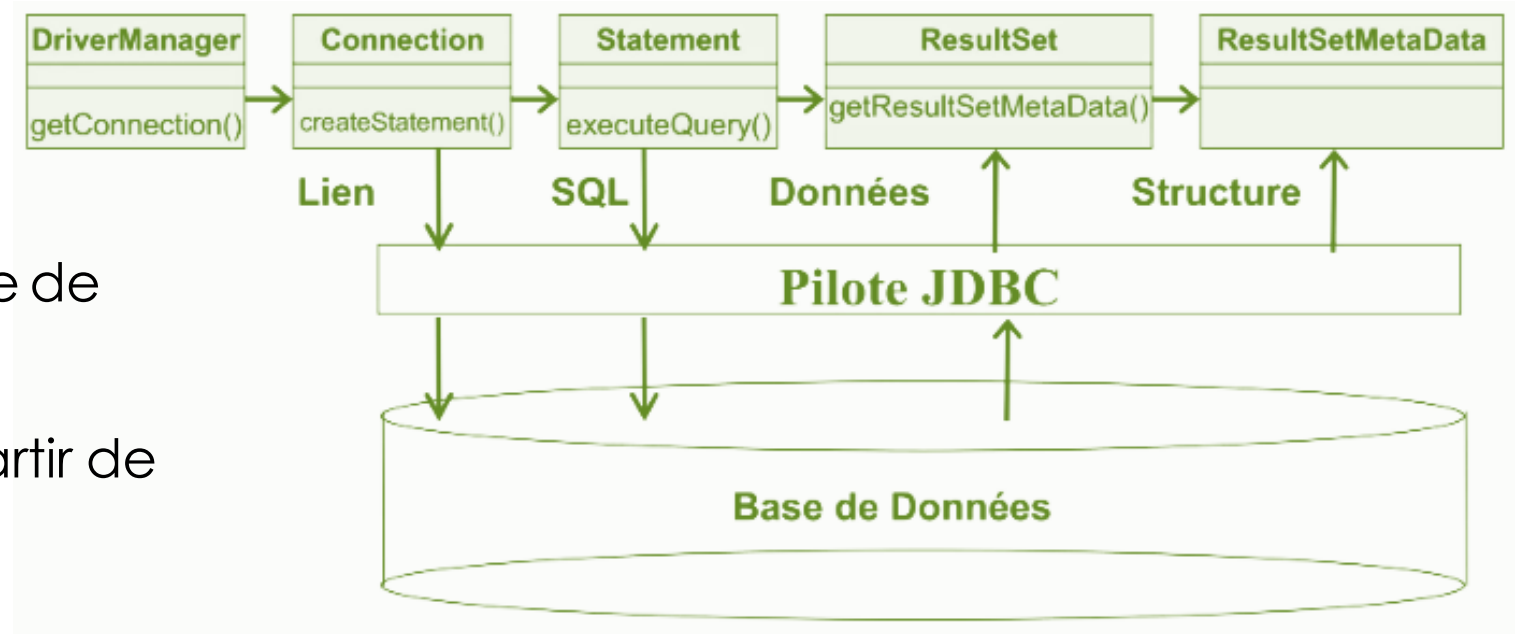
1. Chargement du pilote **JDBC** ;
2. Identification du source de données ;
3. Allocation d'un objet **Connection**;
4. Allocation d'un objet Instruction **Statement** (ou **PreparedStatement**);



CRÉER UNE APPLICATION JDBC

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

5. Exécution d'une requête à l'aide de l'objet **Statement** ;
6. Récupération de données à partir de l'objet envoyé **ResultSet** ;
7. Fermeture de l'objet **ResultSet** ;
8. Fermeture de l'objet **Statement** ;
9. Fermeture de l'objet **Connection** ;



DÉMARCHE JDBC

- **Charger les pilotes JDBC :**

- Utiliser la méthode **forName** de la classe **Class**, en précisant le nom de la classe pilote.

Exemple :

- Pour charger le pilote jdbc de Mysql :

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

CRÉER UNE CONNEXION

- **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 SGBD, en utilisant les sockets.

