Nom, prénom :

Contrôle final de Compléments en Programmation Orientée Objet
(Correction)

À composer en 2 heures 30 minutes.

Quelques rappels et indications :

- Type fini : type dont le nombre d'instances, présentes et futures, est borné par un certain entier positif donné.
- Type immuable : type tel que toutes ses instances, présentes et futures, sont non-modifiables (en profondeur : aussi bien attributs qu'attributs des attributs et ainsi de suite).
- Type scellé : type dont l'ensemble des sous-types (présents et futurs...) est fixé à la compilation de ce type.
- Le type `Map<K,V>` (interface `java.util.Map`) comporte notamment les méthodes :
  - `public V get(K key)` retournant la valeur associée à la clé passée en argument dans la `Map` courante (`null` si cette clé n'existe pas dans cette `Map`) ;
  - et `public V put(K key, V value)`, ajoutant l'association (`key`, `value`) dans la `Map` courante et supprimant, le cas échéant, l'association relative à `key` qui existait déjà.
- La classe immuable `java.util.Optional` : sert à représenter une valeur qui peut être présente ou pas. Ainsi, il est possible d'écrire une méthode qui parfois n'a pas de résultat en déclarant comme type de retour `Optional<Truc>` au lieu de `Truc`.
  Pour obtenir un optionnel vide, on appelle `Optional.empty()`. Pour obtenir un optionnel contenant la valeur `x`, on appelle `Optional.of(x)`.
  Pour extraire la valeur contenue dans optionnel `opt`, on teste d'abord sa présence :

```
1  Optional<Truc> opt = ...;
2  if (opt.isPresent()) {
3      Truc valeur = opt.get(); // ici valeur est garantie être non null
4      // faire quelque chose avec valeur
5  } else { /* comportement alternatif (afficher une erreur ?) */ }
```

  Remarque : un optionnel non vide ne peut pas contenir la valeur `null`.
- La classe `java.lang.Object` contient notamment les méthodes suivantes :
  - `void wait() throws InterruptedException` : met le *thread* courant en attente d'une notification sur le moniteur de l'objet. Typiquement utilisé dans une boucle `while`.
  - `void notify()` : notifie (réveille) un *thread* en attente sur le moniteur de l'objet.
  - `void notifyAll()` : notifie tous les *threads* en attente sur le moniteur de l'objet.

  Sous peine de déclencher une exception, ces méthodes doivent être appelées dans un bloc synchronisé (`synchronized`) sur l'objet sur lequel elles sont appelées.
- `java.util.concurrent.atomic.AtomicReference` : les instances de `AtomicReference<V>` sont des boîtes encapsulant juste une valeur de type `V`, mais celles-ci sont munies d'accesseurs avec des garanties d'atomicité. Voici un extrait des méthodes de cette classe :
  - `public V get()` : retourne la valeur encapsulée
  - `public boolean compareAndSet(V expected, V update)` : si la valeur encapsulée est égale à `expected`, affecte `update` à celle-ci et retourne `true` ; sinon retourne `false` sans rien modifier. La séquence vérification et mise-à-jour est atomique [1].

1. Et cette atomicité n'utilise pas de moniteur. Son implémentation est très efficace car elle utilise une instruction dédiée des microprocesseurs, qui effectue une vérification et une mise-à-jour de façon atomique.

Cette classe a un constructeur avec la signature suivante : `AtomicReference(V initialValue)`. Celui-ci instancie une référence atomique avec pour contenu `initialValue`.

Exercice 1 : Questionnaire

*Pour chaque case, inscrivez soit "***V***"(rai) soit "***F***"(aux), ou bien ne répondez pas.*
Note = max(0, nombre de bonnes réponse - nombre de mauvaises réponses), ramenée au barême.
Sauf mention contraire, les questions concernent Java 8.

1. [F] Tout seul, le fichier `Z.java`, ci-dessous, compile :

```
1 import java.util.function.*;
2 public class Z { Function<Object, Boolean> f = x -> { System.out.println(x); }; }
```

Correction : La lambda expression donnée ici ne peut pas implémenter la méthode `apply` de `Function`, car son type de retour est `void` (ou plutôt : elle ne retourne rien et, en tout cas, certainement pas `Boolean`).
Donc l'inférence vers `Function<Object, Boolean>` n'est pas possible.

2. [V] Une méthode déclarée à la fois `private` et `abstract` ne compile pas.

Correction : En effet, `abstract` demande une redéfinition, or pour redéfinir, il faut hériter, mais les membres `private` ne sont pas héritables. Cette combinaison est donc absurde donc interdite par le compilateur.

