Info – Bloc 1 – UE09 : Programmation Orientée Objet

Laboratoire 6

Durée prévue : 4 heures

Objectifs visés

À la fin du laboratoire, les étudiants seront capables de définir et utiliser des classes implémentant des interfaces en veillant à masquer les détails internes.

Concrètement, les étudiants implémenteront les interfaces `Iterable<T>` et `Iterator<T>`.

Pour faire ce laboratoire, les étudiants devront être capables de manipuler en Java les collections implémentant List et Set.

Créer le projet

Durée estimée : 05 minutes

Crée un projet Eclipse intitulé `poo.labo06`. Dans les propriétés du projet :

- Ajoute un répertoire de sources pour les tests.
- Active PMD et renseigne le fichier de règles *poo-labo06-ruleset.xml*, disponible en annexe.
- Ajoute JUnit 5 comme dépendance au projet

Ajoute enfin les fichiers proposés en annexe. Tu devrais obtenir la structure de départ suivante.

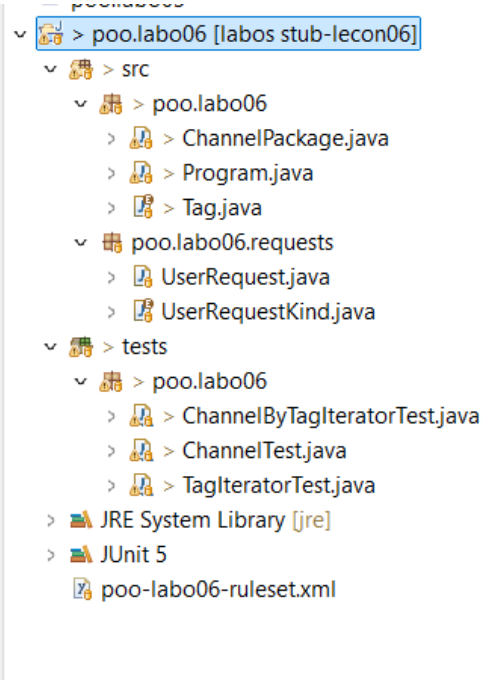

Figure 1 Structure attendue

Exercices

Nous souhaitons une application simulant des parcours séquentiels des chaînes d'un décodeur de télévision. La plupart des décodeurs permettent de parcourir les chaines (*channel*) par ordre de leur numéro ou selon un marqueur (*tag*, ex : cinéma, cartoon, culture, etc.).

Nous te fournissons les marqueurs via l'énumération `Tag`.

Exercice 1 : Définir les chaînes

Durée estimée : 20 minutes

Objectifs :

- Définir une classe composée d'une liste
- Encapsuler les attributs

Déclare la classe `src://poo.labo06.Channel` pour représenter une chaîne télévisée. Définis un constructeur prenant en paramètre le nom de la chaîne et une Collection de tags. Le constructeur valide les paramètres : le nom est un string non blanc et la collection de tags est définie. Pense à créer une copie défensive de la collection.

`Channel` expose les méthodes d'objet suivantes :

- `public String getName()` qui retourne le nom de cette chaîne ;
- `public boolean hasTag(Tag tag)` qui retourne `true` si et seulement si cette chaîne possède le marqueur tag. Lance un `NullPointerException` si tag vaut `null`.

Valide ta classe à l'aide de `ChannelTest` et des alertes PMD.

Exercice 2 : Découverte des itérateurs

Durée estimée : 40 minutes

Objectifs :

- Définir un itérateur et l'utiliser
- Implémenter l'interface `Iterator`
- Implémenter l'interface `Iterable` et l'exploiter avec la boucle for[each] en Java.

En POO, il est fréquent de parcourir les éléments d'une collection à l'aide d'itérateurs. Un itérateur représente un parcours d'une collection, indépendante de la structure de la collection. Cet exercice introduit les itérateurs en Java.

Les itérateurs sont utilisés dans une boucle : tant que l'itérateur a encore des éléments, la boucle demande et traite l'élément suivant.

