Examen de session 2 de Compléments en Programmation Orientée Objet

- **Durée : 2 heures.**
- **Tout document ou moyen de communication est interdit. Seule exception : une feuille A4 recto/verso avec vos notes personnelles.**
- **Répondez au premier exercice directement sur le sujet et rendez la feuille du sujet concernée avec votre copie (le sujet sera mis en ligne après l'épreuve).**
- **Pour tout code à « écrire » ressemblant beaucoup à un code déjà écrit, vous pouvez juste indiquer les différences, à condition que ce soit clair.**

Quelques rappels sur les interfaces fonctionelles :
Dans `java.lang` :

```
1  public interface Runnable { void run(); }
```

Dans `java.util.function` :

```
1  public interface Supplier<T> { T get(); }
```

I) Questions de cours

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. ☐ Retirer une annotation `@Override` d'un programme peut l'empêcher de compiler.
2. ☐ Quand un objet est instancié (pour l'affecter à un attribut), la JVM réserve de la mémoire dans le tas pour le stocker.
3. ☐ Changer quelque part `public` en `private` peut empêcher un programme de compiler.
4. ☐ Changer quelque part `private` en `public` peut empêcher un programme de compiler.
5. ☐ `String` est un type primitif.
6. ☐ L'exécution d'une méthode marquée `synchronized` est garantie atomique.
7. ☐ `Object` est supertype de `Runnable`.
8. ☐ `Object` est supertype de `boolean`.
9. ☐ Il peut y avoir, à tout moment, plus de *threads* en exécution (démarrés et non terminés) qu'il y a de cœurs de CPU dans la machine.
10. ☐ `ArrayList<Integer>` est sous-type de `ArrayList<Object>`.
11. ☐ `ArrayList<Integer>` est sous-type de `List<Integer>`.
12. ☐ L'objet référencé par un attribut `private` de la classe `C` ne peut être lu que depuis `C`.
13. ☐ Ajouter `final` à un attribut `private` peut empêcher le programme de compiler.
14. ☐ Ajouter `final` à une méthode `private` peut empêcher le programme de compiler.
15. ☐ Pour une méthode, `abstract` et `final` sont incompatibles.
16. ☐ L'*upcasting* doit toujours être demandé explicitement (notation `(TypeCible)expression`).

II) Propriétés

Typiquement une classe de données Java se présente sous la forme suivante :

```java
public class DataTuple {
    // propriété 1
    private Type1 data1;
    public Type1 getData1() { return data1; }
    public void setData1(Type1 data) { data1 = data; }

    // propriété 2
    private Type2 data2;
    public Type2 getData2() { return data2; }
    public void setData2(Type2 data) { data2 = data; }

    // et ainsi de suite...

    public DataTuple(Type1 data1, Type2 data2/*, ...*/) { this.data1 = data1; this.data2 = data2; }
    // equals, hashCode et toString doivent aussi être redéfinies, mais nous laissons cela de côté !
}
```

Les méthodes `getDataX` et `setDataX` sont souvent de simples lectures et écritures, mais pas forcément : d'autres modalités d'accès sont possibles (évaluation paresseuse, observabilité, ...).

Afin d'éviter de « réinventer la roue » à chaque fois qu'on écrit une telle classe, on peut utiliser la composition : on délègue alors l'implémentation des modalités d'accès à un objet pour chaque propriété. Ces objets sont instances d'un petit nombre de classes définissant chacune un mode d'accès différent mais implémentant des interfaces communes :

```java
public interface ReadProperty<T> {
    public T get();
}
```

```java
public interface ReadWriteProperty<T> extends ReadProperty<T> {
    public void set(T newVal);
}
```

Les instances de ces interfaces représentent des propriétés « contenant » une valeur lisible via la méthode `get` et modifiable via la méthode `set` (si définie). Exemple d'implémentation :

```java
public class SimpleProperty<T> implements ReadWriteProperty<T> {
    private T value;
    public SimpleProperty(T initial) { value = initial; }
    @Override public T get() { return value; }
    @Override public void set(T newVal) { value = newVal; }
}
```

