

Programmation Orientée Objets avec Java

Chapitre 5 : Les Collections

Stéphane Malandain / Yassin Rekik
Semestre d'automne 2022

Exercice 1

Réalisez des fonctionnalités sur des listes. Complétez le code ci-dessous:

```
1 public class ListHelper {
2
3     /* Retourne une nouvelle liste où toutes les valeurs sont doublées */
4     public static List<Integer> doubleThat(List<Integer> is) {
5     }
6     /* Retourne une nouvelle liste où toutes les valeurs sont plus grandes ou égale à un seuil */
7     public static List<Integer> filterUpper(List<Integer> is, int value) {
8     }
9
10    public static void main(String[] args) {
11        /* Vos tests éventuels */
12    }
13 }
```

```
1 import java.util.List;
2 import java.util.ArrayList;
3
4 public class ListHelper {
5
6     /* Retourne une nouvelle liste où toutes les valeurs sont doublées */
7     public static List<Integer> doubleThat(List<Integer> is) {
8         List<Integer> result = new ArrayList<>();
9         for(int i: is) {
10             result.add( i * 2 );
11         }
12         return result;
13     }
14
15     /* Retourne une nouvelle liste où toutes les valeurs sont plus grandes ou
16        égales à un seuil */
17     public static List<Integer> filterGreaterOrEqual(List<Integer> is, int value) {
18         List<Integer> result = new ArrayList<>();
19         for(int i: is) {
20             if( i >= value ){
21                 result.add( i );
22             }
23         }
24         return result;
25     }
26 }
```

```

26
27     /* Retourne une nouvelle liste où toutes les valeurs positives sont
28     multipliées par deux. Les valeurs négatives sont omises */
29     public static List<Integer> filterPositiveAndThenDoubleThem(List<Integer> is) {
30         // Faites simple, ne cherchez pas l'efficacité
31         return doubleThat( filterGreaterOrEqual(is, value: 0) );
32     }
33
34
35     Run | Debug
36     public static void main(String[] args) {
37
38         System.out.println( filterPositiveAndThenDoubleThem(List.of(-3, -2, -1, e4: 0, e5: 1, e6: 2, e7: 3)) );
39         // ==> 0, 2, 4, 6
40     }
41 }

```

Exercice 2

Réalisez une classe `StringUtil`.

Cette classe doit fournir une première méthode statique qui prend un argument une chaîne de caractères de type `String` et retourne une `Map` dont les clés sont les caractères qui se trouvent dans la chaîne et la valeur le nombre d'occurrences de la lettre dans la chaîne.

Elle fournit une seconde méthode statique prenant une liste de chaînes de caractères en argument et compte la taille moyenne d'une chaîne dans la liste.

Exercice 3

Ecrivez une classe `ListUtil` qui comprend deux méthodes statiques :

- Une méthode qui prend une liste de doubles en argument et retourne la moyenne de tous les éléments.
- Une méthode qui prend une liste de doubles en argument et retourne une nouvelle liste de tous les éléments positifs appartenant à la première.

Exercice 4

Soit la collection ci-dessous :

```

1  Deque<String> stack = new LinkedList<>();
2  stack.addFirst("Bonjour");
3  stack.addFirst("Hello");
4  stack.addFirst("Hi");

```

Complétez les méthodes permettant de parcourir et d'afficher les éléments de cette collection en utilisant les méthodes associées au type de l'argument (`Iterable`, `Iterator` ou `Collection`) :

```

1 interface Loop {
2     public static void loop(Iterable<String> iterable) { ... }
3     public static void loop(Iterator<String> iterator) { ... }
4     public static void loop(Collection<String> coll) { ... }
5 }

```

Laquelle des trois méthodes sera appelée lors de cet appel:

```
Loop.loop( stack );
```

Exercice 5

Réécrivez correctement une classe `Properties` qui supprime totalement la notion d'héritage. Choisissez judicieusement la structure à utiliser pour stocker des propriétés. (indication : du côté des map 😊)

```

1 class Properties {
2     public String getProperty(String key) {
3         /* TODO */
4     }
5     public String getPropertyOrElse(String key, String defaultValue) {
6         /* TODO */
7     }
8     public void addProperty(String key, String value) {
9         /* TODO */
10    }
11    public List<String> keys() {
12        /* TODO */
13    }
14    public List<String> values() {
15        /* TODO */
16    }
17    public ????? allProperties() {
18        /* TODO */
19    }