3. [V] Une classe `abstract` peut contenir une méthode `final`.

Correction : Ici, pas de contradiction. Cela veut juste dire qu'une partie de l'implémentation ne sera pas modifiable par les sous-classes.

4. [V] `HashSet<Integer>` est sous-type de `Set<Integer>`.

Correction : Oui : d'une part `HashSet` implémente `Set`, d'autre part, les deux types génériques sont ici paramétrés avec le même paramètre (`Integer`).

5. [V] Une interface peut contenir une `enum` membre.

Correction : C'est autorisé par le langage. À noter que l'`enum` membre est alors statique.

6. [F] Quand, dans une méthode, on définit et initialise une nouvelle variable locale de type `int`, sa valeur est stockée dans le tas.

Correction : Les variables locales sont stockées en pile ce qui permet de récupérer la mémoire quand on sort de la méthode (décrémentation du pointeur de pile de la taille du *frame*).

7. [V] Tout objet existant a l'exécution est instance de `Object`.

Correction : `Object` est par définition le type de tous les objets... et de `null`.

8. [V] Pour faire un *downcasting*, on doit demander explicitement le transtypage.

Correction : Le *downcasting* est une opération périlleuse (primitifs : perte d'information ; références : exception si à l'exécution l'objet référencé n'a pas le type demandé), il doit donc être demandé explicitement.

9. [V] La classe d'un objet donné est connue et interrogeable à l'exécution.

Correction : Cf. la correction de la question 25.
Ces différents cas de figure fonctionnent car tout objet contient une référence vers l'objet-classe de sa classe.

10. [F] Le type d'une expression est calculé à l'exécution.

Correction : La notion d'expression n'a pas de sens à l'exécution : c'est un morceau de texte du programme qui n'existe plus dès que le code est compilé.
Il se trouve que l'expression est l'unité de base pour la vérification de type statique.

11. [V] Le passage d'information entre deux *threads* de Java se fait via des variables partagées.

Correction : Tout à fait, c'est le principe du modèle des *threads*, même s'il est possible de simuler d'autres mécanismes (comme le passage de messages), en utilisant les variables partagées et les primitives de synchronisation fournies par l'API des *threads* Java.

12. [F] En multipliant par deux le nombres de *threads* utilisés par un programme, on multiplie par deux, ou presque, sa vitesse d'exécution.

Correction : C'est faux pour de nombreuses raisons :
- Il faut que le travail du programme soit par nature parallélisable (nombreux calculs indépendants).
- Il faut que le matériel permette cette parallélisation (avoir autant de processeurs matériels que de *threads*).
- Puis, à cela s'ajoute typiquement de la synchronisation qui va ralentir le tout, dans des proportions variables.

Exercice 2 : Échecs

Le jeu d'échecs se joue avec des pièces de 6 sortes différentes (pion, cavalier, fou, tour, dame, roi) qui peuvent être de 2 couleurs différentes (blanches ou noires).

À faire : (n'écrivez qu'un seul programme, seul le résultat final compte)

1. Écrire les 2 types finis `Couleur` (blanc ou noir) et `Valeur` (pion, ...) servant à caractériser les différentes pièces des échecs.
2. Écrire un type `Piece` immuable, où une pièce se caractérise par une couleur et une valeur.
3. Faites en sorte qu'il soit impossible d'instancier deux pièces identiques (pour simplifier, on suppose ici qu'il n'y a pas 8 pions simples par couleur, comme dans le vrai jeu d'échecs, mais 1 seul). Remarque : cela fera de `Piece` un type fini.

   Indication : rendez le constructeur privé pour forcer à passer par une fabrique statique que vous écrirez. Un attribut statique de type `Map<Couleur, Map<Valeur, Piece>>` permettra de retrouver l'instance unique de pièce pour un couple couleur/valeur donné.

