



Programmation en Java (API)

Par cysboy



*Licence Creative Commons BY-NC-SA 2.0
Dernière mise à jour le 13/12/2011*

Sommaire

Sommaire	2
Partager	1
Programmation en Java (API)	3
Partie 1 : Java DataBase Connectivity	4
En premier : la base de données	4
Une base de données, quésaco ?	4
Laquelle utiliser	5
Installation de PostgreSQL	5
PostgreSQL	10
Préparer sa BDD	10
Créer la BDD	11
Créer ses tables	13
Se connecter à sa BDD	20
Faisons le point	21
Connexion ! Es-tu là ?	23
Fouiller dans sa BDD	25
Le couple Statement - ResultSet	26
Comment ça fonctionne	28
Entraînons-nous	30
Allons un peu plus loin	36
Statement	36
Les requêtes préparées	36
ResultSet 2 : le retour	39
Après la lecture : l'édition	42
Modifier des données	43
Statement, toujours plus fort	45
Gérer les transactions manuellement	47
N'avoir qu'une instance de sa connexion	50
Pourquoi se connecter qu'une seule fois ?	51
Le pattern singleton	51
Le singleton dans tous ces états	54
TP : un testeur de requête	57
Cahier des charges	58
Quelques captures d'écran	58
Correction	58
Le pattern DAO (1/2)	62
Avant toute chose	63
Le pattern DAO : définition	70
Contexte	70
Le pattern DAO	70
Premier test	78
Le pattern DAO (2/2)	80
Le pattern factory	81
Fabriquer vos DAO	82
D'une usine à une multinationale	85



Programmation en Java (API)

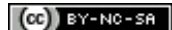
Par



cysboy

Mise à jour : 17/06/2009

Difficulté : Difficile



2 766 visites depuis 7 jours, classé 50/782

Bonjour amis ZérOs.

Me revoici pour un autre big-tuto.

Comme toujours, celui-ci concerne Java. Si vous ne connaissez rien à ce langage, je vous invite à lire [ce tuto](#).

Par contre, si c'est à un point particulier du langage, aux API standards présentes dans le JDK que vous souhaitez vous intéresser, vous êtes à la bonne place !

Nous allons voir ce qu'il vous faut pour utiliser des bases de données, comment faire de la programmation réseau... Nous nous baladerons aussi dans quelques *frameworks* (boîte à outils).

Tout ceci est présent dans le JDK, dans les API standards. Je vous propose d'en faire un tour d'horizon, le tout en partant de ZérO, bien évidemment ! 😊

Partie 1 : Java DataBase Connectivity

Dans cette partie nous allons voir comment interagir avec des bases de données via Java !
Il n'y a rien de compliqué puisque tous les objets dont vous aurez besoin existent déjà...

Il ne vous reste plus qu'à apprendre à les utiliser.

Toutefois, pour les ZérOs qui n'y connaissent rien en bases de données, un petit rappel s'impose...

Celui-ci sera bref et succinct, mais je vous laisserai des liens pour améliorer vos connaissances en la matière. 😊

En premier : la base de données

Avant de voir comment Java utilise les bases de données, il nous en faut une !

Sinon, vous connaissez le proverbe : *pas de bras, pas de chocolat...*

Nous allons donc voir à quoi sert une base de données, mais aussi en installer une afin de pouvoir illustrer la suite de cette partie.

Allez ! Zou...

Une base de données, quésaco ?

Lorsque vous réalisez un logiciel, un site web, ou je ne sais quoi d'autre, vous êtes confrontés, à un moment ou un autre, à une question : *comment vais-je faire pour sauvegarder mes données ?*

Les bases de données (ou BDD) sont une alternative sûre et pérenne.

Ce système de stockage des données existe depuis très longtemps et a fait ses preuves ! Ceux qui souhaiteraient en savoir plus sur les BDD peuvent suivre [ce lien](#).



Mais concrètement, comment ça fonctionne ?

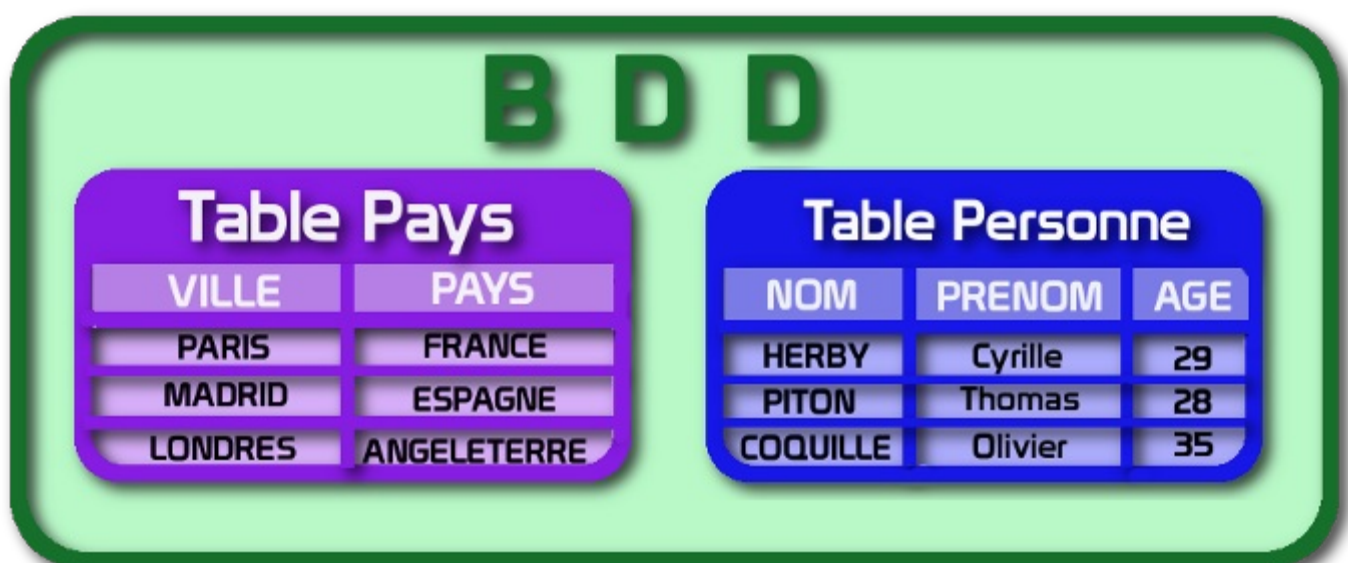
Résumé à l'extrême, il s'agit d'un système de fichiers qui contiennent les données de votre application. Mais ces fichiers sont totalement transparents pour l'utilisateur d'une base de données, donc, **totalement transparents pour VOUS !**

Ces données sont ordonnées par "**tables**", c'est-à-dire par regroupements de plusieurs valeurs.

C'est vous qui allez créer vos propres tables, en spécifiant quelles données vous souhaitez y intégrer.

En fait, imaginez qu'une base de données est une gigantesque armoire à tiroirs dont vous spécifiez les noms et qui contiendront une multitude de fiches dont vous spécifierez aussi leur contenu !

Je sais, un schéma est toujours le bienvenu :



Dans cette base de données, nous avons deux tables : une qui a pour rôle de stocker des personnes avec noms, prénoms et âges, et une table qui s'occupe de stocker des pays, avec leur nom et leur capitale !

Si je reprends ma comparaison de tout à l'heure, la BDD symbolise l'armoire, chaque table un tiroir et chaque ligne de chaque table, une fiche de ce tiroir !

Ensuite, ce qui est formidable avec les BDD, c'est que vous pouvez les interroger en leur posant des questions via un langage. Vous pouvez leur demander des trucs comme :

- donne-moi la fiche de la table **Personne** pour le nom **HERBY**;
- donne-moi la fiche de la table **Pays** pour le pays **France**;
- ...

Le langage qui vous permet d'interroger des bases de données est le langage SQL, où, en français, "Langage de Requête Structurées". Nous aurons l'occasion d'en faire un bref rappel lors du chapitre suivant... 😊

Ainsi, grâce aux BDD, vos données sont stockées, classées par vos soins et identifiables facilement sans avoir à gérer notre propre système de fichiers.



C'est vrai que c'est alléchant, mais comment fait-on pour s'en servir ? On doit télécharger quelque chose ?

Oui, deux choses en fait : la base de données et ce qu'on appelle le SGBD.

Laquelle utiliser

Ben... il existe plusieurs bases de données et toutes sont utilisées par beaucoup de développeurs !

En voici une liste non exhaustive recensant les principales :

- PostgreSQL
- MySQL
- SQL Server
- Oracle
- Access
- ...

Toutes permettent de faire ce que je vous expliquais plus haut.

Chacune a des spécificités :

- certaines sont payantes (Oracle) ;
- certaines sont assez laxistes avec les données qu'elles contiennent (MySQL) ;
- d'autres ont un système de gestion très simple à utiliser (MySQL) ;
- ...

C'est à vous de faire votre choix en regardant sur le web ce que les utilisateurs en disent.

Sinon, il y a un comparatif intéressant, [ici](#).

Pour le tuto sur JDBC, mon choix s'est porté sur PostgreSQL.

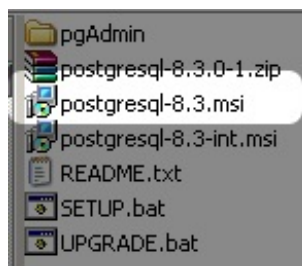
Alors continuons.

Installation de PostgreSQL

Vous pouvez télécharger une version de PostgreSQL [ici](#).

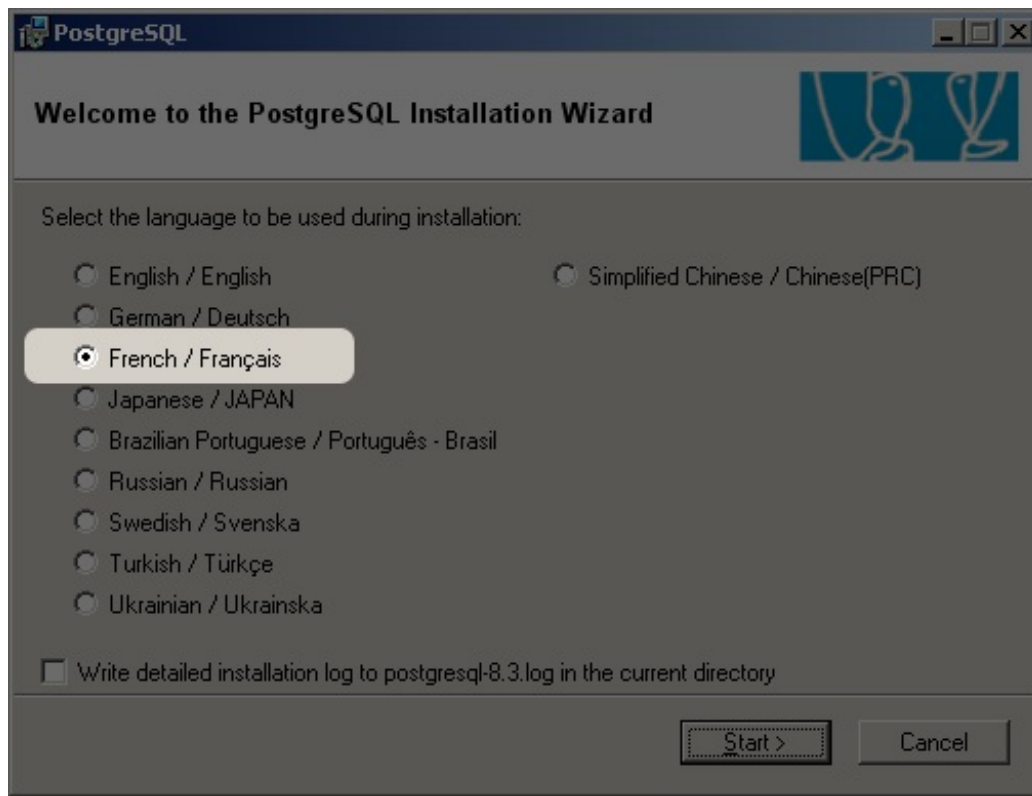
Celle-ci est pour Windows, mais pour les autres OS, vous pouvez faire un tour [par-là](#).

Ensuite, je vous invite à décompresser l'archive téléchargée et à exécuter ce fichier :

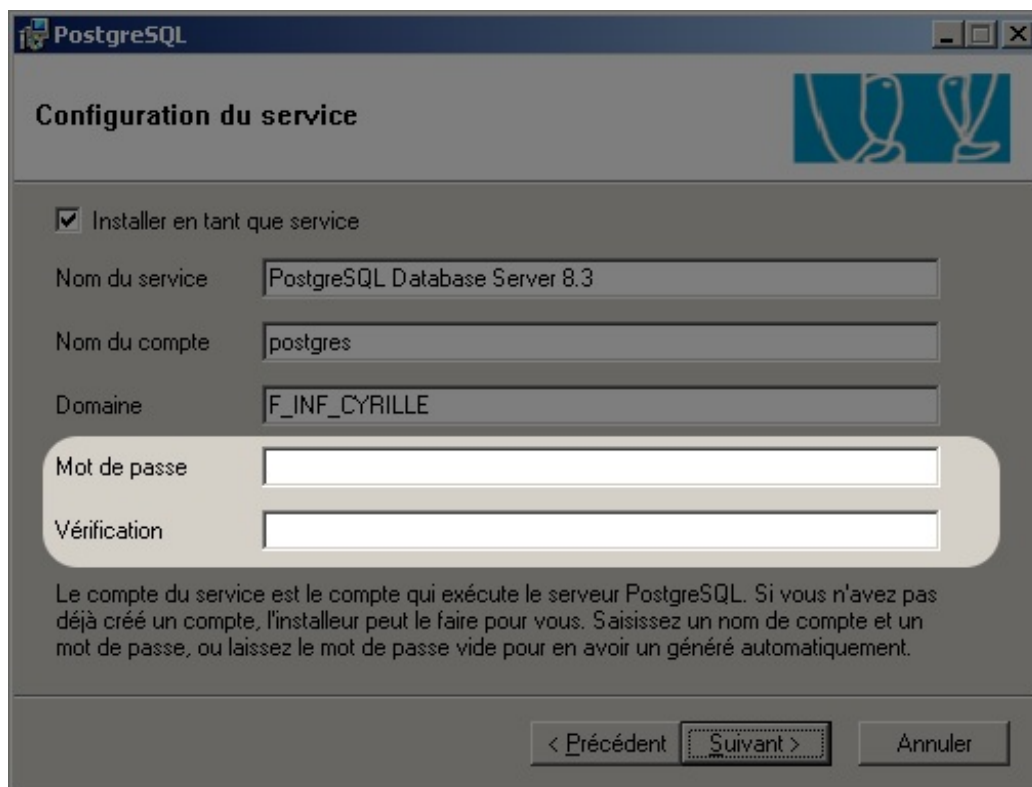


À partir de maintenant, si je ne vous spécifie pas de fenêtre particulière, vous pouvez laisser les réglages par défaut.

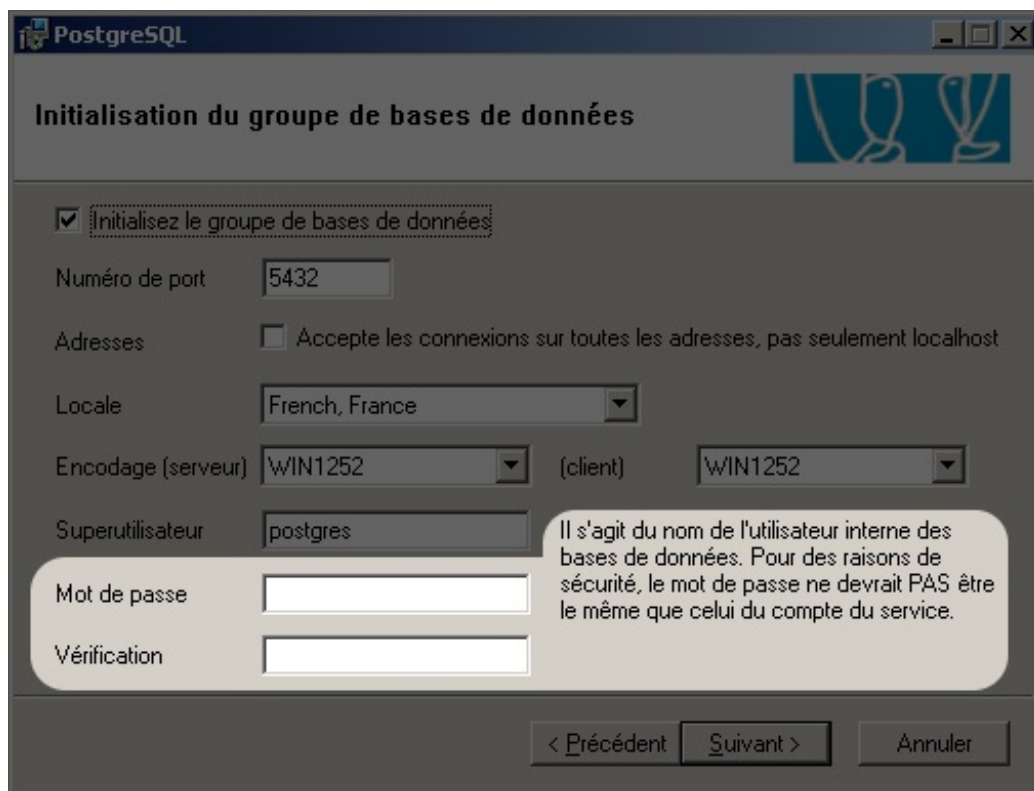
L'installation commence en vous demandant votre langue, choisissez la vôtre :



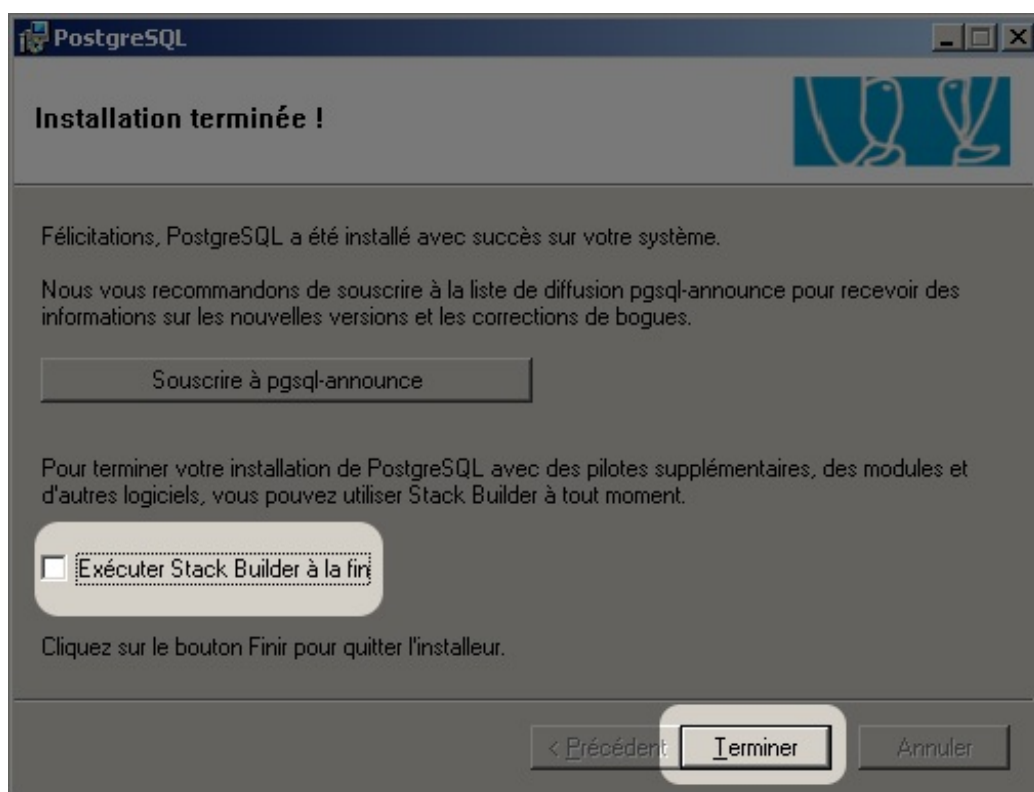
On vous demandera, par la suite, de saisir un mot de passe pour l'utilisateur, comme ceci :



Et un autre pour le super-administrateur :



À la fin de la pré-installation, on vous demandera si vous voulez exécuter le "**Stack Builder**"; ce n'est pas nécessaire, ça permet juste d'installer d'autres logiciels en rapport avec PostgreSQL... Nous n'en avons pas besoin. 😊

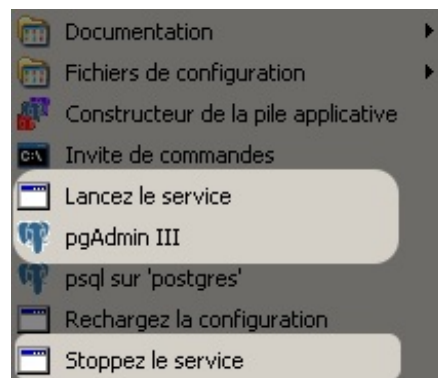


Normalement, le serveur est installé et je dirais même qu'il en est de même pour le SGBD !



Ah bon ?

Oui, regardez dans le menu "**Démarrer**" et allez dans "**Tous les programmes**", vous devriez avoir ceci dans l'encart "**PostgreSQL 8.3**" :

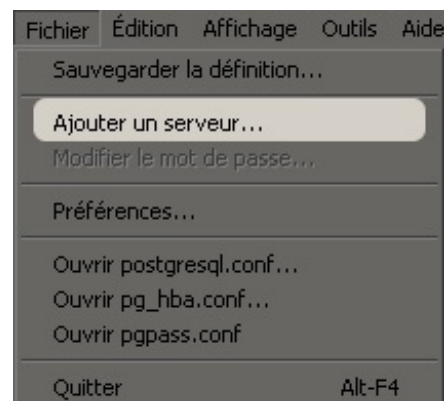


Il y a deux exécutables qui permettent respectivement de lancer le serveur ou de l'arrêter et le dernier, **pgAdmin III**, c'est notre SGBD !

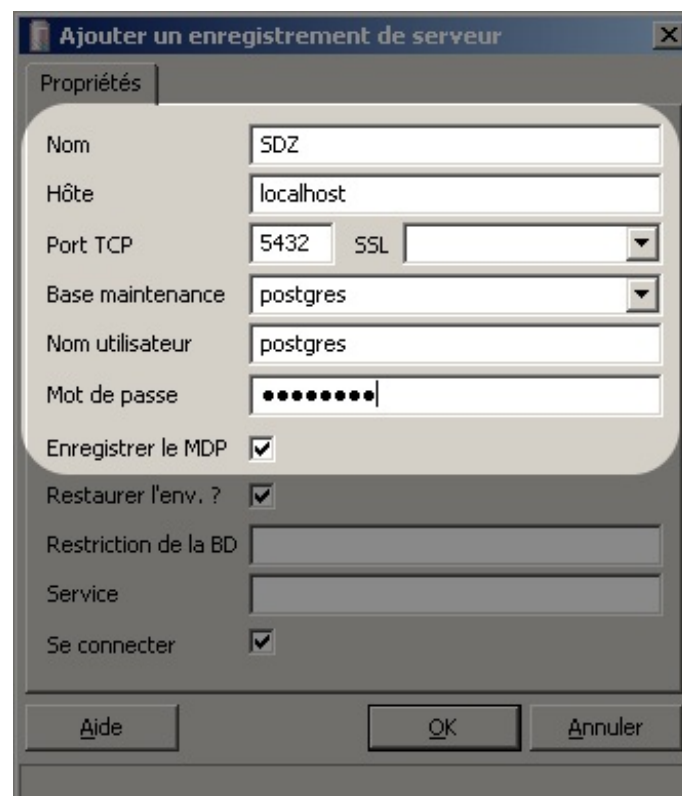
Lancez-le, nous allons configurer notre serveur.

C'est bon ? Vous êtes prêts ?

