Aix*Marseille université
Initiative d'excellence
Année 2020-21

Domaine Sciences et Technologies
Licence informatique

Programmation 2 : TP 5
Code UE : SIN3U02

ArrayList

L'objectif de ce TP est d'implémenter notre propre version de la classe `ArrayList`. Cela va nous permettre :
— de mettre en pratique nos connaissances sur les génériques,
— d'utiliser encore un peu les tableaux,
— de s'habituer à tester les méthodes systématiquement,
— d'approfondir notre compréhension de la classe `ArrayList`, qui est très utilisée.

1 Définition de la classe `MyArrayList`

La classe `MyArrayList` implémente l'interface `List`. Cette dernière définit un ensemble de méthodes permettant de créer et de manipuler des listes d'éléments. Il s'agit d'une interface générique : elle ne précise pas la nature des éléments qui sont stockés dans la liste.

Vous trouverez sur dépôt `git` de ce TP (`https://etulab.univ-amu.fr/nasr/tp5.git`), le fichier `MyArrayList.java`, qui comporte les signatures des méthodes que la classe `MyArrayList` doit implémenter.

Une liste de la classe `MyArrayList` est représentée par un tableau suffisamment grand pour contenir tous les éléments de la liste, aux indices consécutifs à partir de 0. Autrement dit, si la liste contient `size` éléments, ils sont stockés dans les cases `0` à `size - 1`, mais le tableau peut être plus grand. On doit donc retenir séparément la longueur du tableau, que nous appellerons **capacité** (`capacity`) et le nombre d'éléments actuellement dans la liste, que nous appellerons **taille** (`size`).

Par ailleurs, les éléments doivent être consécutifs dans la liste. Ainsi, pour ajouter un élément en milieu de liste, il faut décaler dans le tableau tous les éléments suivants. De même pour supprimer un élément de la liste. Enfin, on peut remplacer un élément de la liste par `null` pour créer un *trou*, qui compte néanmoins pour un élément.

Dans un premier temps, nous implémenterons toutes les méthodes comme si le tableau était de longueur infinie. Plus tard nous verrons comment agrandir le tableau lorsqu'il devient plein.

— Définir les propriétés de `MyArrayList` qui permettront de représenter la liste.
— Ajouter deux constructeurs. Le premier prend en paramètre la capacité initiale du tableau interne. Le deuxième ne prend pas d'argument, et utilise comme capacité initiale une valeur par défaut (par exemple 10). Pour initialiser le tableau, qui est un tableau d'un type générique, il faut créer un tableau d'`Object`, puis faire une conversion de type vers le type désiré `T[]`, en utilisant la syntaxe du *cast* : `(T[]) new Object[capacity]`. C'est le seul usage de *cast* que nous autoriserons lors de ce TP. Ce *cast* étant potentiellement dangereux, *javac* émet un avertissement lors de la compilation. Afin d'éviter cela, vous pouvez rajouter une ligne d'annotation `@SuppressWarnings("unchecked")` avant la déclaration du constructeur. N'oubliez pas que le deuxième constructeur peut appeler le premier avec la syntaxe `this(args)`. Si la capacité initiale donnée en argument est négative, vous devrez lever une exception de type `IllegalArgumentException`.

2 Tests unitaires

Nous allons vérifier chaque méthode de la classe `MyArrayList` à l'aide de tests unitaires. Pour cela, nous avons défini la classe `TestMyArrayList` qui possède une méthode pour chaque test unitaire. Vous trouverez dans cette classe trois tests déjà implémentés. Pour chaque méthode que vous implémentez, pensez à la manière dont vous pouvez vérifier si elle répond bien aux spécifications et écrivez les tests correspondants.

Dans la mesure où la classe `MyArrayList` doit se comporter de la même manière que la classe `ArrayList`, une manière de vérifier que les méthodes de `MyArrayList` sont correctes, consiste à faire en parallèle les mêmes opérations sur une `ArrayList` et une `MyArrayList`. Les listes obtenues doivent être équivalentes. Pour vérifier

cela, vous pouvez utiliser la méthode `equals` de `ArrayList`, comme dans l'exemple de la méthode `testAdd1` de `TestMyArrayList`. Mais attention, la méthode `equals` de `ArrayList` repose sur l'itérateur de `MyArrayList`. Il faut donc faire attention à bien tester cet itérateur et il faut aussi, bien entendu, que vous ayez implémenté cet itérateur.

