Structure de données génériques en Java

TP 6
Programmation objets, web et mobiles en Java
Licence 3 Professionnelle - Multimédia

Pierre Talbot (ptalbot@hyc.io)
Université de Pierre et Marie Curie

7 novembre 2014

1 Préliminaire

Lisez tout le TP avant de commencer.

1.1 Modalité

- Ce TP est à réaliser *seul* avec un gestionnaire de version.
- Si le TP vous semble trop dur ou trop facile, n'hésitez pas à nous contacter pour adapter le sujet.

Vous pouvez récolter des points bonus :

- +1 point si le code et la documentation sont en anglais.
- +1 point si le rapport est en anglais.
- Spécifiez dans le rapport toutes fonctionnalités supplémentaires non précisées dans l'énoncé si vous pensez que ça mérite des points supplémentaires.

Les points donnés pour le code prennent en compte ces paramètres :

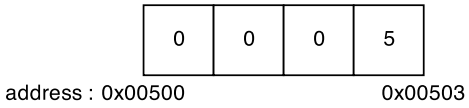
- Clarté du code (bon découpage en petites méthodes, bon nommage des variables, ...).
- Redondance du code (pas de répétition !).
- L'extensibilité (peut-on ajouter facilement une fonctionnalité ?).
- La justesse du code.

1.2 La mémoire en Java

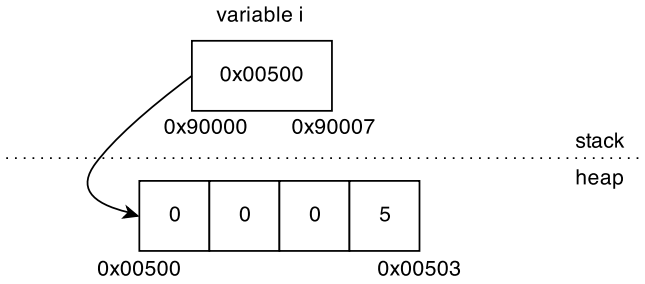
La mémoire est organisée de façon linéaire et vous pouvez, via Java, demander des blocs de mémoire avec l'opérateur `new`. Nous utiliserons la classe suivante en guise d'exemple :

```
public class Integer {
  private int x;
  public Integer(int x) { this.x = x; }
}
```

Que se passe t'il quand on fait `Integer i = new Integer(5);` au niveau de la mémoire ? Il faut réserver une zone de mémoire d'une taille suffisante pour contenir un entier, généralement codé sur 4 octets pour pouvoir stocker le nombre, on a donc :



Mais ce n'est pas tout, en Java toutes variables contenant un objet utilise une *indirection*, c'est-à-dire que la variable contient en faite l'adresse de la zone mémoire concernée [1], ainsi on a :



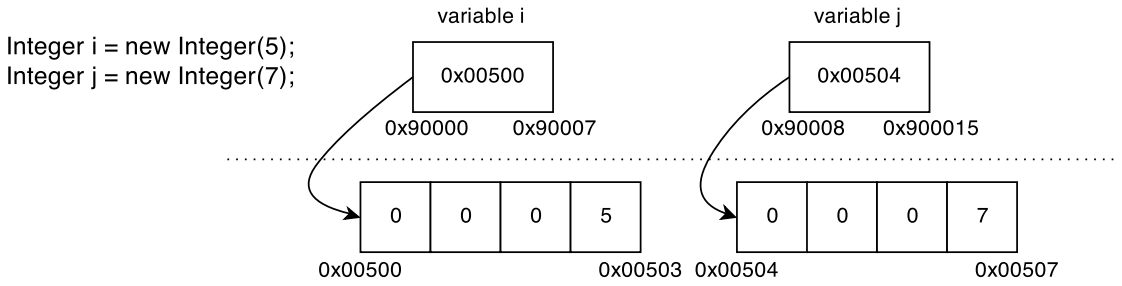
On notera que la mémoire est divisée en deux, d'un côté la *pile* (ou *stack*) et de l'autre le *tas* (ou *heap*). Toutes les variables sont stockées sur la pile mais les zones mémoires vers lesquelles elles pointent sont allouées sur le tas. Il faut juste retenir que dans un programme, on accède au tas que via une variable contenant l'adresse de la zone mémoire du tas, et que cette variable est sur la pile.

