TD Listes Chaînées

Ce sujet permet, en plus de l'algorithmique liée aux listes, d'aborder : les *types génériques* et les *classes internes*.

Exercice 1 : Listes et itérateurs

Une liste est une collection ordonnée d'objets. On rappelle la définition récursive d'une liste. Une liste est :

- soit une liste vide
- soit un couple *(valeur,suivant)* où *valeur* représente le premier élément de la liste (sa "tête) et *suivant* est la **liste** des autres éléments de la liste (le "reste").

Q 1 . Définissez une classe `MyList<E>`, où `E` est le type des éléments de la liste, qui permet la représentation de listes sous forme de listes chaînées.
On ne respectera pas les interfaces `List` ni `Collection`.

Les méthodes à mettre en œuvre sont les suivantes :

- ▷ `isEmpty` pour tester si la liste est vide ou non
- ▷ `add` pour ajouter un élément en tête de la liste
- ▷ `remove` permet de supprimer le premier élément de la liste
- ▷ `head` pour récupérer l'élément en tête de la liste
- ▷ `contains` permet de tester l'appartenance à la liste d'un objet
- ▷ `size` fournit la taille de la liste
- ▷ `iterator` pour obtenir un iterator sur la liste

Les méthodes `head` et `remove` lèvent une exception (`IllegalStateException` par exemple) si la liste est vide.

Les itérateurs obtenus par la méthode `iterator` seront conformes à l'interface `java.util.Iterator`[1] (voir Annexe).
Comme dans les listes de l'api `java.util`, la classe des itérateurs sera définie comme classe interne de `MyList`, pourquoi ?
On gèrera également le caractère "fail-fast" des itérateurs : les méthodes `next` et `remove` lèvent une `ConcurrentModificationException` (voir Annexe) si la liste manipulée est modifiée pendant l'usage de l'itérateur.

Annexe

`java.util.Iterator<E>`

`public boolean hasNext()` Returns true if the iteration has more elements. (In other words, returns true if next would return an element rather than throwing an exception.)

`public E next()` Returns the next element in the iteration.

Thows `NoSuchElementException` - iteration has no more elements.

`public void remove()` Removes from the underlying collection the last element returned by the iterator (optional operation). This method can be called only once per call to next. The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method.

Throws `IllegalStateException` - if the next method has not yet been called, or the remove method has already been called after the last call to the next method.

[1] Ce qui permet de faire implémenter l'interface `Iterable` par `MyList`;

`java.util.ConcurrentModificationException`

This exception may be thrown by methods that have detected concurrent modification of an object when such modification is not permissible.

For example, it is not generally permissible for one thread to modify a Collection while another thread is iterating over it. In general, the results of the iteration are undefined under these circumstances. Some Iterator implementations (including those of all the general purpose collection implementations provided by the JRE) may choose to throw this exception if this behavior is detected. **Iterators that do this are known as fail-fast iterators**, as they fail quickly and cleanly, rather that risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that this exception does not always indicate that an object has been concurrently modified by a different thread. If a single thread issues a sequence of method invocations that violates the contract of an object, the object may throw this exception. **For example, if a thread modifies a collection directly while it is iterating over the collection with a fail-fast iterator, the iterator will throw this exception.**

(...)