

Programmation Orientée Objets avec Java

Chapitre 6 : Types Imbriqués

Chapitre 7 : Les exceptions

Chapitre 8 : La généricité

Stéphane Malandain / Yassin Rekik

Semestre d'automne 2023

Exercice 1

Complétez le code pour que les appels ci-dessous fonctionnent :

```
1 interface Pushable {  
2     void push();  
3 }
```

```
1 public class Test {  
2     public static void push(Pushable p) {  
3         p.push();  
4         System.out.println("Button has been pushed");  
5     }  
6  
7     public static void main(String[] args) {  
8         /* complétez le code ci-dessous */  
9         push(  
10  
11  
12  
13     ); /* Doit afficher:  
14         Push  
15         Button has been pushed  
16     */  
17 }  
18 }
```

Exercice 2

Reprenez l'exercice 6 de la série 3_4, le simulateur de machine; Modifiez-le pour intégrer `On`, `Off` et `Err` en tant que classes internes de l'interface `Status`.

Exercice 3

Reprenez l'exercice 6 de la série 5 sur les listes d'entiers (`ListInt` - `ArrayListInt`) et réalisez votre propre itérateur sur celle-ci.

```

1
2 ListInt list = new ArrayListInt();
3 list.insert(0);
4 list.insert(3);
5 list.insertAll(2,1);
6
7 for (int i = 0; i < list.size(); i+=1) {
8     int v = list.get(i);
9     System.out.println("Value: " + v);
10 }
11
12 /* l'itérateur permet maintenant le code ci-dessous */
13 for (int v: list) {
14     System.out.println("Value: " + v);
15 }
16
17 /* ou */
18 Iterator<Integer> it = list.iterator();
19 while (it.hasNext()) {
20     System.out.println("Value: " + it.next());
21 }
22
23 /* et même */
24 list.forEach( v -> System.out.println("Value: " + v));
25

```

Exercice 4

Ecrivez une classe `IntegerArrayStack` qui hérite de l'interface `IntegerStack`. Utilisez une `ArrayList` pour simuler le comportement de votre pile. Retournez l'exception (unchecked) `EmptyIntegerStack` lorsque c'est nécessaire.

Fournissez **le maximum** de méthodes concrètes (au moins quatre) dans l'interface.

```

1 import java.util.Optional;
2
3 public interface IntegerStack {
4     int pop(); // retourne et enlève l'élément de tête
5     Optional<Integer> popOption();
6     void ifHeadIsPresent(Consumer<Integer> consumer);
7     /* un consumer est un interface fonctionnelle en Java représentant une
8      * Opération qui accepte un paramètre d'entrée et ne renvoie rien; elle 'consomme'
9      */
10    void push(int i); // ajoute un élément
11    int size(); // retourne la taille
12    int peek(); // lit la valeur de tête tout en la conservant
13    Optional<Integer> peekOption();
14    boolean isEmpty();
15 }

```

Extrait d'utilisation :

```

1 IntegerStack stack = new IntegerArrayStack();
2 stack.push(1);
3 stack.push(2);
4 stack.push(3);
5 stack.ifHeadIsPresent( v -> System.out.println("head: " + v) ); // head: 3
6

```

Exercice 5

Reprenez l'exercice 3 déjà réalisé (qui reprend l'exercice 6 de la série 5...) sur les listes d'entiers et ajoutez une nouvelle classe `LinkedListInt` qui implémente `ListInt`. Concrétisez cette liste en réalisant vous-même une liste chaînée.

L'illustration de la figure 1 représente l'état de la liste après avoir exécuté le code ci-dessous :

```
1  ListInt list = new LinkedListInt();
2  list.insert(12);
3  list.insertAll(99, 37);
4
```

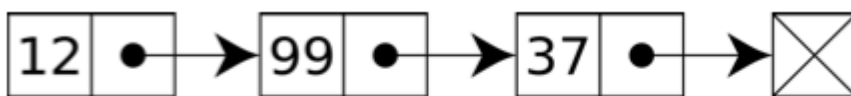


Figure 1: Liste chaînée - source: wikipedia.org

Vous êtes libres d'insérer les éléments à la fin ou en tête de liste. Cependant, le parcours d'un itérateur doit respecter l'ordre d'insertion. Rappelez-vous que `ListInt` est un `Iterable`.

Exercice 6

Réalisez une méthode test qui prend en argument un `Supplier<Integer>` et retourne un `Optional<Integer>`. Cette méthode a pour but de déléguer une exécution et de retourner un optionnel vide en cas de problème.

- ❑ si l'exécution du fournisseur n'a pas levé d'exception, la valeur est encapsulée dans un `Optional` (lignes 1 et 2)
- ❑ retourne un `Optional` vide sinon (lignes 3 à 7)

Exemple d'utilisation:

```
1  test( () -> 3 ) // retourne Optional contenant 3
2  test( () -> 3 + 4 ) // retourne Optional contenant 7
3  test( () -> 3 / 0 ) // retourne Optional vide
4  test( () -> null ) // retourne Optional vide
5  test( () -> {
6  |    throw new RuntimeException()
7  }) // retourne un Optional vide
8
```

Remarque : Le `Supplier` est utilisé pour déléguer une opération. Si la méthode `test` prenait un `int` au lieu d'un `Supplier<Integer>`, le problème surviendrait avant d'appeler la méthode, c'est-à-dire, lors de l'évaluation des arguments !

Exercice 7

L'interface `Comparable<T>` indique qu'il est possible de comparer un type. Elle oblige à redéfinir la méthode `int compareTo(T o)`

La classe `Integer` hérite de `Comparable<Integer>` ; la classe `String` hérite quant à elle de `Comparable<String>`

```
1 public final class Integer implements Comparable<Integer> ... {
2     ...
3     public int compareTo(Integer anotherInteger) { ... }
4 }
5 public final class String implements Comparable<String> ... {
6     ...
7     public int compareTo(String anotherString) { ... }
8 }
```

Modifiez la classe `Box<T>` pour permettre de comparer deux boîtes:

- Une `Box` doit respecter l'interface `Comparable`
- Pour comparer des `Boxs`, il suffit de comparer la valeur qu'ils encapsulent. Le paramètre est donc lui aussi comparable !
- Créer une méthode statique utilitaire pour préciser si une `Box` est plus grande qu'une autre

Exemple d'utilisation :

```
1 Box<Integer> b1 = new Box<>(1);
2 Box<Integer> b2 = new Box<>(2);
3 Util.isBigger(b1, b2); // doit retourner false
```

Exercice 8

Reprenez l'exercice sur la pile (`IntegerStack`) et rendez-la générique.