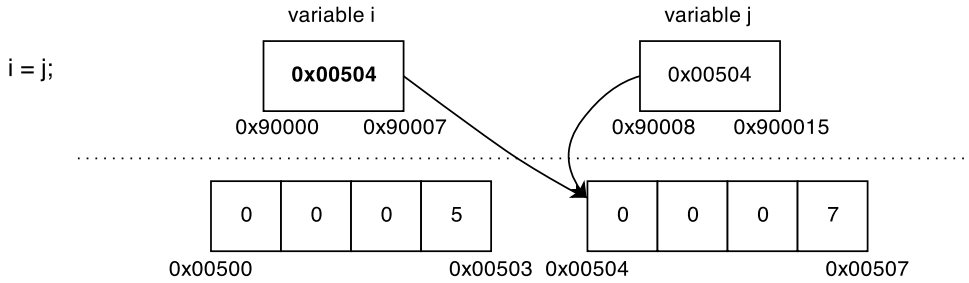
Admettons maintenant qu'on ait la séquence de code suivant :

```
Integer i = new Integer(5);
Integer j = new Integer(7);
i = j;
```

Que se passe t-il au niveau de la mémoire ? Comme on peut le voir sur le schéma suivant, la variable i pointe vers la même zone mémoire que j, ce qui signifie qu'on peut modifier un même objet via deux variables.

1. On appelle ça un pointeur dans les langages bas-niveau comme `C`, il n'y a aucune différence technique avec les références.





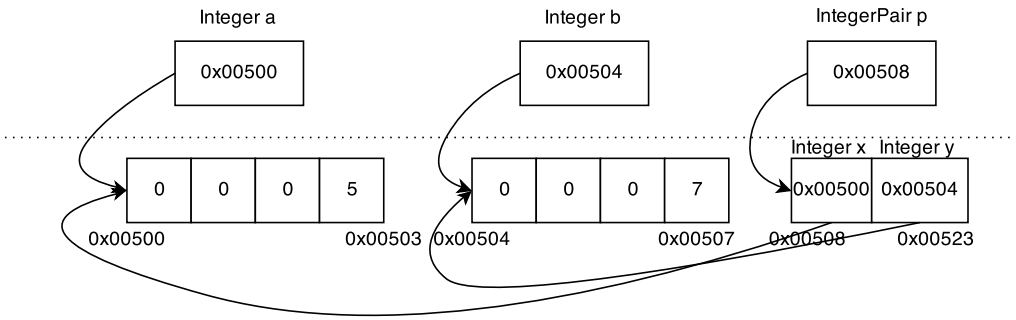
La zone mémoire sur laquelle i pointait est maintenant inaccessible, on ne pourra plus jamais la réutiliser et ça sera le *garbage collector* qui nettoiera cette zone et la rendra libre à nouveau.

Il se peut également que les variables membres d'un objet pointent vers d'autres objets, considérons le code suivant.

```
public class IntegerPair {
  private Integer x;
  private Integer y;
  public IntegerPair(Integer x, Integer y) {
    this.x = x;
    this.y = y;
  }
}

Integer a = new Integer(7);
Integer b = new Integer(5);
IntegerPair p = new IntegerPair(a, b);
```

On peut représenter la mémoire de cet objet comme sur le schéma suivant. Remarquez que les membres x et y pointent vers la même zone mémoire que a et b.



Finalement, il faut faire la distinction entre les types primitifs (`int`, `double`, `char`, ...) et les objets (`String`, `ArrayList`, ...) car les premiers ne demande pas de blocs de mémoire sur le tas via un `new`, mais ils sont automatiquement alloués sur la pile. On a vu avant que les variables sur la pile contiennent des adresses vers le tas, au lieu d'une adresse on aura directement un entier ou un autre élément primitif. Une des conséquences de cette différence est, que lorsqu'on fait une affectation `int i=9; int j=2; j=i;`, la valeur est copiée et non l'adresse de la valeur. Donc on a bien deux éléments distincts, la modification de l'un ne changera pas l'autre. Notons que pour réellement copier des objets, il faut utiliser la méthode `clone` (qui doit être implémentée à la main par l'objet en question).

