Java Avancé

Fiche 12 : exercices récapitulatifs

Table des matières

1 Objectifs

Pas de nouveaux objectifs, ce sont les objectifs de toutes les fiches précédentes qui sont applicables.

2 Concepts

Pas de concepts spécifiques à cette fiche, le but de cette fiche est de reprendre l'ensemble des concepts des fiches du cours d'Ateliers Java.

3 Exercices

3.1 Introduction

L'exercice qui suit et un exercice récapitulatif du cours d'Ateliers Java et devrait pouvoir être réalisé en trois heures **sans internet**, tout en pouvant consulter les solutions des ateliers, les fiches théoriques, la JavaDoc, le manuel IntelliJ et les slides de COO.

C'est le questionnaire d'examen de la session d'août 2021.

Cet exercice comporte 2 parties, chacune calibrée pour nécessiter 1h15 minutes de travail. Chaque partie est découpée en plusieurs questions. Il est donc offert ~30 minutes pour lire cet énoncé, vous imprégner des projets.

Les questions sont globalement assez indépendantes les unes des autres et doivent parfois même être réalisées dans leur propre projet ou module. Ceci implique que vous pouvez **répondre aux questions dans l'ordre que vous voulez** !

1. L'archive **AJ-reacp-partie1.zip** contient le projet associé à la partie 1 de l'exercice et l'archive **AJ-reacp-partie2.zip** contient le projet IntelliJ associé à la partie 2.
2. Désarchivez ces deux archives.
3. Ouvrez chaque projet dans IntelliJ en sélectionnant le bon répertoire (contenant à sa racine un répertoire caché **.idea**).
4. Configurez **si nécessaire** le JDK (uniquement s'il n'est pas trouvé) ; dans IntelliJ : File -> Project Structure -> Project -> SDK
5. Répondez aux questions dans l'ordre que vous voulez, toujours dans le bon projet

3.2 Partie 1 : Gestion du transport de marchandises

3.2.1 Introduction

Une chaîne de magasins possède sa propre flotte de camions afin de transporter ses marchandises entre ses différents entrepôts. Les marchandises sont transportées dans des caisses dont les dimensions sont toujours les mêmes mais dont le poids varie.

Les camions effectuent donc des trajets afin de transporter les caisses. Un camion ne peut pas faire deux trajets le même jour. De plus, si un camion doit faire deux trajets pendant deux jours consécutifs, il faut que la ville d'arrivée du premier trajet soit la ville de départ du deuxième trajet. De plus, il faut veiller à ne pas prévoir plus de caisses ou plus de poids pour le trajet que ce que le camion peut transporter.

Vous trouverez, dans le projet **exam-22-08-partie1**, 4 modules. Chaque module contient un package domaine dans lequel se trouvent 3 classes : `Caisse`, `Trajet` et `Camion`. Ces classes permettent déjà de créer des trajets et de gérer les caisses qu'ils comportent. Elles permettent également l'ajout d'un trajet à un camion (en vérifiant les conditions évoquées ci-dessus) et d'ajouter une palette pour un camion s'il y a moyen de trouver un trajet pour lequel on peut le faire. Pour plus de détails, consultez les commentaires indiqués dans les classes (les paramètres ne sont pas toujours testés mais vous ne devez pas ajouter les tests manquants).

Pour chaque question, vous allez devoir apporter des modifications à ces classes dans le module prévu à cette effet (le nom du module indique la question à laquelle il est destiné). Dans les trois premiers modules, vous trouverez aussi, dans la package main, une classe Main permettant de vérifier ce que vous faites. Pour chaque affichage effectué, il est indiqué l'affichage attendu. Ce n'est pas parce que vous avez toujours l'affichage attendu que vos classes sont d'office correctes.

3.2.2 Question 1 : Enuméré

En fait, il n'existe que trois gabarits (dépendant du nombre d'essieux) de camions. La charge maximale autorisée ainsi que le nombre de caisses qu'un camion peut contenir dépend de son gabarit.

Vous devez donc créer, dans la classe Camion, un énuméré Gabarit ayant trois constantes : `DEUX_ESSIEUX`, `TROIS_ESSIEUX`, `QUATRE_ESSIEUX`. Un gabarit comprendra également deux champs : `chargeMaximale` et `nbMaxCaisses`. Les camions avec deux essieux ont une charge maximale de 16400 kg et peuvent transporter maximum 20 caisses, ceux avec trois essieux ont une charge maximale de 22000 kg et peuvent transporter maximum 30 caisses et ceux ayant quatre essieux ont une charge maximale de 28600 kg et peuvent transporter au maximum 40 caisses.

Vous devez ensuite remplacer les attributs `nbMaxCaisses` et `chargeMaximal` de la classe Camion par un unique champ gabarit de type Gabarit et adapter le constructeur et les méthodes `ajouterTrajet` et `rechercherTrajet` de la classe Camion ainsi que la méthode `main` de la classe Main afin que tout fonctionne encore.

3.2.3 Question 2 : Collections

Dans la classe Camion, garder les trajets dans un Set n'est pas le plus adéquat pour gérer les problèmes de date. Remplacez le `Set<Trajet>` par une `SortedMap<LocalDate,Trajet>` qui, pour une date, retient comme valeur le trajet prévu à cette date. Vous devez ensuite adapter et optimiser les méthodes `ajouterTrajet, trouverTrajet` et `rechercherTrajet` de la classe `Camion`.

