Haute École Léonard de Vinci

Haute école Leonard de Vinci — Session de janvier 2023

Examen de Programmation Java : avancé

Titulaire(s) : Stéphanie Ferneeuw, Laurent Leleux, Grégory Seront, Raphaël Baroni

Année(s) d'études : Bloc 2

Durée de l'examen : 3 h ; pas de sortie durant les 60 premières minutes

Modalités : théorie, solutions et JAVADOC accessible, internet non autorisé !

/!\ Lisez les consignes avant de démarrer l'examen /!\

Consignes

L'examen comporte 2 parties, chacune calibrée pour nécessiter 1h15 minutes de travail. Chaque partie est découpée en plusieurs questions. Vous avez donc 30 minutes pour lire cet énoncé, vous imprégner des projets, remettre votre travail final…

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 que vous avez dézippée contient :
    - Les projets sur lesquels vous allez travailler
    - Les slides de COO
    - Les fiches de théorie de AJ
    - Les solutions des fiches de l'année
    - Le manuel IntelliJ (au cas où)
    - L'énoncé que vous êtes en train de lire
2. La JavaDoc est disponible dans C:\Progs\Java\...
3. Ouvrez le projet dans IntelliJ.
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
6. Une fois terminé, faites une archives ZIP **UNIQUEMENT** pour chacun de vos deux projets IntelliJ finaux.
    - Nom des deux archives : **examen-partie1-NOM-PRENOM.zip, examen-partie2-NOM-PRENOM.zip**
7. L'archive doit être à la racine du lecteur réseau (U:).
8. Vérifiez une dernière fois que vos deux archives contiennent bien vos projets modifiés.
9. Ecrivez votre Nom et Prénom sur votre feuille de login, et **remettez-la au professeur**.

Partie 1 : Gestion de la production et de la vente d'œufs (12 points)

Une ferme possède plusieurs poulaillers. Dans chaque poulailler peut se trouver un lot actuel de poules. Pour ajouter un nouveau lot à un poulailler, il est vérifié que le poulailler est vide (qu'il n'y a pas de lot actuellement dedant) et que le lot à ajouter n'a pas déjà été affecté à un poulailler.

Au niveau d'un lot, il est possible d'initialiser les ventes d'œufs ou d'en enregistrer une.

Vous trouverez, dans le projet **examen-partie1**, 4 modules. Chaque module contient un package **domain** dans lequel se trouvent 6 classes et un package **main** contenant une seule classe. Ces classes permettent déjà de gérer des lots de poules et des ventes d'œufs. Pour plus de détails, consultez le code des classes et les commentaires indiqués dans ces classes.

Pour chaque question, vous allez devoir apporter des modifications à ces classes dans le module prévu à cet 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 **GestionPoulaillers** 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.

Question 1 : Enuméré

Répondez à cette question dans le module **partie1-q1** de votre projet **exam-partie1**.

On souhaite identifier tous les acheteurs dans un énuméré qui devra se trouver imbriqué dans la classe **Ferme**.

L'énuméré **Acheteur** devra reprendre ces 5 constantes : **DELHAIZE**, **CARREFOUR**, **JETJ**, **ROSEMARIE**, et **PARTICULIER**.

Un **Acheteur** comprendra deux champs : **type** et **nom**.
Pour identifier les valeurs de ces deux champs pour chacune de 5 constantes, veuillez consulter la classe **GestionPoulaillers** pour les trouver.
Pour un **type** dont la valeur vaut « Particulier », le **nom** prendra d'office la valeur « Particulier » elle aussi.

Pour que le **toString** de la classe **VenteOeufs** puisse être adapté, vous allez devoir ajouter une méthode à **Acheteur**, un **getter** ou un **toString**, selon votre choix.

