Feuille 10 - Iterator et Lambda

1 Iterator

Considérez le programme suivant :

```
1  List<Integer> l = new LinkedList<>();
2  for(int i=0; i<100000; i++) l.add(i);
3  System.out.println("Done adding!");
4  int sum;
5  /* Loop 1 */
6  System.out.println("Calculate with loop");
7  sum=0;
8  for(int i=0;i<100000; i++) sum+=l.get(i);
9  System.out.println("Done loop 1: sum = "+sum);
10 /* Loop 2 */
11 System.out.println("Calculate with implicit iterator");
12 sum=0;
13 for(Integer i:l) sum+=i;
14 System.out.println("Done loop 2: sum = "+sum);
```

Le programme contient deux boucles qui font la même chose.

— Y a-t-il une différence entre les deux méthodes ? Quelle boucle est plus efficace ?

Regardons un peu plus attentivement la deuxième boucle. Comment est-que ça marche ? Pour que la notation `for(Integer i:l)` soit acceptée par le compilateur il faut que le type de `l` soit un sous-type de l'interface `Iterable<Integer>`. En fait, si on regarde la documentation de Java on voit que la classe `List` implémente l'interface `Iterable<T>`. L'interface `Iterable` déclare une méthode `Iterator<T> iterator()` qui retourne un objet **capable de parcourir la liste**. Un objet de type `Iterator<T>` nous donne deux méthodes :

— `boolean hasNext()` qui teste si on est au bout de la liste.

— `T next()` qui retourn l'élément suivant de la liste et avance l'iterator.

Alors, une autre manière, plus explicite, d'écrire la deuxième boucle est :

```
1 System.out.println("Calculate with explicit iterator");
2 sum=0;
3 Iterator<Integer> it = l.iterator();
4 while(it.hasNext()) sum += it.next();
5 System.out.println("Done loop 3: sum = "+sum);
```

2 Écrire un Iterator

Considérez la class suivante :

```
1  class myList {
2          private Integer [] data;
3          private final int SIZE = 50;
4          public myList() {
5                  data = new Integer[SIZE];
6                  for(int i=0; i<SIZE; i++)
7                          data[i] = i;
8          }
9          public Iterator<Integer> evenIterator(){
10                 /*..*/
11         }
12         public static void main(String [] args){
13                 myList l = new myList();
14                 Iterator<Integer> it = l.evenIterator();
15                 while(it.hasNext()) System.out.println(it.next());
16         }
17 }
```

Le méthode `evenIterator` doit retourner un objet de type `Iterator<Integer>` qui permet de parcourir que les éléments de la liste qui se trouvent dans des positions paires. Alors, la boucle de la ligne 15 doit afficher les éléments $0, 2, 4, 6, 8, \ldots$.

Pour implémenter cette méthode vous devez déclarer une nouvelle classe qui implémente l'interface `Iterator<Integer>` et qui redéfinit les méthodes `next` et `hasNext`. **NB :** il vaut mieux mettre cette classe comme classe interne de la class `myList`, puisque cette nouvelle classe a besoin d'utiliser des attributs internes de l'objet actuel (comme, par exemple, `data,SIZE`). En plus, puisque la méthode `evenIterator` ne va construire qu'un seul objet de cette classe, il vaut mieux utiliser une classe anonyme.

3 Iterator avec une condition

Maintenant on va ajouter une nouvelle méthode à notre classe :

```
1  import java.util.*;
2  import java.util.function.*;
3  class myList {
4          private Integer [] data;
5          private final int SIZE = 50;
6          public myList() { ... }
7          public Iterator<Integer> iteratorIf(Predicate<Integer> p){
8                  /*...*/
9          }
10         public static void main(String [] args){
11                 System.out.println("Hi!");
12                 myList l = new myList();
13                 Iterator<Integer> it = l.iteratorIf((x-> x%5==2));
14                 while(it.hasNext()) System.out.println(it.next());
15         }
16 }
```

La méthode `iteratorIf` prend comme paramètre un `Predicate<Integer>`, c'est-à-dire, un objet qui (appartient à une classe qui) a redéfini la méthode `boolean test(Integer i)`. (Regardez la documentation de l'interface `Predicate` pour le vérifier). Alors, l'idée est que la méthode `iteratorIf` parcourt la liste mais ignore tous les éléments pour lesquels la méthode `test` de l'objet `p` retourne false. La ligne 13 va, donc, afficher les éléments $2, 7, 12, \ldots$.

Programmez la méthode `iteratorIf`. Puis, modifiez la ligne 13 en sorte que le programme n'affiche que les éléments de la liste qui sont de nombres premiers.