Correction :

```java
import java.util.HashMap;
import java.util.Map;

public final class Piece {
    private static final Map<Couleur, Map<Valeur, Piece>> known = new HashMap<>();
    public final Couleur couleur;
    public final Valeur valeur;

    private Piece(Couleur couleur, Valeur valeur) {
        this.couleur = couleur;
        this.valeur = valeur;
    }

    public static Piece of(Couleur couleur, Valeur valeur) {
        if (known.get(couleur) == null) known.put(couleur, new HashMap<Valeur, Piece>());
        Map<Valeur, Piece> knownForThisColor = known.get(couleur);
        if (knownForThisColor.get(valeur) == null) knownForThisColor.put(valeur, new
             Piece(couleur, valeur));
        return knownForThisColor.get(valeur);
    }

    public static enum Couleur {BLANC, NOIR}

    public static enum Valeur {PION, FOU, CAVALIER, TOUR, DAME, ROI}
}
```

Exercice 3 : Liste chaînée immuable

Ci-dessous, une implémentation de liste chaînée :

```java
import java.util.Iterator;

public abstract class LC<T> {
    private LC() { }
    public static <T> LC<T> vide() { return new Vide<>(); }
    public static <T> LC<T> cons(T tete, LC<T> queue) { return new Cons<>(tete, queue); }
    public abstract T tete();
    public abstract LC<T> queue();
    public abstract boolean estVide();

    public static class OperationOnEmptyListException extends RuntimeException { }

```

```
13      private static final class Vide<T> extends LC<T> {
14          @Override public T tete() { throw new OperationOnEmptyListException(); }
15          @Override public LC<T> queue() { throw new OperationOnEmptyListException(); }
16          @Override public boolean estVide() { return true; }
17      }
18
19      private static final class Cons<T> extends LC<T> {
20          private final T tete; private final LC<T> queue;
21          private Cons(T tete, LC<T> queue) { this.tete = tete; this.queue = queue; }
22          @Override public T tete() { return tete; }
23          @Override public LC<T> queue() { return queue; }
24          @Override public boolean estVide() { return false; }
25      }
26  }
```

1. À quoi sert le constructeur privé de `LC` ?

   Correction : Celui-ci supprime le constructeur public par défaut ; ainsi `LC` n'a plus que des constructeurs privés. Cela assure que cette classe ne peut être ni instanciée directement ni étendue directement en dehors de son corps. Ses seules sous-classes directes seront donc des classes imbriquées et ses instances directes seront créées seulement depuis ses méthodes et celles de ses classes imbriquées.

2. Montrez que le type `LC` est scellé.

   Correction : Le constructeur privé assure déjà que toutes les sous-classes directes sont imbriquées ; or ces dernières étant `final`, elles ne sont pas extensibles. Ainsi `LC` n'a pas de sous-types indirects. Ses seuls sous-types sont donc, et seront toujours, `LC.Nil` et `LC.Cons`.

3. Montrez que si `Truc` est un type immuable, alors le type `LC<Truc>` est immuable.

   Correction : Comme `LC` est scellée, il suffit de montrer que les instances directes de `LC`, `LC.Nil` et `LC.Cons` sont non modifiables. C'est bien le cas pour `LC.Nil` (`final` et sans attribut). Quant `LC.Cons`, elle est `final` et contient un attribut immuable (`tete`) et un attribut de type `LC` (`queue`), donc une instance de `LC.Cons` est non modifiable si et seulement si son attribut `queue` est non modifiable.
   Ceci se résoud par une preuve par récurrence sur la longueur de la liste (normal, car la définition de `LC` est récursive) :
   - cas de base, taille $= 0$: cela correspond à une instance de `LC.Nil`. C'est OK.
   - cas récurrent, taille $\geqslant 1$: cela correspond une instance de `LC.Cons`, mais on a vu que cette instance était non-modifiable si et seulement si son attribut `queue` était non modifiable. Or `queue` est instance de `LC` de longueur strictement inférieure. L'hypothèse de récurrence est donc satisfaite.

Exercice 4 : Pile "*more-or-less safe*"

Nous utilisons la liste immuable de l'exercice précédent pour implémenter une pile LIFO (mutable, mais dont les instances utilisent une telle liste pour représenter leur état) :

```
1   import java.util.Collection;
2   import java.util.Optional;
3
4   public final class Pile<T> {
```

```
5     private volatile LC<T> contenu = LC.vide(); // (en réalité volatile n'est pas utile pour LC)
6     public void empile(T elem) { contenu = LC.cons(elem, contenu); }
7     public Optional<T> depile() {
8         LC<T> sauvegardeContenu = contenu; // noter la sauvegarde locale
9         if (sauvegardeContenu.estVide()) return Optional.empty();
10        else {
11            contenu = sauvegardeContenu.queue();
12            return Optional.of(sauvegardeContenu.tete());
13        }
14    }
15    public boolean estVide() { return contenu.estVide(); }
16 }
```

On peut montrer (**ne le faites pas !**), et supposer vrai dans la suite, que les méthodes de `Pile` respectent les spécifications suivantes (y compris en cas d'exécution *multi-thread*) :

1. Au retour de `empile` la pile obtenue est la même qu'au début de son exécution, mais avec la valeur passée en paramètre ajoutée, en plus, à son sommet.
2. Au retour de `depile`, si la valeur retournée est un optionnel non vide, la pile obtenue est la même que celle du début de son exécution, privée de son sommet. Dans ce cas, le contenu de l'optionnel est l'élément qui était précédemment en sommet de pile.
3. `depile` retourne l'optionnel vide si et seulement si la pile était vide au début de son exécution. Dans ce cas, `depile` ne modifie pas la pile.
4. `empty` retourne `true` si et seulement si la pile est vide ; elle ne modifie pas celle-ci.