3 Fonctionnalités de base

Implémenter les méthodes de bases : `add(T elt)`, `size()`, `isEmpty()`, `get(int index)`. Créer des tests pour ces méthodes. Afin de tester si l'indice donné en argument de la méthode `get` est correct, vous pouvez utilise la méthode `static int checkIndex(int index, int length)` de la classe `Objects`.

4 Conversion en `String`

Implémenter la méthode `toString` en utilisant la classe `StringBuilder` pour construire la chaîne résultat. Il vaut mieux utiliser `StringBuilder` que concaténer directement des `String`. En effet, les `String` sont immuables, une concaténation avec des chaîne de caractères longues coûte cher car elle demande de créer une nouvelle `String` (différente des deux `String` concaténée).

5 Itérateur

Afin de pouvoir parcourir une instance de `MyArrayList` sans connaître la manière dont cette classe implémente une liste, on fait appel à un itérateur. Afin d'appliquer la fonction `action`, par exemple, à tous les éléments de la liste `maListe`, on procède de la manière suivante :

```
List<Integer> maListe = new MyArrayList<Integer>();
...
Iterator itr = maListe.iterator();
while(itr.hasNext())
  action(itr.next());
```

ou bien, de manière plus compacte :

```
List<Integer> maListe = new MyArrayList<Integer>();
...
for (int element: mal)
  action(element);
```

Implémenter la méthode `iterator`. Pour cela, créer une classe générique `ArrayIterator<E>` implémentant l'interface `Iterator<E>`. Son constructeur reçoit une copie de la partie utilisée du tableau contenant les éléments de la liste. (la classe `Arrays` contient une méthode qui vous aidera). Compléter la classe `ArrayIterator`.

6 Gestion dynamique de la capacité

Lorsque le tableau devient trop petit pour contenir les éléments de la liste, il faut le remplacer par un tableau plus grand.

— Ajouter une méthode `ensureCapacity(int capacity)`, qui sans modifier la liste, s'assure que le tableau peut contenir `capacity` éléments. Pour cela, `ensureCapacity` peut au besoin remplacer le tableau par un nouveau tableau plus grand, dans lequel on aura pris soin de copier toutes les valeurs de la liste dans les cases de mêmes indices. Comme le coût de la copie est élevé, on ne veut pas avoir à redimensionner le tableau trop souvent. On fera donc en sorte que lorsque le tableau est redimensionné, sa capacité soit doublée. À nouveau, pensez à utiliser les méthodes de la classe `Arrays`.

— Dans toutes les méthodes modifiant le nombre d'éléments dans la liste, ajouter un appel à `ensureCapacity` pour s'assurer que la liste ne déborde jamais du tableau.

7 Davantage de méthodes

— Implémenter `equals`. Pour cela, compléter les instructions suivantes :

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
    return true;
  }
  if (!(obj instanceof List<?>)) {
    return false;
  }
  List<?> other = (List<?>) obj;
  // TODO: check whether this and other are equal.
}
```

— Implémenter et tester les méthodes `set` et `add(int index, T elt)`.
— Implémenter et tester les méthodes `contains`, `indexOf`, `lastIndexOf`.
— Implémenter et tester les méthodes `remove(int)`, `remove(Object)`

8 Partie optionnelle

Finir l'implémentation de la classe `MyArrayList` en implémentant toutes les méthodes manquantes de l'interface `List`. Tester le comportement de ces méthodes.

On voudrait aussi s'assurer que le tableau n'est pas trop grand par rapport à la liste, pour ne pas gâcher de la mémoire pour rien. Pour cela, modifier `ensureCapacity` pour réduire la taille du tableau de moitié lorsque seulement 25% de sa taille est utilisée.