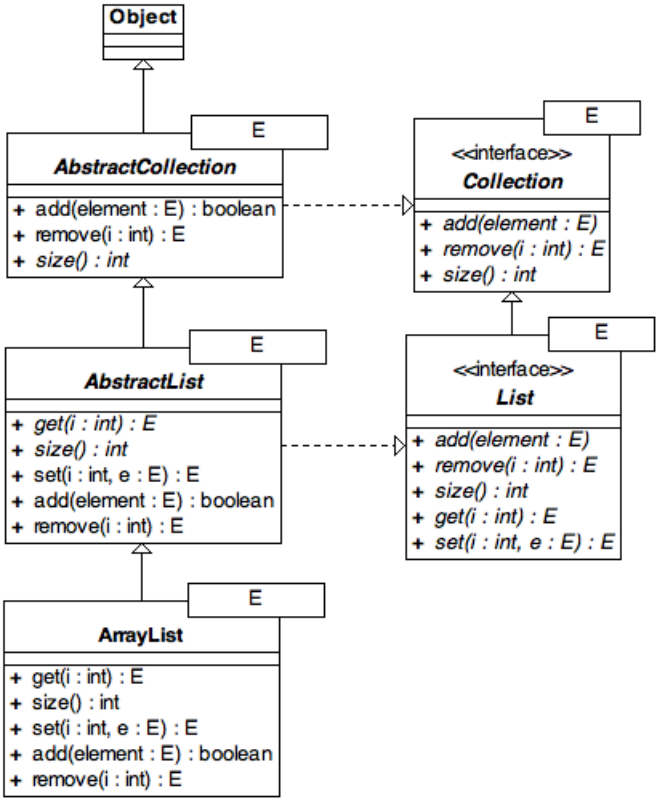
Développement Web 2 (L3)

FICHE DE TP N^{O} 2

08 février 2018

1 Généricité

Nous avons vu en cours les notions de classes abstraites et d'interfaces. En UML les classes et méthodes abstraites sont indiquées avec le mot {abstract} ou leur nom écrit en italique. Les interfaces sont indiquées par {interface} ou «`interface`».



Ces notions sont très usitées dans les librairies fournies avec Java. Notamment, les collections, que vous avez déjà utilisées en sont un bon exemple.

Le schéma ci-dessus présente une partie des héritages et des implémentations à partir de la classe concrète `ArrayList<E>` (notons qu'un type passé en paramètre d'une classe est noté par un rectangle en haut à droite de la classe).

En partant de ArrayList, nous pouvons constater :

— ArrayList hérite de la classe abstraite AbstractList et implémente donc les méthodes get et size qui sont abstraites dans AbstractList.

— AbstractList hérite de la classe abstraite AbstractCollection ce qui lui fournit notamment les méthodes add, remove et size provenant de l'interface Collection (voir ci-dessous), et implémente également l'interface List qui lui fournit surtout les méthodes get et set pour accéder en lecture et en écriture à des éléments particulier de la liste.

— AbstractCollection hérite directement de la classe Object et implémente l'interface Collection nécessitant les méthodes add, remove et size.

— L'interface List hérite de l'interface Collection, et lui ajoute les méthodes get et set.

Donc par polymorphisme, une ArrayList peut également être utilisée comme une AbstractList, comme une AbstractCollection, comme un Object, comme une List et comme une Collection.

EXERCICE 1. *Écrivez une classe SortedListOfStrings représentant une liste triée d'objets du type String. Le tri est effectué à chaque ajout d'élément dans la liste (méthode add). Basez-vous sur une `ArrayList<String>` (héritage). Pour comparer deux chaînes entre elles, utilisez la méthode `compareTo()` de la classe String.*

EXERCICE 2. *Ecrivez une classe SortedListOfStringsTest contenant une méthode main pour tester votre liste.*

1.1 Limites de l'héritage

L'héritage est un outil puissant permettant de factoriser les parties de code redondantes, par exemple. Il permet également de développer des programmes génériques, grâce à la notion de polymorphisme. Cependant, quelques limites existent, notamment, le fait qu'une sous-classe hérite de toutes les méthodes de sa super-classe. Ce qui peut entraîner des problèmes d'intégrité des données. Prenons l'exemple de la classe SortedListOfStrings. Est-elle intègre ? Peut-on assurer qu'elle est toujours triée ? **Non !**

En effet, on peut utiliser la méthode set(int index, String element) si elle hérite de la classe ArrayList pour remplacer un élément par un autre, non respectueux de l'ordre de tri ; à moins que l'on redéfinisse toutes les méthodes d'accès à la liste... Ceci impose donc de réécrire une quantité de code importante pour assurer l'intégrité de la classe.

