

## TD/TP2 –

## COMPLEMENTS JAVA - COLLECTIONS

EXERCICE 1 : *ANNUAIRE TELEPHONIQUE*

## 1.1. Annuaire Simple

Programmez une classe annuaire téléphonique qui associe à des noms (*String*) des numéros de téléphone (*String*). La classe **Annuaire** hérite de **HashMap** et offre, en plus du constructeur de l'annuaire vide, 5 méthodes :

- *domaine* : affichage de l'ensemble des noms de l'annuaire (appelé aussi le domaine de la table).
- *acces* : accès au numéro associé au nom donné en paramètre; si ce nom n'existe pas, la méthode retourne *null*.
- *adjonction* : adjonction d'un couple (nom, numéro) dans l'annuaire; le nom ne doit pas déjà être une entrée de la table; il le devient après l'adjonction; si nom existe déjà, la méthode ne fait rien.
- *suppression* : suppression d'une entrée de la table; si le nom n'existe pas, ne fait rien.
- *changement* : changement du numéro associé au nom donné en paramètre; si nom n'existe pas, la méthode ne fait rien.

Ecrire une classe de test unitaire pour vérifier le bon fonctionnement de ces méthodes.

## 1.2. Annuaire Complexe

Un nom ne permet pas d'identifier de manière certaine un interlocuteur. Pour rendre l'annuaire plus sûr, on souhaite désormais associer un numéro de téléphone à un couple (*nom, prenom*).

Ce couple (*nom, prenom*) est représenté par une classe **Personne** à définir.

Ecrire une classe **AnnuaireComp** avec les méthodes analogues à la classe précédente. Bien entendu, la méthode *acces* utilisera désormais deux *String* comme paramètres: le nom et le prénom de la personne que l'on cherche. Et ainsi de suite.

Testez avec attention votre nouvelle classe. Il est possible qu'elle ne fonctionne pas correctement. Expliquez pourquoi et remédiez à la situation.

## EXERCICE 2 : EXTRACTION D'ADRESSES IP

On dispose d'un fichier texte décrivant les logs des opérations effectuées sur un serveur. Ce fichier de logs (**logs.txt** fourni sur arche) est structuré de la manière suivante :

- chaque ligne correspond au log d'une opération,
- chaque opération est décrite par l'adresse IP de la machine à l'origine de l'opération suivie d'un espace et du descriptif de l'opération (pensez à utiliser la méthode **split** de la classe **String**),
- les lignes sont triées par date (non utilisée dans ce sujet) et ne sont pas triées par IP.

```
64.233.166.94 ajout_nouveau_produit_dans_la_base
193.54.21.201 suppression_produit_dans_la_base
193.54.21.201 acces_a_la_base
192.168.0.14 ajout_nouveau_produit_dans_la_base
```

On souhaite pouvoir construire l'ensemble des adresses IP des machines qui ont effectué des opérations sur le serveur à partir du fichier de logs. Chaque adresse IP doit être présente au plus une seule fois même si la machine a effectué plusieurs opérations sur le serveur.

- 2.1. Quelle classe utiliser pour assurer l'unicité des adresses IP dans la structure de données? Si besoin, quelles adaptations sont nécessaires pour utiliser cette classe ?
- 2.2. Ecrire une classe **ListeIp** chargée de construire cet ensemble. Cette classe possédera un attribut **ips** de type **Set<String>** contenant les IP des machines sous la forme d'une chaîne de caractères. Elle possédera en outre un constructeur, une méthode **chargerFichier(String name)** chargée de lire le fichier et de remplir l'ensemble et une méthode **toString** destinée à afficher l'ensemble des adresses IP.
- 2.3. On souhaite ajouter une option pour afficher les adresses IP triées dans l'ordre alphabétique. Le constructeur de **ListeIp** doit désormais prendre en paramètre un **boolean** permettant de savoir si le résultat doit être trié ou non. La classe concrète de l'attribut **ips** dépendra de la valeur de ce paramètre et permettra de changer le comportement de l'objet **ListeIp** construit. Prenez le temps de regarder quelles classes concrètes sont possibles avant de chercher à répondre à la question.
- 2.4. L'ordre alphabétique ne permet pas de trier les adresses IP comme attendu puisque les chaînes sont comparées caractère par caractère (les chiffres étant comparés dans l'ordre, 64.233.166.94 arrive après 192.114.51.21). On souhaite utiliser une classe intermédiaire **AdresseIp** à écrire pour résoudre le problème. L'attribut **ips** est désormais de type **Set<AdresseIp>**. Modifiez et testez les classes pour obtenir le comportement attendu (unicité des adresses IP, adresses IP triées dans l'ordre des adresses quand cela est demandé, ...).

