**Projet Hopital :**

**Modifications importantes :**

**1) Singleton :**

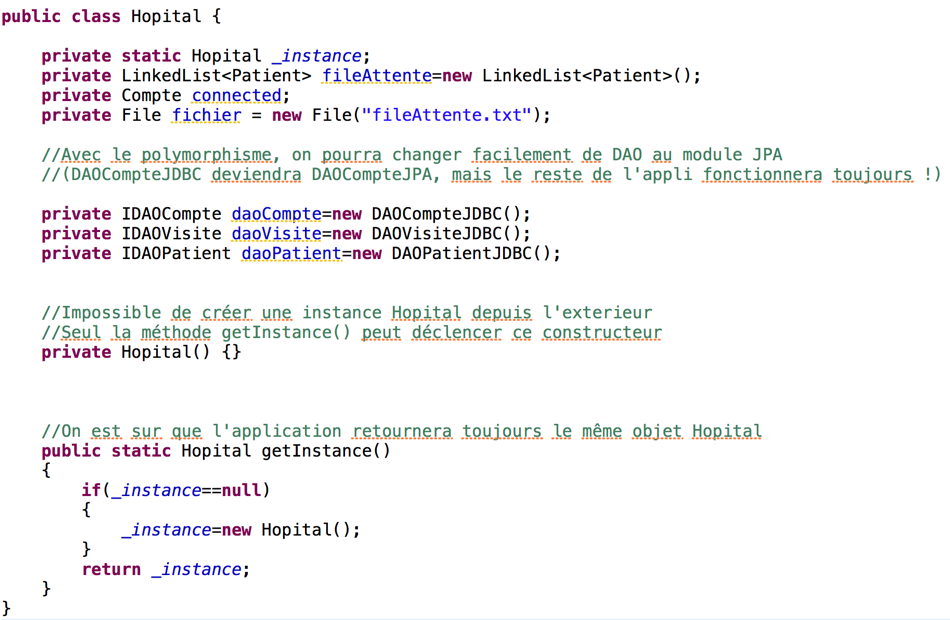
Plutôt que d’utiliser toutes les variables en static en haut de la classe, on va les stocker dans un objet qu’on récupèrera partout dans l’application.

Pour ça, on va utiliser le ***Design Pattern*** : **Singleton**. Le but c’est d’avoir une classe qui fournit un objet unique dans toute l’application.

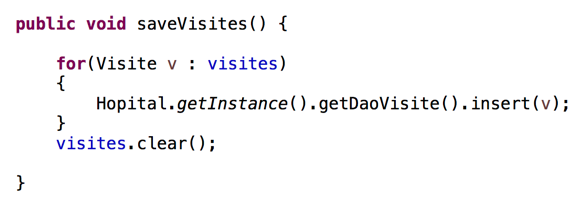
Comment ça marche :

* Créer une classe qui va stocker toutes les variables (dans notre exemple, Hopital)
* Placer une variable static du type de la classe ( ***\_instance*** ici)
* Mettre un constructeur en private
* Mettre une methode static qui retournera l’objet unique

La méthode static est la seule à pouvoir créer l’objet unique s’il n’est pas déjà présent en mémoire



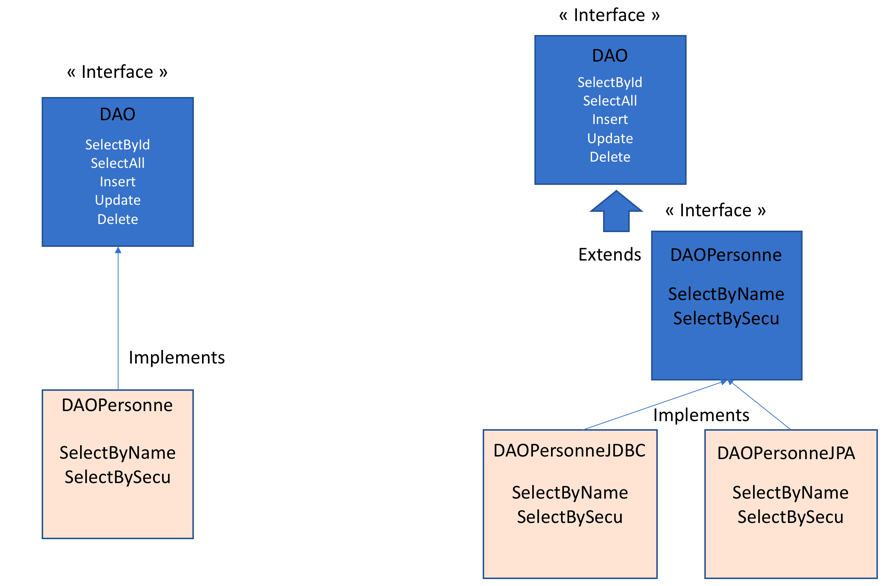
Dans le reste de l’application, on pourra accéder aux attributs en passant par getInstance()



**2) Modification des DAO :**

Avec le module JPA, vous allez refaire tous les DAO (mais garder l’interface IDAO, que vous utilisiez JDBC ou JPA, vous avez besoin des 5 methodes du CRUD quoi qu’il arrive)

La solution que l’on va appliquer est de rajouter une interface intermédiaire, c’est là qu’on ira placer les méthodes relatives à un model précis, la fonction checkConnect ira dans l’interface IDAOCompte, on est ainsi sûr de l’implémenter dans les versions JDBC et JPA

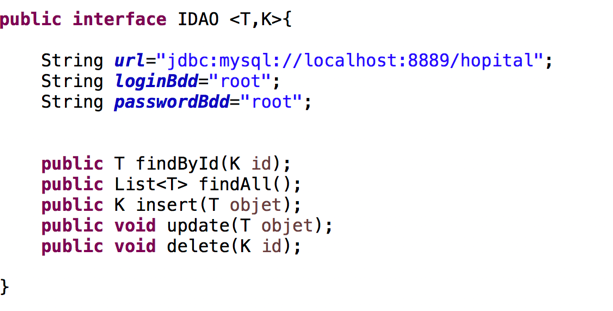


Modification de l’interface IDAO :

Les infos de connexions sont directement dans l’interface (pensez à les modifier !)

La fonction insert retourne désormais le nouvel id généré par la base.

Pour ça, on ajoute un paramètre à la fonction conn.prepareStatement() et on le récupère à l’aide d’un ResultSet





Suppression en Cascade :

Dans notre base de données, notre visite est composée de deux clés étrangères (id du patient et du médecin). Lorsque l’on supprime un médecin ou un patient, leurs visites doivent être supprimées à la main (Sinon SQL retourne une exception)