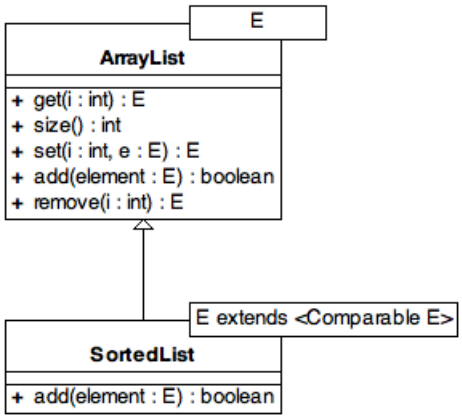
Une solution à ce problème récurrent est l'usage de la délégation : au lieu d'hériter d'une classe A, une classe B va encapsuler un objet de la classe A et fournir les accès nécessaires et suffisant pour l'intégrité des données. Par

exemple, au lieu d'hériter de la classe ArrayList, la classe SortedListOfString va posséder un attribut privé instance de la classe ArrayList et fournir uniquement les accès utiles (création, ajout d'élément, etc.). Elle possèdera bien les mêmes comportements, mais par contre ne pourra pas être utiliser comme une ArrayList (pas de polymorphisme, donc) puisque ce n'est pas une sous-classe.

EXERCICE 3. *Créez une nouvelle classe SortedListOfStringsUsingDelegation représentant une liste ordonnée de String mais par délégation au lieu d'héritage. Les opérations fournies sont la création, l'ajout, la taille, l'obtention d'un élément à une position donnée, la suppression du premier élément et la conversion en chaîne de caractères.*

1.2 La généricité en Action

Dans l'exercice 1 , vous avez défini une liste triée de chaînes de caractères : SortedListOfStrings. Par héritage vous avez pu définir qu'une liste triée est une ArrayList<String> dont la méthode add est modifiée de telle sorte que la chaîne à insérer soit à la bonne place dans la liste au lieu d'être ajoutée en fin de liste. Sans le savoir vous avez utilisé l'interface Comparable qu'implémente la classe String de Java, en utilisant la méthode compareTo. Donc, votre méthode pour insérer une chaîne de caractère peut être utilisée pour n'importe quelle classe implémentant l'interface Comparable (à quelques modifications de déclarations près).



EXERCICE 4. *En vous basant sur votre solution, écrivez une classe SortedList permettant de ranger n'importe quel objet à partir du moment où cet objet est comparable aux objets contenus dans la liste (et inversement). En d'autres termes, ceci signifie qu'au lieu de ranger des objets de type String votre classe doit ranger des objets de type quelconque E qui soit comparable à des objets de type E.*

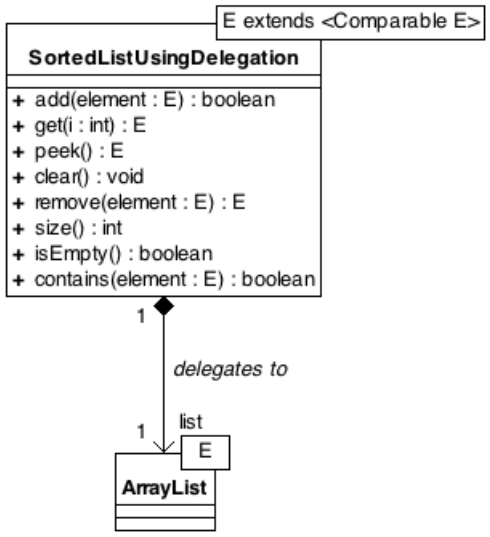
En Java, on note ce type E extends Comparable<E> comme présenté dans la figure 5 et le code suivant :

```
public class SortedList<E extends Comparable<E» extends
ArrayList<E> { ... }
```

L'idée est de pouvoir utiliser des listes contenant des types différents comme dans l'exemple suivant :

```
SortedList<String> sls = new SortedList<String>();
sls.add("Hello");
sls.add("World");
SortedList<Integer> sli = new SortedList<Integer>();
sli.add(1);
sli.add(2);
```