Questions :

1. Dites (et prouvez-le) si, dans un contexte *mono-thread*, la classe `Pile` respecte les spécifications suivantes :
   (a) Après avoir exécuté n fois `empile` sur une pile vide, la pile contient n éléments.

   > **Correction :** Les appels consécutifs s'exécutent séquentiellement. Donc l'effet d'un appel s'applique au résultat de l'appel précédent. Donc, par récurrence, appeler n fois `empile` sur une pile ajoute n éléments à la pile initiale.

   (b) Après avoir exécuté n fois `empile` sur la pile vide, on peut exécuter n fois `depile` sans jamais récupérer d'optionnel vide.
   Le $n+1$ème appel retournera un optionnel vide.

   > **Correction :** Après les `empile` la liste a au moins n éléments. Or un appel à `depile` décroit la taille de 1. L'exécution est séquentielle, donc un appel à `depile` décroit la liste obtenue après le précédent appel. Donc les appels de 1 à n se font sur des piles de taille n à 1, donc jamais vide, donc retournent toujours un optionnel non vide.
   > Le $n+1$ème se fait sur la pile vide et retourne donc l'optionnel vide.

   (c) Toute valeur contenue dans un optionnel retourné par `depile` a été auparavant passée en paramètre d'appel à `empile`.

   > **Correction :** La seule méthode de `Pile` qui alimente la liste encapsulée est `empile`. De plus cette dernière est un attribut privé de type immuable (donc pas de modification de l'attribut ni de l'objet référencé possibles depuis l'extérieur). Donc à tout moment, cette liste ne contient que des éléments qui ont été ajoutés par `empile`.

Or `depile` ne fabrique des optionnels qu'à partir d'éléments pris de cette liste.

2. Même question dans un contexte *multi-thread*.

Correction :

(a) Les appels à `empile` augmentent bien la taille de la pile, mais 2 appels parallèles peuvent augmenter la même pile initiale et affecter le résultat à leur retour sans prendre en compte le résultat de l'autre appel à `empile`. Donc le résultat de 2 appels parallèles peut très bien être une pile ne contenant qu'un seul élément de plus.
C'est donc faux.

(b) En supposant une exécution séquentielle des `depile`, si les `empile` n'avaient au final pas ajouté n éléments, il se peut qu'un des n premiers `depile` retourne `Optional.empty()`.
À l'opposé, comme pour les `empile` des `depile` qui s'exécuteraient en parallèle pourraient ignorer le résultat d'un autre `depile` ; ainsi, exécuter $n+1$ `depile`s ne garantit pas (même à supposer que la pile contienne initialement $n+2$ valeurs) qu'on supprime tous les éléments de la pile.

(c) Le raisonnement pour prouver cette propriété dans le cas séquentiel n'utilise pas la séquentialité. Il reste donc valable ici. Donc cette garantie reste vraie même avec plusieurs *threads*.

Dans la suite, les questions sont posées en supposant qu'on utilise `Pile` dans un programme a priori multi-thread.

3. Montrez à l'aide d'un contre-exemple que de conditionner tout appel à `depile` à un appel à `estVide` qui retournerait `false` ne permet pas de garantir que `depile` ne retournera pas `Optional.empty()`.
Vous pouvez, par exemple, considérer un programme où l'on traite tous les éléments d'une pile depuis plusieurs threads différents qui exécuteraient tous une boucle comme celle-ci :

```
1 while (!pile.estVide()) traite(pile.depile()); // où traite est une méthode quelconque
```

Remarque : ce problème justifie l'intérêt pour `depile` de retourner `Optional<T>` plutôt que seulement `T` (avec levée d'exception en cas d'appel sur pile vide, qu'il faudra traiter). Il est, en effet, plus pratique de manipuler des `Optional<T>` que de traiter des exceptions.

