

Introduction

Le but de ce TP est de vous faire implémenter 2 collections (l'une utilisant un tableau et l'autre utilisant une liste doublement chaînée) en se basant sur l'implémentation partielle de l'interface `Collection<E>` apportée par la classe abstraite `AbstractCollection<E>`.

Celle-ci implémente déjà en grande partie l'interface `Collection<E>` grâce à l'utilisation de la template méthode `Iterator<E> iterator()` qui permet d'utiliser un itérateur dans la classe `AbstractCollection<E>` même si seules les classes filles de cette classe fourniront un itérateur concret. Implémenter une nouvelle collection à partir de la classe `AbstractCollection<E>` ne requiert donc que peu de membres. Il faudra donc dans nos classes :

- Un conteneur interne : un tableau, une liste chaînée, ou bien une autre collection.
- Des constructeurs
 - Un constructeur par défaut pour créer une nouvelle collection vide.
 - Un constructeur de copie à partir d'une autre collection.
- Une surcharge de la méthode `boolean add(E e)` car celle apportée par la classe `AbstractCollection<E>` ne fait que lever une `UnsupportedOperationException`.
- Une implémentation de la méthode `int size()`.
- Une implémentation des méthodes `boolean equals(Object o)` ainsi que `int hashCode()` car la classe `AbstractCollection<E>` ne surcharge pas ces méthodes héritées de la classe `Object`.
- Une implémentation de la factory méthode `Iterator<E> iterator()` fournissant un itérateur sur notre conteneur interne.
 - Il faudra donc pour ce faire implémenter une classe interne **dans** chaque nouvelle collection et qui implémentera l'interface `Iterator<E>`, permettant ainsi de parcourir les éléments de notre conteneur interne. Il faudra donc dans nos itérateur implémenter :
 - La méthode `boolean hasNext()` qui permet de savoir s'il reste des éléments à itérer sur le conteneur interne.
 - La méthode `E next() throws NoSuchElementException` qui renvoie l'élément courant du conteneur interne correspondant à l'élément courant de l'itération et se positionne sur le prochain élément (si celui-ci existe).
 - La méthode `void remove() throws IllegalStateException` qui permet (éventuellement) de supprimer du conteneur interne l'élément qui vient d'être renvoyé par la méthode `next()` (si celle-ci a été appelée au préalable).

Implémentation d'une `Collection<E>` en utilisant un tableau : `MyArrayCollection<E>`

Créez une nouvelle classe dans le package `collections` : `public MyArrayCollection<E> extends AbstractCollection<E> implements Capacity<E>` qui contiendra donc :

- `E[] array` : Comme conteneur interne un simple tableau d'éléments.
- `int size` : Un entier indiquant le nombre d'éléments actuellement stockés dans le tableau.
- `int capacity` : Un entier indiquant la taille (actuelle) du tableau.
- `int capacityIncrement` : Un entier indiquant le nombre de cases à rajouter au tableau lorsque celui-ci s'avèrera trop petit pour stocker tous les éléments requis de la collection. Nous serons donc amenés à réallouer le tableau interne de temps à autre.
- Il faudra des constructeurs adéquats pour initialiser tous ces attributs. Ces constructeurs devront lever des `IllegalArgumentException` si les arguments fournis aux constructeurs sont invalides. Typiquement : la capacité initiale doit être positive ou nulle et la capacité d'incrément strictement positive.

L'interface `Capacity<E>` du package `collections.utils` définit les méthodes caractéristiques d'une classe possédant un tableau interne (comme c'est le cas de notre collection) que l'on devra éventuellement faire grandir lorsque le nombre d'éléments à mettre dans la collection dépassera le nombre de cases de notre tableau interne :

- `abstract int getCapacity()` permettra d'obtenir le nombre d'éléments que l'on peut actuellement stocker dans notre tableau interne (qui peut être différent du nombre d'éléments actuellement stockés dans notre tableau (voir l'attribut `size`)).
- `abstract int getCapacityIncrement()` permettra d'obtenir le nombre de cases à rajouter au tableau interne lorsqu'il faudra l'agrandir.
- `abstract void grow(int amount)` : permettra de réallouer le tableau interne avec `amount` cases.
- `default void ensureCapacity(int minCapacity)` : permet de réallouer (si besoin) le tableau interne avec au moins `minCapacity` cases.
- `static <E> E[] resizeArray(E[] array, int requiredSize)` : permet de réallouer un tableau `array` avec `requiredSize` tout en préservant son contenu (si `requiredSize` est plus grand que `array.length`).x

Dans la classe `MyArrayCollection<E>`, créez une classe interne `private ArrayIterator implements Iterator<E>` qui contiendra (au moins) :

- Un `int index` indiquant l'état courant de l'itération (entre 0 et `size()` - 1 dans la collection).
- Un `boolean nextCalled` indiquant `true` lorsque que la méthode `next()` vient d'être appelée et qui pourra être remis à `false` dans la méthode `remove()`.
- Ainsi qu'une implémentation des méthodes `hasNext()`, `next()` et `remove()` bien sûr.

Vous pourrez tester cette collection avec la classe `CollectionTest` du package `tests`.

Implémentation d'une `Collection<E>` en utilisant une liste doublement chaînée : `MyLinkedCollection<E>`

Créez une nouvelle classe dans le package `collections` : `public MyLinkedCollection<E> implements AbstractCollection<E>` qui contiendra donc comme conteneur interne la tête d'une liste chaînée (un nœud de la liste) : `Node<E> head`.

La classe `Node<E>` vous est fournie dans le package `collections.utils` et représente les nœuds doublement chaînés d'une liste doublement chaînée. Ce Nœud contient donc :

- `E data` : une donnée
- `Node<E> previous` : une référence au nœud précédent qui peut être null s'il n'y a pas de nœud précédent.
- `Node<E> next` : une référence au nœud suivant qui peut être null s'il n'y a pas de nœud suivant.

Dans la classe `MyLinkedCollection<E>`, créez une classe interne `private NodeIterator implements Iterator<E>` qui contiendra (au moins) :

- `Node<E> current` : Le nœud courant de l'itération qui pourra être initialisé à `head` lors de la construction de l'itérateur.
- `Node<E> previous` : Le nœud précédent de l'itération qui pourra être initialisé à null, puis mis à jour lors de l'appel de méthode `next()` et pourra être utile lors de l'appel à la méthode `remove()` puisqu'il faudra alors supprimer le nœud précédent en reliant le nœud précédant du précédent (le pénultième) au nœud courant de l'itération.
- `boolean nextCalled` : Un boolean indiquant `true` lorsque que la méthode `next()` vient d'être appelée et qui pourra être remis à `false` dans la méthode `remove()`.

Vous pourrez tester cette collection avec la classe `CollectionTest` du package `tests`.

Annexe

La Figure 1 page 4 résume la hiérarchie de classes dans laquelle vous allez travailler.

Les classes que vous devez implémenter sont :

- `MyArrayCollection<E>`
 - `ArrayIterator`
- `MyLinkedCollection<E>`
 - `NodeIterator`

