Aix*Marseille université
Initiative d'excellence
Année 2018-19

Domaine Sciences et Technologies
Licence informatique

Programmation 2 : TP 6
Code UE : SIN3U02

ArrayList

1 Implémentation de la classe `ArrayList`

1.1 Consignes pour démarrer le TP

Comme pour les TP précédents, on va utiliser git pour la gestion de versions. Il vous faut donc vous reporter aux consignes du premier TP.

Une fois le dépôt téléchargé, vous pouvez compiler et tester le programme en cliquant deux fois sur `myArrayList-***` -> `Tasks` -> `verification` -> `test` dans l'onglet *Gradle* (il faut pour cela avoir ajouter votre projet en tant que projet *Gradle* ; si l'IDE ne l'a pas proposé, vous pouvez cliquer droit sur le fichier `build.gradle` du projet et sélectionner l'option `import gradle project`. Si l'onglet *gradle* ne s'affiche pas, aller dans le menu `View` -> `Tool windows` -> `Gradle`).

Lien vers la classroom du TP : lien.

1.2 Objectifs du TP

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,
- à approfondir notre compréhension de la classe `ArrayList` que nous utilisons extrêmement souvent.

Toutes les méthodes que nous allons écrire seront testées en vérifiant les retours, comme nous savons déjà faire, mais aussi en comparant au résultat de la méthode correspondantes dans `ArrayList`. Pour cela, il suffira d'effectuer le même appel de méthode sur une instance d'`ArrayList` et sur une instance de `MyArrayList` égale, et de comparer que les valeurs retournées sont égales et que les deux instances sont toujours égales. Ainsi, nous testerons que les deux classes ont la même sémantique.

1.3 Consignes pour le début du TP

Modifiez le fichier `README.md`. Mettez votre nom, votre **numéro de groupe** ainsi que le nom et le **numéro de groupe** de votre éventuel co-équipier. Faites un *commit* avec pour message "inscription d'un membre de l'équipe", puis un *push*.

N'oubliez pas de **faire des *commit* régulièrement !**

1.4 Tâche 1 : Mise en place

La classe `MyArrayList` est déjà créée dans le fichier correspondant, mais pour l'instant elle est vide.
- Corriger sa déclaration, pour en faire une classe générique. Ajouter à la déclaration que cette classe implémente l'interface des listes. La classe est alors considérée erronée puisque pour l'instant elle ne contient pas l'implémentation des méthodes de l'interface.
- En utilisant l'IDE pour corriger l'erreur, générer toutes les méthodes manquantes de l'interface, en veillant bien **à cocher la case *check JavaDoc*** . Attention, les méthodes ainsi générées ne font rien et retourne des valeurs par défaut, il faut donc maintenant les compléter, ce que nous allons faire petit à petit. Noter que la documentation des opérations des listes a été copiée dans votre fichier, ce qui vous permettra de plus facilement savoir ce que font les méthodes à implémenter.
- Ajouter une méthode `toString`. Pour l'instant elle retourne une chaîne vide.

— Ajouter une méthode `public boolean equals(Object o)`. Pour l'instant, elle doit retourner `true` si et seulement si l'objet passé en paramètre est `this`.
— Il est temps de faire un *push*, si vous n'en avez pas encore fait.

1.5 Tâche 2 : Représentation et premières méthodes

Internement, une liste de la classe `ArrayList` est représentée par un tableau suffisamment long 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 long. 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.

— Choisir 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 du 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. N'oubliez pas que le deuxième constructeur peut appeler le premier constructeur avec la syntaxe `this(args)`. Si la capacité initiale donnée en argument est négative, vous devrez lever une exception de type `IllegalArgumentException`. Ce *cast* étant potentiellement dangereux, *javac* émet un avertissement loirs de la compilation. Afin d'éviter celà, vous pouvez rajouter une ligne d'annotation `@SuppressWarnings("unchecked")` avant la déclaration du constructeur.
— Ajouter un autre constructeur, prenant une collection en argument. Laisser ce constructeur non-implémenté pour l'instant.
— À ce point, vous pouvez lancer les tests. Les tests devraient échouer pour la plupart, c'est normal étant donné que tout reste à implémenter.
— 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`.
— Implémenter le troisième constructeur, celui-prenant une collection en paramètre, et remplissant la liste initiale avec tous les objets de la collection.
— Implémenter la méthode `iterator`. Pour cela créer une classe générique `ArrayIterator` 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`.
— Implémenter `toString` en utilisant la classe `StringBuilder` pour construire la chaîne résultat. Il faut utiliser `StringBuilder` au lieu de 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).
— Vous pouvez maintenant tester le comportement de `MyArrayList` par rapport à `ArrayList`. Pour l'instant, les tests reposent sur la méthode `equals` de `ArrayList`, qui elle-même repose sur l'itérateur de `MyArrayList`. Il faut donc faire attention à bien tester l'itérateur. Certains tests vont échouer car vous n'avez pas encore implémenté les méthodes nécessaires. Vérifier que les tests concernant les méthodes que vous avez implémenté passent (faites les corrections nécessaires). D'autres tests vont échouer parce

que la capacité du tableau est fixe, nous corrigerons cela dans la prochaine tâche.
— Il est temps de faire un autre *push* si vous ne l'avez pas encore fait.

1.6 Tâche 3 : 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.

1.7 Tâche 4 : 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 equals.
}
```

— 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)`

1.8 Tâche 5 : 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.