Correction : Supposons 2 *treads* t_1 et t_2 exécutant la boucle proposée. Supposons qu'à un moment donné, la pile ne contient d'un seul élément. L'entrelacement suivant est possible : $t_1 \rightarrow$ `stack.estVide()` (retourne `false`) ; $t_2 \rightarrow$ `stack.estVide()` (retourne `false`) ; $t_1 \rightarrow$ `stack.depile()` (retourne `Optional.of(x)`) ; $t_2 \rightarrow$ `stack.estVide()` (retourne `Optional.empty()`).

Cette classe `Pile` n'offre donc pas toutes les garanties d'une pile classique quand elle est utilisée par plusieurs *threads*. Néanmoins, celle classe est intéressante car l'absence de synchronisation accélère les traitements. La seule question est si notre application concurrente peut se contenuer de ces garanties a minima. On peut quand-même essayer de donner plus de garanties, sans pour autant sacrifier toute la performance. C'est l'objet de la suite.

Exercice 5 : Pile synchronisée

Pour l'instant, il s'agit de rendre la classe `Pile` *thread-safe* sans penser aux performances.

Observons d'abord que les problèmes repérés dans l'exercice précédent étaient dûs à la non-atomicité des méthodes. Il faut donc faire en sorte que les méthodes de `Pile` soient atomiques.

1. Modifiez les méthodes de `Pile`, afin qu'elles se synchronisent sur le moniteur de `this` pour garantir leur atomicité (indiquez juste ce qu'il faut insérer et où).

   Correction : Il suffit pour cela d'ajouter le mot-clé `synchronized` avant chaque déclaration de méthode. Comme elles se synchronisent toutes sur le même objet (`this`), il sera impossible que des méthodes appelées sur la même instance de `Pile` s'entrelacent (= atomicité). Cela suffit à garantir une exécution séquentielle et donc à donner les mêmes garanties en *multi-thread* qu'en *mono-thread*.

2. Écrivez la méthode `public T depileBloquante()throws InterruptedException`, qui attend que la pile soit non-vide et dépile un élément dès qu'il est présent (et le retourne). Si nécessaire, modifiez les autres méthodes pour que `depileBloquante` fonctionne.

   Correction :

```
1   public synchronized T depileBloquante throws InterruptedException {
2       while (contenu.estVide()) wait();
3       T ret = contenu.tete();
4       contenu = contenu.queue();
5       return ret;
6   }
```

   Pour que cette méthode soit notifiée qu'un élément est disponible si jamais elle a été mise en attente, il faut modifier `empile` :

```
1   public synchronized void empile(T elem) {
2       contenu = LC.cons(elem, contenu);
3       notify();
4   }
```

Exercice 6 : Pile à variable atomique

Le mécanisme précédent a l'inconvénient de régulièrement mettre des *threads* en attente de moniteur et de les réveiller (très coûteux). À la place, nous vous proposons d'utiliser la classe `java.util.concurrent.atomic.AtomicReference` décrite en introduction.

À faire : Réécrivez la classe `Pile` (de l'exercice 4) en utilisant `AtomicReference`, de sorte à ce que toute la spécification donnée dans l'exercice 4 soit vraie pour toute exécution concurrente.

Quelques indications :

— `contenu` ne sera plus de type `LC<T>`, mais de type `AtomicReference<LC<T>>`.

— `compareAndSet` peut être réitérée indéfiniment jusqu'à ce qu'elle retourne `true` [2].

— dans `depile`, on fera toujours attention à ne pas appeler `tete` ou `queue` sur la pile vide !

Correction :

```
1  import java.util.Collection;
2  import java.util.Optional;
3  import jaav.util.concurrent.atomic.AtomicReference;
```

2. Cela finira par se produire très vite. Le temps perdu dans les quelques essais infructueux est négligeable par rapport au temps perdu à mettre un *thread* en attente puis à le réveiller.

```
4
5  public final class Pile<T> {
6      private final LC<T> content = new AtomicReference<>(LC.nil());
7      public void empile(T elem) {Opti
8          while (true) {
9              LC<T> vieuxContenu = contenu.get();
10             if (contenu.compareAndSet(vieuxContenu, LC.cons(elem, vieuxContenu)) return;
11         }
12     }
13     public Optional<T> depile() {
14         while (true) {
15             LC<T> vieuxContenu = contenu.get();
16             if (vieuxContenu.estVide) return Optional.empty();
17             if (contenu.compareAndSet(vieuxContenu, vieuxContenu.queue())
18             return Optional.of(vieuxContenu.tete());
19         }
20     }
21     public boolean estVide() { return contenu.estVide(); }
22 }
```