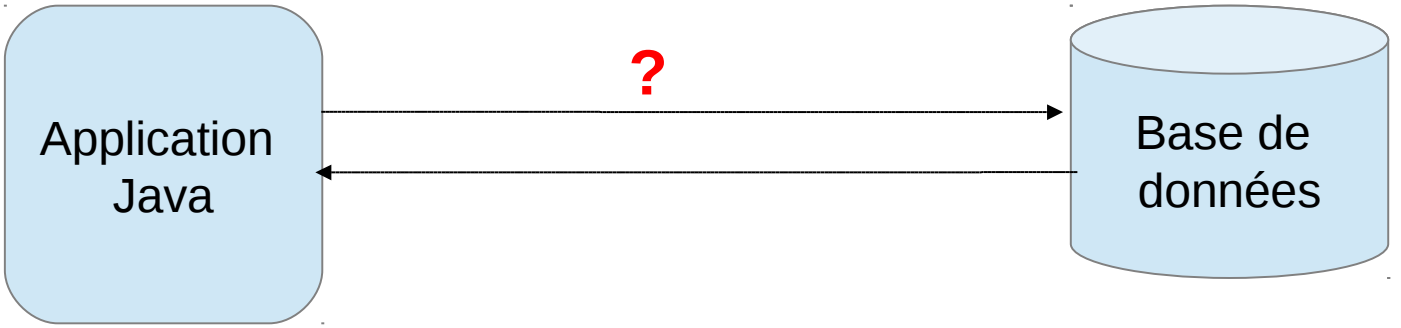
Java et les bases de données

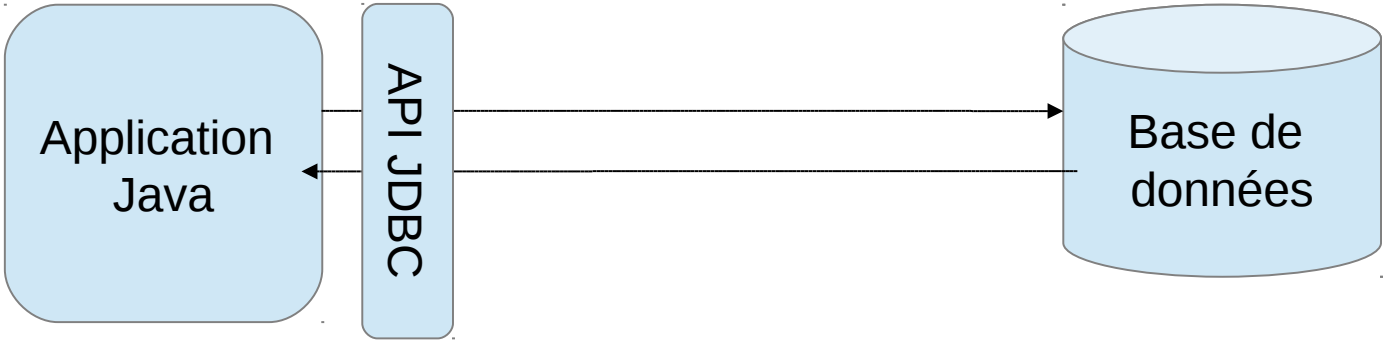
Introduction



Besoin d'un framework permettant l'accès aux bases de données (SQL) dans un programme Java :

