

Programmation Orientée Objets avec Java

Chapitre 6 : Types Imbriqués

Chapitre 7 : Les exceptions

Chapitre 8 : La généricité

Stéphane Malandain / Yassin Rekik

Semestre d'automne 2022

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(x: "Button has been pushed");  
5     }  
6  
7     Run | Debug  
8     public static void main(String[] args) {  
9         /* completez le code ci-dessous */  
10        push(  
11  
12  
13        ); /* Doit afficher:  
14            Push  
15            Button has been pushed  
16        */  
17    }  
18 }
```

Correction :

```
5
6 public class Test {
7
8     public static void push(Pushable p) {
9         p.push();
10        System.out.println(x: "Button has been pushed");
11    }
12
13    Run | Debug
14    public static void main(String[] args) {
15        /* completez le code ci-dessous */
16        push( new Pushable(){
17            public void push() {
18                System.out.println(x: "Push");
19            }
20        });
21        /* Doit afficher:
22        Push
23        Button has been pushed
24        */
25        // avec interface fonctionnelle (fonctionne même sans @FunctionalInterface)
26        push( () -> System.out.println(x: "Push version 2"));
27    }
28 }
```

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 .

Correction :

```
1  import java.util.Random;
2
3  interface Status {
4
5      static class On implements Status {
6          private On() {} /* maintenant le constructeur peut être privé
7              |         |         |         |         * (visible à l'intérieur de l'interface Status) */
8          public boolean isOn() { return true; }
9      }
10
11     static class Off implements Status {
12         private Off() {} // maintenant le constructeur peut être privé
13         public boolean isOff() { return true; }
14     }
15
16     static class Err implements Status {
17         private String msg;
18         private Err(String msg) { // maintenant le constructeur peut être privé
19             this.msg = msg;
20         }
21         public String getErrorMessage() { return this.msg; }
22     }
23
24     static Status process() {
25         int chance = new Random().nextInt(bound: 3);
26         if( chance == 0 ) {
27             return new On(); // Ok, constructeur visible
28         } else {
29             return chance == 1 ? new Off() : new Err(msg: "Oops"); // Ok, constructeurs visibles
30         }
31     }
32
33     static Err makeError(String message) {
34         return new Err(message); // Ok, constructeur visible
35     }
36
37     default boolean isOn() { return false; }
38     default boolean isOff() { return false; }
39     default String getErrorMessage() { throw new UnsupportedOperationException(message: "..."); }
40 }
41
42
43 public class StatusApp {
44     Run | Debug
45     public static void main(String[] args) {
46
47         Status s1 = Status.process();
48         if( s1.isOn() ) {
49             System.out.println(x: "L'appareil fonctionne");
50         } else if ( s1.isOff() ) {
51             System.out.println(x: "L'appareil est éteint");
52         } else {
53             System.out.println("L'appareil est instable: " + s1.getErrorMessage());
54         }
55         Status s2 = Status.makeError(message: "Oops");
56         // Status s3 = new Status.Err("Oops"); // KO: constructeur privé
57     }
```

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
```

Correction :

```
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 }
53

```

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.