```

```
1  import java.util.Map;
2  import java.util.HashMap;
3  import java.util.List;
4  import java.util.ArrayList;
5
6  public class Properties {
7
8      private Map<String, String> props = new HashMap<>();
9
10     public String getProperty(String key) {
11         if (this.props.containsKey(key)) {
12             return this.props.get(key);
13         }
14         throw new RuntimeException("oups..");
15
16         /*
17         String res = props.get(key);
18         if (res == null) {
19             throw new RuntimeException("oups..");
20         }
21         return res;
22         */
23     }
24
25     public String getPropertyOrElse(String key, String defaultValue) {
26         return this.props.getOrDefault(key, defaultValue);
27
28         /*
29         if (this.props.containsKey(key)) {
30             return this.props.get(key);
31         }
32         return defaultValue;
33         */
34     }
35
36     public void addProperty(String key, String value) {
37         this.props.put(key, value);
38     }
39
40     public List<String> keys() {
41         return new ArrayList<>(this.props.keySet());
42         // return List.copyOf( this.props.keySet() );
43     }
44 }
```

```

45     public List<String> values() {
46         // return new ArrayList<>(this.props.values());
47         return List.copyOf( this.props.values() );
48     }
49
50     public List<Map.Entry<String,String>> allProperties() {
51         return List.copyOf( this.props.entrySet() );
52     }
53
54     @Override
55     public boolean equals(Object o) {
56         if (this == o) {
57             return true;
58         }
59         if (o == null || o.getClass() != this.getClass()) {
60             return false;
61         }
62         Properties that = (Properties)o;
63         return this.props.equals(that.props);
64     }
65
66     @Override
67     public int hashCode() {
68         return this.props.hashCode();
69     }
70
71     @Override
72     public String toString() {
73         return "Properties(" + this.props.toString() + ")";
74     }
75
76     public static void main(String[] args) {
77         /* tests */
78         Properties p = new Properties();
79         p.addProperty("server-a", "192.168.1.5");
80         p.addProperty("server-b", "192.168.1.6");
81         p.addProperty("login", "root");
82         Properties p2 = new Properties();
83         p2.addProperty("server-a", "192.168.1.5");
84         p2.addProperty("server-b", "192.168.1.6");
85         p2.addProperty("login", "root");
86
87         System.out.println( p.equals(p2) );
88     }
89 }

```

Exercice 6

Vous devez réaliser une implémentation d'une liste dynamique d'entiers. Votre implémentation, appelée `ArrayListInt` doit **impérativement** utiliser un tableau statique d'entiers pour simuler le comportement d'une telle liste.

Voici une utilisation possible :

```
1 public class exo6 {
2     public static void main(String[] args) {
3         ListInt list = new ArrayListInt();
4         list.insert(0);
5         list.insert(3);
6         list.insertAll(2,1); // insertAll prend un nombre arbitraire d'éléments
7         System.out.println( "Size: " + list.size() );
8         for (int i = 0; i < list.size(); i+=1) {
9             int v = list.get(i);
10            System.out.println("Value: " + v);
11        }
12        list.clear();
13        System.out.println( "Size: " + list.size() );
14        /*
15         * Cet exemple afficherait
16         * Size: 4
17         * Value: 0
18         * Value: 3
19         * Value: 2
20         * Value: 1
21         * Size: 0
22         */
23    }
24 }
```

Vous êtes libre d'utiliser un tableau de type `int` primitif (`int[]`) ou un tableau d'objets `Integer` (`Integer[]`).

