TP n° 5

Exercice 1. Points techniques

Cet exercice a pour but de vérifier que la connexion à la base fonctionne correctement.

1. Se rendre sur `http://www.lacl.fr/~msassolas/enseignement/SQL-Prog_Licence/`.
2. *(Droits de `psql`)*
   a) Télécharger `pg_hba.conf`.
   b) Se placer en root à l'aide de `sudo su`.
   c) Remplacer le fichier `/etc/postgresql/9.3/main/pg_hba.conf` par le `pg_hba.conf` téléchargé :
   ```
   mv pg_hba.conf /etc/postgresql/9.3/main/pg_hba.conf
   ```
   d) Relancer le service de base de données :
   ```
   /etc/init.d/postgresql restart
   ```
   e) Quitter le mode root avec `exit`.
3. *(Fichiers fournis)* Tous ces fichiers sont à sauvegarder dans le répertoire dans lequel le TP sera effectué (là où seront les `.java`).
   a) (Re)télécharger `classicmodelsDB.sql` (il a légèrement changé).
   b) Télécharger la version actuelle de JDBC.
   c) Télécharger `ConnexionMerit.java`.
   d) Compiler `ConnexionMerit` :
   ```
   javac ConnexionMerit.java
   ```
   e) Exécuter `ConnexionMerit` sans oublier d'ajouter le classpath. En l'occurrence :
   ```
   java -classpath .:./postgresql-9.4.1207.jar ConnexionMerit
   ```
   Aucune erreur ne devrait apparaître.
4. *(Données dans la base)* Exécuter `psql merit -f classicmodelsDB.sql` afin d'être sûr que les données soient dans la base de donnée.

ATTENTION : dans la suite, nous allons écrire des requêtes pour la base de donnée. Souvenez-vous que `classicmodels` est un schéma. Ainsi, il faut
- soit indiquer comme nom de table `classicmodels.maTable` au lieu de simplement `maTable` ;
- soit avant chaque requête (une seule fois est nécessaire dans chaque fonction tant que la connexion n'est pas redéfinie), il faut se placer dans le bon schéma avec la commande suivante (où `st` est un `Statement`) :
```
st.executeUpdate("SET search_path TO classicmodels");
```

Exercice 2. Rechercher/Remplacer

Dans cet exercice, on va écrire des petits programmes qui n'auront qu'une seule classe (qui sera donc principale).

1. *(`Rechercher.java`)*
   a) Écrire une fonction `param2req` qui à partir des paramètres suivants : un mot clef, un nombre de résultats par page, un numéro de page n, retourne la requeête SQL à exécuter pour obtenir la n-ième page de résultat des produits (« products ») dont la description contient le mot clef ; les produits sont triés par ordre alphabétique de leur nom.
   b) Écrire la fonction qui effectue cette recherche et affiche le résultat sur la console.
   c) Écrire la fonction principale qui lit les arguments dans cet ordre depuis la ligne de commande et effectue la recherche.

2. *(`Remplacer.java`)*
   a) Écrire une fonction qui remplace dans toutes les descriptions des produits un mot par un autre. Quelques indications :
      — Le principe est d'effectuer une recherche sur le premier mot clef et d'agir pour chaque ligne du `ResultSet` que l'on obtiendra.
      — Cela implique d'utiliser un deuxième `Statement` pour faire les mises à jour.
      — Le remplacement en lui même est aisé car la classe `String` de Java dispose de l'opération `replace` (cf documentation).
   b) Écrire la fonction principale qui lit les chaînes à rechercher et remplacer depuis les arguments de la ligne de commande.
3. Tester ces programmes en cherchant un mot, puis en le remplaçant et en le recherchant à nouveau, en observant que le changement a bien été effectué.

Exercice 3. L'employé du mois

On souhaite créer un programme qui fournisse le meilleur vendeur à partir des données présentes dans la base de donnée. Nous allons globalement procéder de la manière suivante, qui sera détaillée dans les questions ci-dessous. Nous allons construire une classe `Client` dont les objets correspondront à un `customer` dans la base. Ceci nous permettra entre autres de savoir combien un client a effectué de commandes. Puis nous créerons une classe `Vendeur` qui correspondra aux employés qui sont des « Sales Rep », et auquel seront associés leurs clients. Nous terminerons par la création d'une `ListeVendeurs` qui rassemblera tous les vendeurs et permettra de savoir lequel est le meilleur [1].