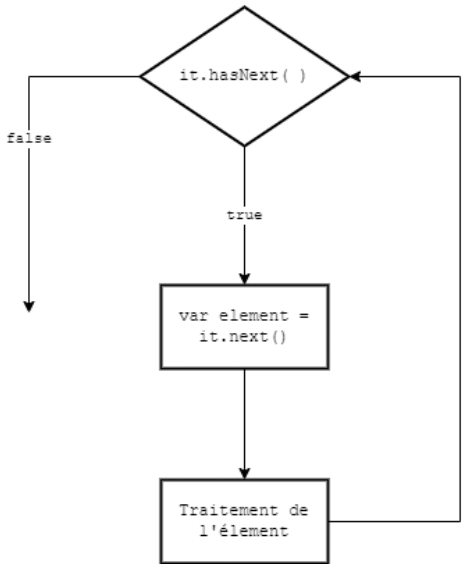

Figure 2 Utilisation classique d'un itérateur

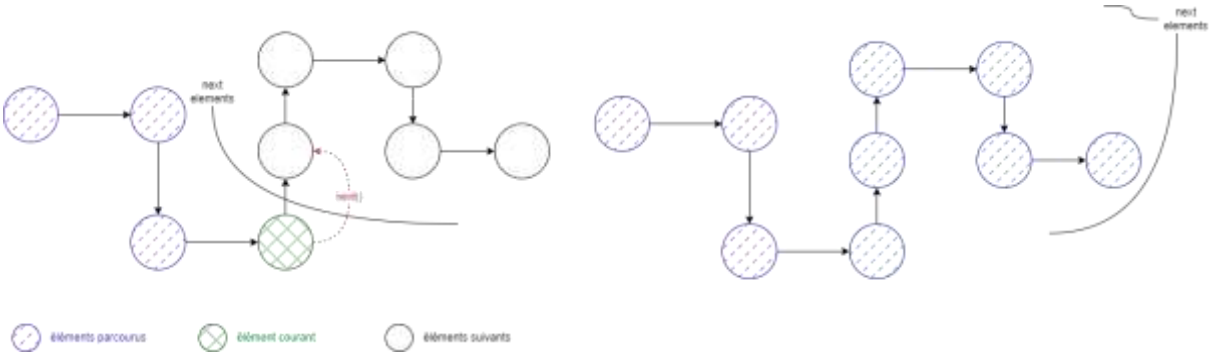

Figure 3 Un itérateur parcourt les éléments d'une collection tant qu'il reste des éléments.

En Java, on demande le plus souvent un itérateur à un objet itérable. Tout itérable implémente une méthode `iterator()` qui retourne un nouvel itérateur.

Déclare la classe `src://poo.labo06.TagIterator`. Cette classe possède un constructeur prenant en paramètre une collection de marqueurs qu'elle utilise pour initialiser une liste de marqueurs. Elle initialise également un attribut mémorisant la position de l'élément courant (que vaut cet élément au début de la vie de l'objet ?).

`TagIterator` expose deux méthodes d'objet :

- `public boolean hasNext()` qui retourne `true` si et seulement si la position courante de cet itérateur est plus petite que la taille de la liste ;
- `public Tag next()` qui retourne l'élément à la position courante, puis incrémente cette position de 1.

Attention

Appeler la méthode `next()` fait muter l'itérateur : il se déplace sur l'élément suivant. Dans une boucle, il est recommandé d'appeler cette méthode une seule fois.

Valide ta classe avec la classe `tests://poo.labo06.TagIteratorTest` et les alertes de PMD. Ajoute les tests suivants, en particulier le second test qui te demandera de construire une boucle while utilisant l'itérateur.

```
Un itérateur de tags
N'a pas d'élément suivant s'il est vide
Soit l'itérateur de tags construit avec la liste vide
Alors cet itérateur n'a pas d'élément suivant
Et le suivant de cet itérateur vaut null.

Parcourt les éléments d'une collection
Soit l'itérateur de tags construit avec la liste [Tag.FRENCH, Tag.DUTCH,
Tag.GENERAL]
Quand je construis un tableau à partir des tags suivants de l'itérateur, tant
qu'il en possède
Alors j'obtiens ce tableau est égal à [Tag.FRENCH, Tag.DUTCH, Tag.GENERAL]
```

Quand ta classe est validée, dans la classe `Channel`, déclare une méthode d'objet publique `TagIterator iterator()` qui retourne **un nouvel objet** `TagIterator` initialisé avec la liste des tags de cette chaîne. Ajoute le test suivant à la classe `ChannelTest`.

```
Une chaine
Fournit de nouveaux itérateurs de tags
Soit la chaîne nommée "La une", associée aux tags Tag.GENERAL, Tag.FRENCH et
Tag.PUBLIC
Quand j'appelle deux fois la méthode iterator()
Alors les résultats sont non-null
Et le résultat du premier appel est un objet distinct du résultat du second appel.
```

Dans l'en-tête de la classe `TagIterator`, ajoute la clause `implements Iterator<Tag>` et sauve ton fichier. Dans l'en-tête de la classe `Channel`, ajoute la clause `implements` Iterable<Tag> et sauve ton fichier.

