Bases de la programmation orientée objet

IUT Montpellier-Sète – Département Informatique

- **Cours:** M2103 - support ici
- **Enseignants:** Marin Bougeret, Romain Lebreton, Sophie Nabitz, Victor Poupet, Petru Valicov
- Le forum Piazza de ce cours pour poser vos questions
- Email pour une question d'ordre privée concernant le cours.

Consignes

- Vous respecterez les différents principes de programmation vues dans le cours et les TPs précédents
- Toutes vos classes doivent résider dans le paquetage `fr.umontpellier.iut`
- Les signatures des méthodes et les noms des classes qui vous sont données doivent rester inchangés.

TP9 : *utilisation des collections Java*

Voici le lien GitHub Classroom pour faire votre fork privé du TP.

Date limite de rendu de votre code sur le dépôt GitHub : **Dimanche 25 avril à 23h00**

Reprenons la classe `Employe` du TP5. Pour vous faciliter la tâche, nous l'avons directement écrite et simplifiée en enlevant les attributs inutiles pour ce TP. Un attribut `dateEmbauche` avec accesseur et modifieur ont été ajoutés à la classe `Employe`. Le but de ce TP est de proposer différentes organisations des employés dans l'entreprise en fonction des besoins du client.

Exercice 1 - mise en place

1. La classe `Entreprise` gère les employés sous forme d'une collection (la plus générale possible). Ajoutez un constructeur sans paramètres instanciant cette collection en tant que `ArrayList`.
2. Complétez les méthodes `void embaucher(Employe e, LocalDate dateEmbauche)` et `void licencier(Employe e)` de la classe `Entreprise` afin mettre à jour la liste d'employés de manière correspondante.
3. Redéfinissez la méthode `String toString()` de la classe `Entreprise` pour afficher ses informations.
4. Vérifiez votre programme en créant dans la classe principale (`GestionEmployes`) une entreprise et en embauchant plusieurs employés. Vous afficherez l'état de l'entreprise après chacune des opérations.

Exercice 2 - organisation des employés

L'entreprise souhaite mieux organiser ses employés.

1. Redéfinissez les méthodes `equals(Object o)` et `hashCode()` de la classe `Employe` afin de distinguer deux employés en fonction de leur numéro INSEE et leur nom (de type `String`). Dorénavant deux employés seront considérés comme des doublons s'ils ont le même numéro INSEE et le même nom.

Remarque : la collection `lePersonnel` peut contenir le même employé (même numéro INSEE et même nom) plusieurs fois si cette personne occupe des postes différents. Donc ici vous ne devez pas toucher au code de l'objet `lePersonnel` de la classe `Entreprise`.

2. Écrivez le corps de la méthode `Collection<Employe> getEmployesDansDesordre()`. À partir de la collection `lePersonnel`, elle devra retourner une autre collection en enlevant tous les doublons et ce **sans invoquer explicitement un algorithme de recherche de doublons**.

   Attention : Pas de modifications du code précédemment écrit (et donc de l'attribut `lePersonnel`).

3. Maintenant, pour une meilleure lisibilité, l'entreprise souhaite pouvoir retrouver l'ensemble de ses employés sans les doublons mais dans l'ordre. L'ordre choisi est l'ordre *croissant* suivant le nom et qui en cas d'égalité, applique l'ordre *décroissant* suivant le numéro INSEE.

   Écrivez le corps de la méthode `Collection<Employe> getEmployesOrdonnes()` qui, à partir de la collection `lePersonnel`, retourne une autre collection respectant ces contraintes. Naturellement, comme dans la question précédente, **il ne faut pas écrire ou invoquer explicitement un algorithme de recherche de doublons, ni un algorithme de tri**.

   Remarque : les deux fonctions `getEmployesDansDesordre()` et `getEmployesOrdonnes()` doivent être totalement indépendantes et ne doivent pas s'appeler entre elles.