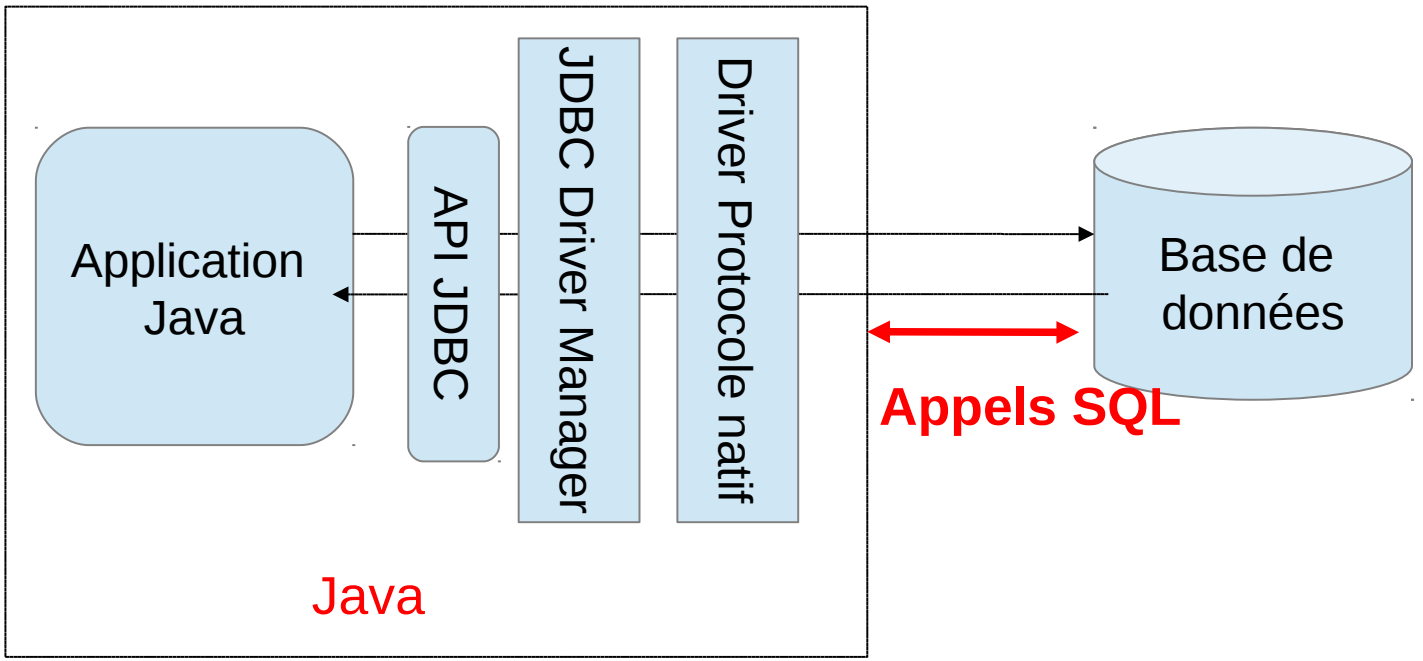
- Indépendamment du type de la base utilisée (mySQL, Oracle, Postgres ...)
- Permet de faire tout type de requêtes (sélection de données, création de table, transactions)

JDBC : Java DataBase Connectivity



- JDBC est une API Java (ensemble de classes et d'interfaces définies par SUN et les acteurs du domaine des BD) permettant d'accéder aux bases de données à l'aide du langage Java via des requêtes SQL. Cette API permet d'atteindre de manière quasi-transparente des bases mySQL, Sybase, Oracle, ... avec le même programme Java JDBC.

- JDBC fait partie du JDK (Java Development Kit).
Paquetage `java.sql : import java.sql.*;`



- Le protocole va établir le lien avec la base de données, en sachant « lui parler ».
Dans JDBC : des classes chargées de gérer un pilote...
Des pilotes existent pour mySQL, postGresSQL, ACCESS,...

- Le `DriverManager` est une classe qui ne contient que des méthodes statiques. Elle fournit des méthodes qui sont des utilitaires pour gérer l'accès aux bases de données par Java et les différents drivers JDBC à l'intérieur d'un programme. Finalement on ne créée ni ne récupère d'objet de cette classe.

- La connexion **ne** peut s'établir **que** si l'on donne l'adresse de la BD a laquelle on veut se connecter...

Principes généraux d'accès à une BDD

- Première étape:

* Préciser le type de driver que l'on veut utiliser
    - Un driver permet de gérer l'accès à un type particulier de SGBD

- Deuxième étape:

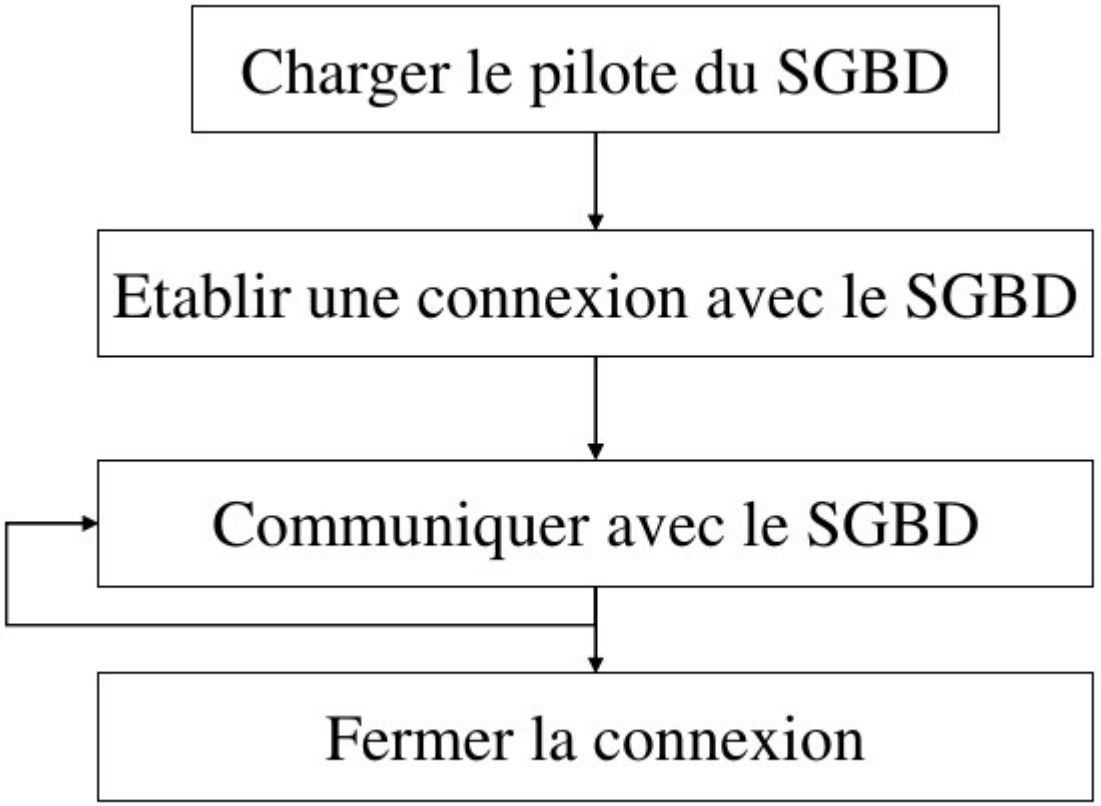
* Récupérer un objet « Connection » en s'identifiant auprès du SGBD et en précisant la base utilisée

- Étapes suivantes:

* A partir de la connexion, créer un « statement » (état) correspondant à une requête particulière
* Exécuter ce statement au niveau du SGBD
* Fermer le statement

- Dernière étape:

* Se déconnecter de la base en fermant la connexion


Charger le pilote du SGBD
Etablir une connexion avec le SGBD
Communiquer avec le SGBD
Fermer la connexion

Exemple

1) charger un pilote driver

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

2) créer un objet Connection

```
Connection maConnection = DriverManager.getConnection(url);
```

`url` : String contenant l'adresse de la base de données

3) créer un objet Statement

```
Statement maRequeteSQL = maConnection.createStatement();
```

4) envoyer la requête et recupérer le résultat dans un `ResultSet`

```
ResultSet monResultat=
maRequeteSQL.executeQuery(texteRequeteSQL);
```

`texteRequeteSQL` : String contenant le texte de la requête,par exemple : "SELECT * FROM Client"

Mise en place du pilote

- Ensemble des classes qui implémentent les interfaces spécifiées par JDBC pour un gestionnaire de bases de données particulier.

- Les protocoles d'accès aux BD étant propriétaires, il y a donc plusieurs drivers pour atteindre diverses BD.

- Le premier pilote (développé par SUN) est le pilote `jdbc:odbc`.
La liste des pilotes disponibles se trouve ici:
http://www.oracle.com/technetwork/indexes/downloads/index.html#database

- Le pilote JDBC est un composant logiciel qui satisfait aux spécifications JDBC établies par Sun. Ce pilote est spécifique à un fabriquant de base de donnée. On trouvera par exemple un pilote pour les bases de données Oracle, un pilote pour la base MySQL... Il doit être ajouté au CLASSPATH.

