Contrôle final de Compléments en Programmation Orientée Objet

- **Durée : 2 heures 30 minutes.**
- **Tout document ou moyen de communication est interdit. Seule exception : une feuille A4 recto/verso avec vos notes personnelles.**
- **Le barême donné est indicatif.**
- **Répondez au dernier exercice directement sur le sujet. Il est imprimé sur une feuille séparée à joindre à la copie que vous rendrez.**

Quelques rappels et indications :

- Type fini : type dont le nombre d'instances, présentes et futures, est borné par un certain entier positif fixé dès la compilaton.
- 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 :
  - `V get(K key)` : retourne la valeur associée à la clé passée en argument dans la `Map` courante (`null` si cette clé n'existe pas dans cette `Map`) ;
  - et `V put(K key, V value)` : supprime, dans la `Map` courante, l'association relative à `key`, si elle existait déjà, et y ajoute la nouvelle association (`key`, `value`).
- 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'aura 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 créer un optionnel contenant la valeur `x`, on appelle `Optional.of(x)`. Pour extraire la valeur stockée dans optionnel, on teste d'abord sa présence avec `isPresent`, puis on appelle `get`. Exemple :

```
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].

Cette classe possède le constructeur `AtomicReference(V initialValue)` qui instancie une référence atomique avec pour contenu initial `initialValue`.

Exercice 1 : Échecs (3 points)

Le jeu d'échecs se joue avec des pièces de 6 sortes (valeurs) 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, cavalier, fou, tour, dame, roi) 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 dans une même exécution du programme.

   Pour simplifier, on suppose ici qu'il n'y a pas, comme aux vrais échecs, 8 simples pions par couleur, mais juste 1 seul.

   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.

Exercice 2 : Liste chaînée immuable (4 points)

Ci-dessous, une implémentation de liste chaînée :

```
1  public abstract class LC<T> {
2      private LC() { }
3      public static <T> LC<T> vide() { return new Vide<>(); }
4      public static <T> LC<T> cons(T tete, LC<T> queue) { return new Cons<>(tete, queue); }
5      public abstract T tete();
6      public abstract LC<T> queue();
7      public abstract boolean estVide();
8
9      public static class OperationSurListeVideException extends RuntimeException { }
10
11     private static final class Vide<T> extends LC<T> {
12         @Override public T tete() { throw new OperationSurListeVideException(); }
13         @Override public LC<T> queue() { throw new OperationSurListeVideException(); }
14         @Override public boolean estVide() { return true; }
15     }
16
17     private static final class Cons<T> extends LC<T> {
18         private final T tete; private final LC<T> queue;
19         private Cons(T tete, LC<T> queue) { this.tete = tete; this.queue = queue; }
20         @Override public T tete() { return tete; }
21         @Override public LC<T> queue() { return queue; }
22         @Override public boolean estVide() { return false; }
23     }
24 }
```

1. À quoi sert le constructeur privé de `LC` ?
2. Montrez que le type `LC` est scellé.

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.

3. Montrez que si `Truc` est un type immuable, alors le type `LC<Truc>` est immuable.
   Indice : il faut faire une récurrence sur la longueur de la liste.

Exercice 3 : Pile "*more-or-less safe*" (6 points)

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.Optional;
2
3  public final class Pile<T> {
4      private volatile LC<T> contenu = LC.vide(); // (en réalité volatile n'est pas utile pour LC)
5      public void empile(T elem) { contenu = LC.cons(elem, contenu); }
6      public Optional<T> depile() {
7          LC<T> sauvegardeContenu = contenu; // noter la sauvegarde locale
8          if (sauvegardeContenu.estVide()) return Optional.empty();
9          else {
10             contenu = sauvegardeContenu.queue();
11             return Optional.of(sauvegardeContenu.tete());
12         }
13     }
14     public boolean estVide() { return contenu.estVide(); }
15 }
```

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, en plus, la valeur passée en paramètre ajoutée 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, mais 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. `estVide` retourne `true` si et seulement si la pile est vide ; elle ne modifie pas celle-ci.

Questions :

1. Dites (et prouvez-le succinctement) si, pour une instance de la classe `Pile` utilisée par un seul *thread*, les méthodes de cette classe respectent les spécifications suivantes :
   (a) Après avoir exécuté n fois `empile` sur l'instance vide, la pile contient n éléments.
   (b) Après avoir exécuté n fois `empile` sur l'instance vide, on peut exécuter n fois `depile` sans jamais récupérer d'optionnel vide.
   Le $(n+1)$ième appel retournera un optionnel vide.
   (c) Toute valeur contenue dans un optionnel retourné par `depile` a été auparavant passée en paramètre d'appel à `empile`.
2. Même question dans le cas où l'instance de `Pile` est utilisée par 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.

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 contenter 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 4 : Pile synchronisée (3 points)

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 dus 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).
2. Ajoutez 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.

Exercice 5 : Pile à variable atomique (3 points, bonus)

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 3) en utilisant `AtomicReference`, de sorte à ce que toute la spécification donnée dans l'exercice 3 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 !

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.

Nom, prénom :

Exercice 6 : Questionnaire (4 points)

*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. ☐ Tout seul, le fichier `Z.java`, ci-dessous, compile :

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

2. ☐ Une méthode déclarée à la fois `private` et `abstract` ne compile pas.
3. ☐ Une classe `abstract` peut contenir une méthode `final`.
4. ☐ `HashSet<Integer>` est sous-type de `Set<Integer>`.
5. ☐ Une interface peut contenir une `enum` membre.
6. ☐ Quand, dans une méthode, on définit et initialise une nouvelle variable locale de type `int`, sa valeur est stockée dans le tas.
7. ☐ Tout objet existant à l'exécution est instance de `Object`.
8. ☐ Pour faire un *downcasting*, on doit demander explicitement le transtypage.
9. ☐ La classe d'un objet donné est connue et peut être obtenue à l'exécution.
10. ☐ Le type d'une expression est calculé à l'exécution.
11. ☐ Le passage d'information entre deux *threads* de Java se fait via des variables partagées.
12. ☐ En multipliant par deux le nombres de *threads* utilisés par un programme, on s'attend généralement à multiplier par deux, ou presque, sa vitesse d'exécution.