Vous devez ensuite :

- remplacer les attributs **`typeAcheteur`** et **`nomAcheteur`** de la classe **VenteOeufs** par un unique champ **acheteur** de type **Acheteur**
- adapter les deux constructeurs de **VenteOeufs** : il ne doit se trouver plus qu'un seul constructeur initialisant notamment une instance d'**Acheteur**.
- adapter les méthodes de la classe **VenteOeufs** qui faisaient appels à **typeAcheteur** et **nomAcheteur**. Ne pas oublier **equals**, **hashCode** et le **toString** !
- adapter **obtenirVentesGroupeesParAcheteurTrieesParDate** de la classe **Ferme**.
- adapter la classe **GestionPoulailler**s afin qu'elle fonctionne toujours et utilise l'énuméré créée !

Faites attention à ce que les résultats après changement (« Ventes par acheteur (q1 et q2) ») correspondent aux résultats avant changement.

Question 2 : Collections

Répondez à cette question dans le module **partie1-q2** de votre projet **exam-partie1**.

Collection non triée

Commencez par changer le type de la collection **ventes** au sein de la classe **Lot** vers un **Set**. En effet, une liste n'est pas adéquate lorsque l'on souhaite éviter les doublons (« duplicate » en anglais) au sein d'une collection non triée. Modifiez le code partout où **ventes** est appelé, y compris dans la classe **Ferme** où **getVentes** est utilisée, afin d'utiliser un **Set** n'autorisant pas les doublons.

Attention à optimiser la méthode **intialiserVentes** de la classe **Lot** en fonction du changement de collection.

Collection triée

Dans la méthode **obtenirVentesGroupeesParAcheteurTrieesParDate** de la classe **Ferme**, garder les ventes d'œufs par acheteur au sein de listes n'est pas le plus adéquat pour gérer le tri par date.

Actuellement, la méthode **ajouterVenteTrieePourAcheteur** permet d'ajouter un élément en respectant le tri suivant :

- par ordre des ventes les plus récentes aux plus anciennes ;
- lorsque plusieurs ventes sont faites le même jour, par ordre des nombres d'œufs vendus du plus grand au plus petit.

Veuillez modifier la collection **ventesGroupeesParAcheteurTrieesParDate** (de type **Map<String, List<VenteOeufs>**) vers un nouveau type permettant d'automatiser le tri des ventes par acheteur : **Map<String, SortedSet<VenteOeufs>**. En utilisant un **SortedSet**, vous devez optimiser la méthode **ventesGroupeesParAcheteurTrieesParDate** afin de ne plus faire appel à la méthode **ajouterVenteTrieePourAcheteur**.

Faites attention à ce que les résultats après changement (« Ventes par acheteur (q1 et q2) ») correspondent aux résultats avant changement.

Question 3 : Stream

Répondez à cette question dans le module **partie1-q3** de votre projet **exam-partie1**.

Complétez les méthodes **`trouverVenteLaPlusGrande`** et **`trouverVentesPourNomAcheteur`** **`calculerOeufsVendusParNomAcheteur`** de la classe **`Ferme`** en tenant compte des commentaires. **Vous devez écrire ces méthodes en une ligne (changer seulement ce qui est après le return) avec les streams.**

Question 4 : Tests

Répondez à cette question dans le module **partie1-q4** de votre projet **exam-partie1**.

Vous devez tester le constructeur et une méthode de la classe **Lot** avec JUnit. Pour cela, vous devez créer une classe de tests JUnit **`LotTest`** (dans le package **domain** du répertoire tests du module **partie1-q4**) dans laquelle vous devez faire les tests suivants :

- q4.1 : vérifiez que si vous appelez **le constructeur** en passant une quantité de volailles plus petite que 1, une **IllegalArgumentException** est lancée.

- q4.2 : Vérifiez que la méthode **`signalerAffectation`** renvoie **`true`** quand on l'appelle sur un lot dont on n'a pas encore signalé d'affectation.
- q4.3 : Vérifiez que la méthode **`signalerAffectation`** renvoie **`false`** quand on l'appelle sur un lot pour lequel on a déjà signalé une affectation.