Correction :

```
1 public class EmptyIntegerStack extends RuntimeException {
2     public EmptyIntegerStack() {
3         super(message: "Stack is empty");
4     }
5 }
```

```
1 import java.util.Optional;
2 import java.util.List;
3 import java.util.ArrayList;
4 import java.util.function.Consumer;
5
6 public class IntegerArrayStack implements IntegerStack {
7
8     private List<Integer> stack = new ArrayList<>();
9
10    public int pop() {
11        if ( isEmpty() ) {
12            throw new EmptyIntegerStack();
13        }
14        return stack.remove( stack.size()-1 );
15    }
16
17    public void push(int i) {
18        stack.add(i);
19    }
20    public int size() {
21        return stack.size();
22    }
23
24    public int peek() {
25        if ( isEmpty() ) {
26            throw new EmptyIntegerStack();
27        }
28        return stack.get( stack.size()-1 );
29    }
30 }
```

```

1  import java.util.Optional;
2  import java.util.function.Consumer;
3
4  public interface IntegerStack {
5
6      int pop();
7
8      default Optional<Integer> popOption() {
9          if ( isEmpty() ) {
10             return Optional.empty();
11          }
12          return Optional.of( pop() );
13      }
14      // version 2 :
15      default Optional<Integer> popOption() {
16          try {
17              return Optional.of( pop() );
18          } catch (EmptyIntegerStack ex) {
19              return Optional.empty();
20          }
21      }
22
23      default void ifHeadIsPresent(Consumer<Integer> consumer) {
24          peekOption().ifPresent( consumer );
25      }
26
27      void push(int i); // ajoute un élément
28
29      int size(); // retourne la taille
30
31
32      default int peek() {
33          int res = pop();
34          push(res);
35          return res;
36      }
37
38      default int peek() {
39          Optional<Integer> res = peekOption();
40          if (res.isPresent()) {
41              return res.get();
42          }
43          throw new EmptyIntegerStack();
44      }
45
46      default Optional<Integer> peekOption() {
47          if ( isEmpty() ) {
48              return Optional.empty();
49          }
50          return Optional.of( peek() );
51      }
52
53      default boolean isEmpty() {
54          return size() == 0;
55      }
56  }
57

```

```

1 public class Test {
2     private static void log(Object o) {
3         System.out.println(o.toString());
4     }
5     public static void main(String... args) {
6         IntegerStack stack = new IntegerArrayStack();
7         stack.push(1);
8         stack.push(2);
9         stack.push(3);
10        stack.ifHeadIsPresent( v -> log("head: " + v) );
11        log( stack.size() );
12        log( stack.pop() );
13        log( stack.pop() );
14        log( stack.pop() );
15        log( stack.pop() ); // should throw
16    }
17 }

```

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

```

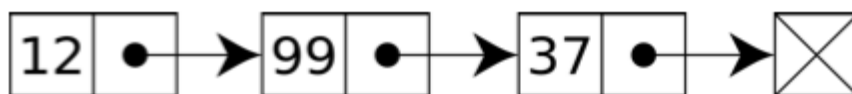


Figure 1: Liste chaînée - source: wikipedia.org

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`.

Correction :


```

1  import java.util.Iterator;
2
3  public class App {
4      public static void main(String[] args) {
5
6          ListInt list = new LinkedListInt();
7          /* Should be the same as:
8           * ListInt list = new ArrayListInt(); */
9
10         list.insert(0);
11         list.insert(1);
12         list.insertAll(2,3,10,11,12,13,21,22,23,24);
13
14         /* l'itérateur permet maintenant le code ci-dessous */
15         for (int v: list) {
16             System.out.println("Value: " + v);
17         }
18
19         /* ou */
20         Iterator<Integer> it = list.iterator();
21         while (it.hasNext()) {
22             System.out.println("Value: " + it.next());
23         }
24
25         /* et même */
26         list.forEach( v -> System.out.println("Value: " + v));
27
28     }
29 }
30

```

```

1  public 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  }
15

```

```

1  import java.util.Iterator;
2
3  public 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             if ( !hasNext() ) {
12                 throw new NoSuchElementException();
13             }
14             int res = ArrayListInt.this.list[index];
15             index += 1;
16             return res;
17         }
18     }
19
20     private final int INC_CAPACITY = 10;
21     private int currentSize = INC_CAPACITY;
22     private int[] list = new int[currentSize];
23     private int nextFree = 0;
24
25     public int get(int indice) {
26         if ( indice >= size() ) {
27             throw new RuntimeException(message: "Empty List");
28         }
29         return list[indice];
30     }
31
32     private void enlargeCapacity() {
33         int[] nextStack = new int[currentSize + INC_CAPACITY];
34         for(int i = 0; i < currentSize; i++){
35             nextStack[i] = list[i];
36         }
37         list = nextStack;
38         currentSize += INC_CAPACITY;
39
40         /* OU */
41         //this.list = Arrays.copyOf(this.list, currentSize + INC_CAPACITY);
42         //currentSize += INC_CAPACITY;
43     }
44
45     public void insert(int value) {
46         if( nextFree == currentSize ) {
47             enlargeCapacity();
48         }
49         list[nextFree] = value;
50         nextFree += 1;
51     }
52
53     public int size() {
54         return nextFree;
55     }
56
57     public void clear() {
58         nextFree = 0;
59     }
60
61     public Iterator<Integer> iterator() {
62         return new ArrayIterator();
63     }
64 }
65

```