EXERCICE 5. *Faites de même avec votre solution de liste par délégation (SortedListOfStringsUsingDelegation) comme l'illustre la figure ci dessous.*



Écrivez et testez donc les méthodes présentes dans la figure :

- *add pour ajouter un élément à la bonne place,*
- *get pour récupérer le ième élément,*
- *peek pour récupérer le premier élement de la liste,*
- *clear pour vider la liste,*
- *remove pour supprimer un élément de la liste,*
- *size pour récupérer la taille de la liste,*
- *isEmpty pour déterminer si la liste est vide,*
- *contains pour déterminer si un élément appartient à la liste.*

2 Threads

Dans cette partie, nous allons voir comment utiliser la programmation concurrente dans Java. Pour cela, nous allons réaliser une série d'exercices afin de maitriser le concept de Thread vu en cours.

EXERCICE 6. *Compiler et exécuter le programme suivant. Essayer de prévoir ce qui devrait s'afficher et comparer à l'exécution.*

```
1  package fr.ujm.tia;
2
3  public class DeuxThreads extends Thread {
4    public void run() {
5            for ( int i = 0; i < 1000; i++ ) {
6                    System.out.println("Nouveau thread");
7            }
8    }
9
10   public static void main(String[] args) {
11           DeuxThreads tt = new DeuxThreads();
12           tt.start();
13
14           for ( int i = 0; i < 1000; i++ ) {
15                   System.out.println("Thread principale");
16           }
17   }
18 }
```

Indication : *le démarrage d'un thread se fait via la méthode start(). La méthode main() se trouve elle-même dans un thread (le thread principal)*

EXERCICE 7. *modifier le programme précédent pour que le Thread créé utilise la propriété name de sa classe parente Thread pour faire l'affichage de son nom dans la boucle (faire appel les methodes setName() et getName() de la classe Thread).*

Indication : *pour acceder au thread de main il faut utiliser la méthode Thread.currentThread().*

EXERCICE 8. *modifier le code précédent afin que les 2 threads se passent la main (utiliser la méthode Thread.yield()) après chaque affichage : les 2 affichages doivent être alternés*

EXERCICE 9. *Ajouter dans la classe la méthode suivante :*

```
1  static void Wait(long milli) {
2         System.out.println("pause de "+milli+" ms") ;
3         try {
4             Thread.sleep(milli);
5         } catch (InterruptedException x) {
6             // ignorer
7         }
8     }
```

Remplacer dans le code précédent les appels à yield() par un appel à Wait(200). Quelle est la différence ?

EXERCICE 10. *Personnaliser le nom des threads. Le nouveau Thread doit s'appeler "bleu" et le thread principale (celui main) "gris".*

EXERCICE 11. *créer à partir de l'exercice précédent une classe permettant de :*

- *Créer un nombre n de threads, n passé en paramètre de main (utiliser String[] args de main). (max de n etant 10).*
- *Chaque thread créé possède un nom de la forme "thread x" avec x de 1 à n*
- *Faire en sorte que le programme main() se bloque jusqu'à ce que les n threads aient terminé leur exécution (utiliser la méthode* `join()` *de Thread).*
  Indication : *par exemple, si main() veut attendre la fin du thread « tt », il appellera : tt.join() ;*

Chaque compteur va devoir marquer une pause aléatoire entre chaque nombre (de 0 à 10 000 millisecondes par exemple). Chaque compteur affiche sont état (exemple : "tictac : 2") et un message lorsqu'il aura fini de compter (exemple : "tictac à fini de compter jusqu'à 10").

Écrivez la classe compteur et testez-la en lançant plusieurs compteurs qui comptent jusqu'à 10. Voyez celui qui a fini le plus vite.

EXERCICE 12. *Comme vu en cours, une autre possibilité d'utiliser les Threads est d'implémenter l'interface Runnable. Cette interface contient une méthode run() qui sera appelé lorsque le thread démarre (cad après l'appel de sa méthode start()). Vous pouvez consulter la documentation en ligne de l'interface Runnable pour refaire l'exercice précédent avec cette méthode.*

EXERCICE 13. *Implémentez les solutions des exercices donnés en cours.*