Contraintes supplémentaires

- Implémentez tous les composants pour que le code ci-dessus compile et s'exécute correctement
- Votre implémentation doit respecter le contrat de `ListInt`
- Les méthodes `isEmpty` et `addAll` doivent être **concrètes** dans `ListInt`
- L'implémentation de `ArrayListInt` réserve initialement une taille de tableau de 10 éléments. A chaque dépassement de capacité, vous devez allouer 10 éléments supplémentaires.

```
1 interface ListInt {
2     int get(int indice);
3     void insert(int value);
4     default boolean isEmpty() {
5         return size() == 0;
6     }
7     int size();
8     default void insertAll(int... is) {
9         for(int i: is) {
10             insert(i);
11         }
12     }
13     void clear();
14 }
```

- Utilisez uniquement ces fonctionnalités suivantes sur les tableaux statiques:

- l'attribut `length` qui retourne la taille d'un tableau
- la notation crochet pour modifier ou extraire une valeur du tableau
- la boucle de parcours

```
1  class ArrayListInt implements ListInt {
2
3      private final int INC_CAPACITY = 10;
4      private int currentSize = INC_CAPACITY;
5      private int[] list = new int[currentSize];
6      private int nextFree = 0;
7
8      public int get(int indice) {
9          if ( indice >= size() ) {
10             throw new RuntimeException(message: "Empty List");
11         }
12         return list[indice];
13     }
14     private void enlargeCapacity() {
15         int[] nextStack = new int[currentSize + INC_CAPACITY];
16         for(int i = 0; i < currentSize; i++){
17             nextStack[i] = list[i];
18         }
19         list = nextStack;
20         currentSize += INC_CAPACITY;
21     }
22     public void insert(int value) {
23         if( nextFree == currentSize ) {
24             enlargeCapacity();
25         }
26         list[nextFree] = value;
27         nextFree += 1;
28     }
29     public int size() {
30         return nextFree;
31     }
32     public void clear() {
33         nextFree = 0;
34     }
35 }
```

Exercice 7

Reprenez l'exercice précédent sur les listes d'entiers et réalisez votre propre itérateur sur celle-ci.

```
1  interface ListInt extends Iterable<Integer> {
2      int get(int indice);
3      void insert(int value);
4      default boolean isEmpty() {
5          return size() == 0;
6      }
7      int size();
8      default void insertAll(int... is) {
9          for(int i: is) {
10             insert(i);
11         }
12     }
13     void clear();
14 }
```

```
1  import java.util.Iterator;
2
3  class ArrayListInt implements ListInt {
4      class ArrayIterator implements Iterator<Integer> {
5          private int index = 0;
6          public boolean hasNext() {
7              return index < ArrayListInt.this.nextFree;
8          }
9
10         public Integer next() {
11             int res = ArrayListInt.this.list[index];
12             index += 1;
13             return res;
14         }
15     }
16
17     private final int INC_CAPACITY = 10;
18     private int currentSize = INC_CAPACITY;
19     private int[] list = new int[currentSize];
20     private int nextFree = 0;
21
22     public int get(int indice) {
23         if ( indice >= size() ) {
24             throw new RuntimeException(message: "Empty List");
25         }
26         return list[indice];
27     }
```



```

28 private void enlargeCapacity() {
29     int[] nextStack = new int[currentSize + INC_CAPACITY];
30     for(int i = 0; i < currentSize; i++){
31         nextStack[i] = list[i];
32     }
33     list = nextStack;
34     currentSize += INC_CAPACITY;
35 }
36 public void insert(int value) {
37     if( nextFree == currentSize ) {
38         enlargeCapacity();
39     }
40     list[nextFree] = value;
41     nextFree += 1;
42 }
43 public int size() {
44     return nextFree;
45 }
46 public void clear() {
47     nextFree = 0;
48 }
49 public Iterator<Integer> iterator() {
50     return new ArrayIterator();
51 }
52 }

```

```

1  import java.util.Iterator;
2
3  public class App {
4      Run | Debug
5      public static void main(String[] args) {
6          ListInt list = new ArrayListInt();
7          list.insert(value: 0);
8          list.insert(value: 1);
9          list.insertAll(...is: 2,3,10,11,12,13,21,22,23,24);
10
11         /* l'itérateur permet maintenant le code ci-dessous */
12         for (int v: list) {
13             System.out.println("Value: " + v);
14         }
15
16         /* ou */
17         Iterator<Integer> it = list.iterator();
18         while (it.hasNext()) {
19             System.out.println("Value: " + it.next());
20         }
21
22         /* et même */
23         list.forEach( v -> System.out.println("Value: " + v));
24     }
25 }
26

```