IUT Montpellier-Sète
Département Informatique
S. Gagné, S. Nabitz, P. Valicov

TD 2 : Livraison de marchandise avec le train

Rappels sur le collections

Le site de l'organisation du cours, où vous allez trouver le projet à forker (TD2) :
https://github.com/IUTInfoMontp-M3105

Consignes :

— Respectez toutes les recommandations du TD précédent (TD1).

— Le but de ce TD est de vous rappeler les notions sur les différentes collections `Java`. Reprendre les transparents du cours de cette année et de l'an dernier vous sera bénéfique.

— Le but sera aussi de programmer de manière incrémentale : en ajoutant du code nécessaire pour chaque question et en limitant le nombre de *modifications* (effacer un code fonctionnel existant & et le remplacer par un autre).

— Pensez à écrire des tests unitaires pour chaque nouvelle fonctionnalité ajoutée. Plus vous en ajoutez, meilleure sera votre programme et votre note.

Date limite de rendu de votre code sur le dépôt GitHub : **Dimanche 22 Septembre à 23h00**

Sujet

Vous allez écrire une petite application qui doit aider à l'organisation du transport des marchandise d'une entreprise ferroviaire basée à Montpellier. Le principe est d'assembler des wagons en file (derrière la locomotive) et de détacher les wagons dans leur gare de destination, où les clients se chargeront du reste. Le but est d'organiser l'assemblage des wagons dans le train de façon à ce que la décharge se fasse de façon optimisée.

L'objectif pédagogique est de vous faire comprendre, selon les questions, quelle est la collection la plus appropriée au problème posé afin de mieux respecter les différents principes de conception et programmation objet.

Exercice 1

1. Un *Wagon* est caractérisé par le nom de son *expéditeur*, son *poids* en kilos (arrondi à l'entier près) et sa *destination*. Écrivez la classe correspondante dans le package `fr.umontpellier.iut` et redéfinissez la méthode `toString()` pour afficher l'intégralité de ses attributs. Ajoutez une méthode accesseur (*getter*) au poids du wagon.
2. Dans la méthode `main(String args[])` de la classe principale (`AppTransports`) :
   (a) instanciez au moins les 4 wagons suivants
      (i) société = "RBus", poids = 110, destination = "Toulouse"
      (ii) société = "LaTouristra", poids = 110, destination = "Paris"
      (iii) société = "LeBonVin", poids = 90, destination = "Bordeaux"
      (iv) société = "PoPoLain", poids = 145, destination = "Blois"
   (b) stockez ces informations dans une *Collection* (que vous pouvez appeler `train`), instanciée pour l'instant avec un type effectif le plus basique possible, par exemple `ArrayList`,
   (c) affichez le contenu de cette collection.

Exercice 2

On veut maintenant ordonner les wagons selon leur destination.

1. Vous allez stocker les informations concernant les distances dans une collection, qui sera un attribut *statique* d'une classe `GestionDistances`. Cette collection fera correspondre une distance (entier) à une ville (`String`). Une ville ne peut être associée qu'à une unique distance. Voici les données à stocker dans la collection :

   `Toulouse` ⇒ `249, Paris` ⇒ `760, Bordeaux` ⇒ `483, Blois` ⇒ `618, Lille` ⇒ `960`.

   Quelle collection vous parait la plus appropriée ?

   (a) Écrivez une méthode *statique* dans la classe `GestionDistances` qui prend en paramètre une ville (`String`) et retourne la distance associée à cette ville.

   (b) Complétez la classe `Wagon`, en y ajoutant une méthode `getDistance()`, qui retourne la distance jusqu'à la ville de destination.

   Remarque : la classe `GestionDistances` est une classe de données, à priori elle n'a pas à être instanciée.

2. À partir de la collection `train`, construisez une nouvelle collection `assemblageTrain` dans la méthode `main(String args [])`, de façon à ce que `assemblageTrain` contienne les wagons selon l'ordre définitif de leur assemblage dans le train : les wagons dont la destination est la plus proche doivent toujours être au début de la collection ordonnée, et ceux dont la destination est la plus lointaine doivent toujours être vers la fin de la collection. Les wagons doivent donc être comparés entre eux selon leur distance jusqu'à leur destination.