Alors, dans le menu "**Fichier**", choisissez "**Nouveau serveur**" :

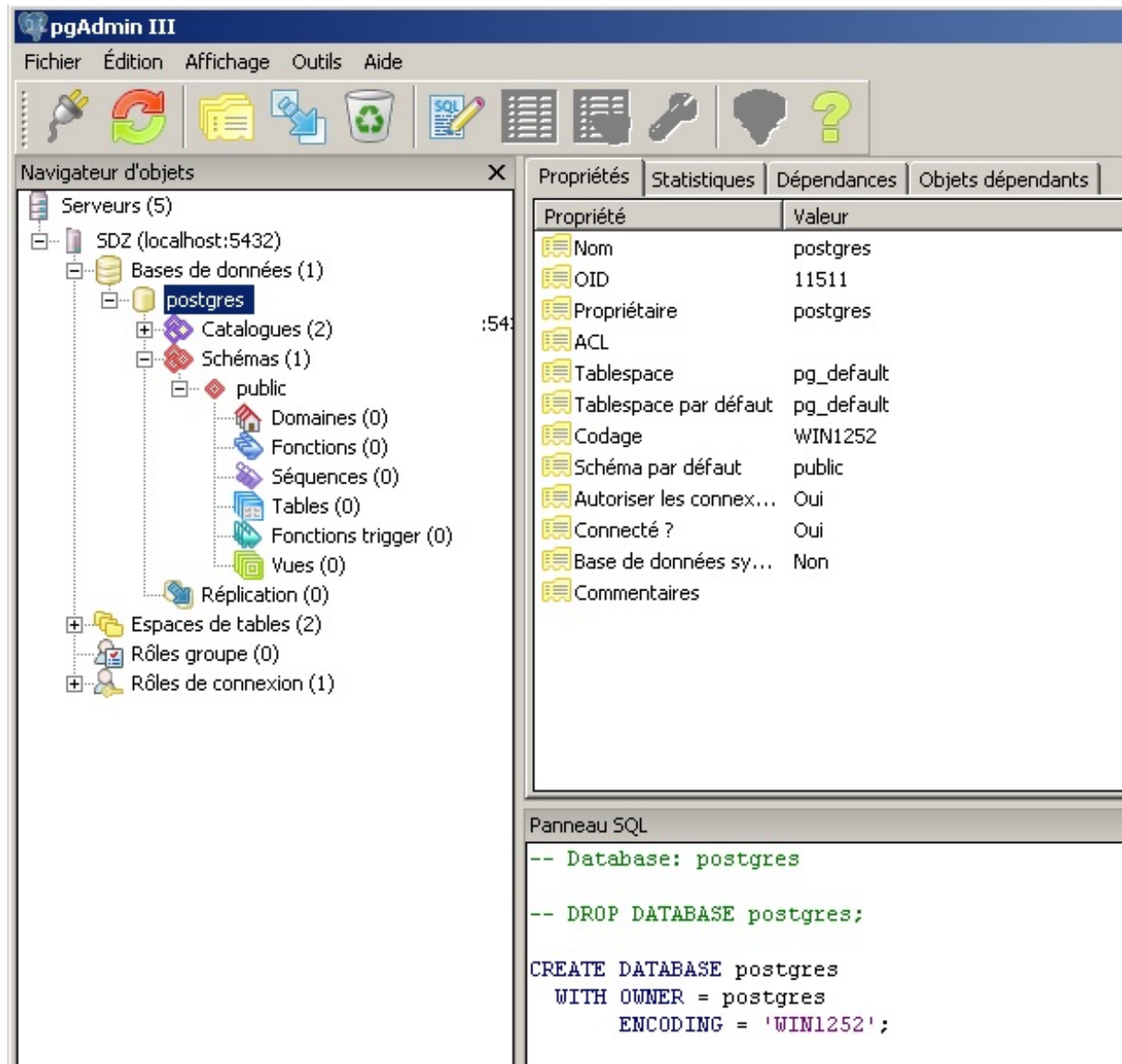


Ce qui vous amène à ceci :



- "Nom" correspond au nom de votre base de données ;
- "Hôte" correspond à l'adresse du serveur sur le réseau ; ici, il est sur votre machine, nous mettons donc "localhost" ;
- vous n'avez pas besoin de toucher au port normalement, sinon la valeur du screen ;
- ensuite, le nom de l'utilisateur et son mot de passe que vous avez défini.

Voilà, vous devriez avoir ceci devant les yeux :



Nous reviendrons sur tout ceci, mais vous pouvez lire que votre serveur, nommé "SDZ", a une base de données appelée "postgres" qui contient 0 table !

Simple et efficace !

Voilà, nous avons installé notre serveur, nous allons voir maintenant comment créer une base, des tables mais surtout faire un bref rappel sur ce fameux langage SQL... 😊

Pas de QCM pour ce chapitre, mais n'y prenez pas trop goût... Tout ceci ne durera pas... 😊

Bon, rien de sorcier ici !

La plupart d'entre vous n'a peut-être rien vu de nouveau, mais il fallait partir de Zéro...

Dès le prochain chapitre, nous allons voir comment Java va "discuter" avec notre base de données...

PostgreSQL

Alors, maintenant, nous allons aborder la prise en main de cet outil qu'est PostgreSQL !

Dans ce chapitre, nous verrons comment créer une base de données, des tables, ajouter des contraintes de clés et d'intégrités... Enfin bref, tout ce dont vous allez avoir besoin pour suivre au mieux ce tutoriel. Ceci, bien sûr, sans rentrer dans les détails : c'est un tuto sur JDBC, pas sur PostgreSQL ou sur SQL !

Bon, il y a déjà du taf, mine de rien...

Préparer sa BDD

Bon, vous êtes connectés à votre BDD préférée et ce n'est pas parce que j'ai choisi PostgreSQL que vous avez dû en faire autant...

Cependant, celles et ceux qui n'ont pas fait le même choix que moi peuvent aller directement vers le sous-chapitre sur SQL (sauf s'ils ont déjà les bases, qu'ils aillent dans ce cas directement au chapitre suivant...).

Déjà, les bases de données servent à stocker des informations, ça, vous le savez ! 😊

Mais ce que vous ignorez peut-être, c'est que, pour ranger correctement nos informations, nous allons devoir analyser celles-ci... Ce tuto n'est pas non plus un tuto sur l'analyse combinée avec des diagrammes entités-associations... Dans le jargon, c'est ce dont on se sert pour créer des BDD, enfin, pour organiser les informations (tables et contenu de tables) !

Nous allons juste poser un thème et nous ferons comme si vous saviez faire tout ça ! 🤖

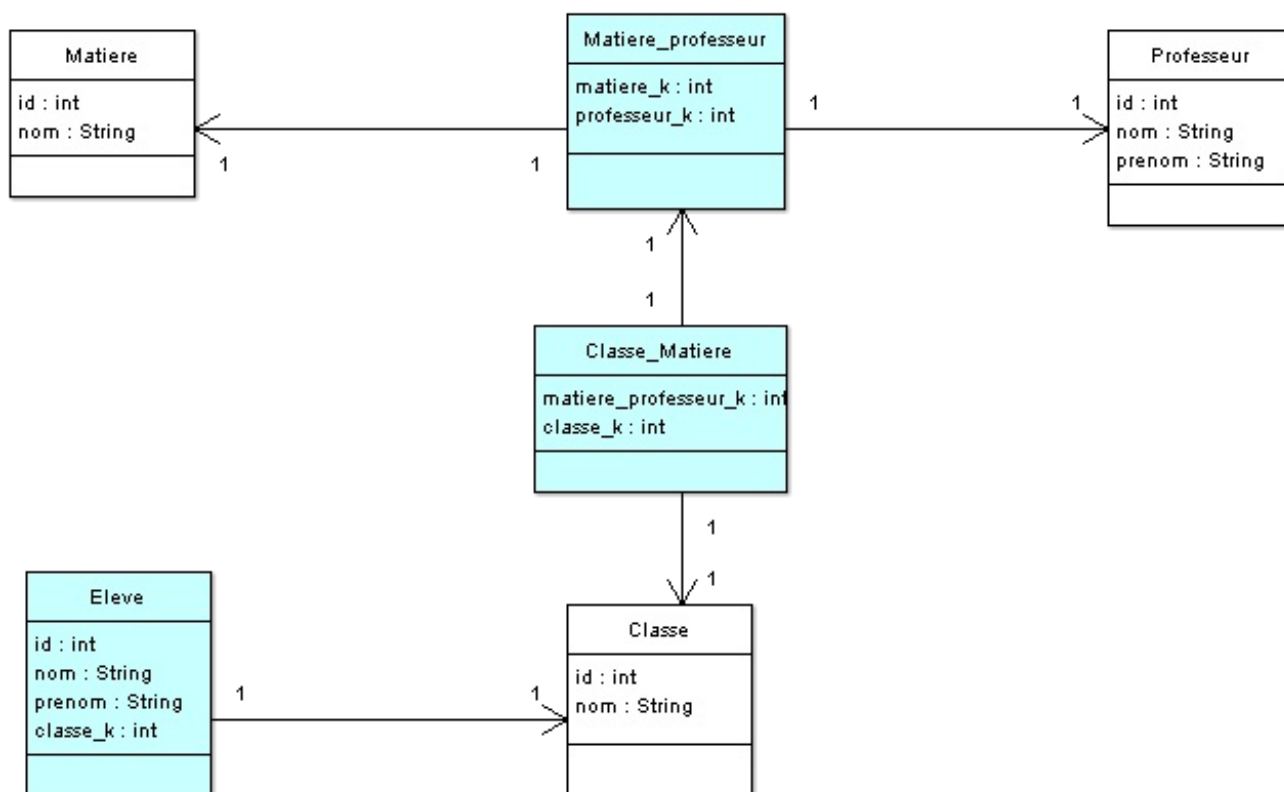
Pour ceux que la réalisation de modèles entités-associations intéressent, vous pouvez faire un tour [ici](#).

Voilà : pour notre base de données, nous allons gérer une école, dont voici les caractéristiques :

- cette école est composée de classes ;
- chaque classe est composée d'élèves ;
- chaque classe a un professeur de chaque matière dispensée par cette école ;
- un professeur peut enseigner plusieurs matières et exercer ses fonctions sur plusieurs classes.

Voilà le point ZÉRO de toute base de données !

Vous vous rendez compte qu'il y a beaucoup d'informations à gérer. Bon, en théorie, nous devrions faire un dictionnaire des données, voir à qui appartient quelle donnée, poursuivre avec une modélisation façon MCD (*Modèle Conceptuel de Données*) et simplifier le tout suivant certaines règles pour terminer avec un MPD (*Modèle Physique de Données*). Nous allons raccourcir le processus et je vais fournir un modèle tout fait, que je vais tout de même vous expliquer... 🤖





Tous ces beaux éléments seront nos futures tables. De plus, les attributs dans celles-ci se nomment des "**champs**".

Vous pouvez voir que tous les acteurs mentionnés se trouvent symbolisés dans ce schéma (classe, professeur, élève...). Vous constaterez que ces acteurs ont un attribut nommé '**id**', ceci correspond à son identifiant : c'est un champ de type entier qui s'incrémentera pour chaque nouvelle entrée, c'est aussi grâce à ce champ que nous créons des liens entre nos acteurs.

Oui... Vous avez remarqué que j'avais colorié des tables en bleu. 😊

Ces tables ont toutes un champ qui a une spécificité : un champ dont le nom se termine par '**_k**'.

D'abord, vous devez savoir que la flèche signifie '**a un**', de ce fait, un élève '**a une**' classe !



Bon, on te suit, mais pourquoi les autres tables ont deux champs comme ça ?

C'est simple, c'est parce que je vous ai dit qu'un professeur pouvait exercer plusieurs matières : dans ce cas, nous avons besoin de ce qu'on appelle une **table de jointure**.

Ainsi, nous pouvons dire que tel professeur exerce telle ou telle matière et qu'une association prof-matière est assignée à une classe ! 😊



La donnée que nous utiliserons pour lier des tables n'est autre que l'identifiant : **id**.

De plus - difficile de ne pas avoir vu ça - chaque champ a un type (entier, double, date, boolean...).

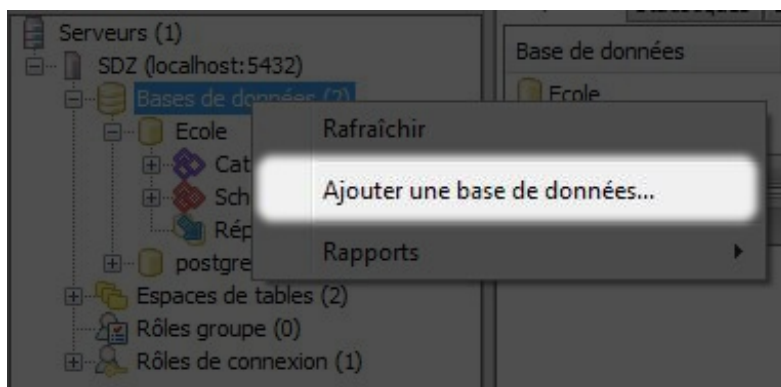
Nous avons tout ce dont nous avons besoin pour construire notre BDD !

Créer la BDD

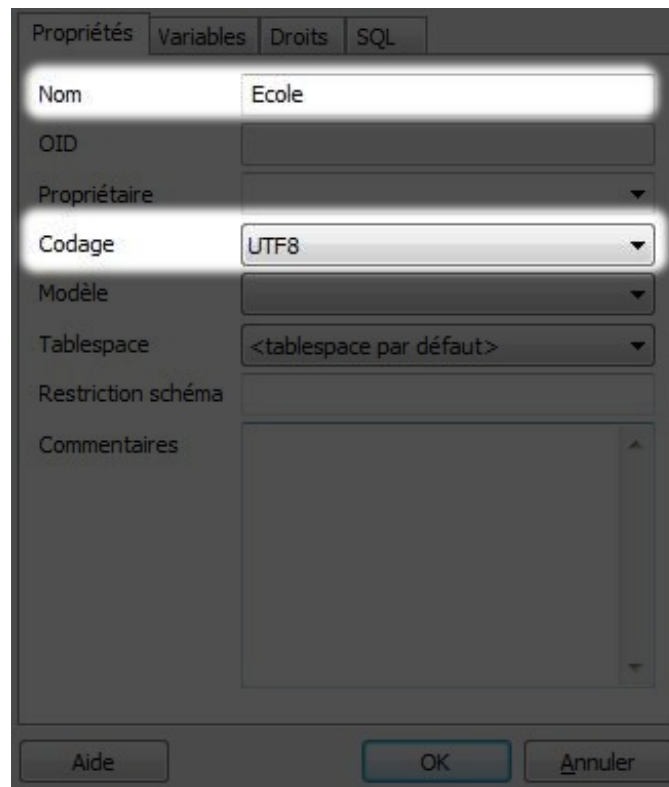
Pour cette opération, rien de plus simple ! 😊

pgAdmin met à votre disposition un outil qui facilite la création de BDD et de tables (créer ses BDD et ses tables à la mano, avec SQL, c'est un peu fastidieux...).

Pour créer une nouvelle base de données, il vous suffit de faire un clic droit sur "**Base de données**" :

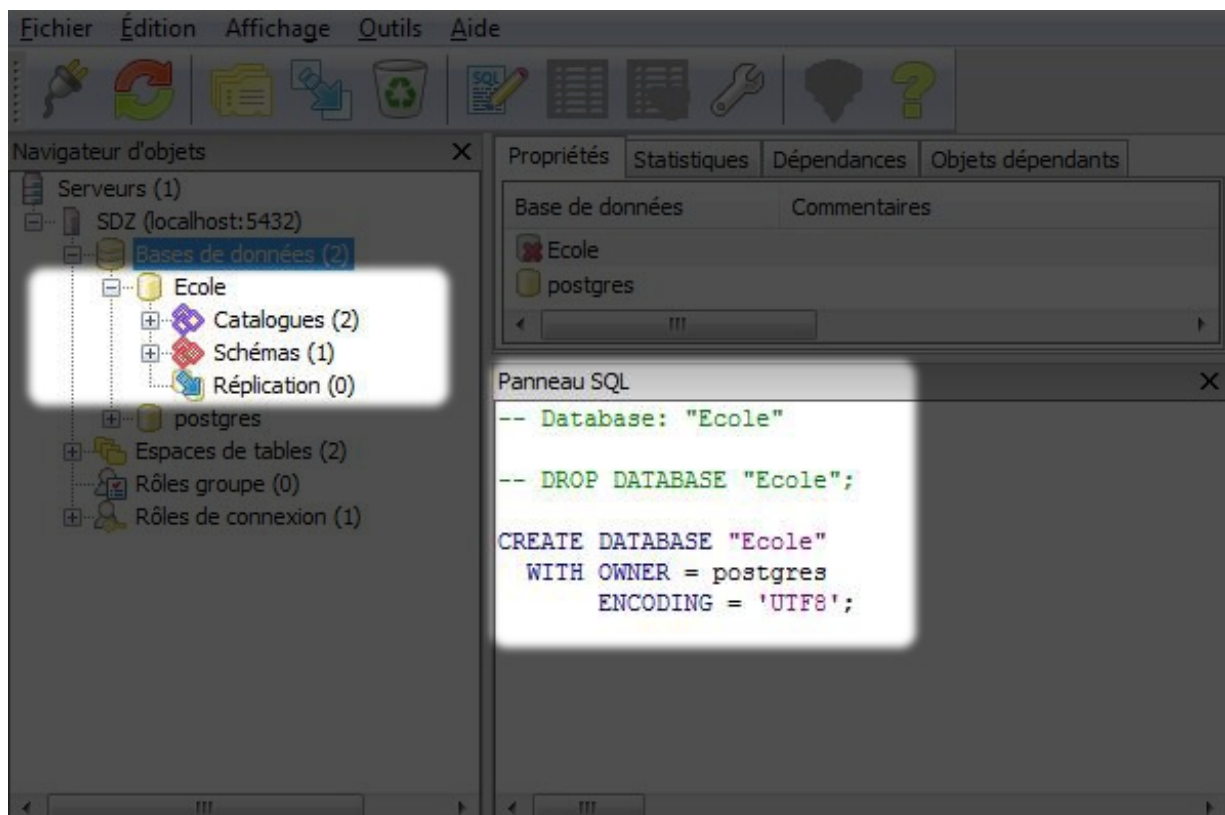


Vous devriez arriver sur cette pop-up :



Renseignez le nom de la base de données et choisissez l'encodage UTF-8. Cet encodage correspond à un jeu de caractères étendu qui autorise les caractères spéciaux !

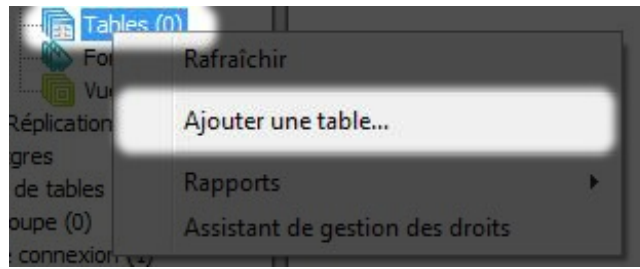
Une fois ceci fait, vous devriez avoir quelque chose comme ça :



Vous pouvez voir votre nouvelle base de données ainsi que le script SQL permettant de créer cette base.

Il ne nous reste plus qu'à créer les tables avec le bon type de données...

Même procédure que pour la séquence, un clic droit sur le noeud "**table**" cette fois, comme ceci :



Créer ses tables

Nous allons maintenant nous attaquer à la création de nos tables afin de pouvoir travailler...

Je vais vous montrer comment faire une table simple, histoire de... Et, pour les plus fainéants, je vous fournirai le script SQL qui finira la création des tables. 😊

Nous commencerons par la table "**classe**", vu que c'est l'une des tables qui n'a besoin d'aucun lien vers une autre table... 😊

Comme dit plus haut, il vous suffit de faire un clic droit sur "**Table**"

Ensuite, PostgreSQL vous demande des informations sur votre future table :

- son nom ;
- le nom de ses champs ;
- le type de ses champs ;
- ...

Ici, vous voyez le moment où vous devez renseigner le nom de votre table.

Ajouter une table...

Propriétés | Colonnes | Contraintes | Droits | SQL

Nom: classe

OID:

Propriétaire:

Tablespace: <tablespace par défaut>

Facteur remplissage:

Avec OID: ☒

Hérite des tables:

Supprimer

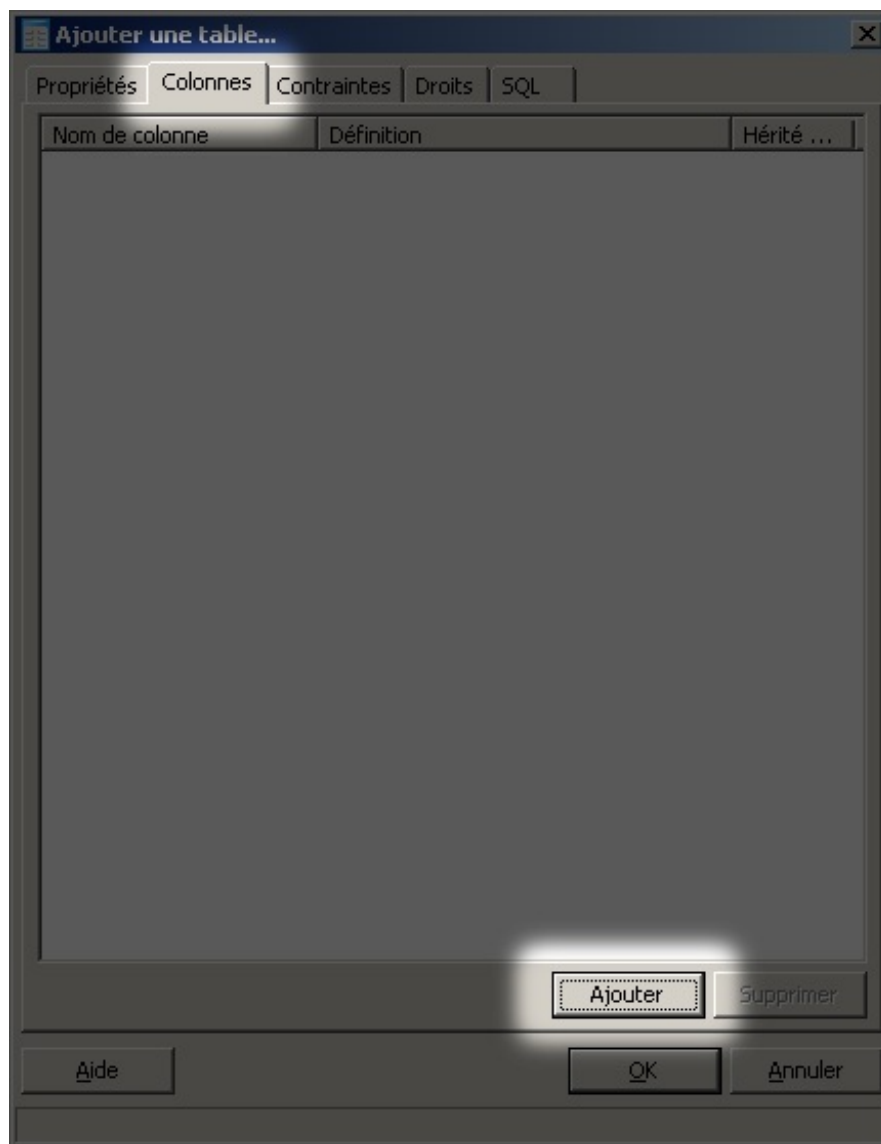
Ajouter

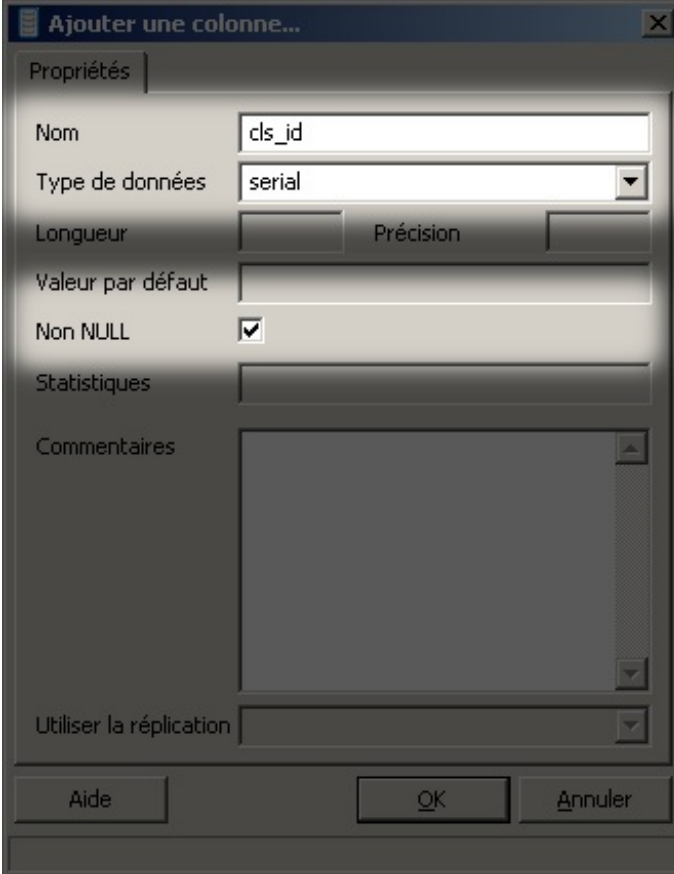
Commentaires: Table contenant les classes de l'école

Utiliser la réplication: ☐

Aide OK Annuler

Ensuite, vous ajouterez des champs (j'ai ajouté des préfixes à mes champs pour ne pas avoir trop d'ambiguïté dans mes requêtes SQL) :

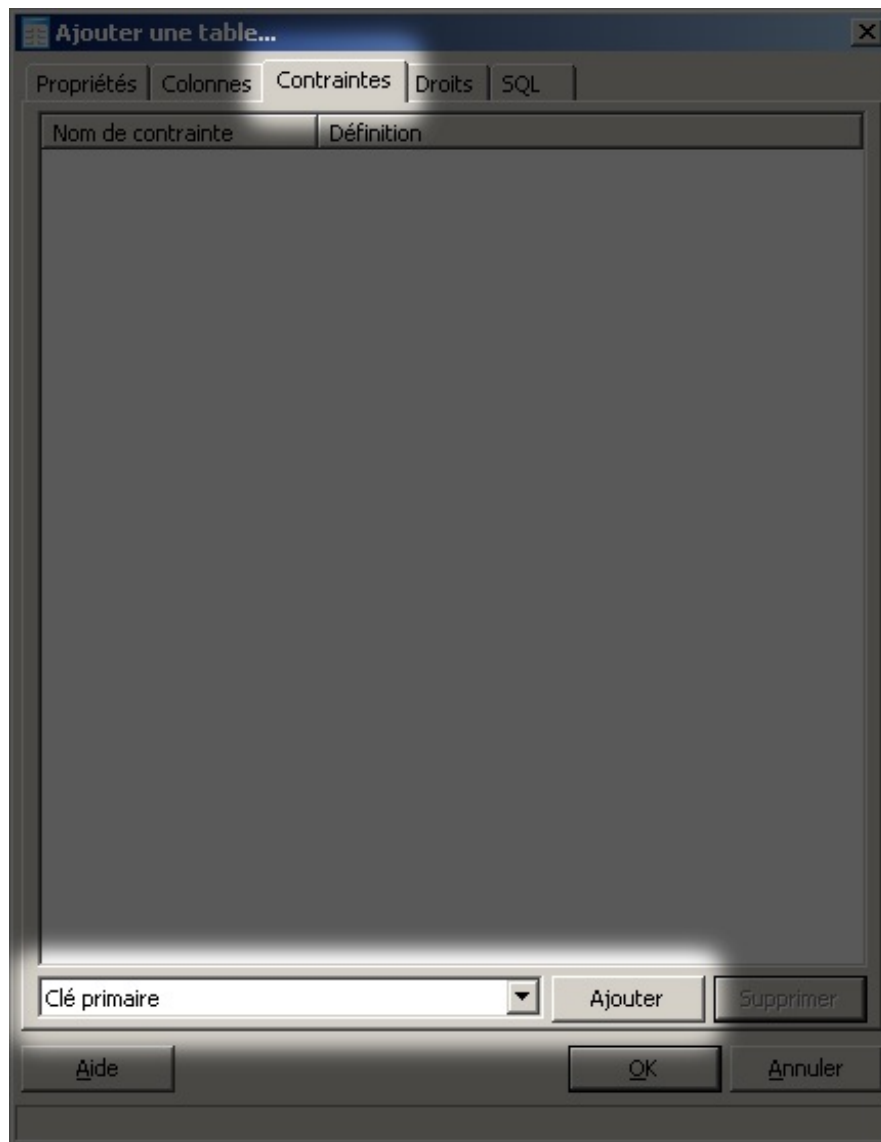




Le champ "cls_id" est de type "serial" afin que celui-ci utilise une séquence. Nous lui ajouterons en plus une contrainte de clé primaire.