- Ce pilote est une classe Java qui implémente l'interface `java.sql.Driver`. Pour le driver MySQL, cette classe est : `com.mysql.jdbc.Driver`, elle doit donc être chargée. Pour cela, on peut utiliser la réflexivité en appelant la méthode `Class.forName()`.

Chargement du pilote (*driver*) de MySQL

```
import java.sql.*;
public class QueryExample {
...
try {
Class.forName("com.mysql.jdbc.Driver");
}
catch (ClassNotFoundException e) {
System.err.println("Driver loading error : " + e);
}
...
}
```

Chargement du pilote (*driver*) de Oracle

```
import java.sql.*;
public class QueryExample {
...
try {
Class.forName("oracle.jdbc.driver.OracleDriver");
}
catch (ClassNotFoundException e) {
System.err.println("Driver loading error : " + e);
}
...
}
```

Chargement du pilote (*driver*) de Microsoft

```
import java.sql.*;
public class QueryExample {
...
try {
Class.forName("connect.micorsoft.MicrosoftDriver");
}
catch (ClassNotFoundException e) {
System.err.println("Driver loading error : " + e);
}
...
}
```

URL de connection

- Afin de localiser la base de donnée sur le serveur de base de données, il est indispensable de spécifier une adresse de connexion sous la forme d'une URL. Ces URL commenceront toutes par "`jdbc:`".

- Généralement, il faudra se référer à la documentation du fournisseur du driver afin de connaître le format de l'URL à utiliser.

- Pour MySQL, l'URL est la suivante :
`jdbc:mysql://host:port/database.`

    * Host : correspond à l'adresse IP du serveur.

    * port : port MySQL ou port par défaut.

    * database : le nom de la base de donnée à laquelle on doit se connecter.

Établir une connection

- Établir une connexion en utilisant 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.

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

Les requêtes

- Afin d'exécuter des requêtes, il convient d'utiliser un object `Statement`. Une instance de cet objet est retournée par un appel à la méthode `Connection.createStatement()` :

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

- Il existe deux types de requêtes :
    * des requêtes de sélection (SELECT), accessibles par la méthode `statement.executeQuery()`. Cette méthode retourne un résultat de type `java.sql.ResultSet` contenant les lignes sélectionnées.

    * des requêtes de modification (UPDATE), d'insertion (INSERT) ou de suppression (DELETE), accessibles par la méthode `statement.executeUpdate()`. Cette méthode retourne un résultat de type `int` correspondant au nombre de lignes affectées par la requête.

- Sélection

```
String query = "SELECT depart,arrivee FROM aeroport";
try {
ResultSet resultSet = statement.executeQuery(query);
}
catch (SQLException e) {
System.err.println("Error executing query: " +
e.getMessage());
}
```

- Modification

```
String query = "UPDATE vols SET
arrivee='toulouse' WHERE depart='Paris'";
try {
int result = statement.executeUpdate(query);
}
catch (SQLException e) {
System.err.println("Error executing query: " +
e.getMessage());
}
```

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).

- 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. Un tableau plus loin précise les relations entre type SQL, type JDBC et méthode à appeler sur l'objet ResultSet.

Les types

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()

BLOB (*Binary Large Object*) : permet de stocker de grandes quantités de données sous forme binaire (fichiers images, musique, etc.).

Quelques méthodes

- `boolean next()` : se dplace sur la ligne suivante s'il y en a une. Elle retourne `true` si le déplacement a été fait, `false` s'il n'y avait pas d'autre ligne.

- `boolean previous()` : se dplace sur la ligne précédente s'il y en a une. Elle retourne `true` si le déplacement a été fait, `false` s'il n'y avait pas de ligne précédente.

- `boolean absolute(int index)` : se dplace sur la ligne numérotée `index`. Elle retourne `true` si le déplacement a été fait, `false` sinon.

- `void close()` : ferme le `ResultSet`.