- Pour un pilote `com.mysql.jdbc.Driver` :

`Connection con= DriverManager . getConnection`

`("jdbc:mysql://localhost:3306/DB","user", "password");`

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 stm=con.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 stock le résultat de la requête dans l'objet **ResultSet**.

```
ResultSet rs=stm.executeQuery( sql: "select * from produits");
```

OBJETS STATEMENT, RESULTSET ET RESULTSETMETADATA

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

```
stm.executeUpdate( sql: "update produits set nom='asus vivo book',marque='asus'," +  
    "prix='2200',quantite='20' where id=1");
```

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

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

OBJET PREPAREDSTATEMENT

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

- Syntaxe de création de l'objet `PreparedStatement`.

```
PreparedStatement pstmt=con.prepareStatement( sql: "select * from produits where marque" +  
" like ? and prix>?");
```

- Définir les valeurs des paramètres de la requête :

```
pstmt.setString( parameterIndex: 1, x: "%" + motCle + "%");  
pstmt.setString( parameterIndex: 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 `Statement`. Cette méthode exécute la requête et stock le résultat de la requête dans l'objet `ResultSet`.

```
pstmt.executeQuery();
```

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

```
pstmt.executeUpdate();
```

CONNEXION À LA BASE DE DONNÉES

```
import java.sql.Connection;

public class App {

    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection connection=DriverManager.getConnection
                ("jdbc:mysql://localhost:3306/catalogue_bd","root","");
            Statement statement=connection.createStatement();
            ResultSet rs=statement.executeQuery("select * from Produits ");
            //pour passer d'une ligne à l'autre et afficher les données
            while(rs.next()) {
                System.out.println(rs.getString("Name_PROD"));
            }
        } catch (Exception e) {

            e.printStackTrace();
        }

    }

}
```

CONNEXION À LA BASE DE DONNÉES

```
import java.sql.*;

public class App {

    public static void main(String[] args) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            Connection connection=DriverManager.getConnection
                ("jdbc:mysql://localhost:3306/catalogue_bd","root","");
            Statement statement=connection.createStatement();
            ResultSet rs=statement.executeQuery("select * from Produits ");
            //Afficher toutes les colonnes
            ResultSetMetaData resultSetMetaData=rs.getMetaData();
            for (int i=1;i<=resultSetMetaData.getColumnCount();i++)
                System.out.print(resultSetMetaData.getColumnName(i)+"\t");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

