Itérateurs

POGL, TP2. Interfaces et classes paramétrées.

Dans ce TP, nous allons réaliser deux classes basiques de conteneurs sur lesquels il est possible d'itérer, et qui implémentent des interfaces extraites de la bibliothèque standard.

Une interface pour les listes

Dans le jargon de Java une liste désigne une collection ordonnée, ou *séquence*, d'éléments. La caractéristique principale d'une telle collection est qu'on peut accéder à un élément en fonction de sa position, donnée par un indice entier. On propose pour cela l'interface suivante, à reproduire dans votre code (extraite de la version plus complète `java.util.List`) :

```
interface List<T> {
    /** Renvoie l'élément d'indice [i]. */
    T get(int i);
    /** Ajoute l'élément [elt] à la fin de la liste. */
    void add(T elt);
}
```

Implémentation par un tableau

Des listes peuvent être représentées en utilisant les tableaux primitifs de Java. On obtient alors des listes dont la capacité est limitée par la taille des tableaux utilisés.

1. Définir une classe `FixedCapacityList` stockant des éléments de type `Object` dans un tableau de type `Object[]`, en respectant les indications suivantes :
   — la classe doit implémenter l'interface `List<Object>`,
   — le constructeur doit prendre en paramètre la capacité souhaitée,
   — la méthode `add` ne doit rien faire si la capacité est atteinte,
   — la méthode `get` n'a pas besoin de vérifier les accès hors limites,
   — pour faciliter la composition avec les autres classes qui composeront le paquet final, les attributs peuvent avoir la visibilité `protected`.

Itérateurs

En Java, l'itération sur les éléments d'une collection se fait via la construction d'un objet itérateur qui est chargé de fournir un à un les éléments de la collection.

L'itérateur d'une collection d'éléments de type `T` a le type `Iterator<T>`, défini par l'interface suivante (importez-la avec la commande `import java.util.Iterator;`) :

```
interface Iterator<T> {
    /** Renvoie vrai s'il existe un prochain élément. */
    boolean hasNext();
    /** Donne le prochain élément et prépare le passage au suivant. */
    T next();
}
```

Chaque appel à la méthode `next()` fait avancer l'itération en renvoyant l'élément suivant.

On manifeste qu'il est possible d'itérer sur une collection en déclarant que cette collection implémente l'interface `Iterable<T>`, qui demande une méthode produisant un itérateur du type correspondant (importez-la avec la commande `import java.lang.Iterable;`).

```
interface Iterable<T> {
    Iterator<T> iterator();
}
```

Par exemple, la classe `ArrayList<T>` implémentant l'interface `Iterable<T>`, on peut afficher l'ensemble des éléments d'une liste de chaînes de caractères grâce au code suivant :

```
ArrayList<String> vent = new ArrayList<String>();
vent.add("'            ,            ,  ≪   -    ≫.");
vent.add("           ,            ,   '           ,   '    ,   '   .");
vent.add("          ,       ,               .");
// Création d'un itérateur
Iterator<String> it = vent.iterator();
// Tant qu'il reste des éléments...
while (it.hasNext()) {
    // prendre le prochain élément et l'afficher.
    System.out.println(it.next());
}
```

2. Modifier l'exemple précédent pour afficher deux fois chaque élément, puis pour n'afficher qu'un élément sur deux.

Des itérateurs pour les tableaux

Nous allons implémenter deux manières d'itérer sur nos listes à capacité fixe, dans l'ordre croissant ou décroissant des indices. Dans chaque cas, il faut créer une classe implémentant l'interface `Iterator<Object>`, et la doter d'attributs permettant de suivre la progression de l'itération.

3. Définir une classe `AscendingIterator` implémentant l'interface `Iterator<Object>` et représentant une itération sur un objet `FixedCapacityList`, qui fournit les éléments dans l'ordre du premier au dernier. Faire ensuite en sorte que la classe `FixedCapacityList` implémente l'interface `Iterable<Object>`.
4. Définir une classe `DescendingIterator` implémentant l'interface `Iterator<Object>` et représentant une itération sur un objet `FixedCapacityList`, qui fournit les éléments dans l'ordre du dernier au premier.

Implémentation par une liste chaînée

Pour implémenter l'interface `List`, une alternative aux tableaux utilisés dans `FixedCapacityList` consiste à définir une structure de liste chaînée : pour une liste dont les éléments sont de type `T`, on utilise plusieurs blocs reliés entre eux et contenant chacun un élément.

Concrètement, on définit une classe `Block<T>` ayant :
— un attribut `contents` de type `T` pour l'élément contenu,
— un attribut `nextBlock` de type `Block<T>` désignant le bloc suivant, ou valant `null` s'il n'y a pas de bloc suivant.

La classe `LinkedList<T>` contient alors deux attributs `firstBlock` et `lastBlock`, tous deux de type `Block<T>`, désignant respectivement les premier et dernier blocs de la liste.

5. Définir des classes `Block<T>` et `LinkedList<T>` stockant des éléments de type `T` dans une liste chaînée, en respectant les indications suivantes :
   — la classe `LinkedList<T>` doit implémenter l'interface `List<T>`,
   — lors de la création d'une liste, les champs `firstBlock` et `lastBlock` doivent être initialisés à `null`,
   — la méthode `get(int i)` doit renvoyer le contenu du premier bloc si `i` vaut `0`, le contenu du bloc suivant si `i` vaut `1`, de celui d'après si `i` vaut `2`, etc,
   — la méthode `add` doit ajouter un bloc après le dernier bloc (et mettre à jour le champ `nextBlock` du dernier bloc).
6. Définir une classe `LinkedListIterator<T>` implémentant l'interface `Iterator<T>` et représentant une itération sur un objet `LinkedList<T>`, dans l'ordre du premier au dernier. Faire ensuite en sorte que la classe `LinkedList` implémente l'interface `Iterable<Object>`.