Remarque : Eclipse te signale-t-il un problème pour TagIterator ? Cela ne devrait pas être le cas si tu as importé `java.util.Iterator`. En effet, les méthodes `hasNext()` et `next()` correspondent aux méthodes à implémenter. As-tu oublié le modificateur `public` ?

Remarque : Eclipse te signale-t-il un problème pour Channel ? Cela ne devrait pas être le cas. En effet, la méthode `iterator()` correspond à la seule méthode à implémenter. As-tu oublié le modificateur `public` ?

En Java, **tout objet dont la classe implémente Iterable<T> peut servir de source dans une boucle for(loopVariable : source).** Pour t'en convaincre, décommente le dernier test de la classe ChannelTest.

Note : Les collections fournies par le JDK implémentent toutes l'interface `Iterable<T>` : tu peux toutes les utiliser avec une boucle `foreach`.

Exercice 3 : Tout ça … pour ça ?!

Durée estimée : 10 minutes

Objectif : modifier l'implémentation d'une classe.

Comme indiqué en conclusion de l'exercice précédent, les collections du JDK implémentent l'interface `Iterable<T>`. Tu peux remplacer le corps de la méthode `Channel.iterator()` par un appel à la méthode `iterator()` de sa liste de tags. Tu constateras que, moyennant de petites adaptations, les tests écrits réussissent toujours.

Écrire ses propres itérateurs reste une approche intéressante si tu souhaites définir un parcours personnalisé :

- Un parcours des éléments d'une liste à rebours,
- Un parcours filtrant certains éléments ne répondant pas à une condition,
- Etc.

Exercice 4 : parcours des chaînes par ordre d'insertion

Durée estimée : 40 minutes

Objectifs :

- Définir une classe composée d'une implémentation de `Set<T>`
- Implémenter plusieurs interfaces au sein d'une classe
- Valider des méthodes à l'aide de tests unitaires

Nous souhaitons définir la classe des bouquets de chaînes (*Channel Package*). Cette classe collectionne des chaînes en veillant à ce qu'elles soient uniques et ordonnées selon l'ordre de leurs insertions. Elle implémente l'interface `Iterable<Channel>`. Sa méthode `iterator()`retourne un nouvel itérateur qui parcourt les chaînes dans l'ordre de leurs insertions.

Pour arriver à nos fins, nous pouvons travailler avec une LinkedHashSet. Contrairement à une HashSet et à une TreeSet, `LinkedHashSet` conserve l'ordre des insertions. Pour utiliser cette collection, la classe `Channel` doit respecter le contrat liant `equals(Object)` à `hashCode()` :

1. Redéfinir la méthode `equals(Object)` pour que deux chaînes soient équivalentes si elles portent le même nom ;
2. Redéfinir la méthode `hashCode()` pour que cette dernière retourne le même entier pour deux chaînes équivalentes.

Note

Eclipse propose des outils pour générer une méthode hashCode() à partir des champs d'un objet. C'est un bon point de départ.

Modifie la classe Channel. Teste-la pour couvrir au moins 90% du code des méthodes `equals(Object)`, `hashCode()`. Valides également ta classe à l'aide de PMD.

Crée ensuite la classe `src://poo.labo06.ChannelPackage`. Cette classe possède un Set de Channel initialisé avec une LinkedHashSet. Elle définit deux méthodes :

- `public void add(Channel…newChannels)` qui ajoute un nombre quelconque de `Channel` à ce bouquet.
- `public int getChannelCount()` qui retourne le nombre de chaînes qui compose ce bouquet.

`ChannelPackage` implémente `Iterable<Channel>`. Implémente la méthode `iterator()` pour que son corps ne contienne qu'une seule instruction `return <expression>`.

Valide le parcours d'un bouquet de chaînes par un test unitaire défini dans la classe `ChannelPackageTest`. Tu peux utiliser l'assertion `assertIterableEquals(Iterable<T>, Iterable<T>)`.

Retour avec le responsable

Durée estimée : 30 minutes

Le responsable propose une correction et insiste sur les éléments suivants :

- En Java, en plus des associations d'héritage, une classe peut implémenter des interfaces.
- Les interfaces sont des types. Tu peux les utiliser pour typer le résultat d'une méthode, une variable, etc.
- Un objet itérateur décrit un seul parcours. Une fois le parcours terminé, l'itérateur ne devrait plus être utilisé.
- L'interface `Iterable<T>` respecte le principe des copies défensives, car chaque appel devrait retourner un nouvel itérateur.
- L'interface `Iterator<T>` contribue à masquer les détails internes d'un objet. En effet, il est difficile de connaître la collection concrète utilisée à partir d'un itérateur. Nous pouvons facilement changer de type d'itérateurs ou de collections.

