Java Avancé

Arelier 5 : JUnits

Table des matières

1 Objectifs

ID	Objectifs
AJ01	Être capable d'écrire des tests avec JUnit 5

2 Concepts

2.1 JUnits

La théorie nécessaire se trouve dans le fichier JUnit5.pdf

3 Exercices

3.1 Introduction

Afin de gérer l'uniformité de ses prix dans l'ensemble de ses magasins, une chaîne commerciale a développé une application. Les prix varient au cours du temps et en fonction de la quantité achetée. À cette fin, elle a déjà implémenté les classes suivantes : `Promo`, `Prix`, `Produit` et `ListeProduits`.

La classe `Promo` est un énuméré reprenant les différents types de promotions applicables à un produit.

La classe `Prix` représente les prix appliqués selon la quantité achetée à un moment donné. De ce fait, elle possède éventuellement un `TypePromo` et, si c'est le cas, une valeur réelle strictement positive `valeurPromo` permettant de quantifier la promotion. Les prix unitaires selon la quantité achetée sont eux gardés dans une `SortedMap<Integer, Double>` dont la clé représente la quantité minimale à acheter pour que le prix unitaire fourni dans la valeur soit appliqué. Par exemple, on pourrait définir que le prix unitaire est de 15 euros à partir d'une unité et de 12 euros à partir de 10 unités. Il est interdit d'acheter pour une quantité qui est inférieure à la plus petite quantité renseignée dans la `SortedMap`.

Un produit possède un nom, une marque et un rayon. Deux produits ayant les mêmes valeurs pour ces attributs sont considérés comme étant identiques. Pour garder les prix appliqués au cours du temps, un produit garde aussi l'historique des prix dans une `SortedMap<LocalDate, Prix>` trié par ordre décroissant de date. La clé correspond la date à partie de laquelle le prix qui est mis comme valeur est d'application.

Finalement, la classe `ListeProduit` garde la liste des produits triés par ordre croissant de leur nom, puis de leur marque et enfin de leur rayon.

Pour voir les méthodes existantes dans ces différentes classes et avoir plus précisément ce qu'elles font, consultez directement les classes fournies.

3.2 Consignes

Importer le projet :

Récupérez le projet aj_atelier05 sur moodle, dézippez-le. Faites un clic droit sur le projet et choisissez Open Folder as IntelliJ Idea project. Il faudra éventuellement ajouter la SDK. Pour cela :

- Allez dans File -> Project Structure…

- Sélectionnez Project dans Project Settings.
- Sélectionnez votre SDK dans le menu déroulant situé dans la section Project SDK et cliquez sur Apply et ensuite sur OK.

Vérifiez que tout fonctionne bien en exécutant la class `Main` du package `main`.

Créer un dossier pour les tests :

- Créez, dans votre projet, un nouveau dossier intitulé tests (clic droit sur le projet et choisir New -> Directory)
- Faites un clic droit sur le dossier tests et sélectionnez Mark Directory as -> Test Sources Root

Créer une classe de test :

1. Ouvrir la classe que vous voulez tester (essayez avec la classe `Prix`)
2. Se positionner sur le nom de la classe au sein du code, faire un clic droit et choisir « Show Context Action » (raccourci ALT + ENTER)
3. Choisir « Create Test »
4. Dans Testing Library, il faut choisir « JUnit5 ». Si le message « JUnit5 library not found in the module » apparaît, cliquez sur Fix et ensuite sur OK.
5. Vérifiez que le package de destination indiqué est bien le même que le package de la classe à tester (package `domaine` pour la classe `Prix`).
6. Dans Generate, sélectionnez setUp/@Before.
7. Sélectionnez ensuite les méthodes que vous voulez tester (toutes sauf `toString` pour la classe `Prix`) et cliquez sur OK. Cela devrait générer une classe de test (`PrixTest` pour la classe de tests de la classe `Prix`) dans le package `domaine` du répertoire tests

3.3 Tests de la classe Prix

Préparation du test

Déclarez 3 attributs (`prixAucune`, `prixPub` et `prixSolde`) qui permettront de faire tous les tests. Ces trois attributs de type `Prix` seront instanciés dans la méthode annotée `@BeforeEach`. Il faut un prix (`prixAucune`) sans promo, un autre (`prixPub`) avec `Promo.PUB` comme `Promo` et enfin un dernier (`prixSolde`) avec `Promo.SOLDE` comme `Promo`. Pour ces deux derniers, mettez les valeurs que vous souhaitez pour autant qu'elles soient strictement positives. Dans `prixAucune` définissez 2 prix : l'un pour signifier qu'à partir d'une unité, le prix est de 20 euros et l'autre pour signifier qu'à partir de 10 unités (quantité), le prix devient 10 euros. Dans `prixPub`, signifiez qu'à partir de 3 unités, le prix est de 15 euros.

N'oubliez pas que chaque test doit se trouver dans une méthode annotée. Nous vous recommandons d'utiliser l'annotation `@DisplayName` afin d'avoir un affichage clair sur ce qui est testé dans la méthode. Vous pouvez aussi faire plusieurs méthodes de tests pour un même point ci-dessous.