Ce champ est un "**character varying**" de taille 64 : ce champ pourra donc contenir 64 caractères.

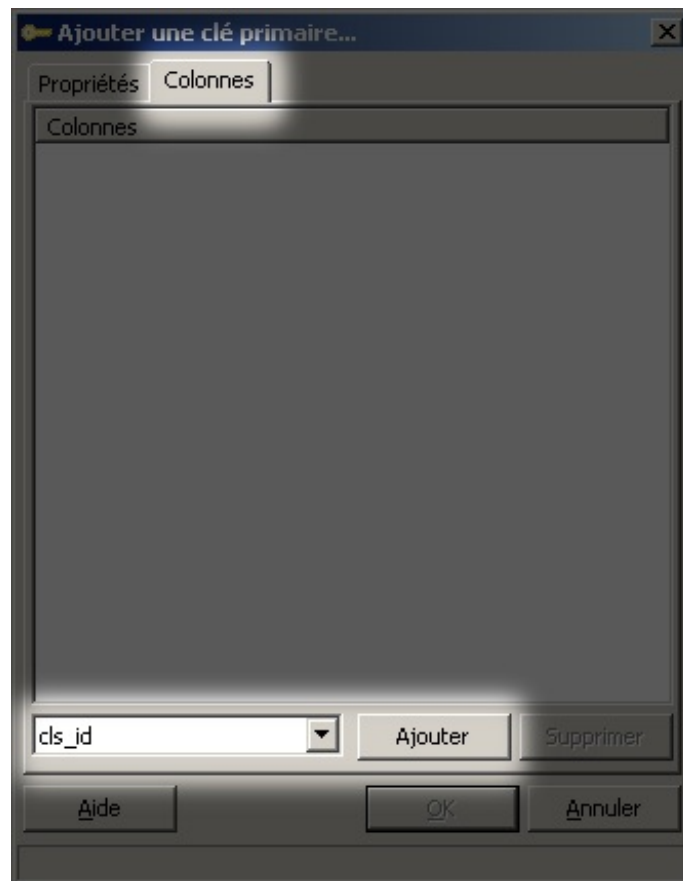
Nous mettons maintenant la contrainte de clé primaire sur notre identifieur :



Vous avez vu comment on crée une table avec PostgreSQL, mais je ne vais pas vous demander de faire ça pour chacune d'entre elles. 🤖

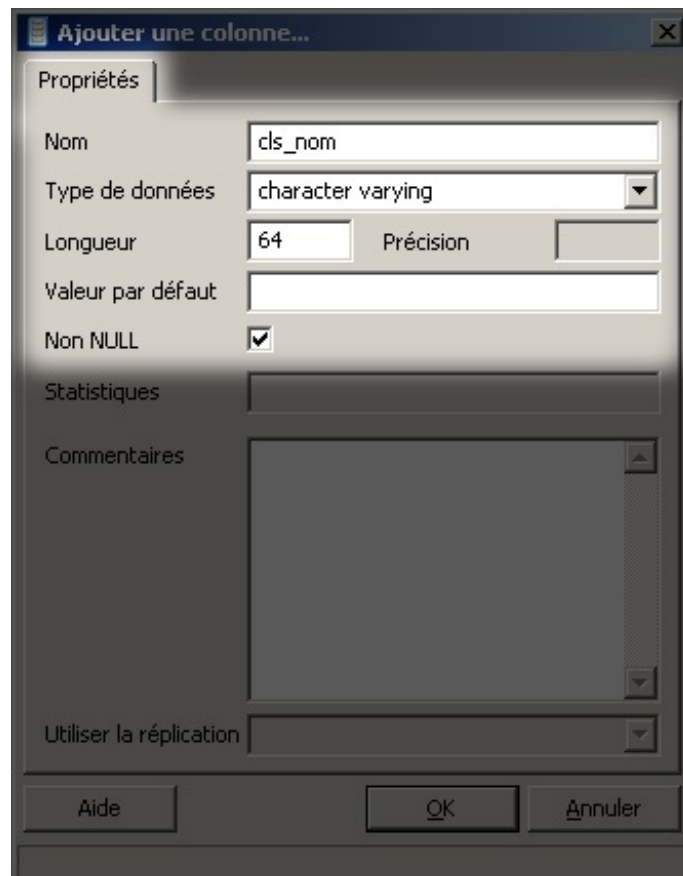
Je ne suis pas vache à ce point... Nous devons voir comment on utilise les BDD avec Java, pas avec le SGBD...

Cliquez sur "**Ajouter**".



Choisissez la colonne "**cls_id**" et cliquez sur "**Ajouter**".
Validez le tout.

Maintenant le deuxième champ :



Voici le code SQL de cette création de table :

```
Panneau SQL
-- Table: classe
-- DROP TABLE classe;

CREATE TABLE classe
(
  cls_id serial NOT NULL,
  cls_nom character varying(64) NOT NULL,
  CONSTRAINT "clé primaire" PRIMARY KEY (cls_id)
)
WITH (oids=true);
ALTER TABLE classe OWNER TO postgres;
COMMENT ON TABLE classe IS 'Table contenant les classes de l''école';
```

Je ne vais pas vous faire créer toutes les tables de cette manière... Le tuto est censé vous apprendre à utiliser les BDD avec Java...

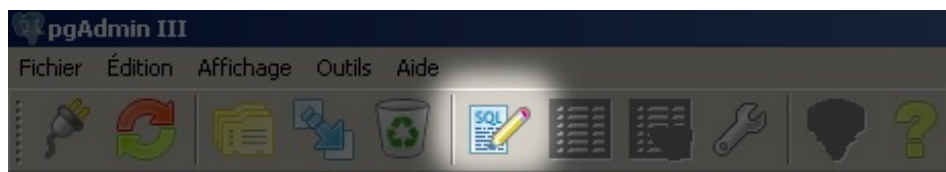
Voici donc [un fichier contenant le script SQL](#) de création des tables restantes ainsi que leurs contenus.

Sympa, le gars. 😊

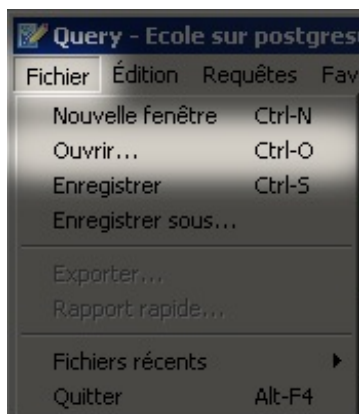


Le dit fichier est dans une archive, pensez à la décompresser avant... 🤖

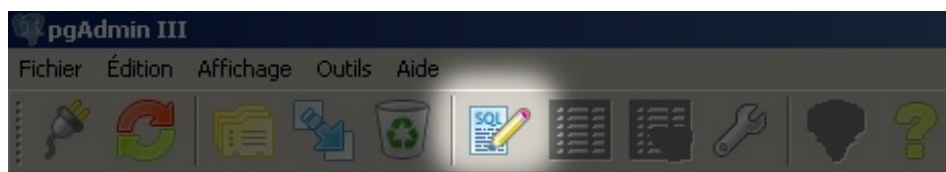
Il ne vous reste plus qu'à ouvrir le fichier avec PostgreSQL en allant dans l'éditeur de requêtes SQL :



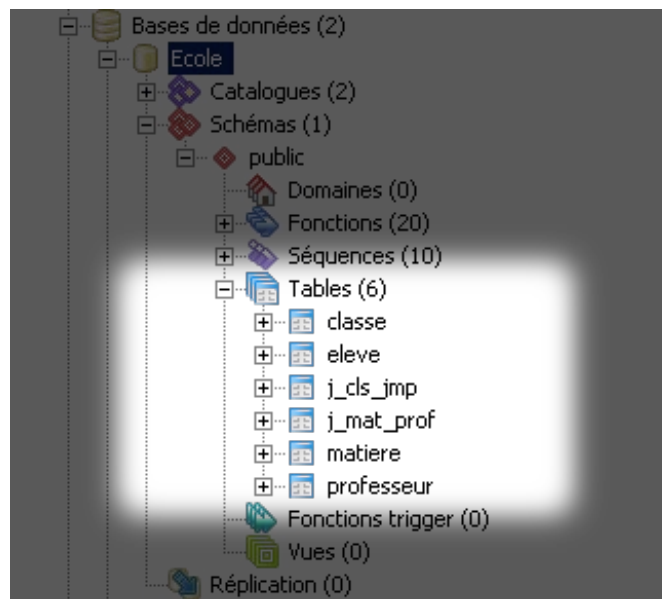
Vous pouvez à présent ouvrir le fichier que je vous ai fourni en faisant "**Fichier / Ouvrir**" et choisir le fichier .sql



Maintenant, exécutez la requête en cliquant sur ce bouton (vous pouvez aussi appuyer sur **F5**) :



Fermez l'éditeur de requête. Vous avez toute la base créée et, en plus, il y a des données. 🧙



Je vous retrouve donc au prochain chapitre afin de commencer notre aventure.

Voilà, les bases sont posées !

Nous allons pouvoir attaquer sévère dans JDBC... Let's go les ZérOs ! 🧙

Se connecter à sa BDD

Nous avons pris le temps de modéliser et de créer notre base de données.

Il est grand temps de prendre le taureau par les cornes et de voir comment accéder à tout ceci dans un programme Java.


Inutile de préciser que savoir programmer en objets est plus que requis et que, mis à part ce point, tout sera pris à partir de Zéro.



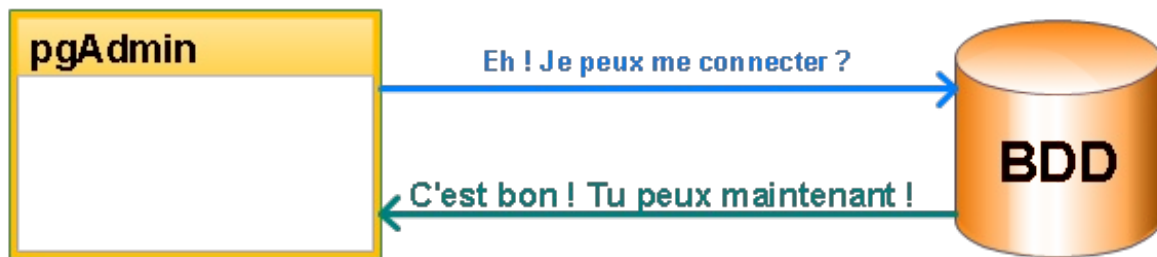
En avant !


Faisons le point

Bon : vous avez utilisé **pgAdmin** jusqu'à maintenant. Le truc, c'est qu'il se passe des choses cachées, peu avouables voire carrément obscures entre **pgAdmin** et **PostgreSQL**.

Non, je plaisante... mais il se passe effectivement des choses entre ces deux-là ! 

Le fait est que vous ne devriez plus faire l'amalgame entre BDD et SGBD ; par contre, vous devez encore ignorer que le SGBD effectue une connexion avec la BDD afin de pouvoir communiquer. Ceci pourrait se schématiser par un dialogue du genre :



Les Zéros qui ont déjà installé une imprimante savent que leur machine a besoin d'un driver pour que la communication puisse se faire entre les deux acteurs ! Ici, c'est la même chose ! 

pgAdmin utilise un **driver** pour se connecter à la base de données. Vu que les personnes qui ont développé les deux soft travaillent main dans la main, pas de souci de communication mais qu'en sera-t-il avec Java ?

En fait, avec Java, vous allez avoir besoin de ces **drivers**, mais pas sous n'importe quelle forme !

Afin de pouvoir vous connecter à une base de données avec Java, il vous faut un fichier .jar qui correspond au fameux pilote.

Il existe donc un fichier .jar qui permet de se connecter à une base PostgreSQL !



Est-ce que ça veut dire qu'il y a un fichier .jar par BDD ?

Tout à fait, il existe un fichier .jar pour se connecter à :

- MySQL ;
- SQL Server ;
- Oracle ;
- ...

Et si vous ne faites pas ceci, vous ne pourrez pas vous connecter !

Un bémol toutefois : vous pouvez aussi vous connecter à une BDD en utilisant les pilotes ODBC présents dans **Windows**, mais ceci nécessite d'installer les pilotes dans Windows et de les paramétrer dans les sources de données ODBC pour, par la suite, utiliser ces pilotes ODBC pour se connecter à la BDD dans un programme Java !

Je ne parlerai pas de cette méthode puisqu'elle ne fonctionne que pour Windows !

Mais bon, pour ceux que le sujet intéresse, allez voir [par là](#) !



D'accord, on voit bien le principe ! Mais alors... où trouve-t-on ce pilote pour Java ?

Une simple recherche sur [Google](#) comblera vos attentes avec une recherche du genre :



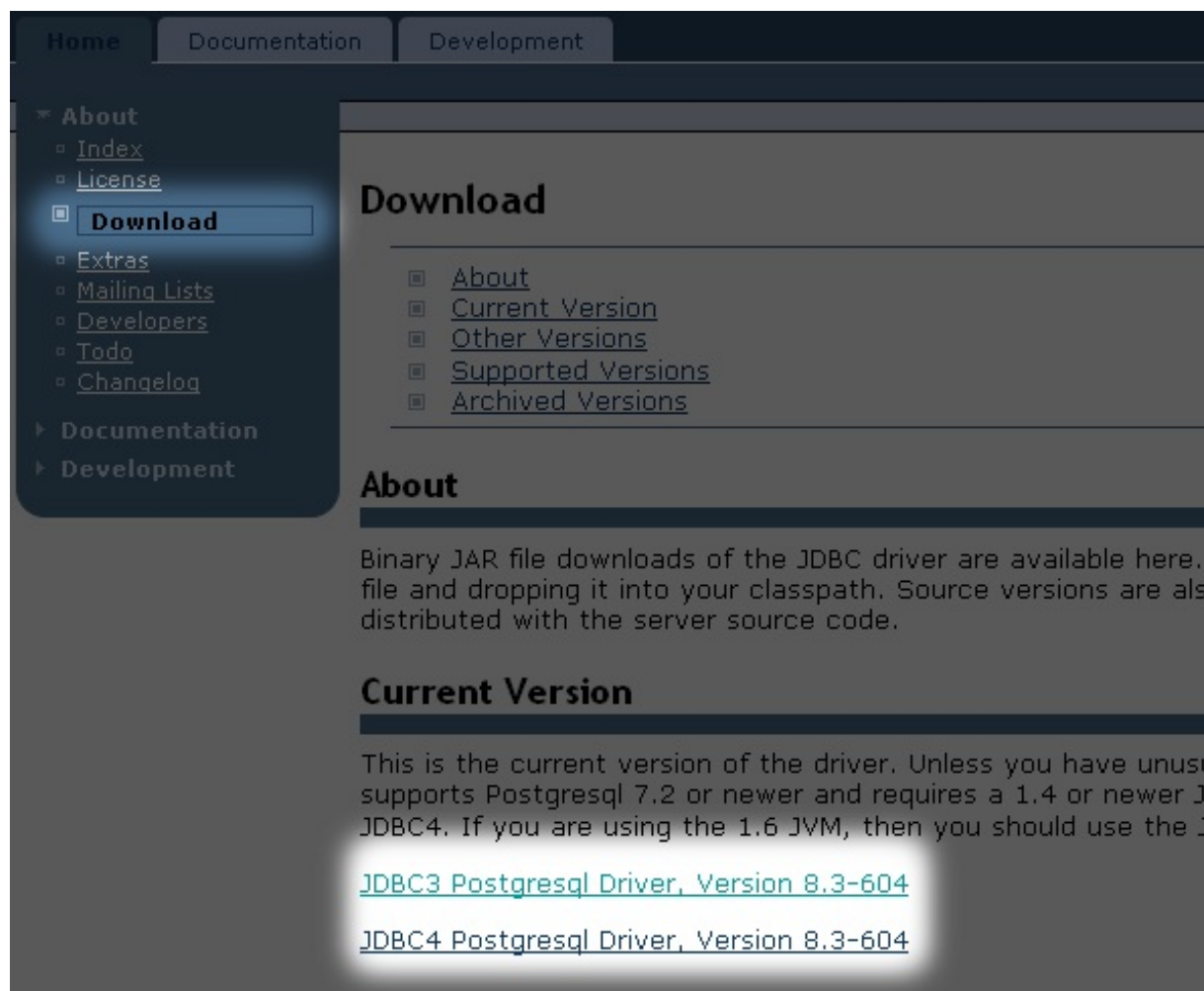
... pour PostgreSQL, ou :



pour MySQL...

Enfin bref, il est assez facile de trouver les pilotes JDBC convoités.

Sur la page de téléchargement des pilotes pour PostgreSQL, il y a la dernière version disponible ; j'ai pris la version JDBC4 :



La version JDBC4 offre des nouveautés et une souplesse dans l'utilisation de JDBC, mais j'avoue, je ne les connais pas trop... Vous trouverez un rapide aperçu [par ici](#) !

Bon ! Téléchargez le fichier .jar, pour ceux qui auraient la flemme de faire une recherche sur Google, c'est par ici ; et maintenant, que faire de cette archive ?



Et comment on l'utilise ?

Pour l'utilisation, nous y arrivons à grand pas, mais une question se pose encore : où mettre l'archive ? Vous avez deux solutions :

- la mettre dans votre projet et l'inclure au CLASSPATH ;
- mettre l'archive de le dossier d'installation du JRE, dans le dossier "**lib/ext**".

Le tout est de savoir si votre application est vouée à être exportée sur différents postes, auquel cas l'approche CLASSPATH est encore la plus judicieuse (sinon, il faudra ajouter l'archive dans tous les JRE...) mais, nous, nous allons utiliser la deuxième méthode afin de ne pas surcharger nos projets !

Je vous laisse donc mettre l'archive téléchargée dans le dossier sus-mentionné. 😊

Connexion ! Es-tu là ?

La base de données est prête, les tables sont créées et remplies et nous avons le driver ! Il ne nous reste plus qu'à nous connecter, **ENFIN** !



C'est vrai que tu auras mis le temps...

Eh, il faut ce qu'il faut ! Et encore, j'aurais pu vous faire réviser le SQL avant d'attaquer ! 😊

Je vous autorise à créer un nouveau projet dans Eclipse avec une classe contenant une méthode `public static void main(String[] args)` .

Voici le code source pour obtenir une connexion :

Code : Java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.util.Enumeration;
import java.util.Properties;

public class Connect {

    public static void main(String[] args) {

        try {
            Class.forName("org.postgresql.Driver");
            System.out.println("DRIVER OK ! ");

            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "postgres";

            Connection conn = DriverManager.getConnection(url, user, passwd);
            System.out.println("Connection effective !");

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```



Tout ça pour avoir une connexion ?

Oui. 😊

Nous allons détailler un peu tout ceci...

Dans un premier temps, nous avons créé une instance de l'objet **Driver** présent dans le fichier .jar que nous avons téléchargé tout à l'heure.

Il est inutile de créer une véritable instance de ce type d'objet ; par-là, j'entends que faire ceci :

Code : Java

```
org.postgres.Driver driver = new org.postgres.Driver();
```

n'est pas nécessaire. Du coup, nous utilisons [la réflexivité](#) afin d'instancier cet objet.

À ce stade, il y a comme un pont entre votre programme Java et votre BDD, mais *el trafique* routier n'y est pas encore autorisé ! Il faut qu'une connexion soit effective afin que votre programme et la BDD puissent communiquer. Ceci se fait avec l'autre ligne de code : `Connection conn = DriverManager.getConnection(url, user, passwd);` .

Nous avons au préalable défini trois **String** contenant respectivement :

- l'URL de connexion ;
- le nom de l'utilisateur ;
- le mot de passe utilisateur.



L'URL de quoi ?



De connexion. Elle est indispensable à Java pour se connecter à n'importe quelle BDD.
Cette URL se décompose comme suit :

jdbc:postgresql: **//localhost:5432** /Ecole

En vert, vous pouvez voir le début de l'URL de connexion. elle commence TOUJOURS par "**jdbc:**". Dans notre cas, nous utilisons PostgreSQL, la dénomination "**postgresql:**" suit le début de l'URL. Si vous utilisez une source de données ODBC, il faudrait avoir "**jdbc:odbc:**".

En fait, ceci dépend du pilote JDBC : cela permet à Java de savoir quel pilote utiliser ! **On parle de protocole utilisé.**

En bleu, vous trouverez la localisation de la machine physique sur le réseau ; ici, nous sommes en local, nous utilisons donc **"//localhost:5432"**. Ah oui, le nom de la machine physique est suivi du numéro de port utilisé. 😊

En orange, pour ceux qui ne l'auraient pas deviné, il s'agit du nom de notre base de données ! 😊



Les informations en bleu et en orange dépendent du pilote JDBC utilisé. Pour en savoir plus, consultez la documentation de celui-ci ! 😊

L'URL, le nom d'utilisateur, le mot de passe et le driver permettent ensemble de créer le pont de connexion et le trafic routier sur ce pont !

Donc, si vous exécutez ce code, vous aurez :

```
Problems  Javadoc  Declaration  Console  ⌵
<terminated> Connect [Java Application] C:\Program Files\Java\jdk1.6.0_
DRIVER OK !
Connection effective !
```



Vous n'êtes obligés de spécifier la totalité des informations pour l'URL... Mais il faudra au moins "**jdbc:postgresql**".



Tout ceci lève une **Exception** en cas de problème !



L'avantage avec les fichiers .jar comme drivers de connexion, c'est que vous n'êtes pas tenus d'initialiser le driver par réflexivité ou autre. Tout se passe en Java ! Vu qu'il y a un rappel du protocole à utiliser dans l'URL de connexion, tout est parfait et Java s'en sort tout seul !

Ne vous étonnez pas si vous ne voyez plus `Class.forName("org.postgresql.Driver");` par la suite...

Bon, on arrive à se connecter. Maintenant, avant de poursuivre, un QCM vous attend ! 🤖

Chapitre très simple, n'est-ce pas ?

Vous savez maintenant établir une connexion avec une BDD. Nous allons maintenant voir comment faire joujou avec tout ceci !

Fouiller dans sa BDD

Nous continuons notre voyage initiatique au pays de JDBC en abordant la manière d'interroger notre BDD.

Eh oui, une base de données n'est utile que si nous pouvons consulter, ajouter, modifier ou encore supprimer les données qu'elle comprend.

Par contre, pour faire ceci, il était IMPÉRATIF de se connecter. Mais vu que c'est chose faite, nous allons voir comment fouiner dans notre BDD.

Le couple Statement - ResultSet

Voici deux objets que vous utiliserez sûrement beaucoup !

