Gestion d'erreurs dans le paquetage `tec`

La réalisation va utiliser le mécanisme des exceptions pour effectuer la gestion d'erreurs.

La première partie reprend l'instanciation de Jauge pour introduire le mécanisme des exceptions (levée/capture).

La deuxième partie vous demande de prendre en compte quelques cas d'erreurs dans le paquetage tec et de tester la levée d'exception dans ces cas d'erreurs.

La dernière partie change la réalisation de la classe Autobus. Le stockage des passagers va utiliser les classes du « Java Collections Framework ».

Chaque tandem effectue ce travail sur une version du projet.

Table des matières

1 L'instanciation de Jauge

Il y a un cas particulier pour l'instanciation de la classe Jauge. Dans TestJauge, on peut écrire le test suivant :

```java
public void testCasLimite() {
  Jauge inverse = null;

  inverse = new Jauge(-42, 10);
  //Quel comportement faut-il vérifier ?
}
```

La documentation de Jauge est imprécise. Nous allons considérer que vigieMax < 0 est invalide.

Dans le cas d'une instanciation avec des paramètres invalides, il est obligatoire d'indiquer une erreur plutôt que de ne rien faire ou d'essayer de mettre l'objet dans un état cohérent. L'instanciation doit donc échouer.

Pour cela, nous allons utiliser le mécanisme des exceptions.

1.1 Lever une exception

Dans le constructeur de la classe Jauge, lever l'exception (non contrôlée) IllegalArgumentException dans le cas d'un paramètre vigieMax < 0.

Exécuter la classe de test pour vérifier la lévee de l'exception.

1.2 Capturer une exception

Au lieu d'effectuer une vérification visuelle, nous allons réécrire le test testExceptionCasLimite() pour vérifier avec une assertion si l'exception est bien levée. Pour cela, il faut capturer l'exception dans le test.

Le traitement à réaliser est le suivant :

— Si l'exception est capturée, c'est-à-dire que l'exception a été levée. Le test ne doit rien afficher puisque c'est le comportement voulu.

— Si l'exception n'est pas capturée, c'est-à-dire que l'exception n'a pas été levée, le test doit échouer. À la fin du bloc try ajouter l'assertion assert false : "Exception non levee";.

Quelle est la valeur de la variable inverse dans la partie catch ?

2 Paquetage `tec`

Pour la gestion d'erreurs du paquetage, nous utilisons à la fois des exceptions contrôlées et des exceptions non contrôlées :

— Les exceptions contrôlées sont utilisées à l'extérieur du paquetage : la méthode monterDans() utilisée par le client propage/lève une exception contrôlée. Cette exception contrôlée indique une erreur provoquée par le client.

— Les exceptions non contrôlées sont utilisées dans le paquetage : les méthodes internes au paquetage comme monteeDemander*() propagent/lèvent des exceptions non contrôlées.

Il n'est pas obligatoire d'indiquer la propagation d'une exception non contrôlée dans le prototype des méthodes. Il est, ainsi, possible d'ajouter/modifier la mise en œuvre de la gestion d'erreur sans changer le prototype des méthodes internes au paquetage.

Après le traitement des cas d'erreur, vous allez développer des tests vérifiant la levée de ces exceptions.

2.1 Cas d'erreurs utilisant des exceptions non contrôlées

Traiter les cas suivants :

— L'instanciation d'un usager avec une valeur négative pour la destination lève l'exception java.lang.IllegalArgumentException.

— L'instanciation d'un transport avec une valeur négative pour le nombre de places assises et/ou debout lève l'exception java.lang.IllegalArgumentException.

— La montée d'un usager avec les méthodes monteeDemander*() lève l'exception java.lang.IllegalStateExcep si l'instance est déjà stockée dans le transport.

2.2 L'exception contrôlée.

Définir dans le paquetage tec, l'exception contrôlée TecException.

Pour correspondre à une exception contrôlée, de quelle classe va-t-elle hériter ?

L'instanciation de la classe TecException doit pouvoir se faire de deux manières :

— en précisant un message d'erreur,

— ou en précisant une autre exception.

En vous servant de la documentation de sa classe de base, donner les deux constructeurs de la classe TecException qui permettent ces instanciations.

Dans l'interface Usager, modifier le prototype de la méthode monterDans() pour spécifier la propagation de l'exception TecException.

Le prototype des méthodes dans les sous-types de cette interface n'a pas besoin d'être modifié. Tant que le code redéfini ne lève pas cette exception contrôlée, il n'est pas nécessaire de préciser la clause de propagation.

Compiler les fichiers du répertoire `src`.