4. Écrivez plusieurs tests unitaires vérifiant la fonctionnalité programmée. Voici le scénario à appliquer dans chaque test :

   - créer une entreprise
   - créer plusieurs employés (au moins 4) avec des noms différents et/ou numéros INSEE différents
   - vérifiez avec des *assertions* (`assertEquals(...)`, `assertNotEquals(...)`, `assertTrue(...)`, `assertFalse(...)` etc.) que la méthode `getEmployesOrdonnes()` fonctionne correctement. Vous vérifierez notamment que les collections retournées par `getEmployesOrdonnes()` et `getEmployesDansDesordre()` sont de même tailles (et contiennent les mêmes employés). Vous trouverez la liste exhaustive des assertions en *Junit 5* dans l'API de la classe `Assertions`.

Exercice 3 - priorité aux anciens

L'entreprise souhaite distribuer des bonus à ses employés en fonction de la date d'embauche. Le problème est que cette somme est évidemment limitée, donc on risque de ne pas pouvoir distribuer des bonus à chaque employé... Cette somme est représentée par l'attribut `double bonusTotal` de la classe `Entreprise`. Un *setter* permet à l'utilisateur de fixer à tout moment la somme d'argent disponible pour distribuer un bonus aux employés.

L'attribut `double bonus` de la classe `Employe` permet de définir la quantité de bonus qu'un employé va recevoir. Les méthodes *setter* et *getter* permettent la gestion de ce bonus.

1. Dans la classe `Employe` la méthode `int getMoisAnciennete()` renvoie le nombre de mois correspondant à l'intervalle de temps entre la date d'embauche et maintenant. Le corps de cette méthode vous est donné et vous ne devez pas le modifier. L'ancienneté est calculée sur le nombre de mois **complets** depuis la date d'embauche à l'aide du type énuméré `ChronoUnit`. Cette classe permet d'effectuer des calculs en fonction de différentes unités temporelles (jours, mois, années, etc.).

   Écrivez plusieurs tests unitaires afin de comprendre le fonctionnement de la méthode `int getMoisAnciennete()` de la classe `Employe`. Vérifiez notamment que deux personnes étant embauchées à des dates différentes, mais ayant effectué le même nombre de mois **complets** aient la même ancienneté. Par exemple, un employé embauché depuis 10 mois et 14 jours aura la même ancienneté qu'un employé embauché depuis 10 mois. Prêtez attention à la variation des longueurs des mois dans une année !

2. Le patron a décidé de donner la priorité aux anciens pour la distribution du bonus. Ainsi, le bonus sera distribué aux employés suivant leurs dates d'embauche : de la plus ancienne, à la plus récente. Le bonus qu'un employé va recevoir est égal à `3*ancienneté`.

Écrivez le corps de la méthode `void distribuerBonus()` qui effectue cette tâche **sans utiliser explicitement** un algorithme de tri et **sans modifier la classe `Employe`** Écrivez des tests unitaires pour vous assurer que chaque employé a bien reçu le bon bonus.

Remarques :

- Pour déterminer l'ordre de distribution du bonus de deux employés embauchés à des dates identiques vous prendrez l'ordre d'apparition dans la collection `lePersonnel`.
- Un employé embauché à plusieurs postes (qui apparaît plusieurs fois dans `lePersonnel`), percevra plusieurs fois le bonus.
- Le bonus étant limité, il se peut que certains employés ne touchent rien (notamment les plus jeunes). De même, si vers la fin de la distribution, la quantité de bonus restante est inférieure à `3*ancienneté`, alors l'employé recevra seulement la quantité de bonus restante et tant pis pour son ancienneté !

3. Modifiez la méthode `toString()` de `Employe` afin qu'elle affiche également le bonus que l'employé a reçu.

4. L'entreprise traverse une période de crise et décide de se séparer d'une partie de ses employés. Afin de fidéliser les anciens employés, ce qui a été décidé c'est de licencier les employés ayant travaillé le moins longtemps dans l'entreprise. Sans modifier le code précédemment écrit, écrivez le code de la méthode `void remercier(int n)` de la classe `Entreprise` afin de licencier `n` employés ayant été embauchés le plus tard.

   Remarques importantes : Comme dans le cas de la question 2, si deux employés sont embauchés à des dates identiques, vous les remercierez dans l'ordre d'apparition dans la collection `lePersonnel`. Également, un employé peut être licencié d'un poste, mais pas d'un autre.