En fait, ce sont ces deux objets qui permettent de récupérer des données de la BDD ou de travailler avec celles-ci... 🤖

Afin de vous faire comprendre tout ceci de façon simple, voici un exemple assez complet (mais pas trop quand même) affichant le contenu de la table **classe** :

Code : Java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;

public class Connect {

    public static void main(String[] args) {

        try {
            Class.forName("org.postgresql.Driver");

            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "postgres";

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

            //Création d'un objet Statement
            Statement state = conn.createStatement();
            //L'objet ResultSet contient le résultat de la requête SQL
            ResultSet result = state.executeQuery("SELECT * FROM classe");
            //On récupère les MetaData
            ResultSetMetaData resultMeta = result.getMetaData();

            System.out.println("\n*****");
            //On affiche le nom des colonnes
            for(int i = 1; i <= resultMeta.getColumnCount(); i++)
                System.out.print("\t" +
resultMeta洗getColumnName(i).toUpperCase() + "\t *");

            System.out.println("\n*****");

            while(result.next()) {
                for(int i = 1; i <= resultMeta.getColumnCount(); i++)
                    System.out.print("\t" + result.getObject(i).toString() + "\t
");

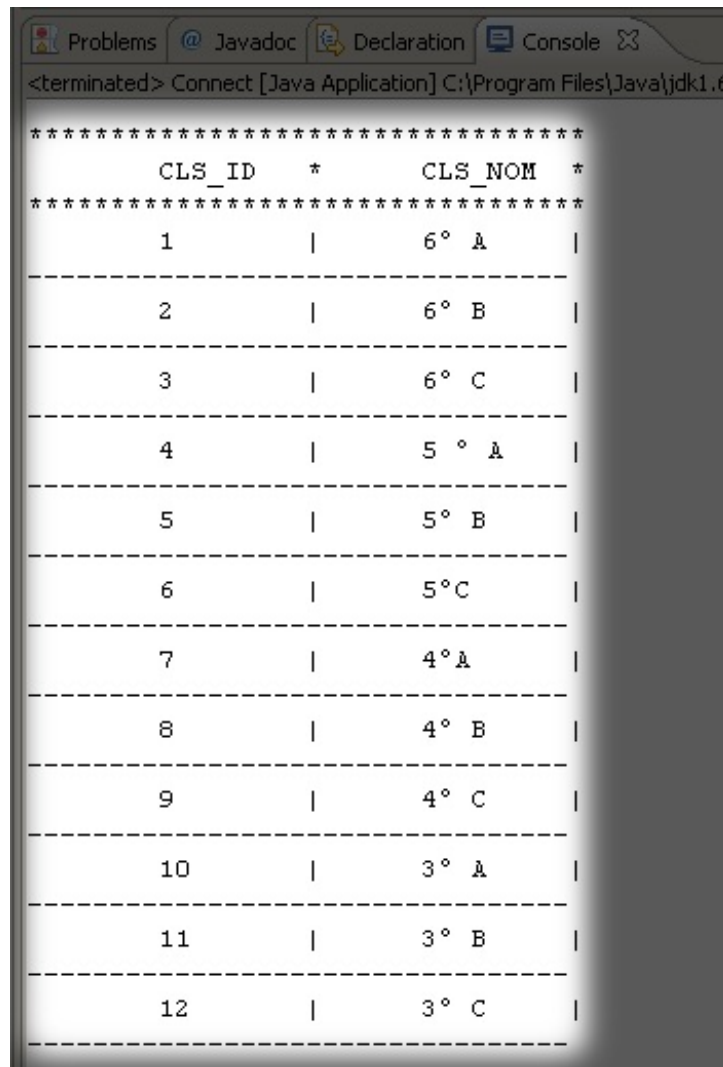
                System.out.println("\n-----");
            }

            result.close();
            state.close();

        } catch (Exception e) {
```

```
e.printStackTrace();
}
}
```

Voici ce que nous donne ce code :



CLS_ID	CLS_NOM
1	6° A
2	6° B
3	6° C
4	5° A
5	5° B
6	5° C
7	4° A
8	4° B
9	4° C
10	3° A
11	3° B
12	3° C

Vous avez sûrement compris que j'ai simplement exécuté une requête SQL et récupéré les lignes trouvées ! Mais détaillons un peu plus ce qu'il s'est passé. 🤔

Déjà, vous aurez pu remarquer que j'ai spécifié l'URL complète pour la connexion : sinon comment voulez-vous savoir quelle BDD attaquer ?...

La connexion à la BDD mise à part, les choses se sont passées en quatre étapes distinctes :

- création de l'objet **Statement** ;
- exécution de la requête SQL ;
- récupération et affichage des données via l'objet **ResultSet** ;
- fermeture des objets utilisés (bien que non obligatoire, c'est recommandé).



Tout ça semble clair, mais tu ne pourrais pas nous en dire un peu plus sur ces objets ?

Mais c'est ce que j'allais faire... 🤔

Dites-vous une chose : objet **Statement** => instruction SQL ! C'est cet objet qui exécute les requêtes SQL et qui retourne les résultats.

Ensuite, lorsque vous entendez **ResultSet**, c'est que votre requête SQL a retourné un certain nombre de lignes à récupérer et à afficher..

Comment ça fonctionne

Je vous ai fourni un morceau de code, il fonctionne, mais comment ça marche ?

Voici le moment où tout vous sera dévoilé.

Comme je vous le disais plus haut, l'objet **Statement** est l'objet qui vous permet de faire des requêtes SQL. Celles-ci peuvent être de type :

- **CREATE** ;
- **INSERT** ;
- **UPDATE** ;
- **SELECT** ;
- **DELETE** .

Vous n'êtes pas sans savoir que, selon le type de requête exécutée, celle-ci retourne un / des résultat(s), dans le cas d'un **SELECT**, ou une validation / un message d'erreur dans les autres cas.

L'objet **Statement** vous est fourni par l'objet **Connection** grâce à l'instruction `conn.createStatement()` ; .



Nous verrons, dans le chapitre suivant, un peu plus de choses concernant l'objet **Statement**.

Ce que j'ai fait ensuite, c'est demander à mon objet **Statement** d'exécuter une requête SQL de type **SELECT**. Vous la voyez, celle-ci :

Code : SQL

```
SELECT * FROM classe
```

Cette requête me retournant un résultat contenant beaucoup de lignes, elles-mêmes contenant plusieurs colonnes, nous avons stocké ce résultat dans un objet **ResultSet**, objet permettant de faire diverses actions sur des résultats de requêtes SQL ! 😊

Après cette ligne de code : `ResultSet result = state.executeQuery("SELECT * FROM classe");` , les résultats sont stockés dans l'objet **ResultSet** et nous n'avons plus qu'à afficher les données.

Ici, j'ai utilisé un objet de type **ResultSetMetaData** afin de récupérer les "*meta data*" de ma requête. Comprenez ceci comme "*récupérer les informations globales de ma requête*". Ici, nous avons utilisé cet objet afin de récupérer le nombre de colonnes renvoyé par la requête SQL ainsi que leurs noms.

Cet objet "*meta data*" permettent de récupérer des informations très utiles comme :

- le nombre de colonnes d'un résultat ;
- le nom des colonnes d'un résultat ;
- le type de données stocké dans chaque colonne ;
- le nom de la table à qui appartient la colonne (dans le cas d'une jointure de table) ;
- ...

Vous voyez que ces informations peuvent être utiles.



Il existe aussi un objet **DatabaseMetaData** qui, lui, fournit des informations sur la base de données !

Vous comprenez mieux ce que signifie cette portion de code :

Code : Java

```
System.out.println("\n*****");
//On affiche le nom des colonnes
for(int i = 1; i <= resultMeta.getColumnCount(); i++)
    System.out.print("\t" + resultMeta.getColumnName(i).toUpperCase() +
"\t *");

System.out.println("\n*****");
```

Nous nous servons de la méthode nous donnant le nombre de colonnes dans le résultat afin de récupérer le nom de la colonne grâce à son index !



ATTENTION : contrairement aux indices de tableaux, les indices de colonnes SQL commencent à 1 !

Ensuite, nous récupérons les données de notre requête en nous servant de l'indice des colonnes :

Code : Java

```
while(result.next()){
    for(int i = 1; i <= resultMeta.getColumnCount(); i++)
        System.out.print("\t" + result.getObject(i).toString() + "\t |");

    System.out.println("\n-----");
}
```

Nous utilisons une première boucle afin que, tant que notre objet **ResultSet** nous retourne des lignes de résultats, nous parcourions ensuite chaque ligne via notre boucle **for** .



La méthode `next()` permet de positionner l'objet sur la ligne suivante dans sa liste de résultat ! Au premier tour de boucle, cette méthode positionne l'objet sur la première ligne. Si vous ne positionnez pas l'objet résultat et que vous tentez de lire des données, une exception sera levée !

Je suis parti du postulat que ne connaissons pas le type de données de nos colonnes, mais vu que nous les connaissons, ce code aurait tout aussi bien fonctionné :

Code : Java

```
while(result.next()){
    System.out.print("\t" + result.getInt("cls_id") + "\t |");
    System.out.print("\t" + result.getString("cls_nom") + "\t |");
    System.out.println("\n-----");
}
```



On a le droit de faire ça ?

Tout à fait ! Nous connaissons le nom de nos colonnes retournées par la requête SQL, nous connaissons aussi leurs types, il nous suffit donc d'invoquer la méthode adéquat de l'objet **ResultSet** en utilisant le nom de la colonne à récupérer !



Par contre, si vous essayez de récupérer le contenu de la colonne "`cls_nom`" avec la méthode `getInt("cls_nom")` , vous aurez une zolie exception !

Il existe une méthode `getXXX()` par type primitif et quelques autres correspondant aux types SQL :

- `getArray(int columnIndex) ;`
- `getAscii(int columnIndex) ;`
- `getBigDecimal(int columnIndex) ;`
- `getBinary(int columnIndex) ;`
- `getBlob(int columnIndex) ;`
- `getBoolean(int columnIndex) ;`
- `getBytes(int columnIndex) ;`
- `getCharacter(int columnIndex) ;`
- `getDate(int columnIndex) ;`
- `getDouble(int columnIndex) ;`
- `getFloat(int columnIndex) ;`
- `getInt(int columnIndex) ;`
- `getLong(int columnIndex) ;`
- `getObject(int columnIndex) ;`
- `getString(int columnIndex) ;`

Et, à la fin, nous fermons nos objets :

Code : Java

```
result.close();  
state.close();
```

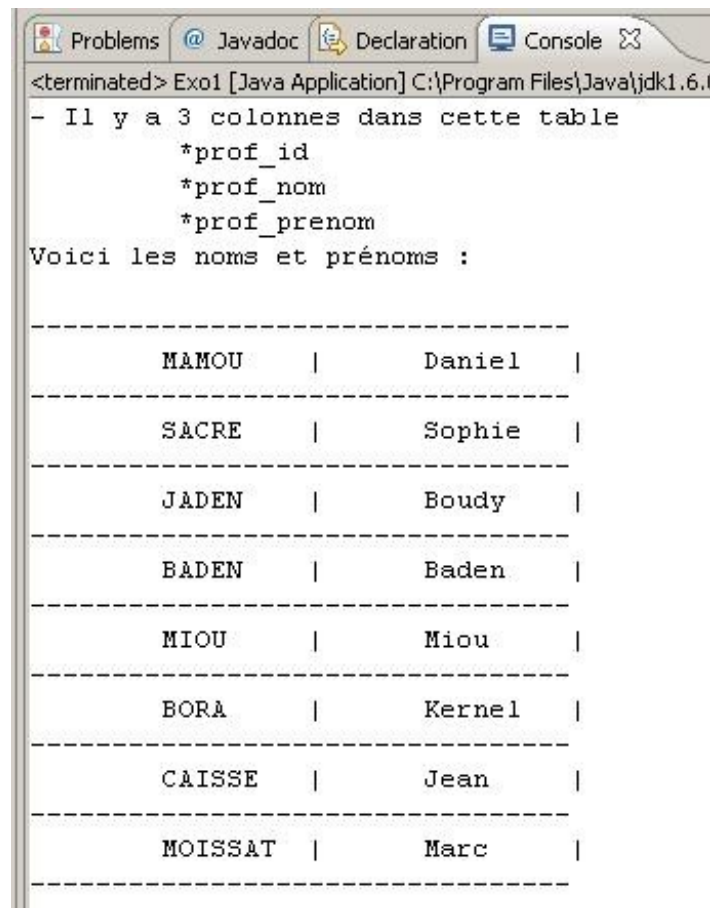
Avant de voir plus en détail les possibilités qu'offrent ces objets, nous allons faire deux-trois requêtes SQL afin de nous habituer à la façon dont tout ça fonctionne ! 😊

Entraînons-nous

Le but du jeu est de me coder les résultats que j'ai obtenus... Je vous laisse chercher dans quelle table... 🤔

Vous êtes prêts ? C'est parti ! 🏁

Voici ce que vous devez me récupérer en premier :



```
<terminated> Ex01 [Java Application] C:\Program Files\Java\jdk1.6.0
- Il y a 3 colonnes dans cette table
  *prof_id
  *prof_nom
  *prof_prenom
Voici les noms et prénoms :

-----
      MAMOU      |      Daniel      |
-----
      SACRE      |      Sophie      |
-----
      JADEN      |      Boudy       |
-----
      BADEN      |      Baden       |
-----
      MIOU       |      Miou        |
-----
      BORA       |      Kernel      |
-----
      CAISSE     |      Jean        |
-----
      MOISSAT    |      Marc        |
-----
```

Cherchez bien...

Bon, vous avez trouvé ! Il n'y avait rien de compliqué ici, voici la correction, enfin, une suggestion de correction :

Secret (cliquez pour afficher)

Code : Java

```
package com.sdz.exo;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;

public class Ex01 {

    public static void main(String[] args) {

        try {
            Class.forName("org.postgresql.Driver");

            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "postgres";

            Connection conn = DriverManager.getConnection(url, user,
            passwd);
            Statement state = conn.createStatement();

            ResultSet result = state.executeQuery("SELECT * FROM
professeur");
            ResultSetMetaData resultMeta = result.getMetaData();

            System.out.println("- Il y a " + resultMeta.getColumnCount() +
" colonnes dans cette table");
```

```

    colonnes dans cette table ;
    for(int i = 1; i <= resultMeta.getColumnCount(); i++)
        System.out.println("\t *" + resultMeta.getColumnName(i));

    System.out.println("Voici les noms et prénoms : ");
    System.out.println("\n-----");

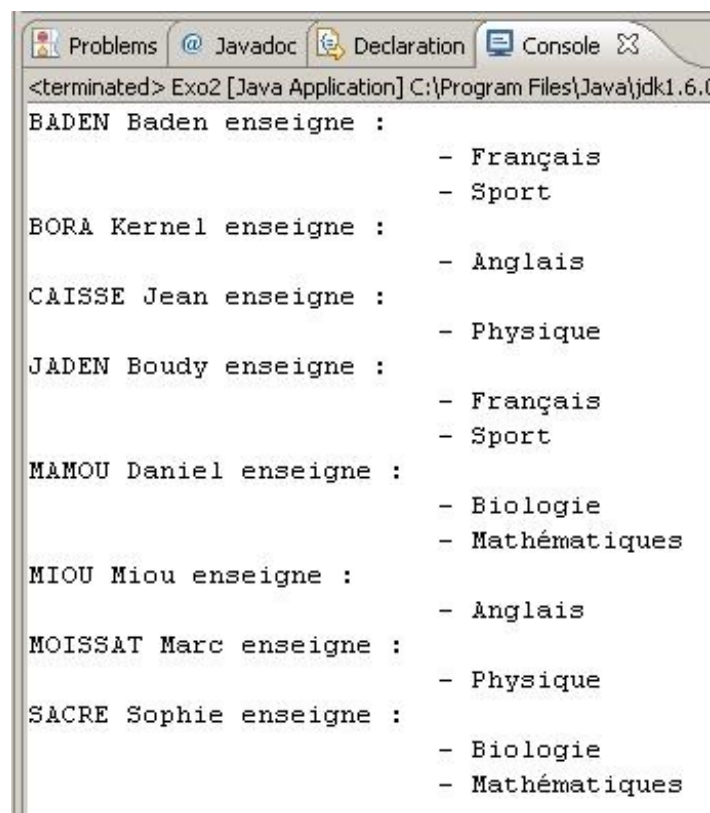
    while(result.next()){
        System.out.print("\t" + result.getString("prof_nom") + "\t"
|");
        System.out.print("\t" + result.getString("prof_prenom") + "\t"
|");
        System.out.println("\n-----");
    }

        result.close();
        state.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Allez, on complique la tâche :



```

<terminated> Exo2 [Java Application] C:\Program Files\Java\jdk1.6.0
BADEN Baden enseigne :
                                - Français
                                - Sport
BORA Kernel enseigne :
                                - Anglais
CAISSE Jean enseigne :
                                - Physique
JADEN Boudy enseigne :
                                - Français
                                - Sport
MAMOU Daniel enseigne :
                                - Biologie
                                - Mathématiques
MIOU Miou enseigne :
                                - Anglais
MOISSAT Marc enseigne :
                                - Physique
SACRE Sophie enseigne :
                                - Biologie
                                - Mathématiques

```

Hou... Ne vous faites pas exploser la cervelle tout de suite... On ne fait que commencer... 😊

Voici une possible solution à ce résultat :

Secret ([cliquez pour afficher](#))

Code : Java


```
package com.sdz.exo;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;

public class Exo2 {
    public static void main(String[] args) {

        try {
            Class.forName("org.postgresql.Driver");

            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "postgres";

            Connection conn = DriverManager.getConnection(url, user,
            passwd);
            Statement state = conn.createStatement();

            String query = "SELECT prof_nom, prof_prenom, mat_nom FROM
            professeur";
            query += " INNER JOIN j_mat_prof ON jmp_prof_k = prof_id";
            query += " INNER JOIN matiere ON jmp_mat_k = mat_id ORDER BY
            prof_nom";

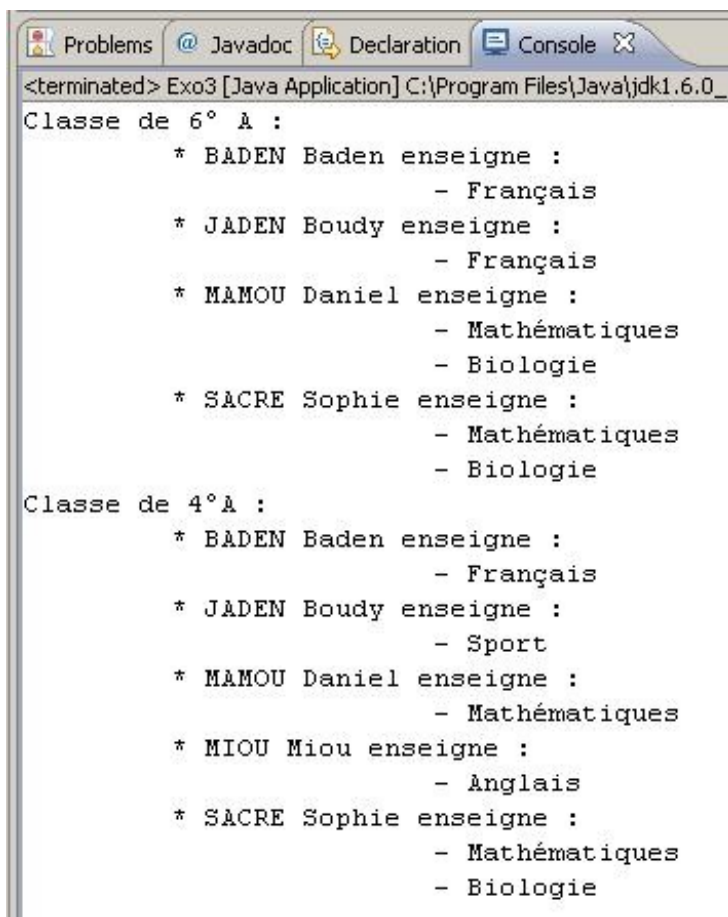
            ResultSet result = state.executeQuery(query);
            String nom = "";

            while(result.next()){
                if(!nom.equals(result.getString("prof_nom"))){
                    nom = result.getString("prof_nom");
                    System.out.println(nom + " " +
            result.getString("prof_prenom") + " enseigne : ");
                }
                System.out.println("\t\t\t - " +
            result.getString("mat_nom"));
            }

            result.close();
            state.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Et le dernier pour la fin :



```
<terminated> Exo3 [Java Application] C:\Program Files\Java\jdk1.6.0_1
Classe de 6° A :
    * BADEN Baden enseigne :
        - Français
    * JADEN Boudy enseigne :
        - Français
    * MAMOU Daniel enseigne :
        - Mathématiques
        - Biologie
    * SACRE Sophie enseigne :
        - Mathématiques
        - Biologie

Classe de 4°A :
    * BADEN Baden enseigne :
        - Français
    * JADEN Boudy enseigne :
        - Sport
    * MAMOU Daniel enseigne :
        - Mathématiques
    * MIOU Miou enseigne :
        - Anglais
    * SACRE Sophie enseigne :
        - Mathématiques
        - Biologie
```

Secret (cliquez pour afficher)

Code : Java

```
package com.sdz.exo;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Exo3 {
    public static void main(String[] args) {

        try {
            Class.forName("org.postgresql.Driver");

            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "postgres";

            Connection conn = DriverManager.getConnection(url, user,
            passwd);
            Statement state = conn.createStatement();

            String query = "SELECT prof_nom, prof_prenom, mat_nom, cls_nom
FROM professeur";
            query += " INNER JOIN j_mat_prof ON jmp_prof_k = prof_id";
            query += " INNER JOIN matiere ON jmp_mat_k = mat_id";
            query += " INNER JOIN j_cls_jmp ON jcm_jmp_k = jmp_id";
            query += " INNER JOIN classe ON jcm_cls_k = cls_id AND cls_id
IN(1, 7) ";
            query += " ORDER BY cls_nom DESC, prof_nom";

            ResultSet result = state.executeQuery(query);
            String nom = "";
```

```
String nom = "",  
String nomClass = "";  
  
while(result.next()){  
    if(!nomClass.equals(result.getString("cls_nom"))){  
        nomClass = result.getString("cls_nom");  
        System.out.println("Classe de " + nomClass + " :");  
    }  
  
    if(!nom.equals(result.getString("prof_nom"))){  
        nom = result.getString("prof_nom");  
        System.out.println("\t * " + nom + " " +  
result.getString("prof_prenom") + " enseigne : ");  
    }  
    System.out.println("\t\t\t - " +  
result.getString("mat_nom"));  
}  
  
        result.close();  
        state.close();  
  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
}
```

Voilà : maintenant que vous vous êtes bien entraînés, nous allons approfondir un peu tout ceci... Mais avant, le QCM des familles. 🐼

C'est un chapitre riche mais pas trop compliqué...

Prenez le temps de voir comment le tout s'articule autour des **ResultSet** et tutti quanti.

Les plus téméraires me retrouveront au prochain chapitre. 😊

Allons un peu plus loin

Nous avons vu comment faire des requêtes SQL, les exécuter et les afficher.
Le moment est venu de creuser un peu la question en allant un peu plus en profondeur...

Nous allons revenir sur les objets **Statement** et **ResultSet** en détaillant un peu le tout ! 😊

Statement

Vous avez vu comment obtenir un objet **Statement** ; par contre, ce que vous ignorez, c'est que je ne vous ai pas tout dit... 😬
Je sais, je suis un vilain cysboy...

Vous savez déjà que, pour récupérer un objet **Statement**, vous devez le demander gentiment à un objet **Connection**, ceci en invoquant la méthode `createStatement()` . Je ne vous apprend rien, là ! Du moins j'espère... 😊

Par contre, ce que vous ne savez pas, c'est que vous pouvez spécifier des paramètres pour la création de l'objet **Statement**...
Ces paramètres permettent différentes actions lors du parcours des résultats via l'objet **ResultSet**.

Paramètres pour la lecture du jeu d'enregistrements :

- **TYPE_FORWARD_ONLY** : le résultat n'est consultable qu'en avant. IMPOSSIBLE de revenir en arrière lors de la lecture ;
- **TYPE_SCROLL_SENSITIVE** : le parcours peut se faire en avant ou en arrière, le curseur peut se positionner n'importe où mais si des changements sont faits dans la base pendant la lecture, ces changements ne seront pas visibles ;
- **TYPE_SCROLL_INSENSITIVE** : comme le précédent sauf que les changements sont directement visibles lors du parcours des résultats.

Paramètres concernant la possibilité de mise à jour du jeu d'enregistrements :

- **CONCUR_READONLY** : les données sont consultables en lecture seule. Pas de mise à jour des valeurs pour mettre la base à jour ;
- **CONCUR_UPDATABLE** : les données sont modifiables et, lors d'une modification, la base est mise à jour.



Par défaut, les **ResultSet** issus d'un **Statement** sont **TYPE_FORWARD_ONLY** pour le type de parcours et **CONCUR_READONLY** pour les actions possibles.

Ces paramètres sont des variables statiques de la classe **ResultSet**, vous savez donc comment les utiliser.. 😊

Voici comment créer un **Statement** permettant à l'objet résultat de pouvoir être lu d'avant en arrière avec possibilité de modification :

Code : Java

```
Statement state =  
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
ResultSet.CONCUR_UPDATABLE);
```

Facile... 😊

Vous avez appris à créer des **Statement** avec des paramètres, mais saviez-vous qu'il existe un autre type de **Statement** ?



Nous verrons comment utiliser les fonctionnalités des **ResultSet** juste après ce point...

Les requêtes préparées

Il va falloir vous accrocher un tout petit peu...

De tels objets sont créés exactement de la même façon que des **Statement** classiques, sauf qu'au lieu d'avoir :

Code : Java

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

nous avons :

Code : Java

```
PreparedStatement stm = conn.prepareStatement("SELECT * FROM  
classe");
```

Jusqu'ici, rien de particulier ! Cependant, une différence est déjà effective à ce stade : **la requête SQL est pré-compilée !** Ceci a pour effet que celle-ci s'exécutera plus vite dans le moteur SQL de la BDD, c'est sûr puisqu'il n'aura pas à la compiler... 🤖
En fait, en règle générale, on utilise ce genre d'objet pour des requêtes ayant beaucoup de paramètres ou des requêtes pouvant être exécutées plusieurs fois...



C'est tout ce qu'il y a comme différence ?

Non, bien sûr. Je vous ai dit dans le titre que **nous allons préparer des requêtes !**.
Il y a une différence de taille entre l'objet **PreparedStatement** et l'objet **Statement** : dans le premier, on peut utiliser des paramètres à trous !



Quoi ? 🤔

Je me doutais que vous auriez du mal à comprendre... En fait, vous pouvez insérer un caractère spécial dans vos requêtes et remplacer ce caractère grâce à des méthodes de l'objet **PreparedStatement** en spécifiant sa place et sa valeur (son type étant défini par la méthode utilisée...).

Voici un exemple :

Code : Java

```
package com.sdz.prepare;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;

public class Prepare {

    public static void main(String[] args) {
        try {
            Class.forName("org.postgresql.Driver");

            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "postgres";

            Connection conn = DriverManager.getConnection(url, user, passwd);
            Statement state =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);

            //On crée notre requête
            String query = "SELECT prof_nom, prof_prenom FROM professeur";
            //Premier trou pour le nom du professeur
            query += " WHERE prof_nom = ?";
            //Deuxième trou pour un id de professeur
            query += " OR prof id = ?";
```

```

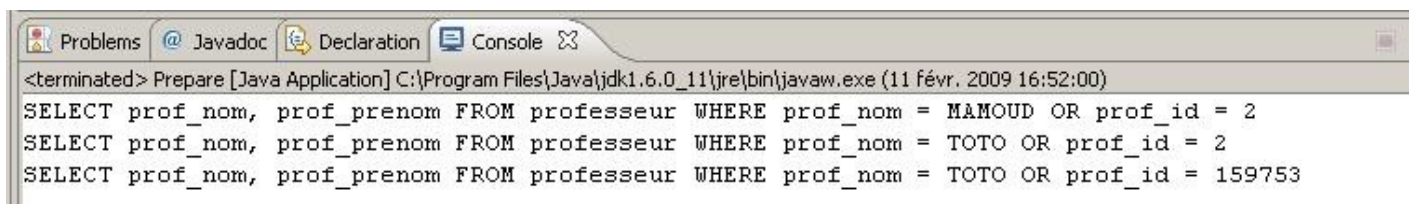
//On crée notre objet avec la requête en paramètre
PreparedStatement prepare = conn.prepareStatement(query);
//On remplace le trou numéro 1 => le nom du professeur
prepare.setString(1, "MAMOUD");
//On remplace le trou numéro 2 => l'id du professeur
prepare.setInt(2, 2);
//On affiche
System.out.println(prepare.toString());
//On remodifie le trou numéro 1
prepare.setString(1, "TOTO");
//On ré-affiche
System.out.println(prepare.toString());
//On remodifie le trou numéro 2
prepare.setInt(2, 159753);
//On ré-affiche
System.out.println(prepare.toString());

        prepare.close();
        state.close();

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Ce qui nous donne :



```

<terminated> Prepare [Java Application] C:\Program Files\Java\jdk1.6.0_11\jre\bin\javaw.exe (11 févr. 2009 16:52:00)
SELECT prof_nom, prof_prenom FROM professeur WHERE prof_nom = MAMOUD OR prof_id = 2
SELECT prof_nom, prof_prenom FROM professeur WHERE prof_nom = TOTO OR prof_id = 2
SELECT prof_nom, prof_prenom FROM professeur WHERE prof_nom = TOTO OR prof_id = 159753

```

Je sais, ça fait toujours ça la première fois... 🤔

Je ne vous cache pas que ceci peut s'avérer très utile.