La compilation de la classe Simple provoque maintenant une erreur car le prototype de la méthode monterDans() utilisée à travers l'interface Usager indique la propagation d'une exception contrôlée.

Dans le code de Simple, le compilateur vous indique de capturer cette exception ou indiquer sa propagation dans le prototype de la méthode main(). Le plus rapide est de spécifier sa propagation dans la méthode main().

Compiler et exécuter Simple.

2.3 Cas d'erreurs dans la méthode monterDans()

Dans la méthode monterDans() de la classe PassagerAbstrait, nous allons ajouter la levée de l'exception contrôlée TecException.

Commencer par ajouter la propagation de l'exception contrôlée TecException dans le prototype de la méthode monterDans().

Compiler les tests unitaires. En cas d'erreur dans les tests, il faut sûrement indiquer la propagation de cette exception dans les méthodes de test qui contiennent un appel à monterDans().

Puis, traiter les cas suivants :

— L'échec de la conversion de type entre Transport et Vehicule (dans la méthode monterDans()) lève l'exception tec.TecException.
La vérification du type se fait à l'aide de l'opérateur instanceof.

— L'exception non contrôlée java.lang.IllegalStateException est capturée dans la méthode monterDans() avec un bloc try/catch autour de la méthode choixPlaceMontee(). Le code de capture lève l'exception TecException. L'instanciation de la TecException se fait avec comme paramètre l'instance de la classe IllegalStateException. De cette manière, l'exception d'origine est conservée (« the specified cause »). Et, le message d'erreur affiché est celui défini par l'instance de IllegalStateException (voir la documentation des constructeurs de la classe java.lang.Throwable).

2.4 Tester les cas d'erreurs

Le test des exceptions se fait sur le modèle du test de l'exception de l'instanciation de Jauge.

Pour les exceptions non contrôlées, écrire les tests unitaires dans les classes de tests de TestPassagerAbstrait et TestAutobus

Pour les tests unitaires de ces exceptions, il est peut-être nécessaire de définir d'autres classes faussaires.

Pour l'exception contrôlée, écrire un autre test d'intégration (en prenant modèle sur la classe Simple) qui provoque les deux erreurs et capture l'exception dans chaque cas.

3 La réalisation de la classe Autobus

L'objectif est de remplacer le tableau de passager dans la classe Autobus par une classe « conteneur » du « framework » collection de l'A.P.I. standard inclus dans la paquetage java.util.

3.1 « Java Collection Framework »

Pour la documentation, vous pouvez consultez les liens suivants :

— la page officielle du framework
— les diagrammes de classe fournis sur la page J.C.F. de wikipédia ;
— le site `http://www.docjar.com` qui donne accès à la documentation officielle mais aussi au code source. Lien sur la documentation du paquetage java.util
— et le dépôt officiel des sources openjdk.

Ce framework est défini à partir de deux catégories de type « abstrait » :

— la structure de données (ou conteneur) avec les interfaces : Collection , Map, List , Set, ... Pour avoir une idée des opérations, se référer à la documentation des interfaces Collection et Map.
— le parcours de la structure de données (ou itérateur) avec l'interface : Iterator . Pour avoir une idée des opérations, se référer à la documentation de l'interface Iterator .

Vous trouverez aussi des classes abstraites (faciles à repérer : leurs noms commencent par Abstract) et des classes concrètes (par exemple ArrayList) qui fournissent une réalisation particulière.

Le mécanisme des types paramétrées est utilisé pour permettre à l'utilisateur de spécifier le type des éléments stocké dans un conteneur.

3.2 Remaniement de la classe Autobus

La réalisation de la classe Autobus utilise un conteneur de type List : ArrayList ou LinkedList . Chaque tandem effectue ce travail avec un conteneur différent.

Remanier la classe Autobus :

— Le type des éléments stockés dans le conteneur est Passager.
— L'ajout et la suppression d'un passager utilise les méthodes add() et remove() définies dans l'interface Collection .
— Le parcours du conteneur dans la méthode allerArretSuivant () se fait par un itérateur en utilisant la boucle for simplifiée (« enhanced for-loop »).

Compiler et exécuter les tests unitaires d'Autobus.

3.2.1 Test d'intégration : coordinateur, nous avons un problème !

Sur le modèle de la classe Simple, écrire un test d'intégration qui reprend le premier scénario avec trois passagers standard : Kaylee destination arrêt numéro 4, Jayne destination arrêt numéro 4 et Inara destination arrêt numéro 5. Ces trois passagers montent dans le transport au premier arrêt.