   Astuce : Pour cette question pensez à vérifier le scénario suivant :

   1. Créer 3 employés comme ceci
      - 2 employés *fifi* et *loulou* avec le même numéro INSEE et le même nom, et des bases différentes
      - 1 employé *toto* avec numéro INSEE, nom et base quelconques
   2. Embaucher d'abord *toto* avec une date d'embauche la plus ancienne (disons 1er janvier 2000), ensuite *fifi* (23 mars 2021) et ensuite *loulou* (25 mars 2021)
   3. Remercier qu'un seul employé en invoquant `remercier(1)` et vérifier que tout fonctionne correctement.

Exercice 4 - indemnités de transport

On souhaite maintenant pouvoir calculer les indemnités de transport pour chaque employé en fonction de la distance entre sa ville de résidence (une donnée de type `String`) et les locaux de l'entreprise. L'attribut `String adresse` de la classe `Employe` correspond à sa ville de résidence. Le *getter* et le *setter* permettent la gestion de cet attribut.

1. La classe `GestionDistances` gère une collection statique faisant correspondre une distance (un entier) à une ville. Une ville ne peut être associée qu'à une unique distance, mais une même distance peut être associée à plusieurs villes. Initialisez cette collection (avec une méthode statique ou au chargement de la classe) avec les données suivantes :

   - Montpellier → 0
   - Sète → 36
   - Sommières → 30
   - Nîmes → 58
   - Lunel → 30
   - Béziers → 80

   Attention : la collection `distances` est statique, donc n'ajoutez pas de constructeur à la classe `GestionDistances`...

2. Déclarez une classe d'exception contrôlée `AdresseInconnueException` héritant de `Exception`. Le constructeur de cette classe aura comme argument un objet `String nomVille` et appellera le constructeur de la classe de base (`Exception`) avec le message : "*La ville " + `nomVille` + " n'existe pas*"
3. La méthode `static int getDistance(String ville)` de `GestionDistances` devra retourner la distance associée à la ville passée en paramètre. Modifiez la signature de cette fonction annonçant qu'elle est **susceptible** de lever une exception `AdresseInconnueException`. Écrivez ensuite le corps de `static int getDistance(String ville)` en veillant à ce qu'elle **lève** une exception `AdresseInconnueException` si la ville n'est pas présente dans la collection `distances`.
4. Écrivez le corps de la méthode `double getIndemniteTransport()` de la classe `Employe`. Elle doit retourner l'indemnité qui est dûe à l'employé. La formule de calcul de cette indemnité est `distance * base`. Si la ville n'existe pas, cette méthode devra traiter l'exception correspondante et retourner `0`.
5. Écrivez des tests unitaires pour vérifier le bon fonctionnement de la méthode `double getIndemniteTransport()`.

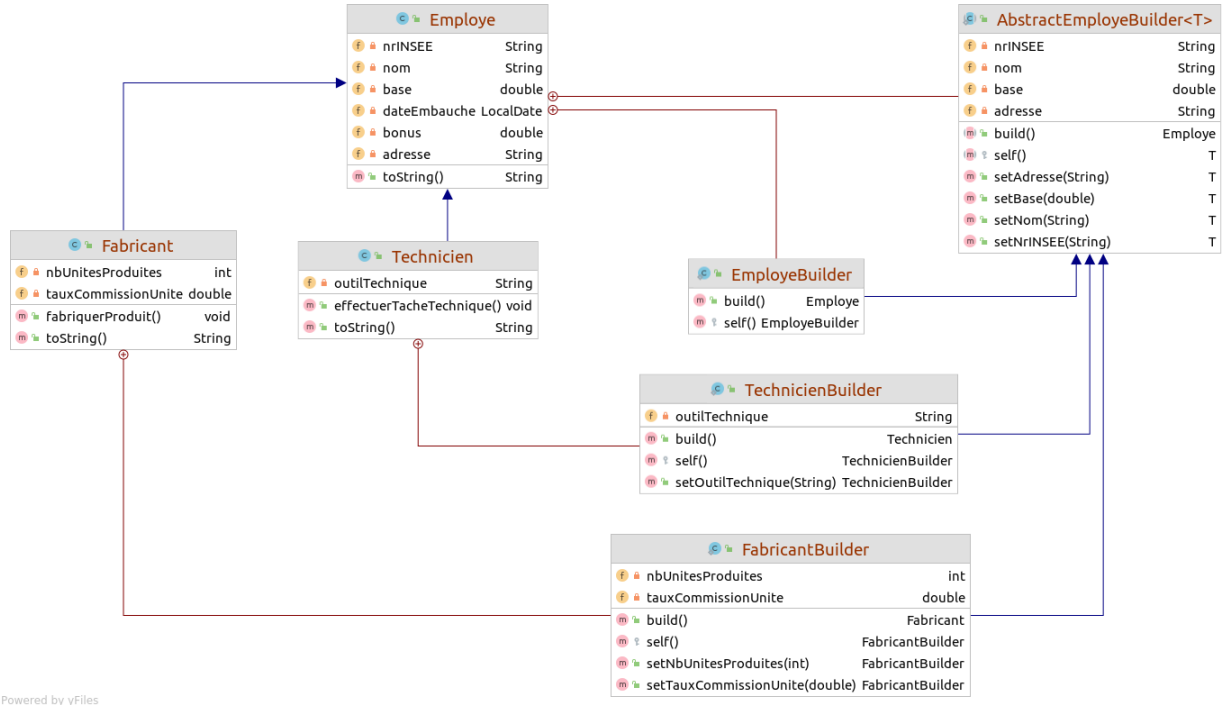
Exercice 5 (bonus) - builders hiérarchiques