Après avoir écrit une méthode de test, exécutez-là et vérifiez que le test réussit !

Test du constructeur

1. Vérifier que le constructeur lance une `IllegalArgumentException` si la promo passée en paramètre est `null`.
2. Vérifier que le constructeur lance une `IllegalArgumentException` si la valeur passée en paramètre est < 0.

Test des getters (Utilisez les attributs définis)

1. Vérifier que la valeur de la promo est initialisée à 0 lors de l'instanciation d'un prix au moyen du constructeur sans paramètre.

2. Vérifier que la valeur de la promo correspond bien à celle passée en paramètre du constructeur.
3. Vérifier que le type de la promo est `null` lors de l'instanciation d'un prix au moyen du constructeur sans paramètre.
4. Vérifier que le type de la promo correspond bien à celle passée en paramètre du constructeur.

Test de definirPrix

1. Vérifier que la méthode `definirPrix` lance une `IllegalArgumentException` si le paramètre quantité est 0 ou négatif.
2. Vérifier que la méthode `definirPrix` lance une `IllegalArgumentException` si le paramètre valeur est 0 ou négatif.
3. Définir un prix de 6 euros à partir de 10 unités pour l'attribut `prixAucune` et vérifier que l'ancien prix a été remplacé.

Test de getPrix

1. Vérifier que la méthode lance une `IllegalArgumentException` si le paramètre quantité est négatif ou nul.
2. Testez les prix renvoyés par la méthode `getPrix` pour l'attribut `prixAucune` : faire le test pour 1 unité, 5 unités, 9 unités, 10 unités, 15 unités, 20 unités et 25 unités.
3. Testez qu'une `QuantiteNonAutoriseeException` est lancée si vous demandez le prix pour 2 unités pour l'attribut `prixPub`.
4. Testez qu'une `QuantiteNonAutoriseeException` est lancée si vous demandez le prix pour 1 unité pour l'attribut `prixSolde`.

3.4 Tests de la classe Produit

Créez la classe `ProduitTest` de façon similaire à `PrixTest`. Ajoutez-y 3 attributs de type Prix initialisez-les comme dans `PrixTest` (copier-coller → juste une fois !). Ajoutez-y aussi 2 attributs `Produits` et dans l'un deux insérez les 3 prix. Attention, on ne peut pas ajouter deux prix à la même date !

Test du constructeur et des getters

1. Testez que le constructeur de la classe Produit lance bien une `IllegalArgumentException` en cas de paramètre invalide (paramètre `null` ou chaîne de caractères constituées uniquement de « blancs »)
2. Testez que les valeurs passées en paramètre lors de l'instanciation d'un produit correspondent bien à celles renvoyées par les getters.

Test des prix (ajouterPrix et getPrix)

1. Vérifier que la méthode `ajouterPrix` lance une `IllegalArgumentException` si un des paramètres est `null`.
2. Vérifier que la méthode `ajouterPrix` lance une `DateDejaPresenteException` si la date est déjà présente.
3. Vérifier que la méthode `ajouterPrix` ajoute effectivement un prix pour une date donnée et vérifier ce prix avec le `getPrix`.
4. Vérifier que lorsqu'on demande un prix pour une date antérieure à la définition d'un prix l'exception `PrixNonDisponibleException` est lancée.
5. Vérifier que lorsqu'on demande un prix pour un produit qui n'en n'a pas, l'exception `PrixNonDisponibleException` est lancée.
6. Vérifier que lorsqu'on demande un prix pour une date comprise entre deux dates pour lesquelles le prix a été défini, c'est bien celui de la date antérieure qui a été renvoyé.

Test des méthodes equals et hashcode

1. Vérifier que la méthode `equals` fonctionne pour 2 instances de `Produit` différentes mais qui ont les même `marque`, `nom` et `rayon`.
2. Vérifier que la méthode `equals` renvoie faux pour deux produits ayant la même marque et le même rayon mais ayant des noms différents.
3. Vérifier que la méthode `equals` renvoie faux pour deux produits ayant le même nom et le même rayon mais ayant des marques différentes.
4. Vérifier que la méthode `equals` renvoie faux pour deux produits ayant le même nom et la même marque mais ayant des rayons différents.
5. Vérifier que la méthode `hashCode` renvoie bien la même valeur pour 2 instances de `Produit` différentes mais qui ont les même `marque`, `nom` et `rayon`.

3.5 Challenge optionnel

Créez la classe `ListeProduitsTest` et testez-y la classe `ListeProduits` du package `usecase`. Pour chaque méthode, pensez à bien tester tous les cas où la méthode doit échouer (quand elle doit renvoyer faux ou lancer une exception) comme cela a été fait, par exemple, pour le test des prix de la classe `Produit`. Dans le cas où la méthode doit réussir, pensez à vérifier que ce qui devait être fait l'a bien été (par exemple, vérifier la présence du produit après l'avoir ajouté). En particulier, pour la méthode permettant d'ajouter un prix à un produit ou celle permettant, pensez à tester que le travail est bien fait sur le bon produit (celui stocké et non celui passé en paramètre) en passant un paramètre un produit égal mais de référence différente à celui stocké. De même pour la méthode permettant de retrouver le prix.