   Quelle collection utiliseriez-vous ? Comment compareriez-vous les wagons entre eux ? Discutez avec votre enseignant.

3. Vérifiez l'assemblage du train dans l'affichage de la méthode `main(String args[])`.

4. Il est possible que plusieurs wagons soient destinés à la même ville. Dans ce cas (et seulement dans ce cas), votre client vous demande de les ordonner en ordre croissant selon le poids. Ajoutez cette fonctionnalité **sans modifier** le code des *classes métiers* (toutes sauf `AppTransports`).

5. Votre client vient de changer de stratégie de distribution. Maintenant il veut que vous *modifiiez* l'ordre d'assemblage en l'ordre inverse selon les distances vers les destinations. En revanche, lorsque les distances pour deux wagons sont identiques, le critère de poids précédemment écrit doit continuer à s'appliquer. Modifiez votre programme afin que dans la collection `assemblageTrain`, les wagons de la destination la plus lointaine (resp. la plus proche) soit au début (resp. à la fin) de la collection. Combien de modifications devriez-vous apporter à votre programme ?

Exercice 3

Le train est parti ! On imagine maintenant qu'on peut charger de nouveaux wagons en cours de trajet. Pour simplifier, on suppose que :

(a) la locomotive est suffisamment puissante pour attacher autant de wagons que l'on veut ;

(b) les nouveaux wagons seront attachés en queue de la file du train, et ce même si la destination du nouveau wagon est très lointaine ;

(c) la destination du nouveau wagon ne sera pas une gare déjà passée.

1. Dans un premier temps, testez la situation suivante : on détache de la `assemblageTrain` un wagon dans la première gare atteinte (par exemple Toulouse), puis on en attache un nouveau à destination de la gare la plus lointaine (par exemple Lille). Que constatez-vous concernant l'organisation du train ?

2. Proposez une amélioration de l'organisation de l'assemblage des trains en choisissant une collection plus adaptée, qui permettrait d'ajouter les nouveaux wagons en queue de train (càd comme premiers éléments de votre collection). Pour cela, l'assemblage initial ne doit pas changer. Écrivez une classe `Chargement` construite à partir d'une collection de wagons. Cette classe doit permettre d'attacher et de détacher les wagons dans le bon ordre.

Exercice 4

L'entreprise veut maintenant se faire rémunérer en calculant le tarif de chaque wagon. Pour proposer une offre tarifaire flexible, vous devez prévoir des possibilités de varier le tarif pour chaque client selon les périodes de l'année, ou les trains affrétés, etc... Pour commencer, on peut appliquer :

- un tarif au poids,
- ou bien au km parcouru,
- ou bien un tarif fixe

La politique tarifaire va sûrement évoluer dans les années à venir et donc toutes les variantes de calcul de tarif ne sont pas identifiées à ce jour. Mais vous n'avez que quelques heures pour finaliser la classe `Wagon` définitivement, en y ajoutant une unique méthode `double getTarif(...)` qui renvoie le coût en fonction du type de facturation décidée. Le schéma d'exécution sera le suivant :

(a) dans la classe principale, le client décide une formule de calcul de tarification pour le wagon,

(b) *en fonction* de cette tarification, la méthode `double getTarif(...)` doit retourner la somme d'argent que ce wagon va coûter.

1. Proposez un diagramme de classes qui modélise la stratégie de calcul du tarif comme un objet générique `Tarif` à appliquer à un `Wagon`. Discutez avec votre enseignant.

2. Implémentez les classes des différentes stratégies de calcul ainsi que la fonction `getTarif(...)` de `Wagon`. Chaque stratégie doit être paramétrable : le prix au km, le prix au kilo, le tarif fixe sont précisés à la construction.

3. Vérifiez dans le programme principal, en appliquant différents tarifs aux wagons de votre train.

4. Voici un nouveau tarif à ajouter rapidement à l'application avant de partir : on applique le minimum entre le tarif distance et le tarif au poids. Ajoutez le code nécessaire **sans modifier** le code des classes précédemment écrites et vérifiez le tout dans le programme principal.