En effet ! Mais qu'y a-t-il comme méthode d'affectation de valeur ?

C'est simple : vous vous souvenez de la petite liste de méthodes de l'objet **ResultSet** pour récupérer des données ? À peu de choses près, la même mais avec **setXXX** à la place de **getXXX**. 😊

Tout comme son homologue sans trou, cet objet peut prendre les mêmes types de paramètres pour la lecture et pour la modification des données lues :

Code : Java

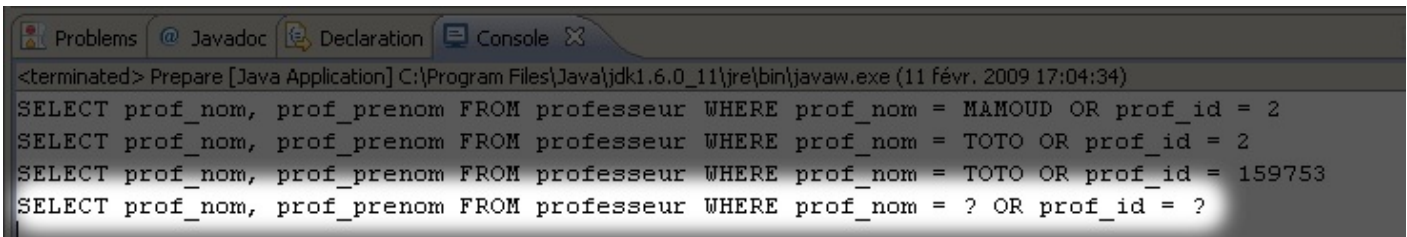
```

PreparedStatement prepare = conn.prepareStatement(
    query,
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY
);

```

Vous avez aussi une méthode retournant un objet **ResultSetMetaData** mais encore une méthode permettant de nettoyer les changements de valeur des trous : `prepare.clearParameters();` .

Si vous ajoutez cette méthode à la fin de ce que je vous ai fait tout à l'heure et que vous affichez à nouveau le contenu de notre objet, vous aurez ceci :



```

<terminated> Prepare [Java Application] C:\Program Files\Java\jdk1.6.0_11\jre\bin\javaw.exe (11 févr. 2009 17:04:34)
SELECT prof_nom, prof_prenom FROM professeur WHERE prof_nom = MAMOUD OR prof_id = 2
SELECT prof_nom, prof_prenom FROM professeur WHERE prof_nom = TOTO OR prof_id = 2
SELECT prof_nom, prof_prenom FROM professeur WHERE prof_nom = TOTO OR prof_id = 159753
SELECT prof_nom, prof_prenom FROM professeur WHERE prof_nom = ? OR prof_id = ?

```

Bon, je pense que le moment est venu de voir l'objet **ResultSet** plus en détail...

ResultSet 2 : le retour

Nous allons voir comment se promener dans nos objets **ResultSet**, vu que nous avons vu comment permettre cela... 😊

En fait, l'objet de résultat offre beaucoup de méthodes afin de pouvoir se promener dans les résultats.

Cela, si vous avez bien préparé l'objet **Statement**.

Vous avez la possibilité de :

- vous positionner avant la première ligne de votre résultat : `res.beforeFirst()` ;
- savoir si vous êtes sur l'avant-première ligne : `res.isBeforeFirst()` ;
- vous positionner sur la première ligne de votre résultat : `res.first()` ;
- savoir si vous êtes sur la première ligne : `res.isFirst()` ;
- vous retrouver sur la dernière ligne : `res.last()` ;
- savoir si vous êtes sur la dernière ligne : `res.isLast()` ;
- vous mettre après la dernière ligne de résultat : `res.afterLast()` ;
- savoir si vous êtes sur l'après-dernière ligne : `res.isAfterLast()` ;
- aller de la première ligne à la dernière : `res.next()` ;
- aller de la dernière ligne à la première : `res.previous()` ;
- vous positionner sur une ligne précise de votre résultat : `res.absolute(5)` ;
- vous positionner sur une ligne par rapport à votre emplacement actuel : `res.relative(-3)` .

Je vous ai concocté un morceau de code mettant en oeuvre tout ceci...

Vous devriez retrouver les informations vu que ce code est commenté. 😊

Code : Java

```

package com.sdz.resultset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Resultset {
    public static void main(String[] args) {
        try {
            //-----
            //TOUT CECI, VOUS CONNAISSEZ
            Class.forName("org.postgresql.Driver");
            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "postgres";

            Connection conn = DriverManager.getConnection(url, user, passwd);
            Statement state =
            conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);

            String query = "SELECT prof_nom, prof_prenom FROM professeur";
            ResultSet res = state.executeQuery(query);
            //-----

```

```

-----

    /**
    int i = 1;

    System.out.println("\n\t-----");
");
    System.out.println("\tLECTURE STANDARD.");
    System.out.println("\t-----");

    while(res.next()){
        System.out.println("\tNom : " + res.getString("prof_nom") + " \t
prénom : "+ res.getString("prof_prenom"));
        //On regarde si nous sommes sur la dernière ligne du résultat
        if(res.isLast())
            System.out.println("\t\t* DERNIER RESULTAT !\n");
        i++;
    }

    //Une fois terminé, nous contrôlons si nous sommes bien à
    l'extérieur des lignes de résultat
    if(res.isAfterLast())
        System.out.println("\tNous venons de terminer !\n");

    System.out.println("\t-----");
    System.out.println("\tLecture en sens contraire.");
    System.out.println("\t-----");

    //Nous sommes donc à la fin
    //Nous pouvons parcourir le résultat en sens contraire
    while(res.previous()){
        System.out.println("\tNom : " + res.getString("prof_nom") + " \t
prénom : "+ res.getString("prof_prenom"));

        //On regarde si on est sur la première ligne du résultat
        if(res.isFirst())
            System.out.println("\t\t* RETOUR AU DEBUT !\n");
    }

    //On regarde si on est sur l'avant-première ligne du résultat
    if(res.isBeforeFirst())
        System.out.println("\tNous venons de revenir au début !\n");

    System.out.println("\t-----");
    System.out.println("\tAprès positionnement absolu du curseur à la
place N° " + i/2 + ".");
    System.out.println("\t-----");
    //On positionne le curseur sur la ligne i/2, peu importe où on
    est
    res.absolute(i/2);
    while(res.next())
        System.out.println("\tNom : " + res.getString("prof_nom") + " \t
prénom : "+ res.getString("prof_prenom"));

    System.out.println("\t-----");
    System.out.println("\tAprès positionnement relatif du curseur à
la place N° " + (i -(i-2)) + ".");
    System.out.println("\t-----");
    //On met le curseur à la ligne actuelle moins i-2
    //Si nous reculons donc de i-2 lignes
    //Si nous n'avions pas mis de signe moins, nous aurions avancé
    de i-2 lignes
    res.relative(-(i-2));
    while(res.next())
        System.out.println("\tNom : " + res.getString("prof_nom") + " \t
prénom : "+ res.getString("prof_prenom"));

        res.close();
        state.close();

```



```

    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Voyez ce que donne ce code :

LECTURE STANDARD.

Nom : MAMOU	prénom : Daniel
Nom : SACRE	prénom : Sophie
Nom : JADEN	prénom : Boudy
Nom : BADEN	prénom : Baden
Nom : MIOU	prénom : Miou
Nom : BORA	prénom : Kernel
Nom : CAISSE	prénom : Jean
Nom : MOISSAT	prénom : Marc

* DERNIER RESULTAT !

Nous venons de terminer !

Lecture en sens contraire.

Nom : MOISSAT	prénom : Marc
Nom : CAISSE	prénom : Jean
Nom : BORA	prénom : Kernel
Nom : MIOU	prénom : Miou
Nom : BADEN	prénom : Baden
Nom : JADEN	prénom : Boudy
Nom : SACRE	prénom : Sophie
Nom : MAMOU	prénom : Daniel

* RETOUR AU DEBUT !

Nous venons de revenir au début !

Après positionnement absolu du curseur à la place N° 4.

Nom : MIOU	prénom : Miou
Nom : BORA	prénom : Kernel
Nom : CAISSE	prénom : Jean
Nom : MOISSAT	prénom : Marc

Après positionnement relatif du curseur à la place N° 2.

Nom : JADEN	prénom : Boudy
Nom : BADEN	prénom : Baden
Nom : MIOU	prénom : Miou
Nom : BORA	prénom : Kernel
Nom : CAISSE	prénom : Jean
Nom : MOISSAT	prénom : Marc

Je pense que la majorité des choses ont été vues ici...

Par contre, faites bien attention à l'endroit où vous vous situez dans le parcours de la requête !



Il y a des emplacements particuliers. Par exemple, si vous n'êtes pas encore positionnés sur le premier élément et que vous faites un `rs.relative(1)`, vous vous retrouvez sur le premier élément. De même, un `rs.absolute(0)` correspond à un `rs.beforeFirst()` !

Donc, lorsque vous voulez spécifier la place du curseur sur la première ligne, vous utiliserez `absolute(1)` ; et ceci, peu importe où vous vous trouvez !



Par contre, ceci nécessite que le **ResultSet** soit de type **TYPE_SCROLL_SENSITIVE** ou **TYPE_SCROLL_INSENSITIVE**, sinon vous aurez une exception !

Bon, rien de bien méchant ici...

Je vous propose donc de clore ce chapitre par un petit QCM avant de continuer sur notre lancée...

J'espère que ce chapitre vous a permis d'y voir plus clair sur le fonctionnement global de tout ce mic-mac...

Après la lecture : l'édition

Nous avons appris à utiliser des requêtes SQL avec des **ResultSet** afin de récupérer des informations provenant de la base. Nous allons voir comment éditer nos tables en insertion, modification et suppression dans nos programmes Java ! Vous allez voir, c'est simplissime !! 😊

Modifier des données

Je sais que beaucoup d'entre vous se sont dit, lors du chapitre précédent :

*"t'es sympa, tu nous dis que les objets **ResultSet** peuvent modifier des données lors de parcours de résultats... et tu ne nous montres même pas ça !"*

J'ai juste préféré séparer le côté lecture du côté lecture et modifications.

D'ailleurs, c'est par ça que nous allons commencer. Bien sûr, il faudra que l'objet **Statement** retourne un **ResultSet** permettant cela !

En fait, durant la lecture, vous pouvez utiliser des méthodes qui ressemblent à celle que je vous ai déjà montrée lors du parcours d'un résultat... Vous vous souvenez, les méthodes comme :

- `res.getAscii()` ;
- `res.getBytes()` ;
- `res.getInt()` ;
- `res.getString()` ;
- ...

Sauf qu'ici, vous remplacerez `getXXX()` ; par `updateXXX()` ; . Ces méthodes de mise à jour des données prennent deux paramètres :

- 1 : le nom de la colonne (**String**) ;
- 2 : la valeur à mettre à la place de la valeur existante (dépend de la méthode utilisée) ;



Comment ça : dépend de la méthode utilisée ?

C'est simple :

- `updateFloat(String nomColonne, float value)` : prend un **float** comme valeur ;
- `updateString(String nomColonne, String value)` : prend une chaîne de caractères en paramètre ;
- ...

Donc, changer la valeur d'un champ est très simple mais il faut, en plus des changements de valeurs, valider ces changements pour qu'ils soient effectifs, et ceci se fait grâce à la méthode `updateRow()` . De la même façon, vous pouvez annuler des changements avec la méthode `cancelRowUpdates()` .



Si vous avez à annuler des modifications, vous devrez le faire avant la méthode de validation, sinon, l'annulation sera ignorée !

Je vous propose de voir un petit exemple de mise à jour :

Code : Java

```
package com.sdz.resultset;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Modif {
    public static void main(String[] args) {
        try {
```

```

//-----
//TOUT CECI, VOUS CONNAISSEZ
Class.forName("org.postgresql.Driver");
String url = "jdbc:postgresql://localhost:5432/Ecole";
String user = "postgres";
String passwd = "postgres";

Connection conn = DriverManager.getConnection(url, user, passwd);
//On autorise la mise à jour des données et la mise à jour de
l'affichage
Statement state =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_UPDATABLE);

//On va chercher une ligne dans la base de données
String query = "SELECT prof_id, prof_nom, prof_prenom FROM
professeur WHERE prof_nom = 'MAMOU'";
ResultSet res = state.executeQuery(query);
//-----

//On se positionne sur la première ligne de donnée
res.first();

//On affiche ce qu'on trouve
System.out.println("NOM : " + res.getString("prof_nom") + " -
PRENOM : " + res.getString("prof_prenom"));

//On met à jour les champs
res.updateString("prof_nom", "COURTEL");
res.updateString("prof_prenom", "Angelo");
//On valide
res.updateRow();

//Et voici les modifications
System.out.println("*****");
System.out.println("APRES MODIFICATION : ");
System.out.println("\tNOM : " + res.getString("prof_nom") + " -
PRENOM : " + res.getString("prof_prenom") + "\n");

//On remet les infos du départ
res.updateString("prof_nom", "MAMOU");
res.updateString("prof_prenom", "Daniel");
//On revalide
res.updateRow();

//Et voilà !
System.out.println("*****");
System.out.println("APRES REMODIFICATION : ");
System.out.println("\tNOM : " + res.getString("prof_nom") + " -
PRENOM : " + res.getString("prof_prenom") + "\n");

res.close();
state.close();

} catch (Exception e) {
e.printStackTrace();
}
}
}

```

Et voici ce que j'obtiens :

```

NOM : MAMOU - PRENOM : Daniel
*****
APRES MODIFICATION :
      NOM : COURTEL - PRENOM : Angelo

*****
APRES REMODIFICATION :
      NOM : MAMOU - PRENOM : Daniel

```

Donc, le temps d'un instant, les données ont été modifiées dans la base de données, nous avons donc réussi notre pari !



Ôte-nous d'un doute, une bête requête SQL n'aurait pas pu faire l'affaire ?

Bien sûr que si ! 😊

Mais au moins, vous avez vu comment modifier des lignes via l'objet **ResultSet**...

Nous allons voir comment exécuter les autres types de requêtes avec Java !

Statement, toujours plus fort

Vous savez depuis quelques temps déjà que ce sont les objets **Statement** qui sont chargés d'exécuter les instructions SQL. Par conséquent, vous devez avoir deviné que les requêtes de type `INSERT`, `UPDATE`, `DELETE` et `CREATE` seront aussi exécutées par ces mêmes objets :

- **INSERT**
- **UPDATE**
- **DELETE**
- **CREATE**

Voici un code d'exemple :

Code : Java

```

package com.sdz.statement;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class State {

    public static void main(String[] args) {

        try {
            Class.forName("org.postgresql.Driver");
            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "postgres";

            Connection conn = DriverManager.getConnection(url, user, passwd);
            //On autorise la mise à jour des données et la mise à jour de
            l'affichage
            Statement state =
            conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
            PreparedStatement prepare = conn.prepareStatement("UPDATE

```

```

professeur set prof_prenom = ? WHERE prof_nom = 'MAMOU');

//On va chercher une ligne dans la base de données
String query = "SELECT prof_nom, prof_prenom FROM professeur
WHERE prof_nom = 'MAMOU'";

//On exécute la requête
ResultSet res = state.executeQuery(query);
res.first();
//On affiche
System.out.println("\n\tDONNEES D'ORIGINE : ");
System.out.println("\t-----");
System.out.println("\tNOM : " + res.getString("prof_nom") + " -
PRENOM : " + res.getString("prof_prenom"));

//On paramètre notre requête préparée
prepare.setString(1, "Gérard");
//On exécute
prepare.executeUpdate();

res = state.executeQuery(query);
res.first();
//On affiche à nouveau
System.out.println("\n\t\t APRES MAJ : ");
System.out.println("\t\t * NOM : " + res.getString("prof_nom") +
" - PRENOM : " + res.getString("prof_prenom"));

//On refait une mise à jour
prepare.setString(1, "Daniel");
prepare.executeUpdate();

res = state.executeQuery(query);
res.first();
//on affiche une nouvelle fois
System.out.println("\n\t\t REMISE A ZERO : ");
System.out.println("\t\t * NOM : " + res.getString("prof_nom") +
" - PRENOM : " + res.getString("prof_prenom"));

        prepare.close();
        res.close();
        state.close();

    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}

```

Ce qui me donne :

DONNEES D'ORIGINE :

NOM : MAMOU - PRENOM : Daniel

APRES MAJ :

* NOM : MAMOU - PRENOM : Gérard

REMISE A ZERO :

* NOM : MAMOU - PRENOM : Daniel

Ici, nous avons utilisé un **PreparedStatement**, histoire de faire compliqué dès le premier coup... 😊

Mais vous auriez pu tout aussi bien utiliser un **Statement** tout simple et invoquer la méthode `executeUpdate(String query)` .

Vous savez quoi ? Pour les autres types de requêtes, il suffit d'invoquer la même méthode que pour la mise à jour... En fait, celle-ci retourne un booléen qui permet de savoir si le traitement a réussi ou non. 😊

Voici quelques exemples :

Code : Java

```
state.executeUpdate("INSERT INTO professeur (prof_nom, prof_prenom)
VALUES ('SALMON', 'Dylan')");
state.executeUpdate("DELETE FROM professeur WHERE prof_nom =
'MAMOU'");
```

C'est très simple à utiliser, vous ne pourrez pas dire le contraire... 😊

Cependant, je ne sais pas si vous savez ceci, mais certains moteurs SQL, comme PostgreSQL, vous proposent de gérer vos requêtes SQL (celles qui ont pour effet une modification de la base, pas celle de visualisation de données) avec ce qu'on appelle des transactions...

Gérer les transactions manuellement

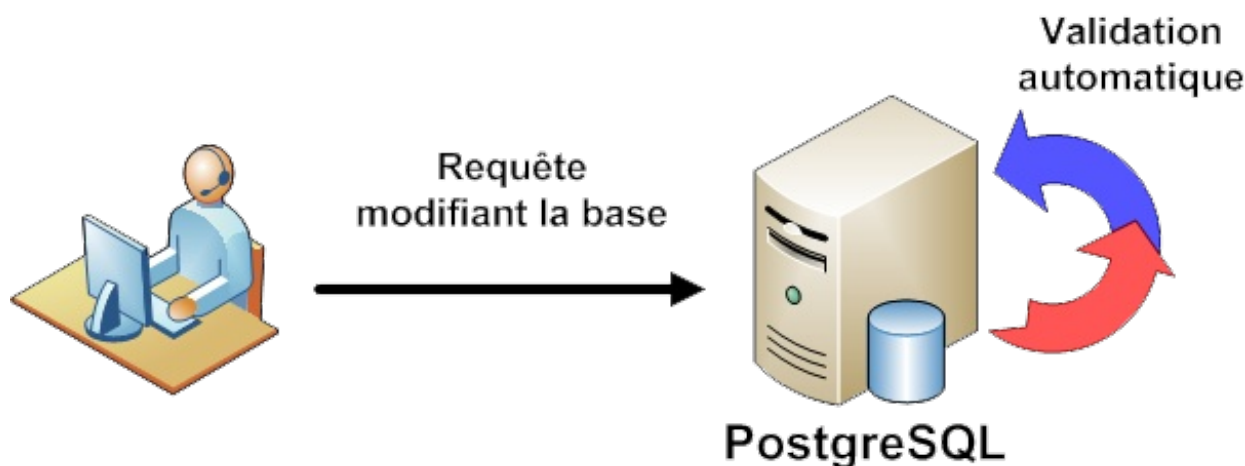


Tu es bien gentil, mais qu'est-ce que c'est que ça ?

Bon, alors, par où vais-je commencer ?...

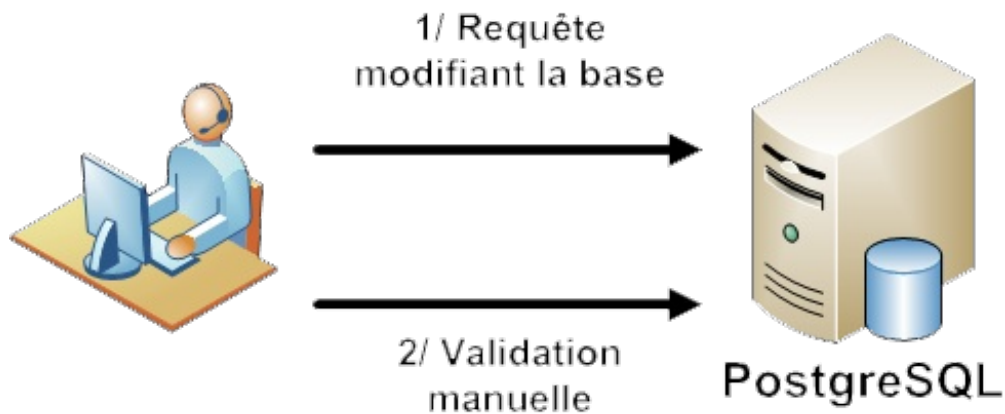
Lorsque vous insérez, modifiez ou supprimez des données dans PostgreSQL, il se passe un événement automatique : **la validation des modifications par le moteur SQL!**

C'est aussi simple que ça... Un petit schéma pour visualiser :



Lorsque vous exécutez une requête de type **INSERT**, **CREATE**, **UPDATE** ou **DELETE**, ces requêtes modifient les données présentes dans la base. Une fois exécutée, le moteur SQL valide directement ces modifications !

Par contre, vous pouvez avoir la main sur ce point :



Comme ceci, c'est vous qui avez le contrôle sur vos données !



Je ne vois pas l'intérêt de faire ça !

C'est simple : pour pouvoir contrôler **l'intégrité de vos données !**

Imaginez que vous devez faire deux requêtes, une modification et une insertion et que vous partez du principe que l'insertion dépend de la mise à jour ! Comment feriez-vous si de mauvaises données étaient mises à jour ? L'insertion qui en découle serait mauvaise... 😞

Ceci, bien sûr, si le moteur SQL valide automatiquement les requêtes faites...



Oui, vu sous cet angle... Comment on gère ça, alors ?

On spécifie au moteur SQL de ne pas valider automatiquement. **Ou plutôt** de valider automatiquement les requêtes SQL avec une méthode qui prend un booléen en paramètre, mais **qui ne concernera pas l'objet *Statement*, mais l'objet *Connection*** :

Code : Java

```

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Transact {

    public static void main(String[] args) {
        try {
            Class.forName("org.postgresql.Driver");

            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "batterie";

            Connection conn = DriverManager.getConnection(url, user, passwd);
            conn.setAutoCommit(false);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
  
```

Par contre, lorsque vous voulez que vos requêtes soient prises en compte, il faut utiliser la méthode `conn.commit()` ; .



En mode `setAutoCommit(false)` , si vous ne validez pas vos requêtes, elles ne seront pas prises en compte !

Vous pouvez revenir à tout moment en mode validation automatique avec `setAutoCommit(true)` ;

Voici un exemple :

Code : Java

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.Statement;

public class Transact {

    public static void main(String[] args) {
        try {
            Class.forName("org.postgresql.Driver");

            String url = "jdbc:postgresql://localhost:5432/Ecole";
            String user = "postgres";
            String passwd = "batterie";

            Connection conn = DriverManager.getConnection(url, user, passwd);
            conn.setAutoCommit(false);
            Statement state =
            conn.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
            ResultSet.CONCUR_UPDATABLE);
            String query = "UPDATE professeur SET prof_prenom = 'Cyrille'
            WHERE prof_nom = 'MAMOU'";

            ResultSet result = state.executeQuery("SELECT * FROM professeur
            WHERE prof_nom = 'MAMOU'");
            result.first();
            System.out.println("NOM : " + result.getString("prof_nom") + " -
            PRENOM : " + result.getString("prof_prenom"));

            state.executeUpdate(query);

            result = state.executeQuery("SELECT * FROM professeur WHERE
            prof_nom = 'MAMOU'");
            result.first();
            System.out.println("NOM : " + result.getString("prof_nom") + " -
            PRENOM : " + result.getString("prof_prenom"));

            result.close();
            state.close();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Vous pouvez exécuter le code autant de fois que vous voulez, vous aurez toujours :