Pour cette partie, vous allez travailler **exclusivement** dans les classes du package `fr.umontpellier.iut.bonus`, le code pour les exercices précédents (du package `fr.umontpellier.iut`) devrait rester intact.

Souvenez-vous que dans le TP5 il y avait toute une hiérarchie de classes héritant de `Employe`, chacune ayant des spécificités. À l'époque, lorsque vous avez généré des builders pour construire des objets de ces classes, plusieurs duplications de code entre les différentes classes builder sont apparues. Il est temps de corriger cela !

Dans le package `fr.umontpellier.iut.bonus`, une solution partielle à ce problème de duplication de code vous est proposée. Une classe abstraite `AbstractEmployeBuilder` permet de factoriser l'ensemble des fonctions de construction communes (à tous les `Employe`). Cette classe est héritée par des builder spécifiques : `EmployeBuilder`, `TechnicienBuilder` et `FabricantBuilder`. En fonction du type concret d'employé, chacune de ces sous-classes ajoute les fonctions supplémentaires pour la construction de l'objet correspondant. Une particularité est que les classes builder sont gérées comme des classes internes statiques (par exemple `TechnicienBuilder` est une classe statique interne de la classe `Technicien`). Ceci pour favoriser l'encapsulation des différents sous-types de `Employe` (pour plus d'infos sur les classes internes voir ce tutoriel Oracle).

Voici le diagramme de classes de cette solution partielle :



La classe `AbstractEmployeBuilder` est paramétrée par un type `T` dont le domaine de définition est borné par `AbstractEmployeBuilder<T>`. Elle possède deux méthodes abstraites importantes : * la méthode `build()` retourne par défaut l'`Employe` construit * la méthode `self()` retourne un objet de type `T` - dans les sous-classes de `AbstractEmployeBuilder`, `T` sera remplacé par le type effectif de builder

Observez comment ces deux fonctions abstraites sont redéfinies dans les trois classes builders : `EmployeBuilder`, `TechnicienBuilder` et `FabricantBuilder`. Pour illustrer le fonctionnement, dans la classe principale `GestionEmployesBuilders` on instancie des différents types d'employé, on les affiche et on leur demande d'exécuter les tâches spécifiques.

Vous remarquerez que dans cette solution il manque le cas des commerciaux. En vous inspirant de la solution existante pour les classes `Employe`, `Fabricant` et `Technicien`, modifiez les classes `Commercial`,`Vendeur` et `Representant` comme suit : * faites hériter `Commercial` de `Employe` * proposez les constructeurs et les méthodes appropriées dans les classes `Commercial`,`Vendeur` et `Representant` afin de permettre l'instanciation des différents commerciaux de manière analogue : avec des builders et en évitant la duplication de code.

Analysez votre solution et discutez avec votre enseignant si nécessaire.

Une explication approfondie concernant les builders hiérarchiques est donnée dans *Effective Java* de J. Blosch, (3ème édition).