La classe `DataTuple` peut alors être réécrite de la façon suivante :

```java
public class NewDataTuple {
    private final ReadWriteProperty<Type1> data1Property;
    public ReadWriteProperty<Type1> getData1Property() { return data1Property; }
    private final ReadWriteProperty<Type2> data2Property;
    public ReadWriteProperty<Type2> getData2Property() { return data2Property; }
    ...
}
```

Exercice 2 : Rétablir une API « classique »

Dans `NewDataTuple`, on accède aux valeurs de la propriété `data1` via les méthodes `get` et `set` de `data1Property`. Pour proposer une interface plus usuelle, il est bien sûr possible d'écrire des méthodes `getData1` et `setData1` qui conservent la même signature et le même comportement que dans `DataTuple`, mais utilisent l'attribut `data1Property` au lieu de `data1`.

Écrivez les méthodes `getData1` et `setData1` décrites ci-dessus.

Exercice 3 : Propriétés en lecture et écriture

Écrivez les implémentations suivantes de `ReadWriteProperty<T>` :

1. `LoggedProperty<T>`, qui, à chaque mise à jour, affiche un message de la forme :

```
Propriété <nom> mise à jour de <ancienne valeur> vers <nouvelle_valeur>.
```

   (le nom de la propriété est un attribut supplémentaire de celle-ci, dont la valeur est passée au constructeur de la propriété, puis ne change pas ensuite)

2. `SynchronizedProperty<T>`, telle que, en cas d'utilisation par deux *threads*, aucun appel à `get` ou `set` ne puisse être exécuté en même temps qu'un autre appel à `get` ou `set`.

Exercice 4 : Propriétés en lecture seule

Écrivez les implémentations suivantes de `ReadProperty<T>` :

1. `ImmutableProperty<T>`, dont la méthode `get` retourne juste la valeur stockée. Faites le nécessaire pour qu'il soit impossible, sans modifier cette classe, de créer des instances de `ImmutableProperty` dont la valeur retournée par `get` pourrait changer entre deux appels.
2. `LazyProperty<T>`, pour laquelle la valeur retournée par `get` est calculée lors du premier appel seulement (et pas avant). Pour cela, le constructeur prend un paramètre de type `Supplier<T>` (au lieu de `T`) seulement utilisé lors du premier appel à `get` (les appels suivants retournent la valeur calculée et stockée au premier appel). Exemple d'exécution :

```
1    Random gen = new Random(System.nanoTime()); // initialisation d'un générateur aléatoire
2    ReadProperty<Integer> someInt = new LazyProperty<>(() -> gen.nextInt());
3    System.out.println(someInt.get()); // tire un entier au hasard et l'affiche
4    System.out.println(someInt.get()); // affiche le même entier sans le tirer à nouveau au hasard
```

3. `SumProperty`, prenant en paramètre deux instances de `ReadProperty<Double>` et dont la méthode `get` retourne l'addition des valeurs des deux propriétés passées en paramètre. Exemple d'exécution :

```
1  ReadWriteProperty<Double> x = new SimpleProperty<>(12.0);
2  ReadWriteProperty<Double> y = new SimpleProperty<>(5.0);
3  ReadProperty<Double> somme = new SumProperty(x, y);
4  System.out.println(somme.get()); // affiche 17.0
5  x.set(3.0);
6  System.out.println(somme.get()); // affiche 8.0
```