```
NOM : MAMOU - PRENOM : Daniel
NOM : MAMOU - PRENOM : Cyrille
```

Vous voyez : malgré la requête de mise à jour, celle-ci est inopérante ! Oh oui ! Vous pouvez voir les modifications durant le temps d'exécution du script, mais vu que vous n'avez pas validé les modifications, celles-ci sont annulées à la fin...

Pour que la mise à jour soit effective, il aurait fallu faire un `conn.commit()` avant la fin du script ! 😊

Bon, je crois qu'on en a assez vu pour ce chapitre... En route pour le QCM ! 🤖

Alors ? Je ne vous avais pas dit que ce chapitre allait être tout simple ?

Chose promise, chose due !

Maintenant, je vous propose de voir comment uniformiser un peu tout ça... C'est vrai qu'établir tout le temps la connexion est un peu fastidieux alors qu'une seule instance de celle-ci suffirait...

C'est ce que je vous propose de voir dans le chapitre qui suit. 😊

N'avoir qu'une instance de sa connexion

L'exécution de commandes SQL n'a plus de secret pour vous, je le sais, c'est moi qui vous ai formés ! 🤖
Non, je blague...

Après vous avoir fait découvrir tout ça, je me suis dis : *montrer une approche un peu plus objet ne serait pas du luxe !*

C'est vrai, se connecter sans arrêt à notre base de données commence à être fastidieux. Je vous propose donc d'y remédier avec ce chapitre. 😊

Pourquoi se connecter qu'une seule fois ?



Pourquoi tu veux absolument qu'on aie une seule instance de notre objet **Connection** ?

Parce que ça ne sert pas à grand-chose de réinitialiser la connexion à votre BDD. Rappelez-vous que la connexion sert à faire le pont entre votre base et votre application. **Pourquoi voulez-vous que votre application se connecte à chaque fois à votre BDD ?** Une fois la connexion effective, pourquoi vouloir la refaire ? Votre application et votre BDD peuvent discuter !



Bon, c'est vrai qu'avec du recul, ça paraît superflu...

Du coup, comment tu fais pour garantir qu'une seule instance de **Connection** existe dans l'application ?

C'est ici que le *pattern singleton* arrive !

Ce pattern est peut-être l'un des plus simples à comprendre même s'il va y avoir un point qui va vous faire bondir... 😬

Le principe de base de ce pattern est d'interdire l'instanciation d'une classe !



On peut savoir comment tu comptes t'y prendre ?

C'est ça qui va vous faire bondir : avec un constructeur déclaré **private** !

Nul ne sert de lambiner, voyons comment rendre tout ceci possible...

Le pattern singleton

Alors ce que nous voulons, c'est une seule instance de notre connexion ; voici une classe qui permet ça :

Code : Java

```
package com.sdz.connection;

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

public class SdzConnection{

    /**
     * URL de connection
     */
    private String url = "jdbc:postgresql://localhost:5432/Ecole";
    /**
     * Nom du user
     */
    private String user = "postgres";
    /**
     * Mot de passe du user
     */
    private String passwd = "postgres";
    /**
     * Objet Connection
     */
    private static Connection connect;

    /**
     * Constructeur privé
     */
}
```

```

*/
private SdzConnection() {
    try {
        connect = DriverManager.getConnection(url, user, passwd);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

/**
 * Méthode qui va nous retourner notre instance
 * et la créer si elle n'existe pas...
 * @return
 */
public static Connection getInstance() {
    if(connect == null) {
        new SdzConnection();
    }
    return connect;
}
}

```

Nous avons ici une classe avec un constructeur privé (même si, ici, ce n'était pas nécessaire) : du coup, impossible d'avoir une instance de cet objet et impossible d'accéder aux attributs puisqu'ils sont déclarés **private** !
 Notre objet **Connection** est instancié dans le constructeur privé et la seule méthode accessible de l'extérieur de la classe est `getInstance()` . C'est donc cette méthode qui aura pour rôle de créer la connexion lorsque celle-ci n'existe pas, et seulement dans ce cas.

Pour en être bien sûrs, nous allons faire un petit test... 😊

Voici le code un peu modifié de la méthode `getInstance()` ...

Code : Java

```

public static Connection getInstance() {
    if(connect == null) {
        new SdzConnection();
        System.out.println("INSTANCIATION DE LA CONNEXION SQL ! ");
    }
    else{
        System.out.println("CONNEXION SQL EXISTANTE ! ");
    }
    return connect;
}
}

```

Ceci a pour but de voir quand la connexion est réellement créée. Ensuite, il ne nous manque plus qu'un code de test. Oh ! Ben ça alors ! J'en ai un sous la main :

Code : Java

```

package com.sdz.connection;

import java.sql.DatabaseMetaData;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;

public class Test {

    public static void main(String[] args) {

        try {
            //1

```

```

        PreparedStatement prepare =
SdzConnection.getInstance().prepareStatement("SELECT * FROM classe
WHERE cls_nom = ?");
        //2
        Statement state = SdzConnection.getInstance().createStatement();
        //3
        SdzConnection.getInstance().setAutoCommit(false);
        //Et 4 appels à la méthode getInstance()
        DatabaseMetaData meta =
SdzConnection.getInstance().getMetaData();

    } catch (SQLException e) {
        e.printStackTrace();
    }
}
}

```

Il y a 4 appels à la méthode en question, [que croyez-vous que ce code va afficher ?](#)

Secret (cliquez pour afficher)

```

INSTANCIATION DE LA CONNEXION SQL !
CONNEXION SQL EXISTANTE !
CONNEXION SQL EXISTANTE !
CONNEXION SQL EXISTANTE !

```

Vous avez la preuve que l'instanciation ne se fait qu'une seule fois et donc, que notre connexion à la BDD est unique !



Bon, ça c'est compris, par contre, pourquoi tu disais que le constructeur n'était pas nécessaire ?

Tout simplement parce que nous aurions pu avoir cela à la place et ça aurait tout aussi bien fonctionné :

Code : Java

```

package com.sdz.connection;

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

public class SdzConnection{

    /**
     * URL de connection
     */
    private static String url =
"jdbc:postgresql://localhost:5432/Ecole";
    /**
     * Nom du user
     */
    private static String user = "postgres";
    /**
     * Mot de passe du user
     */
    private static String passwd = "postgres";
    /**
     * Objet Connection
     */
    private static Connection connect;

```

```

/**
 * Méthode qui va nous retourner notre instance
 * et la créer si elle n'existe pas...
 * @return
 */
public static Connection getInstance() {
    if(connect == null) {
        try {
            connect = DriverManager.getConnection(url, user, passwd);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
    return connect;
}
}

```



Par contre, vous devrez rajouter la déclaration **static** de vos paramètres de connexion...

Vous pouvez relancer le code de test, vous verrez qu'il fonctionne toujours !

J'ai mis un constructeur privé d'abord car vous deviez savoir que cela existait, mais c'était superflu dans notre cas...



D'accord, mais j'ai une application *multi-threads*, tu es sûr qu'il n'y aura pas de conflit ?

Hum ! Vous savez déjà répondre à cette question si vous avez lu le [chapitre sur les threads du tuto Java](#) !

Il vous suffit de synchroniser la méthode `getInstance()` et le tour est joué...

Mais, parce qu'il y a un *mais*... cette méthode ne règle le problème qu'avant que la connexion ne soit instanciée. Autrement dit, une fois la connexion instanciée, la synchronisation ne sert plus à rien... 🤔

Le problème de *multi-threading* ne se pose pas vraiment pour une connexion à une BDD puisque ce *singleton* sert surtout de passerelle entre votre BDD et votre application, mais il peut y avoir d'autres objets que des connexions SQL qui ne doivent être instanciés qu'une fois et tous ne sont pas aussi laxistes concernant le multi-threading...

Voyons comment parfaire ce pattern avec un autre exemple qu'une connexion SQL...

Le singleton dans tous ces états

Alors... Nous allons travailler avec un autre exemple et vu que j'étais très inspiré, voici notre super singleton :

Code : Java

```

package com.sdz.connection;

public class SdzSingleton {

    /**
     * Le singleton
     */
    private static SdzSingleton single;

    /**
     * Variable d'instance
     */
    private String name = "";

    /**
     * Constructeur privé
     */
    private SdzSingleton() {
        this.name = "Mon singleton";
        System.out.println("\t\tCREATION DE L'INSTANCE ! ! !");
    }

    /**
     * Méthode d'accès au singleton
     */
}

```

```

* @return SdzSingleton
*/
public static SdzSingleton getInstance() {
    if(single == null)
        single = new SdzSingleton();

    return single;
}

/**
* Accesseur
* @return
*/
public String getName() {
    return this.name;
}
}

```



Quoi, c'est ça ton singleton ?

Oui, ce n'est pas que je manquais d'inspiration, c'est juste qu'avec une classe toute simple, on comprend mieux les choses...

Et voici notre classe de test :

Code : Java

```

package com.sdz.connection;

public class TestSingleton {

    public static void main(String[] args) {
        for(int i = 1; i < 4; i++)
            System.out.println("Appel N° " + i + " : " +
                SdzSingleton.getInstance().getName());
    }
}

```

Ce qui nous donne :

```

                CREATION DE L'INSTANCE ! ! !
Appel N° 1 : Mon singleton
Appel N° 2 : Mon singleton
Appel N° 3 : Mon singleton

```

La politique du singleton est toujours bonne. Maintenant, je vais vous poser une question : [quand croyez-vous que la création d'une instance soit la plus judicieuse ?](#)



Qu'est-ce que tu veux dire par là ?

C'est simple : ici, nous avons exécuté notre code et l'instance est créée lorsque qu'on la demande pour la première fois ! C'est le principal problème que pose le singleton et le multi-threading : la première instance... Une fois celle-ci créée, il y a moins de problème.



Parce que tu vois un autre endroit où créer l'instance, toi ?



Oui, au chargement de la classe par la JVM et ceci se fait en instanciant notre singleton à sa déclaration dans la classe, soit, comme ceci :

Code : Java

```
package com.sdz.connection;


public class SdzSingleton {

    /**
     * Le singleton
     */
    private static SdzSingleton single = new SdzSingleton();
    /**
     * Variable d'instance
     */
    private String name = "";

    /**
     * Constructeur privé
     */
    private SdzSingleton() {
        this.name = "Mon singleton";
        System.out.println("\n\t\tCREATION DE L'INSTANCE ! ! !");
    }

    /**
     * Méthode d'accès au singleton
     * @return SdzSingleton
     */
    public static SdzSingleton getInstance() {
        return single;
    }

    /**
     * Accesseur
     * @return
     */
    public String getName() {
        return this.name;
    }
}
```

Avec ce code, c'est la machine virtuelle qui va se charger de charger l'instance du singleton, bien avant que n'importe quel *thread* vienne taquiner la méthode `getInstance()` ... 

Il y a une autre méthode permettant de faire ceci, mais elle ne fonctionne parfaitement que depuis le JDK 1.5...

On appelle cette méthode : "**le verrouillage à double vérification**".

Cette méthode consiste à utiliser le mot clé **volatile** combiné au mot clé **synchronized**.

Pour les ZérOs qui l'ignorent, déclarer une variable **volatile** permet de s'assurer un accès ordonné des threads à une variable (plusieurs threads peuvent accéder à cette variable), Ceci marque le premier point de verrouillage.

Ensuite, la double vérification s'effectuera dans la méthode `getInstance()` , on synchronise UNIQUEMENT lorsque le singleton n'est pas créé.

Voici ce que ça nous donne :

Code : Java

```
package com.sdz.connection;

public class SdzSingleton {
```



```

/**
 * Le singleton
 */
private volatile static SdzSingleton single;
/**
 * Variable d'instance
 */
private String name = "";

/**
 * Constructeur privé
 */
private SdzSingleton() {
    this.name = "Mon singleton";
    System.out.println("\n\t\tCREATION DE L'INSTANCE ! ! !");
}

/**
 * Méthode d'accès au singleton
 * @return SdzSingleton
 */
public static SdzSingleton getInstance() {
    if(single == null) {
        synchronized(SdzSingleton.class) {
            if(single == null)
                single = new SdzSingleton();
        }
    }

    return single;
}

/**
 * Accesseur
 * @return
 */
public String getName() {
    return this.name;
}
}


```

Voilà : vous êtes désormais incollables sur le singleton !

Je pense qu'après tout ça, un QCM s'impose...


Bon : vous devez vous rendre compte que travailler avec JDBC n'a rien de compliqué, en définitive !

Je vois dans vos regards que vous avez perdu le côté mystifié de JDBC, maintenant ; vous devriez être à même d'utiliser cette API dans vos programmes Java.

Tiens, avec ce que nous venons de voir, ça me donne une idée de TP... 

TP : un testeur de requête

Bon, vous avez appris un tas de choses et il est grand temps de faire un peu de pratique !

Dans ce TP, je vais vous demander de réaliser un testeur de requête SQL... Vous ne voyez pas où je veux en venir ? Lisez donc la suite... 

Cahier des charges

Alors... Le but du jeu est de :

- pouvoir avoir une IHM permettant la saisie d'une requête SQL dans un champ ;
- lancer l'exécution de la requête grâce à un bouton ;
- ce bouton devra être dans une barre d'outils ;
- dans le cas où la requête renvoie 0 ou plusieurs résultats, afficher ceux-ci dans un **JTable** ;
- le nom des colonnes devra être visible ;
- en cas d'erreur, une pop-up (**JOptionPane**) contenant le message s'affichera ;
- un petit message affichera le temps d'exécution de la requête ainsi que le nombre de lignes retournées en bas de fenêtre.

Vous avez de quoi faire, ici...

Bon, si vous ne savez pas comment faire pour le temps d'exécution de la requête, je vous donne un indice :

`System.currentTimeMillis()` ; retourne un **long**...



Pour les ZérOs n'ayant pas lu la partie événementielle du tuto Java, je vous autorise à faire une version en mode console ; par contre, celle-ci n'aura pas de correction... 

Quelques captures d'écran

Bon... Voici ce que j'ai obtenu avec mon code. Inspirez-vous en pour faire votre programme...



Je n'ai plus qu'à vous souhaiter bonne chance et bon courage !


Let's go !

Correction

DONG !

Le temps imparti est écoulé !

Bon : comme toujours, il s'agit d'une correction possible et non pas de **LA CORRECTION** !

J'espère que vous vous êtes bien pris la tête sur ce TP. Bien sûr, pas dans le sens où il vous a torturé l'esprit durant des heures jusqu'à vous rendre malades, mais plutôt dans le sens où celui-ci vous a permis de réfléchir et de découvrir des choses. 

Bon, assez tergiversé, vous devez être impatient de voir ce que j'ai fait :

Secret (cliquez pour afficher)

Classe SdzConnection.java :

Code : Java

```
package com.sdz.connection;

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

import javax.swing.JOptionPane;

public class SdzConnection {
```

```

/**
 * URL de connection
 */
private static String url =
"jdbc:postgresql://localhost:5432/Ecole";
/**
 * Nom du user
 */
private static String user = "postgres";
/**
 * Mot de passe du user
 */
private static String passwd = "postgres";
/**
 * Objet Connection
 */
private static Connection connect;

/**
 * Méthode qui va retourner notre instance
 * et la créer si elle n'existe pas...
 * @return
 */
public static Connection getInstance(){
    if(connect == null){
        try {
            connect = DriverManager.getConnection(url, user, passwd);
        } catch (SQLException e) {
            JOptionPane.showMessageDialog(null, e.getMessage(), "ERREUR DE
CONNEXION ! ", JOptionPane.ERROR_MESSAGE);
        }
    }
    return connect;
}
}

```

Classe Fenetre.java

Code : Java

```

package com.sdz.tp;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

import javax.swing.ImageIcon;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSplitPane;
import javax.swing.JTable;
import javax.swing.JTextArea;
import javax.swing.JToolBar;

import com.sdz.connection.SdzConnection;

public class Fenetre extends JFrame {

```

```

    /**
    * Toolbar pour le lancement des requêtes
    */
    private JToolBar tool = new JToolBar();
    /**
    * Le bouton
    */
    private JButton load = new JButton(new
    ImageIcon("img/load.png"));
    /**
    * Le délimiteur
    */
    private JSplitPane split;
    /**
    * Le conteneur de résultat
    */
    private JPanel result = new JPanel();
    /**
    * Requête par défaut pour le démarrage
    */
    private String requete = "SELECT * FROM classe";
    /**
    * Le composant dans lequel taper la requête
    */
    private JTextArea text = new JTextArea(requete);

    /**
    * Constructeur
    */
    public Fenetre() {
        setSize(900, 600);
        setTitle("TP JDBC");
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        initToolbar();
        initContent();
        initTable(requete);
    }

    /**
    * Initialise la toolbar
    */
    private void initToolbar() {
        load.setPreferredSize(new Dimension(30, 35));
        load.setBorder(null);
        load.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                initTable(text.getText());
            }
        });

        tool.add(load);
        getContentPane().add(tool, BorderLayout.NORTH);
    }

    /**
    * Initialise le contenu de la fenêtre
    */
    public void initContent() {
        //Vous connaissez ça...
        result.setLayout(new BorderLayout());
        split = new JSplitPane(JSplitPane.VERTICAL_SPLIT, new
        JScrollPane(text), result);
        split.setDividerLocation(100);
        getContentPane().add(split, BorderLayout.CENTER);
    }

    /**

```

```

* Initialise le visuel avec la requête saisie dans l'éditeur
* @param query
*/
public void initTable(String query){

    try {
        //On crée un statement
        long start = System.currentTimeMillis();
        Statement state = SdzConnection.getInstance()
            .createStatement(
                ResultSet.TYPE_SCROLL_INSENSITIVE,
                ResultSet.CONCUR_READ_ONLY
            );

        //On exécute la requête
        ResultSet res = state.executeQuery(query);
        //Temps d'exécution

        //On récupère les meta afin de récupérer le nom des colonnes
        ResultSetMetaData meta = res.getMetaData();
        //On initialise un tableau d'Object pour les en-têtes du
        //tableau
        Object[] column = new Object[meta.getColumnCount()];

        for(int i = 1 ; i <= meta.getColumnCount(); i++){
            column[i-1] = meta.getColumnName(i);
        }

        //Petite manipulation pour obtenir le nombre de lignes
        res.last();
        int rowCount = res.getRow();
        Object[][] data = new
        Object[res.getRow()][meta.getColumnCount()];

        //On revient au départ
        res.beforeFirst();
        int j = 1;

        //On remplit le tableau d'Object[][]
        while(res.next()){
            for(int i = 1 ; i <= meta.getColumnCount(); i++){
                data[j-1][i-1] = res.getObject(i);

                j++;
            }

            //on ferme le tout

            res.close();
            state.close();

            long totalTime = System.currentTimeMillis() - start;

            //On enlève le contenu de notre conteneur
            result.removeAll();
            //On y ajoute un JTable
            result.add(new JScrollPane(new JTable(data, column)),
            BorderLayout.CENTER);
            result.add(new JLabel("La requête à été exécuter en " +
            totalTime + " ms et a retourné " + rowCount + " ligne(s)"),
            BorderLayout.SOUTH);
            //On force la mise à jour de l'affichage
            result.revalidate();

        } catch (SQLException e) {
            //Dans le cas d'une exception, on affiche une pop-up et on
            //efface le contenu
            result.removeAll();
            result.add(new JScrollPane(new JTable()), BorderLayout.CENTER);
            result.revalidate();
            JOptionPane.showMessageDialog(null, e.getMessage(), "ERREUR !

```

```
        ", JOptionPane.ERROR_MESSAGE);
    }

}

/**
 * Point de départ du programme
 * @param args
 */
public static void main(String[] args) {
    Fenetre fen = new Fenetre();
    fen.setVisible(true);
}
}
```

Bien sûr, ce code n'est pas la perfection même, vous pouvez l'améliorer grandement !

Vous pouvez utiliser un autre composant que moi pour la saisie de la requête, un **JTextPane** par exemple : pour la coloration syntaxique... 😊

Vous pourriez avoir un menu qui vous permette de sauvegarder vos requêtes, un tableau interactif autorisant la modification des données...

Bref, ce n'est pas les améliorations qui manquent. 🤔

Un TP assez riche et qui a dû vous demander quelques instants de réflexion... 🐱

Mais bon, rien d'insurmontable pour les ZérOs avertis que vous êtes.

Je vous propose maintenant de voir comment faire en sorte d'utiliser des objets Java correspondant à vos données dans votre BDD !

J'imagine que vous aspirez à faire ceci depuis longtemps... Alors, rendez-vous au prochain chapitre ! 😊

Le pattern DAO (1/2)

Bon, vous voulez pouvoir utiliser vos données dans des objets, et c'est normal ! 😊

Avec le pattern DAO, vous allez voir comment faire ceci mais en plus, comment rendre ça stable !

En fait, je suppose que vous avez dû essayer de faire en sorte que les données de votre base collent à vos objets, avec des méthodes :

- de récupération ;
- de création ;
- de mise à jour ;
- de suppression.

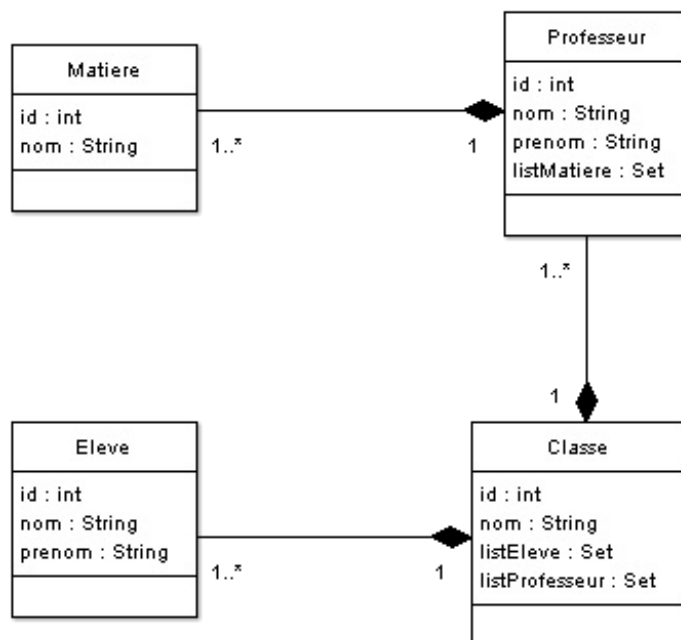
Nous allons voir tout ceci dans ce chapitre et le suivant ! 😊

Avant toute chose

Comme je le disais, vous avez dû essayer de faire en sorte que les données de la base puissent être utilisées via des objets Java. En tout premier lieu, il faut créer une classe par entité (les tables hors tables de jointures...), ce qui nous donnerait les classes suivantes :

- **Eleve** ;
- **Matiere** ;
- **Professeur** ;
- **Classe**.

Et, si nous suivons la logique des relations entre nos tables, nous avons des classes liées suivant le diagramme de classes suivant :



Nous voyons les liens entre les objets avec ce diagramme : une classe est composée de plusieurs élèves et de plusieurs professeurs, et un professeur peut exercer plusieurs matières !

Les tables de jointures de la base sont symbolisées par la composition dans nos objets.

Une fois ceci fait, nous devons coder ces objets avec les accesseurs et les mutateurs adéquats :

- getters et setters pour tous les attributs de toutes les classes ;
- méthode d'ajout et de suppression pour les objets ayant des listes d'objets.

On appelle ce genre d'objet des POJO, pour **Plain Old Java Object** !
Ce qui nous donne ces codes sources :

Secret (cliquez pour afficher)

Classe Eleve.java

Code : Java

```
package com.sdz.bean;

public class Eleve {

    /**
     * ID
     */
    private int id = 0;

    /**
     * Nom de l'élève
     */
    private String nom = "";

    /**
     * Prénom de l'élève
     */
    private String prenom = "";

    //*****
    // CONSTRUCTEUR DE L'OBJET
    //*****

    /**
     * @param id
     * @param nom
     * @param prenom
     */
    public Eleve(int id, String nom, String prenom) {
        this.id = id;
        this.nom = nom;
        this.prenom = prenom;
    }
    public Eleve(){};

    //*****
    // ACCESSEURS ET MUTATEURS
    //*****

    /**
     * @return the id
     */
    public int getId() {
        return id;
    }

    /**
     * @param id the id to set
     */
    public void setId(int id) {
        this.id = id;
    }

    /**
     * @return the nom
     */
    public String getNom() {
```



```

        return nom;
    }

    /**
     * @param nom the nom to set
     */
    public void setNom(String nom) {
        this.nom = nom;
    }

    /**
     * @return the prenom
     */
    public String getPrenom() {
        return prenom;
    }

    /**
     * @param prenom the prenom to set
     */
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
}

```

Classe Matiere.java

Code : Java

```

package com.sdz.bean;

public class Matiere {

    /**
     * ID
     */
    private int id = 0;

    /**
     * Nom du professeur
     */
    private String nom = "";

    /**
     * *****
     * // CONSTRUCTEUR DE L'OBJET
     * *****
     */
    @param id
    @param nom
    public Matiere(int id, String nom) {
        this.id = id;
        this.nom = nom;
    }

    public Matiere() {}

    /**
     * *****
     * // ACCESSEURS ET MUTATEURS
     * *****
     */
    /**
     * @return the id
     */
}

```

```

        public int getId() {
            return id;
        }
        /**
         * @param id the id to set
         */
        public void setId(int id) {
            this.id = id;
        }
        /**
         * @return the nom
         */
        public String getNom() {
            return nom;
        }
        /**
         * @param nom the nom to set
         */
        public void setNom(String nom) {
            this.nom = nom;
        }
    }

```

Classe Professeur.java

Code : Java

```

package com.sdz.bean;

import java.util.HashSet;
import java.util.Set;

public class Professeur {

    /**
     * ID
     */
    private int id = 0;
    /**
     * Nom du professeur
     */
    private String nom = "";
    /**
     * Prénom du professeur
     */
    private String prenom = "";
    /**
     * Liste des matières dispensées
     */
    private Set<Matiere> listMatiere = new HashSet<Matiere>();

    /**
     * @param id
     * @param nom
     * @param prenom
     */
    public Professeur(int id, String nom, String prenom) {
        this.id = id;
        this.nom = nom;
        this.prenom = prenom;
    }

```

```

    }

    public Professeur() {}

    //*****
    // ACCESSEURS ET MUTATEURS
    //*****

    /**
     * @return the id
     */
    public int getId() {
        return id;
    }