Vous devez maintenant réaliser un test unitaire de la méthode **`ajouterLot`** de la classe **`Poulailler`** en utilisant **Mockito**. Pour cela, vous devez créer une classe de tests JUnit **`PoulaillerTest`** (dans le package **domain** du répertoire tests du module partie1-4) dans laquelle vous devez faire le test suivant :

- q4.4 : Dans le setup de votre test, veuillez notamment créer un poulailler ne contenant aucun lot. Veuillez ensuite vérifier, quand vous tentez d'ajouter un lot pour lequel on a signalé son affectation, que l'ajout échoue. Pensez à vérifier aussi, une fois l'échec de la tentative d'ajout d'un lot, qu'aucun lot ne soit présent au niveau du poulailler.

Partie 2 : Framework de tests unitaires (8 points)

Dans cette partie, nous allons redévelopper un framework pour l'exécution de tests unitaires du même type que JUnit. Nous allons clairement ré-inventer la roue, mais ça va nous permettre de démystifier une technologie pas si complexe que ça...

Le framework va permettre d'exécuter des classes de test qui doivent être placées dans le package `be.vinci.tests` de notre projet. Comme dans JUnit, les méthodes de test peuvent avoir n'importe quel nom, mais doivent être annotées par `@Test`.

Ouvrez le projet **examen-partie2.**

Vous trouverez dans le package `be.vinci.tests` un certain nombre de classes de test pour vérifier que tout fonctionne comme il faut.

Prenez le temps de comprendre le principe de l'application avant de commencer.

Dans les grosses lignes, pour comprendre le flux global de l'application :

- On commence par instancier un `TestClassRunner`.
- On lui demande d'exécuter tous les tests. Pour cela :
  - Il parcourt les classes dans le package tests
  - Pour chaque classe, il instancie un objet du domaine `TestClass`
  - Il lui demande de charger la classe de tests
  - Il lui demande d'exécuter les tests de la classe
- Pour charger une classe de test
  - On cherche dans la classe toutes les méthodes qui respectent les critères d'une méthode de test (annotation `@Test`, public...)
  - Pour chaque méthode trouvée, on instancie un objet `TestMethod()` qu'on ajoute à la liste des tests de la classe `TestClass`
- Pour exécuter tous les tests d'une classe
  - On instancie un objet de la classe de test (considérons qu'il y a toujours un constructeur sans paramètres)
  - On parcourt toutes les `TestMethod` et on les exécute une à une en donnant l'objet précédemment créé en paramètre.

/!\ Pour cette partie, toutes les questions doivent être répondues dans le même projet. /!\

Question 1 : Introspection, annotations

Complétez les 3 méthodes suivantes :

- La méthode `run(Object o)` de la classe `TestMethod`
- La méthode `loadFromClass(Class classToLoad)` de la classe `TestClass`
- La méthode `runAllTests()` de la classe `TestClass`

Consultez bien la javadoc des méthodes à écrire pour comprendre ce qu'elles doivent faire. Si nécessaire, n'hésitez pas à relire l'explication du flux global ci-dessus.

Astuce : pour vérifier si une méthode renvoie void, vérifiez si le type de retour est égal à void.class. **Attention, pas Void.class avec une majuscule !**

Question 2 : Threads

Exécuter des tests peut prendre un certain temps. C'est pourquoi on aimerait paralléliser l'exécution des classes de test. On ne souhaite pas paralléliser l'exécution des tests au sein d'une même classe, mais plutôt les classes entre elles.

Par exemple, l'exécution de la classe de test `FakeTest1` sera en parallèle avec la classe `FakeTest2`. Mais le thread de la classe `FakeTest1` exécutera séquentiellement tous les tests de la classe, pareil pour `FakeTest2`.

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

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

Question 3 : Les interfaces, factory et injection de dépendances

Comme vous pouvez le constater, le package `domain` contient des classes 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).