3.2.4 Question 3 : Stream

Complétez les méthodes `trouverTrajetsAvecPlaceRestante()` et `trouverDateTrajet(…)` de la classe `Camion` ainsi que la méthode `nombreDeCaissesParDateLimite()` de la classe `Trajet` en tenant compte des commentaires. **Vous devez écrire ces méthodes en une ligne (changer seulement ce qui est après le return) avec les streams.**

3.2.5 Question 4 : Tests

a) Vous devez tester la méthode `peutAjouter` de la classe Trajet avec JUnit. Pour cela, vous devez créer une classe de tests JUnit `TrajetTest` (dans le package domaine du répertoire tests du module 4-tests) dans laquelle vous devez faire les tests suivants :
    a. Vérifiez que la méthode `peutAjouter` lance une `IllegalArgumentException` lorsqu'on lui passe `null` en paramètre.

b. Vérifiez que la méthode `peutAjouter` renvoie `false` quand on lui passe une caisse dont la ville de départ ne correspond pas à celle du trajet.
c. Vérifiez que la méthode `peutAjouter` renvoie `true` lorsqu'on lui ajoute une caisse qui peut être ajouter au trajet.

b) Vous devez maintenant tester la méthode `ajouterTrajet` de la classe `Camion` en utilisant Mockito. Pour cela, vous devez créer une classe de tests JUnit `CamionTest`(dans le package domaine du répertoire tests du module 4-tests) dans laquelle vous devez faire le test suivant :
   a. Vérifiez que la méthode ajouterTrajet ne permet pas d'ajouter un trajet s'ils comportent trop de caisses par rapport à la capacité du camion.

3.3 Partie 2 : Serveur web

Dans cette partie, dans le projet **exam-22-08-partie2,** nous allons travailler sur le framework d'un serveur web très simple. Celui-ci ne répond pas réellement à des requêtes HTTP sur le réseau, mais permet de répondre à des requêtes HTTP en mode « texte » dans la console.

Par exemple, lorsque nous écrirons : `GET /hello HTTP/1.1` dans la console, le serveur devra exécuter une fonction qui permet de traiter cette requête. Nous pouvons décomposer cette requête en 3 parties :

1. La « méthode », à savoir `GET, PUT, POST, DELETE`…
2. L'URL, à savoir ici `/hello`
3. La version du protocole (nous n'en ferons rien ici)

Vous trouverez dans le package `resources` un certain nombre de classes. Chacune de ses classes contient un certain nombre de fonctions. Ce sont ces fonctions qui vont répondre aux requêtes qui sont envoyées au serveur.

Pour savoir quelle fonction exécuter pour chaque requête, ces méthodes sont « préfixées » par un certain nombre d'annotations. Nous en utiliserons 2 ici :

```
1. @Path("/hello")
2. @GET
```

La 1e permet de définir quelle est l'URL associée avec la fonction. La 2e permet de définir quelle méthode est associée à la fonction. Voici un exemple de fonction qui répond à la requête en exemple ci-dessus.

```
@Path("/hello")
@GET
public void getHello() {
    System.out.println("Hello World !") ;
}
```

Le nom de la fonction n'a aucune importance.

3.3.1 Question 1 : getMethod() - introspection, annotations

Complétez la méthode `getMethod(Request request)` qui reçoit en paramètre la requête et qui va chercher dans toutes les classes du package `resources` (utilisez la méthode "getResourcesClasses()" pour cela) s'il existe une méthode qui est capable de répondre à la requête :

- La méthode doit avoir une annotation @PATH avec une valeur qui correspond à l'URL dans l'objet `Request`.

- La méthode doit avoir une annotation `@GET` ou `@POST` en fonction de la méthode stockée dans la `Request`. Attention, pour vérifier la méthode dans la `Request`, il faut être insensible à la casse. On doit donc pouvoir avoir `get, GET, Get`…
- La méthode doit être `public`
- La méthode ne peut pas être `static`

Si une méthode respecte ces critères, elle peut être renvoyée.

Si aucune méthode ne correspond, il faut renvoyer `null`.

Si plusieurs méthodes correspondent, la première que vous trouvez doit être renvoyée.

Astuce : pour récupérer la valeur passée en paramètre d'une annotation, on peut récupérer directement l'annotation grâce à la ligne suivante : `Path path = m.getAnnotation(Path.class);` Ensuite on peut récupérer la valeur de l'annotation (cfr. Javadoc)

3.3.2 Question 2 : Threads

Cette application fonctionnera à terme dans un contexte Web. Comme vous vous en doutez, il faut donc pouvoir traiter un nombre important de requêtes, et idéalement en parallèle, pour tirer parti des architectures multicœurs de nos processeurs.

Modifiez votre application pour que chaque appel à la méthode `handleRequest(request);` se fasse dans un nouveau Thread.

Astuce : nous vous conseillons de définir le Thread en classe interne pour faciliter les accès aux méthodes existantes.

3.3.3 Question 3 : Les interfaces, factory et injection de dépendances

Comme vous pouvez le constater, le package `domain` contient une classe dont l'implémentation ne devrait pas être visible aux autres packages.

Remédiez à ce problème en **plaçant des interfaces aux bons endroits, cachant les implémentations et en proposant une `factory` qui va distribuer les objets du domaine**.

Cette `factory`, ainsi que tous les autres objets de service (s'il y en a) devront être injectés par injection de dépendances à ceux qui en ont besoin. Le `main` s'occupera de cela en créant les objets de service et en les injectant. Inutile de procéder par introspection, faites la version "simple" avec des `new`.

Faites également **attention à l'encapsulation** (visibilité des classes/interfaces).