Exemple

```
try {
    while(resultSet.next()) {
    System.out.println(resultSet.getString("id_vol") + "\t" +
            resultSet.getString("depart")+"\t" +
            resultSet.getString("arrivee") );
    }
        }
catch (SQLException e) {
}
```

Fermeture de la connexion

Objectif : Libérer les ressources

```
try {
statement.close();
connection.close();
}
catch (SQLException e) {
System.err.println("Error closing connection:
" + e.getMessage());
}
```

Exercice

On va créer une application pour la gestion des vols depuis et vers un aéroport donné. Ce programme doit communiquer avec une base de données contenant les vols en cours, et même les prochains vols de la journée, avec pour chaque vol le numéro de vol, la ville de départ et la ville d'arrivée. Ce programme permet aussi de mettre à jour la base de données pour ajouter des vols, modifier les villes de destination pour les vols qui subissent un changement d'itinéraire,...

```
String sql2 = "UPDATE lignes SET arrivee='Nice' WHERE depart='Toulouse'";
String sql3 = "INSERT INTO lignes  VALUES (212,'Bordeaux','Marseille','13:00')";
```

```
public static void main(String[] args) {
        String driver = "com.mysql.jdbc.Driver";
        String url = "jdbc:mysql://localhost/Vol";
        String sql = "SELECT * from lignes";
        String sql3 = "INSERT INTO lignes  VALUES
(219,'Bordeaux','Lyon','17:00')";
        String query = "UPDATE lignes SET arrivee ='toulouse' WHERE
depart ='Paris'";
        try {Class.forName(driver);
        } catch (ClassNotFoundException e) {
            System.err.println("Driver loading error : " + e);}
        try {Connection connection = DriverManager.getConnection(url,
"root", "xxxxx");
            Statement statement = connection.createStatement();
             int result = statement.executeUpdate(sql3);
             ResultSet r = statement.executeQuery(sql);
             while(r.next()) {System.out.println(r.getString("id") +
             "\t" + r.getString("depart")+"\t" +
                    r.getString("arrivee") + "\t" +
                   r.getString("heure") );}
            r.close();
            statement.close();
            connection.close();
        } catch (SQLException e) {
            System.err.println("Error opening SQL connection: "
                    + e.getMessage());
        }
```

Instructions SQL paramétrées

- La plupart des SGBD ne peuvent analyser qu'une seule fois une requête exécutée un grand nombre de fois.

- JDBC permet de profiter de ce type de fonctionnalité par l'utilisation de requêtes paramétrées ou de procédures stockées.

- Les requêtes paramétrées sont associées aux instances de l'interface `PreparedStatement` qui hérite de l'interface `Statement` :

```
PreparedStatement pstmt =
connexion.preparedStatement("UPDATE emp SET sal=? WHERE
nom=?;");
```

- Les '?' indiquent les emplacements des paramètres.

- Les valeurs des paramètres sont données par les méthodes `setXXX(n,valeur)`.

- On choisit la méthode `setXXX` suivant le type SQL de la valeur que l'on veut mettre dans la requête.

- C'est au développeur de passer une valeur Java cohérente avec le type.

Exemple

```
PreparedStatement pstmt =
conn.preparedStatement("UPDATE emp SET sal=? WHERE nom=?;");

for(int i=0; i<=10,i++)
{
   pstmt.setDouble(1, salaire[i]);
   pstmt.setString(2,nom[i]);
   pstmt.executeUpdate();
}
```

- On peut passer la valeur NULL avec `setNull(n,type)` (type de la classe Types).

Avantages des requêtes paramétrées :

- Traitement plus rapide si elles sont utilisées plusieurs fois avec plusieurs paramètres.

- Amélioration de la portabilité (indépendance des `setXXX` avec les SGBD).

Exercice

Commencer par créer une base de données d'employés pour une entreprise. Cette base contiendra les champs suivants :
- Id : entier unique
- nom : String
- catégorie : 1,2,3...
- salaire : réel positif

1) Écrire ensuite le programme Java qui permet de communiquer avec cette base.
2) Écrire le code permettant d'afficher l'ensemble des salariés appartenant à des catégories que l'utilisateur spécifiera au clavier.
3) Écrire le code permettant de promouvoir un ou plusieurs employés d'une catégorie à la catégorie suivante. La base de donnée doit être mise à jour après la promotion.

Mettez en œuvre un design pattern adapté à ce problème pour garantir l'extensibilité et la maintenabilité du programme, éventuellement par rapport à un changement de type de base de données.