CONNEXION À LA BASE DE DONNÉES

```
public class App {  
  
    public static void main(String[] args) {  
        try {  
            Class.forName("com.mysql.jdbc.Driver");  
            Connection connection=DriverManager.getConnection  
                ("jdbc:mysql://localhost:3306/catalogue_bd","root","");  
            Statement statement=connection.createStatement();  
            ResultSet rs=statement.executeQuery("select * from Produits ");  
            //Afficher toutes les colonnes  
            ResultSetMetaData resultSetMetaData=rs.getMetaData();  
            for (int i=1;i<=resultSetMetaData.getColumnCount();i++)  
                System.out.print(resultSetMetaData.getColumnName(i)+"\t");  
            System.out.println();  
            //Afficher toutes les lignes  
            while(rs.next()) {  
                for(int i=1;i<=resultSetMetaData.getColumnCount();i++)  
                    System.out.print(rs.getString(i)+"\t");  
                System.out.println();  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

ID_PROD	Name	PRIX	QUANTITE
1	PR1	1546	50
2	PR2	26899	26

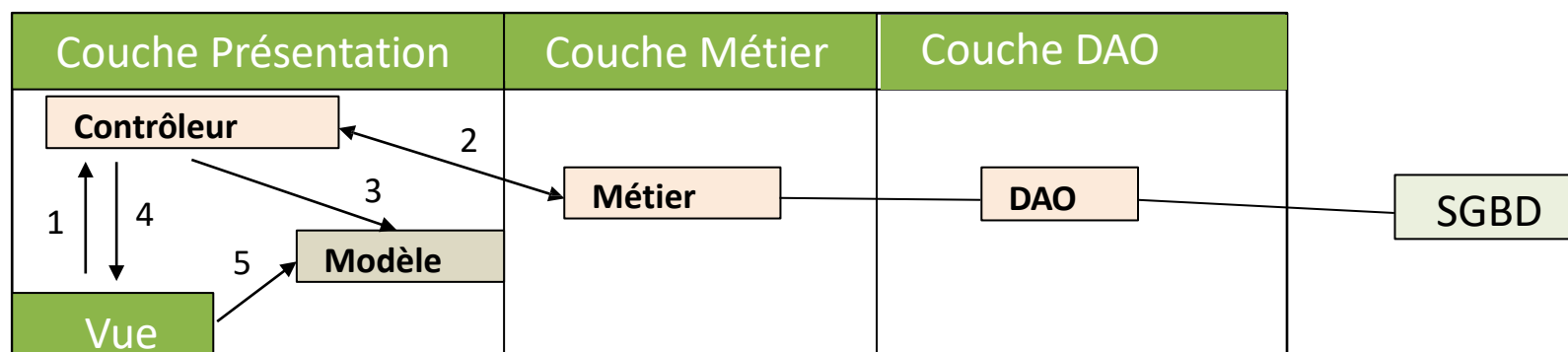
ARCHITECTURE D'UNE APPLICATION

- Dans la pratique, on cherche toujours à séparer la logique métier de la logique présentation.
- On peut diviser une application en trois couches principales :
 - **La couche DAO** qui s'occupe de l'accès aux données de l'application, ces données sont souvent stockées dans une base de données.
 - **La couche Métier** qui s'occupe des traitements que l'application doit effectuer.
 - **La couche présentation** qui s'occupe de la saisie, le contrôle et de l'affichage des résultats.

ARCHITECTURE D'UNE APPLICATION

Généralement la couche présentation respecte le design 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 validation de ces données, il fait appel à la couche métier pour exécuter les traitement.
3. Le contrôleur stocke le résultat dans le 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.



LE MAPPING OBJET RELATIONNEL

- D'une manière générale, les applications sont orientées objet :
 - Manipulation des objets et des classes.
 - Utilisation de l'héritage et l'encapsulation.
 - Utilisation de polymorphisme.
- D'autre 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...).

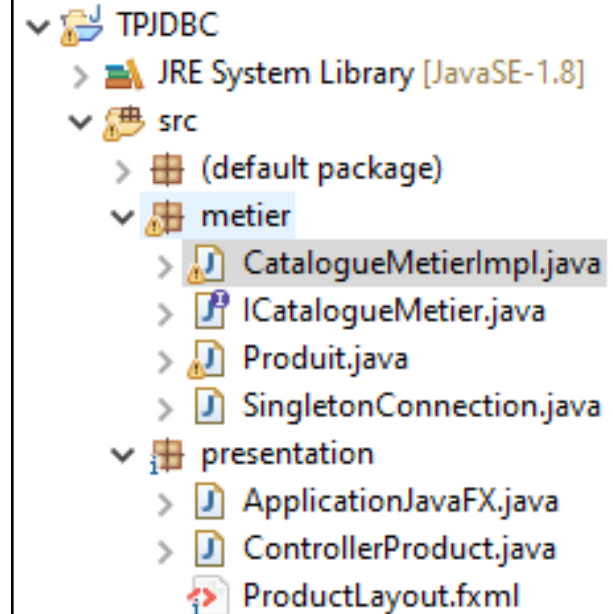
COUCHE MÉTIER

La classe *SingletonConnection* pour déclarer une variable static, pour garantir une seule connection quelque soit le nombre de clients :

```
package metier;

import java.sql.Connection;
import java.sql.DriverManager;

public class SingletonConnection {
    private static Connection connection;
    static {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection=DriverManager.getConnection
                ("jdbc:mysql://localhost:3306/catalogue_bd","root","");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    public static Connection getConnection() {
        return connection;
    }
}
```



COUCHE MÉTIER

La classe Produit