Exercice 5 : parcours et filtre des chaînes par tag

Durée estimée : 30 minutes

Objectif : définir un type d'itérateur particulier.

Nous souhaitons parcourir les chaînes possédant un tag particulier (parcourir les chaînes publiques, par exemple). Ainsi, nous représenterons ces parcours à l'aide d'itérateurs.

Déclare la classe `ChannelByTagIterator` qui implémente l'interface `Iterator<Channel>`. Cette classe prévoit un constructeur prenant en paramètre une `Collection<Channel>` et le tag désiré. Les méthodes à implémenter auront les comportements suivants :

- `hasNext()` retourne `true` si et seulement s'il existe encore un `Channel` qui possède le tag désiré ;
- `next()` retourne la prochaine chaîne et recherche ensuite la chaîne suivante qui possède le tag désiré.

Important

Contrairement à la méthode `next()`, qui fait muter l'itérateur, la méthode `hasNext()` ne devrait pas le faire muter.

Valide ton code avec la classe `ChannelByTagIteratorTest` et les alertes PMD.

Une fois l'itérateur défini, dans la classe `ChannelPackage`, ajoute une méthode d'objet publique `Iterator<Channel> iteratorForTag(Tag tag)`. Cette méthode crée et retourne un itérateur filtrant les chaînes qui contiennent le marqueur tag.

Valide ta méthode par des tests unitaires. Veille notamment à ce que `iteratorForTag(Tag)` retourne un itérateur sur une collection vide si le paramètre vaut null.

Exercice 6 : un programme de test

Durée estimée : 45 minutes

Objectifs :

- Programmer une application utilisant plusieurs types d'itérateurs
- Exploiter les méthodes `values` et `valuesOf` définies pour tout type énuméré
- Exploiter les mécanismes de polymorphisme par sous-typage appliqués à l'interface `Iterator<T>`

Complète les méthodes de la classe `poo.labo06.Program`. Dans le `main`, déclare une variable de type `ChannelPackage`. Cette variable est initialisée avec un bouquet composé des chaînes ci-dessous (de nouveaux tags sont à ajouter dans l'énumération).

Numero	Nom	Tags
1	La Une	GENERAL, PUBLIC, FRENCH
2	La Deux	GENERAL, PUBLIC, FRENCH
3	La Trois	KIDS, PUBLIC, FRENCH
4	Rtl-Tvi	GENERAL, PRIVATE, FRENCH
5	Club-Rtl	CINEMA, KIDS, PRIVATE, FRENCH
6	Plug-Rtl	CINEMA, PRIVATE, FRENCH
7	VRT	GENERAL, PUBLIC, DUTCH
8	Canvas	CINEMA, PUBLIC, DUTCH
9	Sporza	SPORT, PUBLIC, DUTCH
10	VTM	GENERAL, PRIVATE, DUTCH

Le programme demande ensuite à l'utilisateur d'encoder une des commandes suivantes :

- **list** qui affiche les chaînes par numéro ;
- **filter-by-tag TAG** où tag est à remplacer par une valeur de l'énumération ;
- **show-tags** qui affiche les tags ;
- **exit** qui sort du programme.

Pour énumérer les tags, appelle la méthode de classe `Tag.values()` qui retourne le tableau de toutes les valeurs de l'énumération `Tag`. Tu peux également appeler la méthode de classe `Tag.valueOf(String)` pour obtenir la valeur de l'énumération correspondant à un Tag

encodé par l'utilisateur. **Attention,** cette méthode est sensible à la casse et émet une IllegalArgumentException en cas de problème.

Exemple d'exécution

```
POO - Labo 06 : Channels
========================
Encodez une commande : list

Chaines par ordre
-----------------
La Une
La Deux
La Trois
Rtl-Tvi
Club-Rtl
Plug-Rtl
VRT
Canvas
Sporza
VTM

Encodez une commande : filter-by-tag KIDS
Chaines KIDS
--------------------
La Trois
Club-Rtl

Encodez une commande : show-tags
Liste des tags
--------------
GENERAL
FRENCH
PUBLIC
PRIVATE
DUTCH
KIDS
CINEMA
SPORT

Encodez une commande : exit
Au revoir
```

Pour ceux qui souhaitent aller plus loin :

La classe `ChannelPackage` n'est pas efficace pour filtrer sur les tags, car il faut parcourir toutes les chaines. Une approche plus efficace consiste à maintenir une map associant un tag aux chaines correspondantes. Modifiez la classe `ChannelPackage` à cette fin. La classe `ChannelByTagIterator` est-elle encore utile ? La supprimer affecte-t-il le programme ?