2 Dessine-moi la mémoire

Maintenant que nous avons vu comment la mémoire est représentée, nous allons nous entraîner à faire ce genre de schéma. Les réponses aux exercices suivants seront donc uniquement des schémas, vous expliquerez aussi avec des mots ce que le schéma représente. Vous pouvez faire les schémas via un programme, ceux de ce TP ont été réalisé avec le site `www.draw.io`, néanmoins vous pouvez simplement dessiner ces schémas à la main (*proprement*) et les ajouter dans votre rapport électronique en les scannant. Vous pouvez également tester et modifier les programmes pour expérimenter vos réponses.

Exercice 1. Représenter l'état de la mémoire des variables i et j à la fin du programme suivant :

```
int i = 9;
Integer j = new Integer(i);
```

Exercice 2. Représenter l'état de la mémoire de la variable *numbers* à la fin du programme suivant. Noter qu'un tableau consiste juste en plusieurs cases mémoires adjacentes.

```
int numbers[] = new int[6];
numbers[3] = 99;
```

Exercice 3. Sachant que `String str = "abc";` est équivalent à :

```
char data[] = {'a', 'b', 'c'};
String str = new String(data);
```

Représentez la mémoire des variables *name*, *subname* et *subname*2 à la fin du programme suivant, consulter la documentation pour les méthodes que vous ne connaissez pas.

```
String name = new String("Giselle");
String subname = name.substring(2, 4);
String subname2 = name.clone().substring(1, 3);
```

Exercice 4. Représenter l'état de la mémoire des variables *me* et *mother* au point (a) et puis au point (b).

```
public class Person {
  private Person mother;
  public Person() { mother = null; }
  public my_mother_is(Person p) { mother = p; }
}

Person me = new Person();
Person mother = new Person();
// (a)
me.my_mother_is(mother);
// (b)
```

Exercice 5. Représenter l'état de la mémoire de la variable *person* au point (a), (b) et (c) et de la variable p au point (b).

```
public Person {
  private int age;
  public Person(int age) {
    this.age = age;
  }

  static void make_new(Person p) {
    p = new Person(9);
    // (b)
  }
}

Person person = new Person(1);
// (a)
Person.make_new(person);
// (c)
```

3 Tableau

L'opérateur `new` de Java permet également de créer des tableaux contenant un certain nombre de bloc mémoire de même taille. Par exemple, `int[] t = new int[10];` contiendra 10 blocs mémoires de 4 octets (taille d'un entier en Java), donc une zone mémoire de 40 octets en tout. La classe `ArrayList` englobe un tableau et propose des opérations supplémentaires comme l'ajout, la suppression, la recherche d'un élément,... Un tableau est par défaut de taille fixe, pour l'agrandir il faut en recréer un nouveau plus grand et copier les données du premier dans le nouveau, cette opération est notamment cachée par la classe `ArrayList`. Le but de cette exercice est de recréer une classe `ArrayList` en se basant sur celle déjà entamée du cours 6. Les opérations qui vous sont demandées sont

données juste après, la spécification de ces opérations correspond exactement à celles de la classe `ArrayList` de Java, utilisez la documentation officielle.

```
public class ArrayList<T> {
  public ArrayList() { /* ... */ }
  public int size() { /* ... */ }
  public void add(T e) { /* ... */ }
  public T get(int i) { /* ... */ }
  public int indexOf(Object o) { /* ... */ }
  public E remove(int index) { /* ... */ }
  public boolean remove(Object o) { /* ... */ }
}
```

4 Liste chaînée

Les tableaux sont seulement une façon de stocker des données, il en existe beaucoup d'autres et notamment les listes chaînées. Avant tout, posons-nous la question, pourquoi est-ce que les tableaux ne suffisent pas ?