```
package metier;
import java.io.Serializable;
public class Produit implements Serializable {
    private int idProduit;
    private String Name;
    private double prix;
    private int quantite;
    public int getIdProduit() {
        return idProduit;
    }
    public void setIdProduit(int idProduit) {
        this.idProduit = idProduit;
    }
    public String getName() {
        return Name;
    }
    public void setName(String name) {
        Name = name;
    }
    public double getPrix() {
        return prix;
    }
    public void setPrix(double prix) {
        this.prix = prix;
    }
    public int getQuantite() {
        return quantite;
    }
    public void setQuantite(int quantite) {
        this.quantite = quantite;
    }
}
```

```
    public Produit() {
        super();
    }
    public Produit(int idProduit, String name, double prix, int quantite) {
        super();
        this.idProduit = idProduit;
        Name = name;
        this.prix = prix;
        this.quantite = quantite;
    }
}
```

COUCHE MÉTIER

L'interface ICatalogueMetier:

```
package metier;

import java.util.List;

public interface ICatalogueMetier {
    public List<Produit> AllProduits();
    public List<Produit> produitsMC(String mc);
    public void addProduit(Produit p);
}
```

COUCHE MÉTIER

Implémentation de l'interface ICatalogueMetier :

```
public class CatalogueMetierImpl implements ICatalogueMetier{

    @Override
    public List<Produit> produitsMC(String mc) {
        List<Produit> products=new ArrayList<>();
        try {
            //Charger le pilote et établir la connexion
            Connection connect=SingletonConnection.getConnection();
            PreparedStatement p=connect.prepareStatement
                ("select * from produits where Name like ?");
            p.setString(1, "%" +mc+"%");
            ResultSet r=p.executeQuery();
            while(r.next()) {
                Produit pr=new Produit();
                pr.setIdProduit(r.getInt("ID_PROD"));
                pr.setName(r.getString("Name"));
                pr.setPrix(r.getDouble("PRIX"));
                pr.setQuantite(r.getInt("QUANTITE"));
                products.add(pr);
            }
            p.close();
            connect.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return products;
    }
}
```

COUCHE MÉTIER

Implémentation de l'interface ICatalogueMetier (Suite):

```
@Override
public List<Produit> AllProduits() {
    Connection connect=SingletonConnection.getConnection();
    List<Produit> products=new ArrayList<>();
    try {
        PreparedStatement p=connect.prepareStatement("select * from produits");
        ResultSet resultSet=p.executeQuery();
        while(resultSet.next()) {
            Produit p1=new Produit(resultSet.getInt("ID_PROD"),
                                    resultSet.getString("Name"),resultSet.getDouble("PRIX"),
                                    resultSet.getInt("QUANTITE"));
            products.add(p1);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return products;
}
```

COUCHE MÉTIER

Implémentation de l'interface ICatalogueMetier (Suite):

```
@Override
public void addProduit(Produit prod) {
    try {
        Connection connect=SingletonConnection.getConnection();
        PreparedStatement p=connect.prepareStatement
            ("insert into produits (Name, PRIX, QUANTITE) values (?, ?, ?)");
        p.setString(1, prod.getName());
        p.setDouble(2, prod.getPrix());
        p.setInt(3, prod.getQuantite());
        //un entier qui représente les enregistrements mis à jour
        int nmbre=p.executeUpdate();
        connect.close();
    } catch (Exception e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

COUCHE PRÉSENTATION

Application JavaFx:

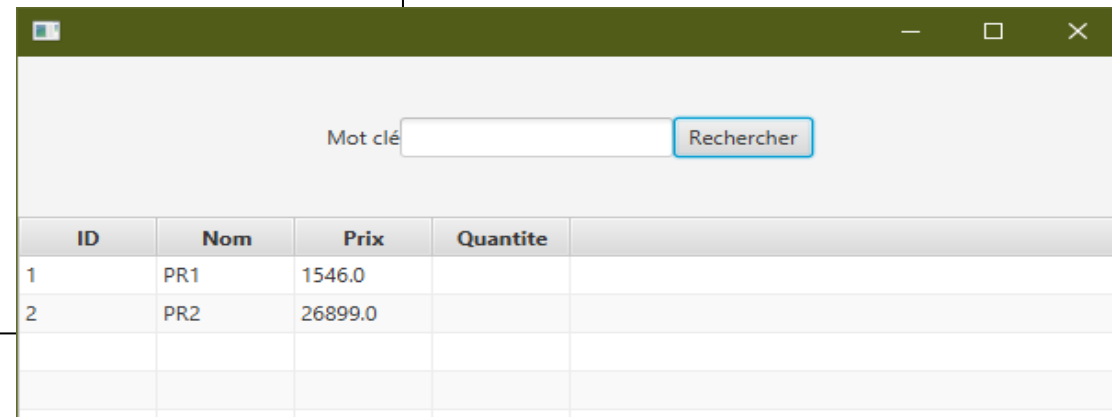
```
public class ApplicationJavaFX extends Application {  
  
    public static void main(String[] args) {  
        launch(args);  
    }  
  
    @Override  
    public void start(Stage primaryStage) throws Exception {  
        BorderPane borderPaneRoot=FXMLLoader.load(getClass()  
            .getResource("ProductLayout.fxml"));  
        Scene scene=new Scene(borderPaneRoot,600,400);  
        primaryStage.setScene(scene);  
        primaryStage.show();  
    }  
}
```


COUCHE PRÉSENTATION

Le fichier ProductLayout.fxml :

```
<BorderPane xmlns="http://javafx.com/javafx/8.0.171" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="presentation.ControllerProduct">

    <top>
        <HBox alignment="CENTER" prefHeight="100.0" prefWidth="200.0" BorderPane.alignment="CENTER">
            <children>
                <Label text="Mot clé" />
                <TextField fx:id="fieldMC" alignment="TOP_CENTER"/>
                <Button fx:id="Research" mnemonicParsing="false" onAction="#Rechercher" text="Rechercher"/>
            </children>
        </HBox>
    </top>
    <center>
        <TableView fx:id="ProductsTable" prefHeight="200.0" prefWidth="200.0" BorderPane.alignment="CENTER">
            <columns>
                <TableColumn fx:id="IDColumn" prefWidth="75.0" text="ID"/>
                <TableColumn fx:id="NameColumn" prefWidth="75.0" text="Nom"/>
                <TableColumn fx:id="PriceColumn" prefWidth="75.0" text="Prix"/>
                <TableColumn fx:id="QuantityColumn" prefWidth="75.0" text="Quantite"/>
            </columns>
        </TableView>
    </center>
</BorderPane>
```



The screenshot shows a JavaFX application window with a title bar. The main content area has a light gray background. At the top, there is a search bar with the text "Mot clé" and a button labeled "Rechercher". Below the search bar is a table with four columns: "ID", "Nom", "Prix", and "Quantite". The table contains two rows of data.

ID	Nom	Prix	Quantite
1	PR1	1546.0	
2	PR2	26899.0	

COUCHE PRÉSENTATION

Le contrôleur ControllerProduct:

```
public class ControllerProduct implements Initializable{

    @FXML
    private TextField fieldMC;
    @FXML
    private Button Research;
    @FXML
    private TableView<Produit> ProductsTable;
    @FXML
    private TableColumn<Produit, Integer> IDColum;
    @FXML
    private TableColumn<Produit, String> NameColumn;
    @FXML
    private TableColumn<Produit, Double> PriceColumn;
    @FXML
    private TableColumn<Produit, Integer> QuantityColumn;
    ObservableList<Produit> observableList1=
        FXCollections.observableArrayList();
    private ICatalogueMetier metier;
    @Override
    public void initialize(URL location, ResourceBundle resources) {
        IDColum.setCellValueFactory(new PropertyValueFactory<>("idProduit"));
        NameColumn.setCellValueFactory(new PropertyValueFactory<>("Name"));
        PriceColumn.setCellValueFactory(new PropertyValueFactory<>("prix"));
        QuantityColumn.setCellValueFactory(new PropertyValueFactory<>("quantité"));
        metier=new CatalogueMetierImpl();
        observableList1.addAll(metier.AllProduits());
        ProductsTable.setItems(observableList1);
    }

    public void Rechercher() {
        String mot_cle=fieldMC.getText();
        List<Produit> produits=metier.produitsMC(mot_cle);
        observableList1.clear();
        observableList1.addAll(produits);
    }
}
```