Exercice 5 : Application

1. Écrivez une classe `Rectangle`, munie des propriétés longueur et largeur (valeur de type `Double`, propriétés modifiables directement via `set`), ainsi que des propriétés périmètre et aire (non modifiables directement : leur valeur dépend de celle des 2 premières propriétés). Vous pouvez supposer déjà définie la classe `ProductProperty` (sur le modèle de `SumProperty`, en remplaçant addition par multiplication).
   Rappel : périmètre = (longueur + largeur) × 2 et aire = longueur × largeur.
2. Même question, mais cette fois-ci, on veut que les propriétés longueur et largeur soient non modifiables (et constantes) ; de plus, on souhaite que le périmètre et l'aire ne soient calculés que lors du premier accès à ces propriétés (c'est-à-dire : ils ne doivent pas être calculés tant qu'on ne souhaite pas connaître leur valeur, ni re-calculés si on demande leur valeur une deuxième fois.).
   Contrainte imposée : n'utilisez que les implémentations de `ReadProperty` déjà décrites dans le sujet.

III) Traitement d'images

Dans l'exercice qui suit, nous allons faire du traitement d'image en parallèle sur tous les processeurs disponibles (cela a du sens quand on traite des images très grandes, par exemple produites par des satellites, des télescopes, ou bien des dispositifs d'imagerie médicale).

Pour cela, des tâches élémentaires (instances de `ForkJoinTask`) vont être soumises au *thread pool* par défaut, qui est configuré pour répartir le travail qui lui est donné sur autant de *threads* que la machine possède de cœurs de CPU.

— Pour créer une instance de `ForkJoinTask`, on peut utiliser une des fabriques `adapt`, par exemple, avec argument de type `Runnable` :

```
1   ForkJoinTask<?> task = ForkJoinTask.adapt(() -> { unCertainCalculElementaire(); } );
```

— Pour soumettre une instance `task` de `ForkJoinTask` au *thread pool* par défaut, il suffit d'appeler `task.fork()`. Celle-ci sera alors exécutée en tâche de fond dès qu'un *thread* sera disponible dans le *thread pool* par défaut.

— Pour attendre la fin de cette tâche, il suffit d'appeler `task.join()` (appel bloquant).

Exercice 6 :

Nous représentons une image par la donnée d'un tableau bi-dimensionnel de pixels. Pour une image en couleur, un pixel est la donnée de 3 valeurs (« canaux ») `double`[1] : une par couleur primaire. Pour une image en niveaux de gris (monochrome), un pixel est juste un `double`.

```
1  public final class PixelRGB {
2      public double rValue, gValue, bValue; // valeurs de rouge, vert et bleu
3  }
```

```
1  public final class ImageRGB {
2      public final int height, width; // hauteur (nb de ligne) et largeur (nb de colonnes)
3      public final PixelRGB[][] pixels; // tableau où est stockée l'image
4      public ImageRGB(int height, int width) {
5          this.height = height; this.width = width; pixels = new PixelRGB[height][width];
6      }
7  }
```

```
1  public final class ImageMono {
2      public final int height, width;
3      public final double[][] pixels;
4      public ImageMono(int height, int width) {
5          this.height = height; this.width = width; pixels = new double[height][width];
6      }
7  }
```

1. Écrivez une fonction `public static ImageMono toMonochrome(ImageRGB image)` qui convertit une image couleur en image monochrome en faisant la moyenne des 3 canaux de couleur pour obtenir l'unique canal gris (gris = (rouge + vert + bleu)/3).
2. Modifiez la fonction précédente afin qu'elle répartisse le travail équitablement sur les tous processeurs disponibles. Pour cela, découpez le travail en de nombreuses petites tâches que vous soumettrez au *thread pool* par défaut.

   Par exemple : « petite tâche » = traitement d'une seule ligne de l'image ; vous pouvez vous aider d'une méthode auxiliaire de la forme `static void convertLine(ImageRGB src, ImageMono dst, int width, int lineNum)`*.*

   Attention : `toMonochrome` ne doit retourner que lorsque tout le travail aura été effectué.

1. En pratique, une fois l'image exportée et affichée, on se contente souvent d'un entier de 8 bits par canal (`byte`). Mais pour les traitements intermédiaires, `double` peut se justifier.