1. *(La classe principale, pour le test)*
   Écrire une classe `MeilleurEmploye` qui dispose d'une fonction principale `main`. Cette fonction servira à faire des tests.
2. *(La classe `Client`)*
   a) Construire une classe `Client` disposant d'un attribut `numero` (un entier), un attribut `nom` (une chaîne de caractères initialisée à `""`) et un attribut `nbCommandes` (un entier initialisé à 0 par défaut).
   b) Écrire un constructeur pour cette classe ne prenant que son numéro (ainsi les autres attributs prendront leur valeur par défaut).
   c) Écrire les accesseurs permettant de lire les valeurs des attributs ; ne pas écrire d'accesseurs modifiant ces valeurs.
   d) Écrire une opération `toString` qui retourne une version textuelle lisible des données de l'objet `Client` ; par exemple : `Client n° 227 (Heintze Collectables): 2 commandes.`
   e) Écrire une opération `fetchNom` qui va chercher dans la base de donnée le nom correspondant au client identifié par son numéro et le renvoie. Si le client n'existe pas dans la base, cette opération devra lancer une `SQLException` avec un message adapté.
   f) Écrire une opération `fetchNbCommandes` qui va chercher dans la base de donnée le nombre de commandes correspondant au client identifié par son numéro et entre les deux dates données en argument (sous forme de `String` que l'on supposera de la forme AAAA-MM-JJ) et le renvoie. Remarquez que Postgres est tout à fait capable de comparer des dates ; à fins de test, sachez qu'il existe pour Postgres des dates `-infinity` et `infinity` qui sont respectivement avant et après toutes les autres. Notez que pour connaître le nombre de telles commandes, il faut récupérer le `ResultSet` correspondant à toutes ces commandes et le parcourir en comptant le nombre de lignes obtenues. Un client qui n'existe pas a 0 commandes.
   g) Écrire une opération `updateFromDb` qui met à jour les informations des attributs à partir de la base de donnée. Cette fonction prend également deux `String` en argument, afin de savoir quelle période de commande est concernée. Cette fonction ne doit pas pouvoir lever d'exception ; en effet, si le client n'existe pas, son nom sera simplement `""`.

1. Le critère utilisé, à savoir le nombre de commandes, est certes discutable, mais il a été imposé par le client.

3. *(La classe `Vendeur`)*
   a) Construire une classe `Vendeur` disposant d'un numéro, d'un nom complet (prénom et nom, initialement `""`) et d'une liste de clients vide par défaut :
   ```
   private ArrayList<Client> listeClients = new ArrayList<Client>();
   ```
   La documentation de la classe `ArrayList<E>` pourra vous être utile.
   b) Écrire le constructeur (ne prenant que le numéro en paramètre), et les accesseurs en lecture pour tous les attributs.
   c) Écrire une opération `toString` qui retourne une version textuelle des informations contenues dans l'objet. On pourra remarquer que `"Liste des clients: "+listeClients` fournit sans problème une chaîne contenant les informations voulues. (À tester à nouveau une fois que cette liste sera non vide.)
   d) Écrire une opération `updateNameFromDb` allant chercher les informations de nom et prénom pour mettre à jour l'attribut correspondant.
   e) Écrire une opération `updateClientsFromDbBrutal` qui met à jour la liste des clients de la manière – brutale – suivante :
   — On enlève tous les éléments de la liste des clients actuels (opération `clear`).
   — On récupère dans la base tous les numéros des clients du vendeur, et pour chacun on crée un nouvel objet `Client`.
   — On s'assure que les données de chaque `Client` ont bien été récupérées depuis la base de donnée.
   — On ajoute ces clients à la liste des clients (opération `add`).
   Cette fonction prend deux chaînes de caractères représentant l'intervalle de temps considéré.
   f) Peu satisfait de la méthode précédente qui crée de nouveaux objets [2] pour chaque client, on préférerait une opération `updateClientsFromDb` un peu plus élégante.
      i – Écrire une opération privée [3] `findClientFromNum(int numClient)` qui retourne – s'il existe dans la liste des clients – l'objet `Client` dont l'identifiant est donné en argument. S'il n'existe pas, un objet `null` est renvoyé.
      ii – Écrire l'opération `updateClientsFromDb` qui fonctionne de la manière suivante :
      — On crée une nouvelle `ArrayList<Client>` vide `newClList`.
      — On récupère les numéros des clients du vendeur.
      — Pour chaque client ainsi identifié, le rechercher dans la liste originale s'il existe.
      — S'il existe, mettre à jour les données du client et l'ajouter à `newClList`.
      — S'il n'existe pas, créer un nouveau client (en prenant soin de récupérer toutes les informations) et l'ajouter à `newClList`.
      — Enfin, remplacer la liste des clients par `newClList`.
   g) Écrire une opération `getNbCommandes` qui calcule le nombre total de commandes dont le vendeur est responsable. Cette fonction utilise les données telles qu'elles sont dans le programme ; en particulier, elle ne s'occupe pas de les mettre à jour à partir de la base de donnée.
4. *(La liste des vendeurs)*
   a) Créer une classe `ClassementVendeurs` dont les attributs sont une `ArrayList<Vendeur>`, initialement vide et deux chaînes correspondant à l'intervalle considéré.
   b) Écrire le constructeur, l'accesseur et l'opération `toString`.
   c) Écrire une opération `updateVendeursFromDb` qui récupère la liste des vendeurs à partir de la base de donnée (rappel, les vendeurs sont des « Sales Rep »). On pourra utiliser la méthode brutale ou la méthode élégante.
   d) Écrire une opération `meilleurVendeur` qui retourne le `Vendeur` ayant le plus de commandes.
5. *(La classe principale, pour livraison)*
   Modifier la classe principale et surtout la fonction `main` pour qu'elle affiche le nom du meilleur vendeur pour la période donnée en argument (par deux chaînes de caractères) lorsque le programme est lancé.

2. Mais que deviennent les anciens ? Est-ce que ça pourrait poser problème ?
3. En effet, on ne souhaite s'en servir qu'au sein de la classe `Vendeur`.