    /**
     * @param id the id to set
     */
    public void setId(int id) {
        this.id = id;
    }

    /**
     * @return the nom
     */
    public String getNom() {
        return nom;
    }

    /**
     * @param nom the nom to set
     */
    public void setNom(String nom) {
        this.nom = nom;
    }

    /**
     * @return the prenom
     */
    public String getPrenom() {
        return prenom;
    }

    /**
     * @param prenom the prenom to set
     */
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }

    /**
     * @return the listMatiere
     */
    public Set<Matiere> getListMatiere() {
        return listMatiere;
    }

    /**
     * @param listMatiere the listMatiere to set
     */
    public void setListMatiere(Set<Matiere> listMatiere) {
        this.listMatiere = listMatiere;
    }

    /**
     * Ajoute une matière à la liste des cours dispensés
     * @param Matiere
     */
    public void addMatiere(Matiere matiere){
        this.listMatiere.add(matiere);
    }

    /**
     * Retire une matière de la liste des cours dispensés
     * @param Matiere

```

```

*/
    public void removeMatiere(Matiere matiere) {
        this.listMatiere.remove(matiere);
    }
}

```

Classe Classe.java

Code : Java

```

package com.sdz.bean;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class Classe {

    /**
     * ID
     */
    private int id = 0;

    /**
     * Nom du professeur
     */
    private String nom = "";

    /**
     * Liste des professeurs
     */
    private Set<Professeur> listProfesseur = new
    HashSet<Professeur>();

    /**
     * Liste des élèves
     */
    private Set<Eleve> listEleve = new HashSet<Eleve>();

    /**
     * @param id
     * @param nom
     */
    public Classe(int id, String nom) {
        this.id = id;
        this.nom = nom;
    }

    public Classe() {}

    /**
     * @return the id
     */
    public int getId() {
        return id;
    }
}

```

```
/**
 * @param id the id to set
 */
public void setId(int id) {
    this.id = id;
}

/**
 * @return the nom
 */
public String getNom() {
    return nom;
}

/**
 * @param nom the nom to set
 */
public void setNom(String nom) {
    this.nom = nom;
}

/**
 * @return the listMatiere
 */
public Set<Professeur> getListProfesseur() {
    return listProfesseur;
}

/**
 * @param listMatiere the listMatiere to set
 */
public void setListProfesseur(Set<Professeur> listProfesseur) {
    this.listProfesseur = listProfesseur;
}

/**
 * @return the listMatiere
 */
public void addProfesseur(Professeur prof) {
    if(!listProfesseur.contains(prof))
        listProfesseur.add(prof);
}

/**
 * @param listMatiere the listMatiere to set
 */
public void removeProfesseur(Professeur prof ) {
    this.listProfesseur.remove(prof);
}

/**
 * @return the listMatiere
 */
public Set<Eleve> getListEleve() {
    return listEleve;
}

/**
 * @param listMatiere the listMatiere to set
 */
public void setListEleve(Set<Eleve> listEleve) {
    this.listEleve = listEleve;
}

/**
 * Ajoute un élève à la classe
 * @param eleve
 */
public void addEleve(Eleve eleve){
    if(!this.listEleve.contains(eleve))
```

```

        this.listEleve.add(eleve);
    }

    /**
     * Retire un élève de la classe
     * @param eleve
     */
    public void removeEleve(Eleve eleve) {
        this.listEleve.remove(eleve);
    }

    /**
     * Méthode equals
     * @param cls
     * @return
     */
    public boolean equals(Classe cls) {
        return this.getId() == cls.getId();
    }
}

```

Voilà, vous avez vos objets tout beaux tout propres !

Nous avons des objets prêts à l'emploi. Maintenant, comment faire en sorte que ces objets puissent recevoir les données de notre base ?

Au lieu de faire des essais à tâtons, nous allons définir le pattern DAO et voir comment il fonctionne avant de l'implémenter.

Le pattern DAO : définition

Contexte


Vous avez des données sérialisées dans une base de données et vous souhaitez y accéder et les manipuler avec des objets Java. Cependant, votre entreprise est en pleine restructuration et vous ne savez pas si vos données vont :

- rester où elles sont ;
- migrer sur une autre base de données ;
- être stockées dans des fichiers XML ;
- ...

Comment faire en sorte de ne pas avoir à modifier toutes les utilisations de nos objets ?

Comment faire un système adaptatif aux futures modifications de supports de données ?

Comment faire en sorte que les objets que nous allons utiliser restent tels qu'ils sont ?

Il pourrait y avoir beaucoup de problématiques de ce genre, mais le pattern DAO est là pour vous ! 

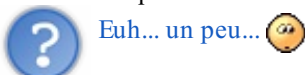
La définition d'un design pattern est toujours un peu pompeuse et mystifiée ; par contre, et ceux qui ont lu les chapitres concernant les design patterns du tuto Java le confirmeront, ils apportent plus de souplesse et de robustesse à vos programmes !

Le pattern DAO

Ce pattern permet de faire le lien entre la couche d'accès aux données et la couche métier d'une application. Il permet de mieux maîtriser les changements susceptibles d'être opérés sur le système de stockage des données, donc, par extension, de préparer une migration d'un système à un autre (BDD vers fichiers XML par exemple...).

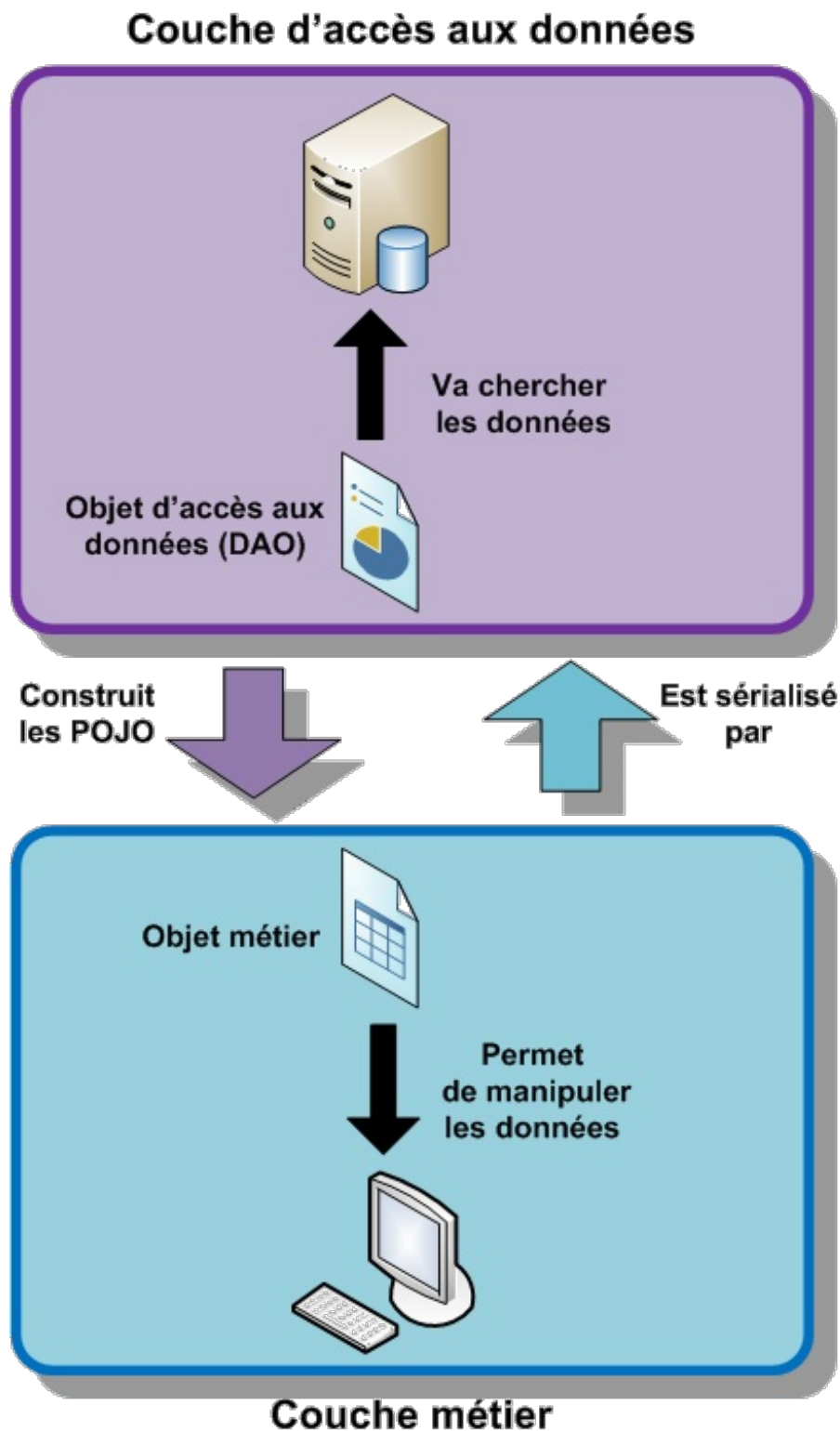
Ceci se fait en séparant accès aux données (BDD) et objets métiers (POJO).

Je me doute que tout ceci doit vous sembler très flou !



C'est normal, mais ne vous en faites pas, je vais tout vous expliquer...

Déjà, il y a une histoire de séparation des "*couches métiers et couches d'accès aux données*". Il s'agit ni plus ni moins de faire en sorte qu'un type d'objet se charge de récupérer les données dans la base et qu'un autre type d'objet (souvent des POJO) soit utilisé pour manipuler ces données. Schématiquement, ça nous donne :



Si on comprend bien, nous allons avoir deux types d'objets !

Tout à fait.

Les objets que nous avons créés plus haut sont nos POJO, les objets utilisés par le programme pour manipuler les données de la base.



Il ne nous reste plus qu'à créer les objets qui vont rechercher les données dans la base !



Oui, mais nous n'allons pas faire n'importe comment...

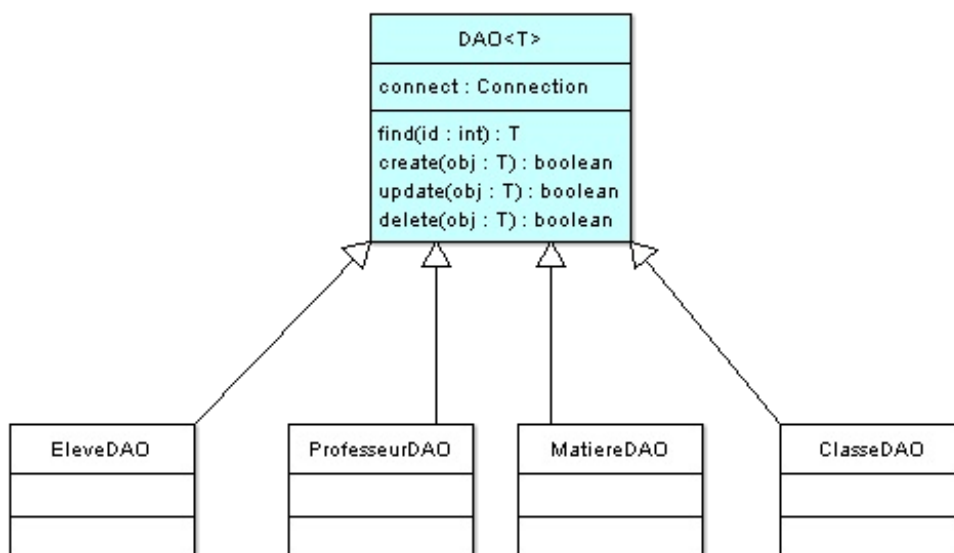
Les dits objets devront être capables de faire des recherches, des insertions, des mises à jour et des suppressions ! Par conséquent, nous pouvons définir un super type d'objet afin d'utiliser au mieux le polymorphisme...

Nous allons devoir créer une classe abstraite (ou une interface) mettant en oeuvre toutes les méthodes sus-mentionnées !



Comment va-t-on faire pour demander à nos objets DAO de récupérer tel type d'objet ou d'en sérialiser tel autre ? Avec des cast ?

Soit en castant, soit en faisant une classe générique ! Comme ceci :



Ah... mais oui !

Vous aviez oublié que nous pouvions créer des classes génériques ? 😞

Bon, je vous pardonne. Maintenant, voyons un peu les codes sources de ces objets :

Secret (cliquez pour afficher)

Classe DAO.java

Code : Java

```

package com.sdz.dao;

import java.sql.Connection;

import com.sdz.connection.SdzConnection;

public abstract class DAO<T> {

    protected Connection connect = null;

    /**
     * Constructeur
     * @param conn
     */
    public DAO(Connection conn) {
        this.connect = conn;
    }
  
```



```

        /**
        * Méthode de création
        * @param obj
        * @return
        */
        public abstract boolean create(T obj);

        /**
        * Méthode pour effacer
        * @param obj
        * @return
        */
        public abstract boolean delete(T obj);

        /**
        * Méthode de mise à jour
        * @param obj
        * @return
        */
        public abstract boolean update(T obj);

        /**
        * Méthode de recherche des informations
        * @param id
        * @return
        */
        public abstract T find(int id);
    }

```

Classe EleveDAO.java

Code : Java

```

package com.sdz.dao.implement;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;

import com.sdz.bean.Eleve;
import com.sdz.dao.DAO;

public class EleveDAO extends DAO<Eleve> {

    public EleveDAO(Connection conn) {
        super(conn);
    }

    public boolean create(Eleve obj) {
        return false;
    }

    public boolean delete(Eleve obj) {
        return false;
    }

    public Eleve find(int id) {

        Eleve eleve = new Eleve();

        try {
            ResultSet result = this.connect
                .createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY
                ).executeQuery(

```

```

        "SELECT * FROM eleve WHERE elv_id = " + id
    );

        if(result.first())
            eleve = new Eleve(id,
result.getString("elv_nom"), result.getString("elv_prenom"));

        } catch (SQLException e) {
            e.printStackTrace();
        }
        return eleve;
    }

    public boolean update(Eleve obj) {
        return false;
    }
}

```

Classe MatiereDAO.java

Code : Java

```

package com.sdz.dao.implement;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;

import com.sdz.bean.Eleve;
import com.sdz.bean.Matiere;
import com.sdz.dao.DAO;

public class MatiereDAO extends DAO<Matiere> {

    public MatiereDAO(Connection conn) {
        super(conn);
    }

    public boolean create(Matiere obj) {

        return false;
    }

    public boolean delete(Matiere obj) {

        return false;
    }

    public Matiere find(int id) {

        Matiere matiere = new Matiere();

        try {
            ResultSet result = this.connect
                .createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY
                ).executeQuery(

```

```

        "SELECT * FROM matiere WHERE mat_id = " + id
    );

        if(result.first())
            matiere = new Matiere(id,
result.getString("mat_nom"));

        } catch (SQLException e) {
            e.printStackTrace();
        }
        return matiere;
    }

    public boolean update(Matiere obj) {

        return false;
    }

}

```

Classe ProfesseurDAO.java

Code : Java

```

package com.sdz.dao.implement;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;

import com.sdz.bean.Eleve;
import com.sdz.bean.Professeur;
import com.sdz.dao.DAO;

public class ProfesseurDAO extends DAO<Professeur> {

    public ProfesseurDAO(Connection conn) {
        super(conn);
    }

    public boolean create(Professeur obj) {

        return false;
    }

    public boolean delete(Professeur obj) {

        return false;
    }

    public Professeur find(int id) {

        Professeur professeur = new Professeur();

        try {
            ResultSet result = this.connect
.createStatement(
ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY

```

```

).executeQuery(

"select * from professeur "+

"left join j_mat_prof on jmp_prof_k = prof_id AND prof_id = "+ id
+

" inner join matiere on jmp_mat_k = mat_id"

);

        if(result.first()){
            professeur = new Professeur(id,
result.getString("prof_nom"), result.getString("prof_prenom"));
            result.beforeFirst();
            MatiereDAO matDao = new

MatiereDAO(this.connect);

            while(result.next())

professeur.addMatiere(matDao.find(result.getInt("mat_id")));
        }
        catch (SQLException e) {
            e.printStackTrace();
        }
        return professeur;
    }

    public boolean update(Professeur obj) {

        return false;
    }
}

```

Classe ClasseDAO.java

Code : Java

```

package com.sdz.dao.implement;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;

import com.sdz.bean.Classe;
import com.sdz.bean.Eleve;
import com.sdz.dao.DAO;

public class ClasseDAO extends DAO<Classe> {

    public ClasseDAO(Connection conn) {
        super(conn);
    }

    public boolean create(Classe obj) {

        return false;
    }

    public boolean delete(Classe obj) {

        return false;
    }
}

```

```

        public Classe find(int id) {
            Classe classe = new Classe();

            try {
                ResultSet result = this.connect
                .createStatement(
                    ResultSet.TYPE_SCROLL_INSENSITIVE,
                    ResultSet.CONCUR_READ_ONLY
                ).executeQuery(
                    "select * from classe WHERE cls_id = " + id
                );

                if(result.first()){
                    classe = new Classe(id,
                    result.getString("cls_nom"));

                    result = this.connect
                    .createStatement()
                    .executeQuery(
                        "select prof_id, prof_nom, prof_prenom from professeur " +
                        "INNER JOIN j_mat_prof on prof_id = jmp_prof_k " +
                        "INNER JOIN j_cls_jmp on jmp_id = jcm_jmp_k AND jcm_cls_k = " + id
                    );

                    ProfesseurDAO profDao = new
                    ProfesseurDAO(this.connect);

                    while(result.next())
                        classe.addProfesseur(profDao.find(result.getInt("prof_id")));

                    EleveDAO eleveDao = new
                    EleveDAO(this.connect);

                    result = this.connect
                    .createStatement()
                    .executeQuery(
                        "select elv_id, elv_nom, elv_prenom from eleve " +
                        "INNER JOIN classe on elv_cls_k = cls_id AND cls_id = " + id
                    );

                    while(result.next())
                        classe.addEleve(eleveDao.find(result.getInt("elv_id")));
                }

                catch (SQLException e) {
                    e.printStackTrace();
                }
                return classe;
            }

        public boolean update(Classe obj) {

```

```

        return false;
    }
}

```



Pour ne pas compliquer la tâche, je n'ai détaillé que la méthode de recherche des données, les autres sont des coquilles vides...
Mais vous devriez être capables de faire ça tout seuls, normalement...

Premier test

Bon : nous avons réalisé une bonne partie de ce pattem, nous allons pouvoir faire notre premier test. 😊



Une bonne partie ? Tu veux dire qu'il y a encore des choses à faire ?

Oui, nous verrons ça dans le prochain chapitre : pour le moment, testons ce que nous avons.



Par contre, je tiens à préciser que j'utilise toujours le singleton créé quelques chapitres plus haut !

Voici un code de test :

Code : Java

```

import com.sdz.bean.Classe;
import com.sdz.bean.Eleve;
import com.sdz.bean.Matiere;
import com.sdz.bean.Professeur;
import com.sdz.connection.SdzConnection;
import com.sdz.dao.DAO;
import com.sdz.dao.implement.ClasseDAO;
import com.sdz.dao.implement.EleveDAO;
import com.sdz.dao.implement.ProfesseurDAO;

public class FirstTest {

    public static void main(String[] args) {

        //testons des élèves
        DAO<Eleve> eleveDao = new
EleveDAO(SdzConnection.getInstance());
        for(int i = 1; i < 5; i++){
            Eleve eleve = eleveDao.find(i);
            System.out.println("Elève N°" + eleve.getId()
+ " - " + eleve.getNom() + " " + eleve.getPrenom());
        }

        System.out.println("\n*****\n");

        //Voyons voir les professeurs
        DAO<Professeur> profDao = new
ProfesseurDAO(SdzConnection.getInstance());
        for(int i = 4; i < 8; i++){
            Professeur prof = profDao.find(i);
            System.out.println(prof.getNom() + " " +

```

```

    prof.getPrenom() + " enseigne : ");
        for (Matiere mat : prof.getListMatiere())
            System.out.println("\t * " +
mat.getNom());
    }

System.out.println("\n*****\n");

    //Et là, c'est la classe
    DAO<Classe> classeDao = new
ClasseDAO(SdzConnection.getInstance());
    Classe classe = classeDao.find(11);

    System.out.println("Classe de " + classe.getNom());
    System.out.println("\nListe des élèves :");
    for (Eleve eleve : classe.getListEleve())
        System.out.println(" - " + eleve.getNom() + "
" + eleve.getPrenom());

    System.out.println("\nListe des professeurs :");
    for (Professeur prof : classe.getListProfesseur())
        System.out.println(" - " + prof.getNom() + "
" + prof.getPrenom());
    }
}

```

Qui me donne :

```
Elève N°1 - HERBY Cyrille
Elève N°2 - COURTEL Angelo
Elève N°3 - PITON Thomas
Elève N°4 - COQUILLE Olivier

*****

BADEN Baden enseigne :
    * Français
    * Sport
MIOU Miou enseigne :
    * Anglais
BORA Kernel enseigne :
    * Anglais
CAISSE Jean enseigne :
    * Physique

*****

Classe de 3° B

Liste des élèves :
    - MONIN Gérald
    - NAEMI Toufic
    - DROUIN Albert

Liste des professeurs :
    - SACRE Sophie
    - MAMOU Cyrille
    - MOISSAT Marc
    - MIOU Miou
```

Vous avez compris comment tout ça fonctionnait ? Ce n'est pas très dur en fin de compte. Je vous laisse quelques instants pour lire, tester, relire, tester à nouveau...

Nous utilisons des objets spécifiques afin de rechercher dans la base des données qui nous servent à instancier des objets Java habituels. 😊

Avant de poursuivre, nous allons faire un tour du côté du QCM...

Bon, vous voyez un peu comment on va procéder.

Le truc, c'est que vous savez comment récupérer les données de votre base pour les utiliser dans des POJO, mais nous n'avons pas vu comment permettre l'utilisation de plusieurs systèmes de base de données, voire même de l'XML.

Ne prenez pas peur, le résultat sera à la hauteur de vos attentes... Même si ne connaissez rien à Java ni à XML (pour l'instant...).



Le pattern DAO (2/2)

Dans ce chapitre, nous allons voir comment peaufiner le pattern DAO afin que celui-ci puisse s'adapter à d'autres systèmes de sauvegarde...

Le but du jeu étant d'avoir le moins possible de changements le cas échéant !

Le pattern factory

Nous allons aborder ici une notion importante : la fabrication d'objets !

En effet, le pattern DAO implémente aussi ce qu'on appelle **le pattern factory**.

Celui-ci consiste à déléguer l'instanciation d'objets à une classe.

En fait, une fabrique ne fait que ça ! 😊

En général, lorsque vous voyez ce genre de code dans une classe :

Code : Java

```
class A{
    public Object getData(int type) {
        Object obj;

        //-----
        if(type == 0)
            obj = new B();
        else if(type == 1)
            obj = new C();
        else
            obj = new D();
        //-----
        obj.doSomething();
        obj.doSomethingElse();
    }
}
```

vous constatez qu'il y a une création d'objet et que ceci est conditionné par une variable. En fait, selon celle-ci, l'objet instancié n'est pas le même. Nous allons donc extraire ce code (celui entre commentaires) pour le mettre dans une classe à part :

Code : Java

```
package com.sdz.transact;

public class Factory {

    public static Object getData(int type) {
        if(type == 0)
            return new B();
        else if(type == 1)
            return new C();
        else
            return new D();
    }
}
```

Du coup, maintenant, lorsque nous voudrions instancier les objets de la fabrique, nous utiliserons celle-ci. Comme ça :

Code : Java

```
B b = Factory.getData(0);
C c = Factory.getData(1);
//...
```



Pourquoi faire tout ce ramdam ? Quel est le problème avec nos instances ?

C'est simple : en temps normal, nous travaillons avec des objets concrets, non soumis au changement. Cependant, dans le cas qui nous intéresse, nos objets peuvent être amenés à changer et j'irais même plus loin : le type d'objet utilisé peut changer !

L'avantage d'utiliser une fabrique, c'est que les instances concrètes (utilisation du mot clé **new**) se fait à **UN SEUL ENDROIT !**

Donc, si nous devons faire des changements, il ne se feront qu'à un seul endroit ! Si nous ajoutons un paramètre dans le constructeur, par exemple...



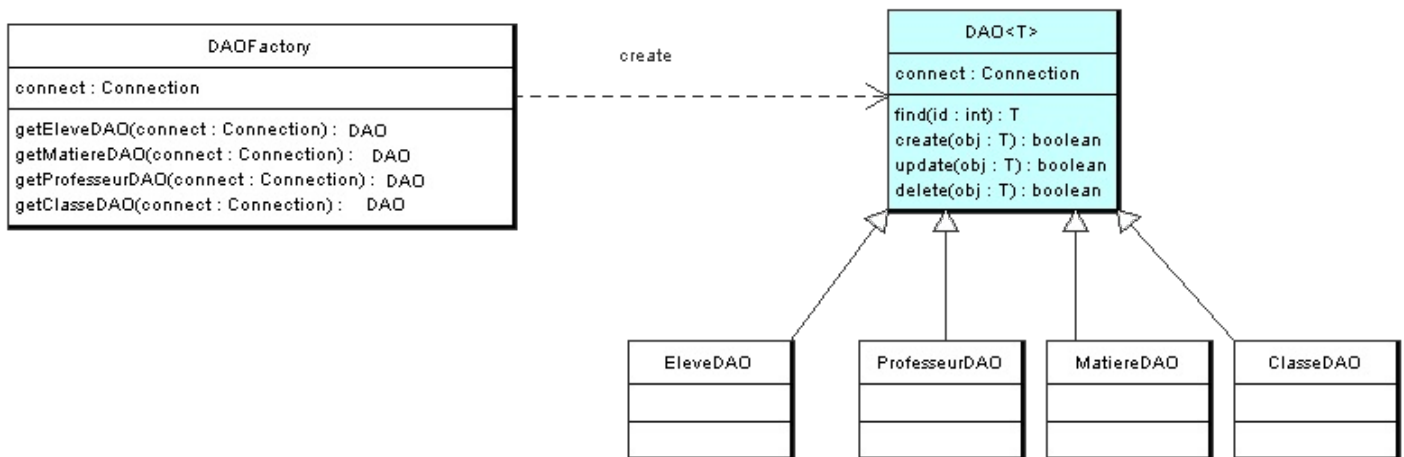
D'accord, on commence à comprendre l'intérêt de cette fabrique.

Je savais que vous comprendriez vite. je vous propose maintenant de voir comment ce pattern est implémenté dans le pattern DAO.

Fabriquer vos DAO

En fait, la factory dans le pattern DAO sert à construire nos instances d'objets d'accès aux données.

Du coup, vu que nous avons un super-type d'objet pour ces objets, nous savons quel type d'objet va retourner notre fabrique.



Voici le code de notre fabrique :

Code : Java