```
1 import java.util.Iterator;
2 import java.util.Map;
3 import java.util.NoSuchElementException;
4 import java.util.Optional;
5
6 public class LinkedListInt implements ListInt {
7
8     private class Element {
9         private int i;
10        private Element next;
11        public Element(int i, Element next) {
12            this.i = i;
13            this.next = next;
14        }
15        public void insertAtTheEnd(int value) {
16            if (this.next == null) {
17                this.next = new Element(value, next: null);
18            } else {
19                this.next.insertAtTheEnd(value);
20            }
21        }
22        public boolean isEmpty() { return next == null; }
23        public int size() {
24            if ( isEmpty() ) {
25                return 0;
26            }
27            return 1 + next.size();
28        }
29        public int get() {
30            return i;
31        }
32        public int get(int index) {
33            if ( index == 0 ) {
34                return this.i;
35            } else if ( !isEmpty() ){
36                return this.next.get(index-1);
37            } else {
38                throw new IndexOutOfBoundsException();
39            }
40        }
41    }
42 }
```

```

42
43     private Element elem = null;
44
45     @Override public boolean isEmpty() { return elem == null; }
46
47     public void clear() { elem = null; }
48
49     public int get(int index) {
50         if (!isEmpty() && index >= 0) {
51             return elem.get(index);
52         } else if (isEmpty()) {
53             throw new NoSuchElementException(s: "...");
54         } else {
55             throw new IndexOutOfBoundsException(s: "index should be positive or zero");
56         }
57     }
58
59     public void insert(int value) {
60         if ( isEmpty() ) {
61             elem = new Element(value, next: null);
62         } else {
63             elem.insertAtTheEnd(value);
64         }
65     }
66
67
68     public int size() {
69         if (isEmpty()) {
70             return 0;
71         }
72         return 1 + elem.size();
73     }
74
75     public Iterator<Integer> iterator() {
76         return new Iterator<>() {
77             private Element cursor = elem;
78             @Override
79             public boolean hasNext() {
80                 return cursor != null;
81             }
82
83             @Override
84             public Integer next() {
85                 int res = cursor.get(); // already throw if hasn't next element
86                 cursor = cursor.next;
87                 return res;
88             }
89         };
90     }
91
92 }

```

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 !

Correction :

```

1  import java.util.Optional;
2  import java.util.function.Supplier;
3
4  @FunctionalInterface
5  interface ThrowableSupplier<T> {
6      T get() throws Exception;
7  }
8
9  public interface Test { // et oui, marche aussi avec un main dans une interface
10
11      public static Optional<Integer> test(Supplier<Integer> s) {
12          try {
13              return Optional.of( s.get() );
14          } catch( Exception e) {
15              return Optional.empty();
16          }
17      }
18
19      public static Optional<Integer> test2(ThrowableSupplier<Integer> s) {
20          try {
21              return Optional.of( s.get() );
22          } catch( Exception e) {
23              return Optional.empty();
24          }
25      }
26
27      public static void log(Object o) { System.out.println(o.toString());}
28
29      Run | Debug
30      public static void main(String... args) {
31          log(test( () -> 3 ));
32          log(test( () -> 3 + 4 ));
33
34          log( test( () -> 3 / 0 ) );
35
36          log(test( () -> null ));
37          log(test( () -> {
38              throw new RuntimeException();
39          }));
40          /* Doesn't compile with non-checked exception because the close `throws` is not
41             * present in the functional interface of Supplier. Let's do our own Supplier
42          log(test( () -> {
43              throw new Exception();
44          }));
45          */
46          log(test2( () -> {
47              throw new Exception();
48          }));
49      }
50
51  }

```

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 `Box`s, 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
4