L'exécution de ce scénario d'intégration se termine par l'erreur [1] :

```
Exception in thread "main"
          java.util.ConcurrentModificationException
```

Cette exception est levée par la méthode next() de l'itérateur. Elle n'apparaît pas dans le prototype de cette méthode car c'est une exception non contrôlée. Pour avoir une idée du problème, il faut consulter la documentation de l'exception et des classes ArrayList ou LinkedList .

Rappeler le problème entre le parcours et la suppression.

Dans la démarche des tests du développeur, une erreur détectée à l'exécution doit être mise en évidence par un test unitaire. Cela permet d'enrichir le jeu de tests et d'indiquer la correction de l'erreur.

1. Si vous n'avez pas d'erreur vous n'utilisez pas d'itérateur et l'exécution ne donne pas le bon résultat.

Écrire ce test unitaire et pour l'instant ce test doit échouer puisque le problème n'est pas corrigé.

3.3 Quelques solutions

Essayons d'envisager les manières d'éviter cette erreur :

1. Remplacer l'instance supprimée par `null` comme dans le code d'origine.
2. Utiliser un itérateur sur une copie de la collection contenant les passagers.

   > Expliquer pourquoi cette copie doit être nécessairement une copie superficielle.

   Cette copie peut-être effectuée par les méthodes `clone()`, `toArray()` ou bien le constructeur de la collection qui prend en paramètre une collection.

   > Pour ces trois manières s'agit-il d'une copie superficielle ou profonde ?

   > Quelle est l'inconvénient d'effectuer cette copie ?

   Réaliser cette solution dans la version sans `fabrique`.
3. Utiliser la méthode `remove()` de l'itérateur pour supprimer le passager dans la méthode `demanderSortie()`.

   > Quel problème cela pose-t-il ? Comment faudrait-il modifier le code ? Conclusion ?

4. > Quelle autre solution pouvez-vous proposer qui évite l'inconvénient de la copie systématique ?

   Réaliser cette solution dans la version avec `fabrique`.

3.4 Boutez vos neurones

En partant du source disponible en ligne des classes `java.util.ArrayList` répondre à la questions suivante :

La classe correspondant à un itérateur sur une instance d'`ArrayList` est définie à l'intérieur de la classe `ArrayList`. Cela permet à une instance de l'itérateur d'accéder aux attributs de l'instance d'`ArrayList`.

Pour répondre aux questions suivantes, vous devez chercher, dans le source de la classe `java.util.ArrayList`, la définition de l'itérateur : la classe sous-type de l'interface `Iterator`.

> Expliquer comment la réalisation de l'itérateur détecte le problème qui provoque la levée de l'exception `ConcurrentModificationException`.

Expliquer comment le code de la méthode remove() de l'itérateur résout le problème de suppression. Conclusion.

4 Automatiser le lancement des tests

Suivant notre standard de codage, le nom d'une méthode de test commence par la chaine test. L'utilisation de l'introspection permet de factoriser l'appel des méthodes de test dans les classes de test.

La classe LancerTests effectue l'appel de tous les méthodes de test sur un ensemble de classe de test. Elle prend en arguments le nom complet des classes de test et appel tous les méthodes commençant par test sur une instance de cette classe de test.

La réalisation de la classe LancerTests définit les deux méthodes de classe suivantes :

1. static private void lancer(Class c) throws Exception à partir de l'instance passée en paramètre :
   — Cherche toutes les méthodes publiques définies dans cette classe (getMethods()).
   — Déclenche chaque méthode de test (invoke()) sur une nouvelle instance (newInstance()) de la classe de test passée en paramètre.
   — Affiche le caractère '.' et comptabilise le nombre de tests valides dans cette classe.
   — à la fin, affiche OK avec le nombre de tests valides.
2. static public int main(String[] args) throws Exception
   — Vérifie la présence de l'option `-ea`.
   — Pour chaque nom du tableau args, charge dans la machine virtuelle la classe de test à partir de son nom complet (forName).
   — Appel la méthode lancer avec comme paramètre la classe de test chargé.

Excécuter les tests et supprimer les méthodes main des classes de test.

Échec d'une assertion Le message d'erreur en cas d'échec d'une assertion n'est plus aussi directe. L'exception reportée dans la méthode main n'est plus java.lang.AssertionError mais java.lang.reflect.InvocationTargetException. En effet, l'échec de l'assertion provoque la terminaison anormale de la méthode invoke() d'où les exceptions supplémentaires levées.

Le message d'erreur lié à l'assertion se trouve dans la partie « Caused by » de l'affichage.

Pour retrouver un affichage plus simple, dans la méthode lancer(Class c) :

1. capturer l'exception java.lang.reflect.InvocationTargetException ;
2. et lever l'exception qui est la cause de java.lang.reflect.InvocationTargetException.