```

package com.sdz.dao;

import java.sql.Connection;

import com.sdz.connection.SdzConnection;
import com.sdz.dao.implement.ClasseDAO;
import com.sdz.dao.implement.EleveDAO;
import com.sdz.dao.implement.MatiereDAO;
import com.sdz.dao.implement.ProfesseurDAO;

public class DAOFactory {

    protected static final Connection conn =
SdzConnection.getInstance();

    /**
     * Retourne un objet Classe interagissant avec la BDD
     * @return
     */
    public static DAO getClasseDAO() {
        return new ClasseDAO(conn);
    }
}
  
```

```

        /**
        * Retourne un objet Professeur interagissant avec la BDD
        * @return
        */
        public static DAO getProfesseurDAO() {
            return new ProfesseurDAO(conn);
        }
        /**
        * Retourne un objet Eleve interagissant avec la BDD
        * @return
        */
        public static DAO getEleveDAO() {
            return new EleveDAO(conn);
        }
        /**
        * Retourne un objet Matiere interagissant avec la BDD
        * @return
        */
        public static DAO getMatiereDAO() {
            return new MatiereDAO(conn);
        }
    }

```

Et voici un code qui devrait vous plaire :

Code : Java

```

import com.sdz.bean.Classe;
import com.sdz.bean.Eleve;
import com.sdz.bean.Matiere;
import com.sdz.bean.Professeur;
import com.sdz.dao.DAO;
import com.sdz.dao.DAOFactory;

public class TestDAO {

    /**
    * @param args
    */
    public static void main(String[] args) {
        System.out.println("");

        //-----
        //On va rechercher des élèves
        //-----

        //On récupère un objet faisant le lien entre la base et
        nos objets
        DAO<Eleve> eleveDao = DAOFactory.getEleveDAO();

        for(int i = 1; i < 5; i++){
            //On fait notre recherche
            Eleve eleve = eleveDao.find(i);
            System.out.println("\tELEVE N°" + eleve.getId() + "
- NOM : " + eleve.getNom() + " - PRENOM : " + eleve.getPrenom());
        }

        System.out.println("\n\t*****");

        //On fait de même pour une classe
        DAO<Classe> classeDao = DAOFactory.getClasseDAO();
        //On cherche la classe ayant pour ID 10
        Classe classe = classeDao.find(10);
    }
}

```

```

        System.out.println("\tCLASSE DE " + classe.getNom());

        //On récupère la liste des élèves
        System.out.println("\n\tCelle-ci contient " +
classe.getListEleve().size() + " élève(s)");
        for(Eleve eleve : classe.getListEleve())
            System.out.println("\t\t - " + eleve.getNom() + " "
+ eleve.getPrenom());

        //Ainsi que la liste des professeurs
        System.out.println("\n\tCelle-ci a " +
classe.getListProfesseur().size() + " professeur(s)");
        for(Professeur prof : classe.getListProfesseur()){
            System.out.println("\t\t - Mr " + prof.getNom() + "
" + prof.getPrenom() + " professeur de :");

            //Tant qu'à faire, on prend aussi les matières...
            ^^
            for(Matiere mat : prof.getListMatiere())
                System.out.println("\t\t\t * " +
mat.getNom());

        }

        System.out.println("\n\t*****");

        //Un petit essai sur les matières
        DAO<Matiere> matiereDao = DAOFactory.getMatiereDAO();
        Matiere mat = matiereDao.find(2);
        System.out.println("\tMATIERE " + mat.getId() + " : " +
mat.getNom());
    }
}

```

Et voilà le résultat que nous donne ce code :

```

ELEVE N°1 - NOM : HERBY - PRENOM : Cyrille
ELEVE N°2 - NOM : COURTEL - PRENOM : Angelo
ELEVE N°3 - NOM : PITON - PRENOM : Thomas
ELEVE N°4 - NOM : COQUILLE - PRENOM : Olivier

*****
CLASSE DE 3° A

Celle-ci contient 3 élève(s)
- TARTUFE Thérèse
- FERNAT Fernand
- JOUBERT Aline

Celle-ci a 4 professeur(s)
- Mr MOISSAT Marc professeur de :
  * Physique
- Mr BORA Kernel professeur de :
  * Anglais
- Mr BADEN Baden professeur de :
  * Sport
  * Français
- Mr MIOU Miou professeur de :
  * Anglais

*****
MATIERE 2 : Français

```

Vous pouvez être fiers de vous ! Vous venez d'implémenter le pattern DAO utilisant une fabrique. C'était un peu effrayant, mais, au final ce n'est rien du tout... 😊



On a bien compris le principe du pattern DAO, même la combinaison DAO - factory. Cependant, on ne voit pas comment gérer plusieurs systèmes de sauvegarde de données ? Il faut modifier les DAO à chaque fois ?

Non, bien sûr... Le fait est que vous pouvez très bien avoir un type de DAO pour chaque type de gestion de données (PostgreSQL, XML, MySQL...). Le vrai problème, c'est de savoir comment récupérer les DAO puisque nous avons délégué leurs instantiations à une fabrique.

Vous allez voir, les choses les plus compliquées peuvent être aussi les plus simples. 😊

D'une usine à une multinationale

Faisons le topo de ce que nous avons :

- des objets métiers ;
- une implémentation d'accès aux données ;
- une classe permettant d'instancier les objets d'accès aux données.

Le fait est que notre structure actuelle fonctionne pour notre système actuel... Ah ! Mais ! Qu'entends-je, qu'ouïe-je ? **Votre patron vient de trancher ! Vous allez utiliser PostgreSQL et du XML !**



C'est bien ce qu'on disait plus haut... Comment gérer ça ? On ne va pas mettre des `if(){...}else{}` dans la fabrique, tout de même ?

Ah ! Je vous arrête ! Vous entendez ce que vous dites :

Citation : Les ZérOs

On ne va pas mettre des `if{...}else{...}` dans la fabrique, tout de même ?

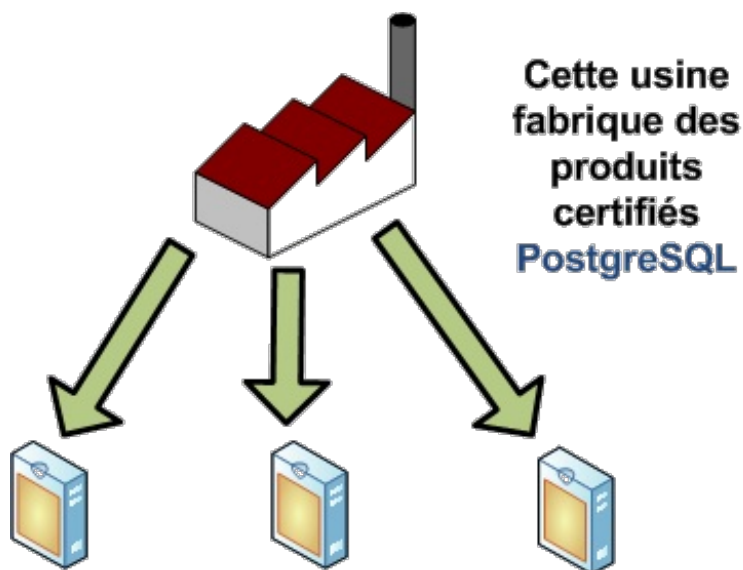
Vous voulez mettre des conditions afin de savoir quel type d'instance retourner : **ça ressemble grandement à une portion de code pouvant être déclinée en fabrique !**



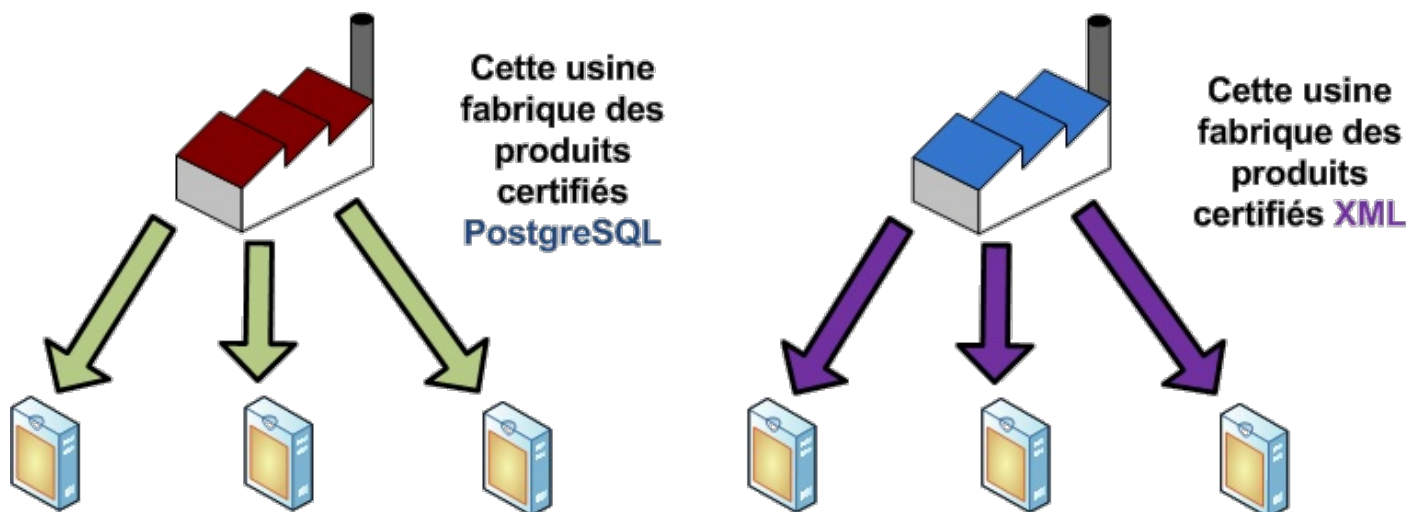
Tu veux faire des fabriques de fabriques ??

Oui ! Notre fabrique actuelle nous permet de construire des objets accédant à des données se trouvant sur une base de données PostgreSQL. Mais la problématique maintenant est de pouvoir aussi utiliser des données provenant de fichiers XML...

Voici un petit schéma représentant la situation actuelle :



Et voilà ce à quoi on aspire :



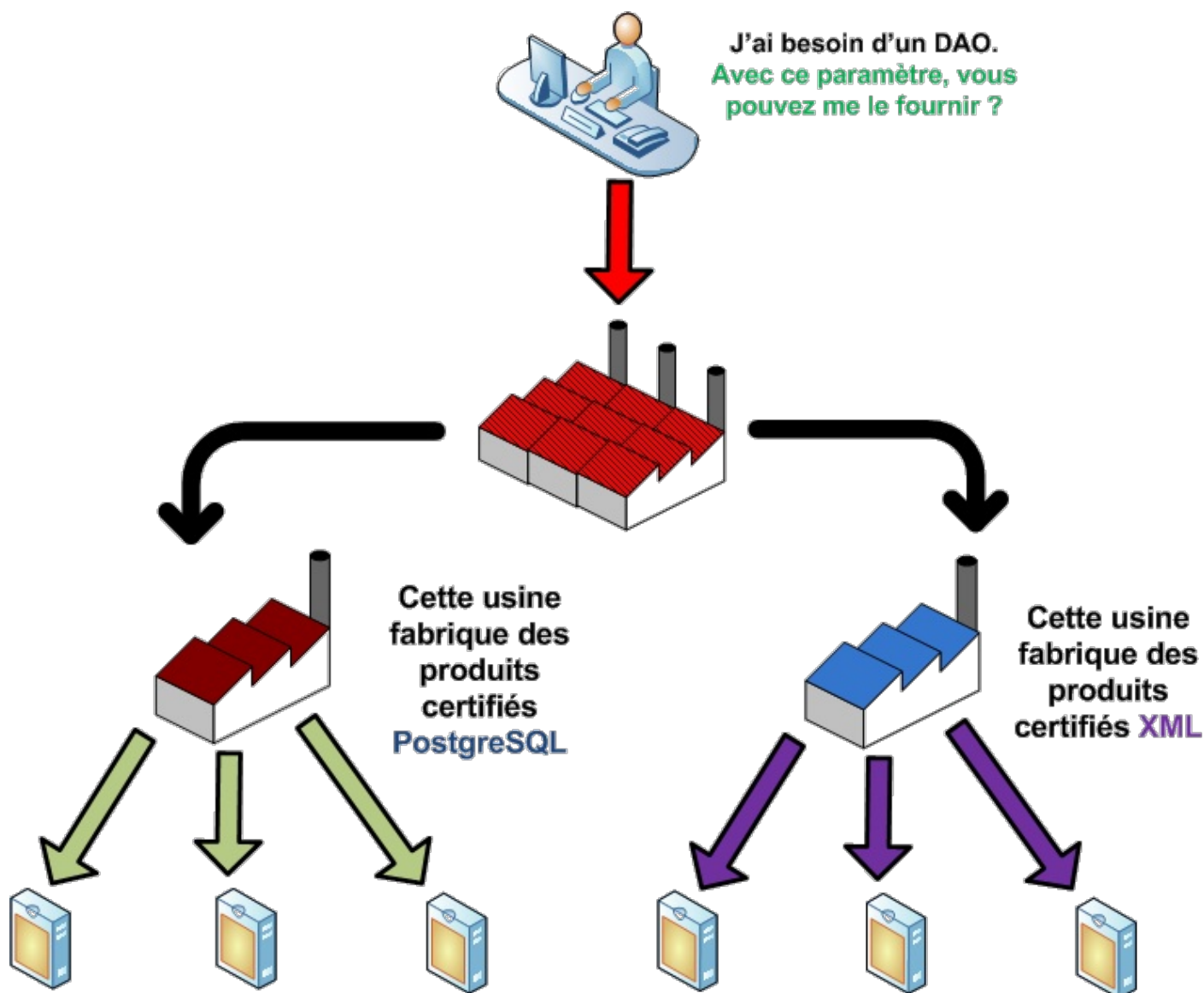
Je pense que vous êtes tous d'accord pour dire que ces deux usines ont un processus de fabrication très similaire.

Par là, j'entends que nous allons utiliser les mêmes méthodes sur les objets sortant de ces deux usines.

Voyez ça un peu comme une grande marque de pain qui aurait beaucoup de boulangeries dans tous les pays du monde ! Cette firme a un savoir-faire évident, mais aussi des particularités : le pain ne se fait pas pareil dans tous les endroits du globe...

Pour vous, c'est comme si vous passiez commande directement au siège social qui, lui, va déléguer à l'usine qui permet de répondre à vos attentes !

Schématiquement, ça donne ceci :

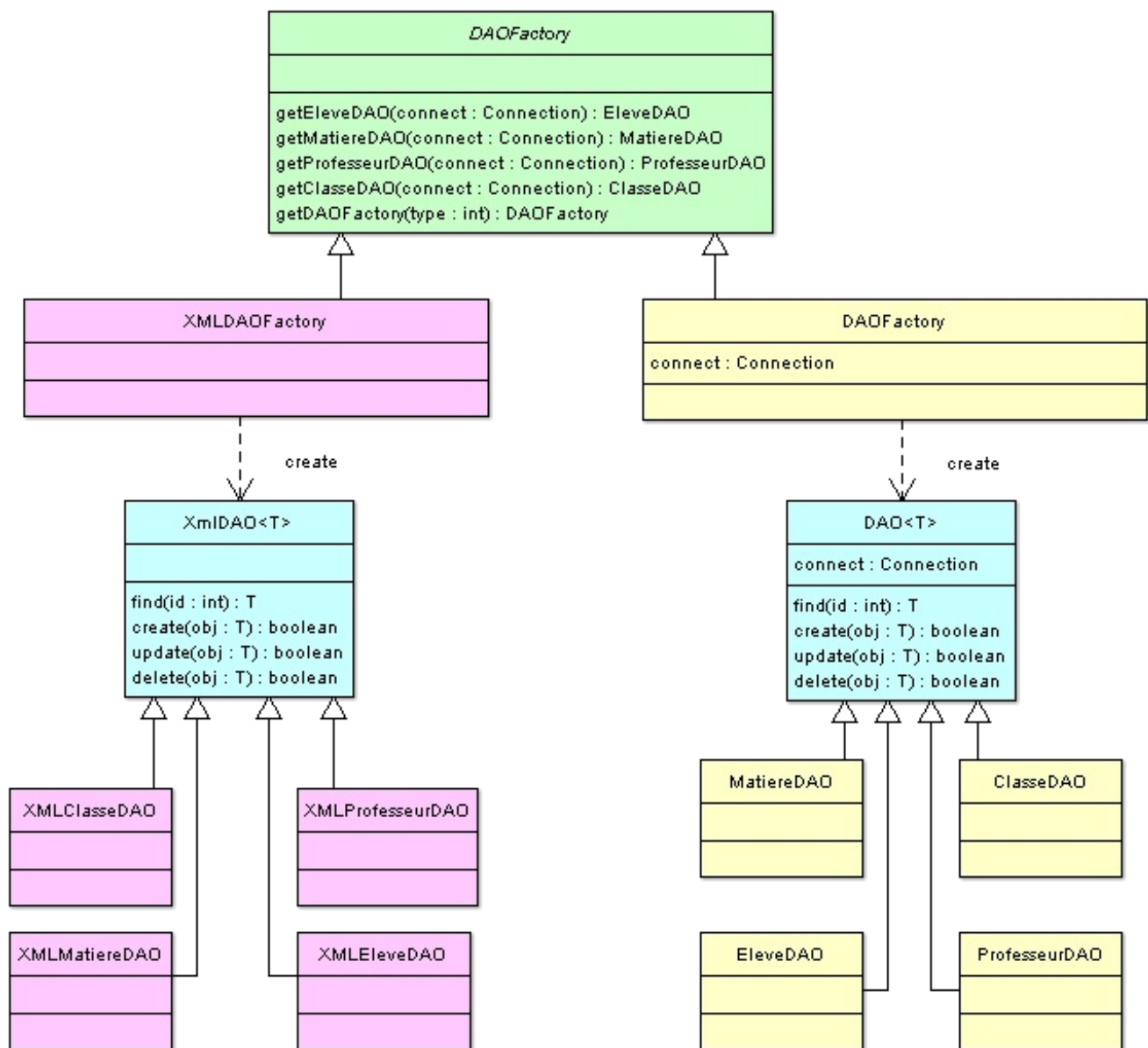


Lorsque je vous dis ça, vous devez avoir une réaction quasi-immédiate : **héritage - polymorphisme !**

Ce qui va changer le plus, par rapport à notre ancienne fabrique, c'est que nous n'utiliserons plus de méthodes statiques, mais des méthodes d'une instance concrète, et pour cause : **impossible de créer une classe abstraite ou une interface avec des méthodes statiques destinée à la redéfinition !**

Donc, nous allons créer une classe abstraite pour nos futurs fabriques, celle-ci devra avoir les méthodes permettant de récupérer les différents DAO **ET une méthode permettant d'instancier la bonne fabrique !**

Je vous ai préparé un diagramme de classe, vous verrez mieux comme ça :



Je vous ai même préparé les codes sources :

Secret ([cliquez pour afficher](#))

Classe AbstractDAOFactory.java :

Code : Java

```

package com.sdz.dao;

public abstract class AbstractDAOFactory {

    public static final int DAO_FACTORY = 0;
    public static final int XML_DAO_FACTORY = 1;

    /**
     * Retourne un objet Classe interagissant avec la BDD
     * @return
     */
    public abstract DAO getClasseDAO();

    /**
     * Retourne un objet Professeur interagissant avec la BDD
     * @return
     */
}
  
```



```

    public abstract DAO getProfesseurDAO();
    /**
     * Retourne un objet Eleve interagissant avec la BDD
     * @return
     */
    public abstract DAO getEleveDAO();
    /**
     * Retourne un objet Matiere interagissant avec la BDD
     * @return
     */
    public abstract DAO getMatiereDAO();

    /**
     * Méthode permettant de récupérer les Factory
     * @param type
     * @return AbstractDAOFactory
     */
    public static AbstractDAOFactory getFactory(int type) {
        switch (type) {
            case DAO_FACTORY:
                return new DAOFactory();
            case XML_DAO_FACTORY:
                return new XMLDAOFactory();
            default:
                return null;
        }
    }
}

```

Classe DAOFactory.java

Code : Java

```

package com.sdz.dao;

import java.sql.Connection;

import com.sdz.connection.SdzConnection;
import com.sdz.dao.implement.ClasseDAO;
import com.sdz.dao.implement.EleveDAO;
import com.sdz.dao.implement.MatiereDAO;
import com.sdz.dao.implement.ProfesseurDAO;

public class DAOFactory extends AbstractDAOFactory {

    protected static final Connection conn =
        SdzConnection.getInstance();

    public DAO getClasseDAO() {
        return new ClasseDAO(conn);
    }

    public DAO getProfesseurDAO() {
        return new ProfesseurDAO(conn);
    }

    public DAO getEleveDAO() {
        return new EleveDAO(conn);
    }

    public DAO getMatiereDAO() {
        return new MatiereDAO(conn);
    }
}

```

Classe XMLDAOFactory.java

Code : Java

```

package com.sdz.dao;

public class XMLDAOFactory extends AbstractDAOFactory {

    public DAO getClasseDAO() {
        return null;
    }

    public DAO getEleveDAO() {
        return null;
    }

    public DAO getMatiereDAO() {
        return null;
    }

    public DAO getProfesseurDAO() {
        return null;
    }
}

```

Vous devez y voir plus clair ; même si la classe **XMLDAOFactory** ne fait rien du tout, vous voyez le principe de base et c'est l'important !

Nous avons maintenant une hiérarchie de classes capables de travailler ensemble. 🧙

Je reprends le dernier exemple que nous avons réalisé, avec un peu de modifications...

Code : Java

```

import com.sdz.bean.Classe;
import com.sdz.bean.Eleve;
import com.sdz.bean.Matiere;
import com.sdz.bean.Professeur;
import com.sdz.dao.AbstractDAOFactory;
import com.sdz.dao.DAO;
import com.sdz.dao.DAOFactory;

public class TestDAO {

    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("");

        //-----
        //On va rechercher des élèves
        //-----
        AbstractDAOFactory adf =
        AbstractDAOFactory.getFactory(AbstractDAOFactory.DAO_FACTORY);
        //On récupère un objet faisant le lien entre la base et nos objets
        DAO<Eleve> eleveDao = adf.getEleveDAO();
    }
}

```

```

    for(int i = 1; i < 5; i++){
        //On fait notre recherche
        Eleve eleve = eleveDao.find(i);
        System.out.println("\tELEVE N°" + eleve.getId() + " - NOM : " +
        eleve.getNom() + " - PRENOM : " + eleve.getPrenom());
    }

    System.out.println("\n\t*****");

    //On fait de même pour une classe
    DAO<Classe> classeDao = adf.getClasseDAO();
    //On cherche la classe ayant pour ID 10
    Classe classe = classeDao.find(10);

    System.out.println("\tCLASSE DE " + classe.getNom());

    //On récupère la liste des élèves
    System.out.println("\n\tCelle-ci contient " +
    classe.getListEleve().size() + " élève(s)");
    for(Eleve eleve : classe.getListEleve()){
        System.out.println("\t\t - " + eleve.getNom() + " " +
        eleve.getPrenom());
    }

    //Ainsi que la liste des professeurs
    System.out.println("\n\tCelle-ci a " + classe.getListProfesseur().size()
    + " professeur(s)");
    for(Professeur prof : classe.getListProfesseur()){
        System.out.println("\t\t - Mr " + prof.getNom() + " " + prof.getPrenom()
        + " professeur de :");
    }

    //Tant qu'à faire, on prend aussi les matières... ^^
    for(Matiere mat : prof.getListMatiere()){
        System.out.println("\t\t\t * " + mat.getNom());
    }

    System.out.println("\n\t*****");

    //Un petit essai sur les matières
    DAO<Matiere> matiereDao = adf.getMatiereDAO();
    Matiere mat = matiereDao.find(2);
    System.out.println("\tMATIERE " + mat.getId() + " : " + mat.getNom());
}
}

```

Et le résultat est le même qu'avant ! Tout fonctionne à merveille !

Ainsi, si vous voulez utiliser l'usine de fabrication XML, vous pouvez faire ceci :

```

AbstractDAOFactory adf =
AbstractDAOFactory.getFactory(AbstractDAOFactory.XML_DAO_FACTORY);

```

Voilà, vous en savez plus sur ce pattern de conception et vous devriez être à même de coder le reste des méthodes (insertions, mise à jour et suppression), il n'y a rien de compliqué : ce sont juste des requêtes SQL... 😊

Allez, et si nous allions faire un tour vers notre QCM ?

Bon, je vous l'accorde, c'est plus difficile qu'il n'y paraît, mais ce n'était pas insurmontable...

Vous avez appris à utiliser un nouvel outil de conception pour structurer vos programmes... Utilisez-le avec sagesse ! 😊

Une partie somme toute assez simple et pratique !

Je pense que vous devez apprécier tout ceci et que l'utilisation d'une base de données avec Java ne vous posera plus de problèmes majeurs. 😊

Ce tuto sur JDBC est maintenant terminé.

J'espère que ce dernier vous a plu et que vous avez appris tout plein de choses...

Nous nous retrouverons bientôt pour une autre API Java à découvrir ! Je ne vous dis pas encore laquelle... 🤪