```

Correction :

```

1  import java.util.Objects;
2  import java.util.function.Function;
3
4  public final class Box<T extends Comparable<T>> implements Comparable<Box<T>> {
5
6      private T value;
7
8      public Box(T value) { this.value = value; }
9
10     public T get() { return this.value; }
11
12     public void set(T newValue) { this.value = newValue; }
13
14     public int compareTo(Box<T> that) {
15
16         return this.value.compareTo(that.value);
17
18     }
19
20     public <R extends Comparable<R>> Box<R> map(Function<T, R> f) {
21         return new Box<R>(f.apply(this.value));
22     }
23
24     @Override
25     public String toString() { return "[" + this.value.toString() + "]"; }
26
27     @Override
28     public boolean equals(Object o) {
29         if (this == o) {
30             return true;
31         }
32         if (o == null || o.getClass() != this.getClass()) {
33             return false;
34         }
35         Box<?> other = (Box<?>)o;
36         return this.value.equals(other.value);
37     }
38
39
40     @Override
41     public int hashCode() {
42         return Objects.hash(this.value);
43     }
44 }
45
46 class Util {
47     public static <T extends Comparable<T>> boolean isBigger(Box<T> b1, Box<T> b2) {
48         return b1.compareTo(b2) > 0;
49     }
50 }
51

```


Exercise 8

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

Correction :

```
1 public class EmptyIntegerStack extends RuntimeException {  
2     public EmptyIntegerStack() {  
3         super(message: "Stack is empty");  
4     }  
5 }
```

```
1 import java.util.Optional;  
2 import java.util.List;  
3 import java.util.ArrayList;  
4 import java.util.function.Consumer;  
5  
6 public class ArrayStack<T> implements Stack<T> {  
7  
8     private List<T> stack = new ArrayList<>();  
9  
10    public T pop() {  
11        if ( isEmpty() ) {  
12            throw new EmptyIntegerStack();  
13        }  
14        return stack.remove( stack.size()-1 );  
15    }  
16  
17    public void push(T t) {  
18        stack.add(t);  
19    }  
20    public int size() {  
21        return stack.size();  
22    }  
23  
24    public T peek() {  
25        if ( isEmpty() ) {  
26            throw new EmptyIntegerStack();  
27        }  
28        return stack.get( stack.size()-1 );  
29    }  
30  
31 }  
32
```

```

1  import java.util.Optional;
2  import java.util.function.Consumer;
3
4  public interface Stack<T> {
5
6      T pop();
7
8      default Optional<T> popOption() {
9          if ( isEmpty() ) {
10             return Optional.empty();
11          }
12          return Optional.of( pop() );
13      }
14
15      default void ifHeadIsPresent(Consumer<T> consumer) {
16          peekOption().ifPresent( consumer );
17      }
18
19      void push(T t); // ajoute un élément
20
21      int size(); // retourne la taille
22
23      T peek();
24
25      default Optional<T> peekOption() {
26          if ( isEmpty() ) {
27             return Optional.empty();
28          }
29          return Optional.of( peek() );
30      }
31
32      default boolean isEmpty() {
33          return size() == 0;
34      }
35  }
36

```

```

1  public class Test {
2      private static void log(Object o) {
3          System.out.println(o.toString());
4      }
5
6      public static void main(String... args) {
7          Stack<Integer> stack = new ArrayStack<>();
8          stack.push(1);
9          stack.push(2);
10         Stack<String> stack2 = new ArrayStack<>();
11         stack2.push("hello");
12         stack2.push("world");
13     }
14 }

```