La plupart du temps, c'est la structure que vous utiliserez néanmoins ils peuvent se révéler peu performant pour certaines opérations. Comme vous l'avez probablement remarqué, l'opération `remove` est inefficace si on veut enlever le premier élément du tableau. Il va décaler tous les autres éléments d'un "cran", ce qui veut dire qu'il y aura une boucle de n étapes. Les listes chaînées permettent entre autres de réaliser cette opération en temps constant (qui ne dépend pas du nombre d'élément dans la liste), mais a également ses propres faiblesses.

Le code suivant vous montre la structure d'une liste chaînée, néanmoins pour les exercices soyez libre de la modifier à votre convenance.

```
public class LinkedList<T> {
  private Node<T> head;

  class Node<T> {
    private T data;
    private Node next;
  }
}
```

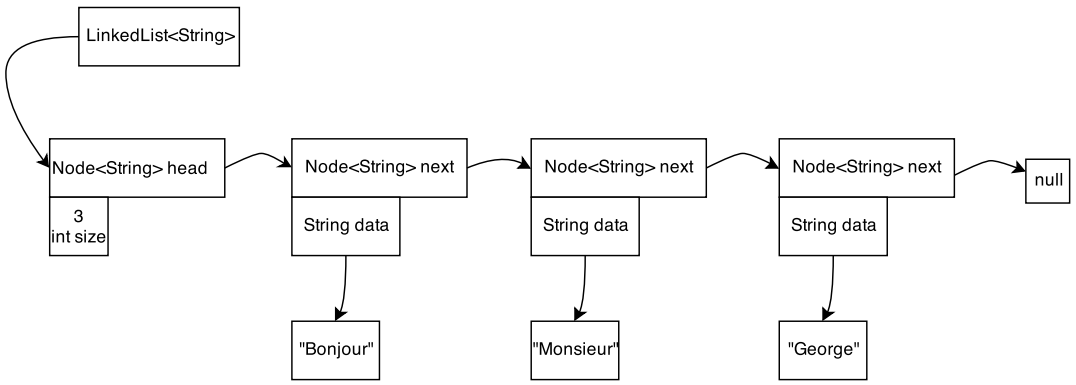
Concrètement on connaît le premier élément de la liste et chaque élément connaît le suivant (en plus de stocker des données). Si la liste est vide alors `head == null`, et plus généralement le dernier élément de la liste a son membre `next` mis à `null`.

On va se familiariser avec cette nouvelle structure en faisant encore des dessins. Nous considérons l'implémentation donnée ci-dessus et les méthodes disponibles sur la classe Java "officiel" `LinkedList`. En guise d'exemple le schéma du programme suivant est donné.

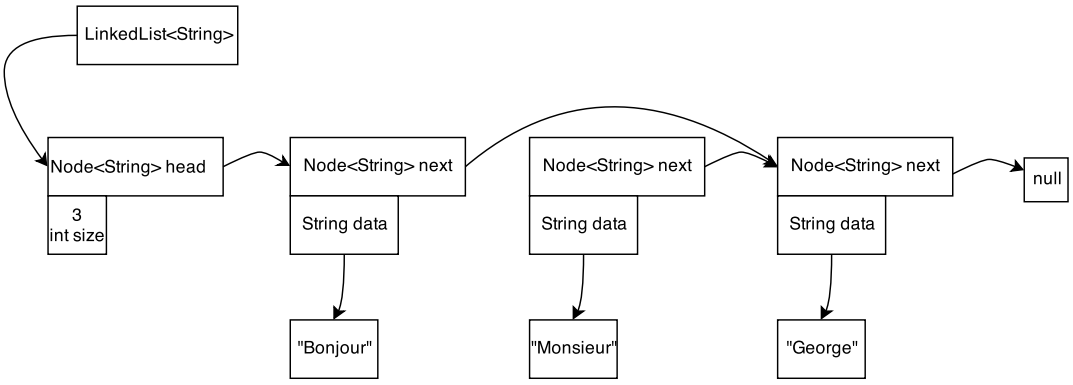
```
LinkedList<String> l = new LinkedList<String>();
l.add(new String("Bonjour"));
l.add(new String("Monsieur"));
```

```
l.add(new String("George"));
// (a)
l.remove(1);
// (b)
```

La figure suivante est donnée sans notion explicite d'adresse vu que vous avez compris les références, on peut maintenant s'abstraire de cette notion (sans pour autant l'oublier). La représentation mémoire est donnée jusqu'au point (a).