## EXERCICE 3 : FORMATION

-----Attention, cet exercice est à faire en binôme-----

### 3.1. TRAVAIL DE CONCEPTION

Lire attentivement et en totalité l'énoncé 3.2. ci-après, réfléchir **avec votre binôme** aux différentes **collections** à utiliser pour répondre au problème posé puis, esquisser sur papier le **diagramme de classes** correspondant. Vérifier la cohérence de votre diagramme en esquissant les **diagrammes de séquences** pour le calcul de la moyenne générale d'un étudiant participant à une formation, pour le calcul de la moyenne dans une matière de tous les étudiants d'un groupe de formation. A partir de ce diagramme, chaque membre du binôme ne devra coder qu'une partie des classes, elles seront mises ensuite en commun, soyez donc précis sur les noms et paramètres des méthodes de chaque classe. Générer le diagramme de classe et le diagramme de séquences demandés en utilisant *PlantUml* ou le logiciel de votre choix.

-----Avant de passer sur machine, faire valider vos choix et le diagramme de classe par votre enseignant.-----

-----Avec votre binôme, sur Github, créer un *repository* (dépôt) qui sera amené à contenir vos fichiers. Partagez ce *repository* avec votre binôme et votre enseignant. Voir le fichier *Utilisation\_git* sur arche pour des rappels sur la marche à suivre.-----

### 3.2. ENONCE DU PROBLEME

Un étudiant est caractérisé par son identité, sa formation et ses résultats. L'identité est définie par un NIP (Numéro d'Identification Personnel), un nom et un prénom (3 chaînes de caractères). Les résultats d'un étudiant sont mémorisés sous la forme d'une collection de notes par matière.

Une formation est définie par un identifiant et une collection des matières qui y sont enseignées avec leurs coefficients.

Une matière est définie par son nom.

Un groupe d'étudiants est défini par une collection des étudiants qui le composent et la formation à laquelle ce groupe appartient.

#### 3.2.1. Ecriture des classes Identite, Etudiant, Formation

##### Travail du membre 1 du binôme :

- Ecrire la classe **Identite** puis déposer cette classe dans votre dépôt.
- Ecrire la classe **Etudiant**. Il faut pouvoir :
  - ajouter une note à un étudiant dans une matière donnée (la note doit être entre 0 et 20), *que faire si la note n'est pas dans le bon intervalle ou/et si la matière n'est pas dans la formation de l'étudiant ?*
  - calculer sa moyenne pour une matière, *que faire si la matière n'est pas dans la formation de l'étudiant ?*
  - calculer sa moyenne générale (attention aux coefficients des matières).
- Faire un *commit* de ces 2 classes
- Charger la classe **Formation** que votre binôme a déposée et écrire pour cette classe une classe de test unitaire pour tester les principales méthodes de **Formation**.

##### Travail du membre 2 du binôme :

- a. Définir la classe **Formation**. On veut pouvoir :
  - ajouter ou supprimer une matière dans une formation,
  - connaître le coefficient d'une matière, *que faire si la matière n'est pas dans la collection attribut de la formation ?*
- b. Faire un *commit* de cette classe.
- c. Charger les classes que votre binôme a déposées et écrire pour la classe **Etudiant** une classe de test unitaire pour tester les principales méthodes de cette classe.