Lorsqu'on efface un élément, au point (b), celui-ci n'est plus pointé par l'élément précédent, mais pointe toujours vers le nœud suivant. Néanmoins, vu qu'il n'est plus accessible, il sera collecté par le *garbage collector* et effacé. On note le nœud précédent pointe maintenant directement sur le dernier nœud.



Lorsque vous implémenterez les opérations démandée ci-après, il est fort utile de se faire un petit exemple sous cette forme pour vérifier que tout se passe bien.

4.1 Implémentation

Vous implémenterez les opérations suivantes en suivant la documentation de Java.

```
public class LinkedList<T> {
  public LinkedList() { /* ... */ }
  public int size() { /* ... */ }
  public void addFirst(T e) { /* ... */ }
  public void addLast(T e) { /* ... */ }
  public T get(int i) { /* ... */ }
  public E pop() { /* ... */ }
  public E poll() { /* ... */ }
  public E peekFirst() { /* ... */ }
  public E peekLast() { /* ... */ }
  public boolean remove(Object o) { /* ... */ }
  public E remove(int index) { /* ... */ }
}
```

4.2 Liste doublement chainée

Même exercice mais avec des listes doublement chainées. Au lieu d'avoir une unique variable `next`, la classe `Node` possède également une variable `previous` pointant vers le nœud précédant. Vous améliorez le code précédent avec cette nouvelle notion.

5 Iterator

En guise d'exemple, prenons ce code :

```
public void print_all(LinkedList<String> names) {
  for(int i = 0; i < names.size(); ++i) {
    System.out.println(names.get(i));
  }
}
```

La méthode `get(i)` n'est pas constante, elle doit elle-même faire une boucle pour arriver à l'élément i, ce qui fait que ce code est très inefficace. Néanmoins vu qu'on parcourt la liste du début à la fin, il ne devrait pas y avoir de surcoût. On pourrait arriver à nos fins avec une méthode qui expose le type `Node` mais on voudrait que ce détail d'implémentation reste caché.

L'astuce est d'utiliser une classe qui "itère" (c'est-à-dire traverse) la collection pour nous. Pour ça il faut que la collection (`ArrayList` ou `LinkedList`) implémente l'interface `Iterable`[2] qui possède une unique méthode `Iterator<T> iterator()`. Le type de retour est donc une classe implémentant `Iterator`[3], cette classe contiendra donc les infos nécessaires pour avancer d'une étape lorsqu'on appelera `next`. Avant de continuer, montrant un exemple pour itérer sur une `ArrayList` :

```
public void print_all(ArrayList<String> names) {
  for(Iterator<String> i = names.iterator();
      i.hasNext(); )
```

2. http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html
3. http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html

```
  {
    System.out.println(i.next());
  }
}
```

Vu que c'est un peu embêtant d'écrire tout ça, Java nous donne un raccourci bien pratique qu'on appele le *for each* :

```
public void print_all(ArrayList<String> names) {
  for(String s : names)
  {
    System.out.println(s);
  }
}
```

C'est strictement équivalent au code précédent.

Exercice. On vous demande donc d'implémenter `Iterator` pour les 2 classes que vous avez créées avant, soit `ArrayList` et `LinkedList`. Noter que la classe `Iterator` est généralement une *inner* classe du container.

6 Graphe

Ce dernier exercice est un travail de recherche et est à réaliser seulement si les autres sont terminés. On vous demande d'implémenter une structure de données de graphe implémenté avec une liste d'adjacence et de fournir l'algorithme du plus court chemin de Dijkstra. Vous pouvez commencer par consulter la page Wikipedia sur les graphes pour les premières explications.