---

**Faire valider vos choix de tests par votre enseignant.**

---

### 3.2.2 La classe Groupe : Gestion des étudiants

- a. Dans la branche principale (master), définir la classe **Groupe**. On doit pouvoir ajouter et supprimer un étudiant du groupe. Attention, on peut ajouter un étudiant à un groupe uniquement si le groupe et l'étudiant ont la même formation,
- b. Ecrire une classe de test pour l'ajout et la suppression d'étudiant dans le groupe.

### 3.2.3 La classe Groupe : Gestion des branches et fonctionnalités complémentaires

Le développement de la classe Groupe va désormais se faire en utilisant des branches. **Chaque binôme sera en charge d'ajouter une fonctionnalité dans une branche dédiée.**

Une fois que la fonctionnalité est ajoutée et validée, le binôme devra faire un merge (fusion) avec la branche principale pour ajouter cette fonctionnalité au projet.

#### Travail du membre 1 du binôme : gestion des moyennes

- a. Créer une branche spécifique "**fct\_moyennes**" destinée à contenir le développement de nouvelles fonctionnalités correspondant aux calculs de moyennes.
- b. Se déplacer dans cette branche pour effectuer la suite du travail sur ces fonctionnalités. Désormais, les modifications se feront sur cette branche "**fct\_moyennes**" jusqu'à ce que le travail soit validé par les tests.
- c. Modifier la classe Groupe pour
  - calculer la moyenne du groupe pour une matière,
  - calculer la moyenne générale.
- d. Ecrire une classe de test permettant de valider que les calculs de moyenne sont corrects.
- e. Lorsque tout est validé, pusher l'ensemble des modifications dans la branche dédiée "**fct\_moyennes**" puis faites un merge dans la branche master pour ajouter cette fonctionnalité au projet.

#### Travail du membre 2 du binôme : gestion des tris

- a. Créer une branche spécifique "**fct\_tris**" destinée à contenir le développement des nouvelles fonctionnalités de tri dans la classe Groupe.

- b. Se déplacer dans cette branche pour effectuer la suite du travail sur cette fonctionnalité. Désormais, les modifications se feront sur cette branche "fct\_tris" jusqu'à ce que le travail soit validé par les tests.
- c. Ajouter dans la classe Groupe,
  - la méthode **triAlpha** qui permet de trier la liste des étudiants du groupe selon l'ordre alphabétique de leur nom (de A vers Z).
  - la méthode **triAntiAlpha** qui permet de trier la liste des étudiants du groupe dans l'ordre inverse de l'ordre alphabétique (de Z vers A).
- d. Ecrire une classe de test permettant de valider que les tris sont corrects.
- e. Lorsque tout est validé, pusher l'ensemble des modifications dans la branche dédiée "fct\_tris" puis faite un merge dans la branche master pour ajouter cette fonctionnalité au projet.

### Travail commun

- a. Une fois que les merges ont été faits, s'assurer que les deux membres du binôme sont bien sur la branche master et disposent des deux fonctionnalités chacun.
- b. Vérifier que les tests des nouvelles fonctionnalités donnent des résultats valides.

#### 3.2.4 La classe Groupe : Tri par mérite

Le travail se fait désormais en binôme dans la même branche et tire parti des travaux réalisés à l'étape précédente.

- a. Créer une branche "fct\_merite" pour ajouter une nouvelle fonctionnalité sans perturber la branche master. Se placer dans cette branche (pour les deux membres du binôme).
- b. Dans cette branche, ajouter la méthode **triParMerite** dans la classe Groupe. Cette méthode doit trier la liste des étudiants en fonction de leur moyenne générale.
- c. Modifier la classe de test pour ajouter des tests vérifiant la validité de la méthode **triParMerite**.
- d. Lorsque la fonctionnalité de tri par mérite est validée, fusionner la branche "fct\_merite" avec la branche principale.