Examen de session 2 de Compléments en Programmation Orientée Objet
(Correction)

- **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. [F] Retirer une annotation `@Override` d'un programme peut l'empêcher de compiler.

   Correction : Non, mais le contraire peut se produire. En effet, cette annotation sert à demander des vérifications supplémentaires.

2. [V] Quand un objet est instancié (pour l'affecter à un attribut), la JVM réserve de la mémoire dans le tas pour le stocker.

   Correction : Tout à fait, pour les valeurs créées à l'exécution, seules les valeurs primitives et les références peuvent être stockées dans les "cases" déjà existantes.
   Les objets (de taille variable) nécessitent une allocation dans le tas.
   La précision "pour l'affecter à un attribut" est là pour écarter le cas où la JVM voudrait optimiser en stockant l'objet directement en pile. Ce phénomène, bien qu'hors-programme pour CPOO, existe bel et bien, mais ne concerne pas les objets affectés aux attributs, qui sont, par nature, persistants.
   L'un dans l'autre, en ignorant ou non l'existence de ces optimisations, la réponse est V.

3. [V] Changer quelque part `public` en `private` peut empêcher un programme de compiler.

   Correction : Tout à fait, pour pas mal de raisons. La plus évidente, c'est que les accès depuis l'extérieur de la classe, qui étaient autorisés, ne le sont plus.

4. [V] Changer quelque part `private` en `public` peut empêcher un programme de compiler.

Correction : Oui, notamment dans une classe étendue par une autre. Les définitions de méthodes de même signature dans la sous-classe sont alors considérées comme des redéfinitions, et peuvent ne pas compiler si leur déclaration est incompatible (mauvais type de retour, visibilité non `public`, clauses `throws`, ...).

5. F `String` est un type primitif.

   Correction : Non, ce type ne fait pas partie de la liste des 8 types primitifs.

6. F L'exécution d'une méthode marquée `synchronized` est garantie atomique.

   Correction : Cela ne suffit pas. Il suffit que les données qu'elle manipule soient aussi manipulés par une autre méthode non synchronisée pour que son exécution ne soit pas atomique par rapport à ces données.

7. V `Object` est supertype de `Runnable`.

   Correction : Comme tous les types définis par les interfaces, `Runnable` est un type référence, donc sous-type de `Object`.

8. F `Object` est supertype de `boolean`.

   Correction : Non car `boolean` est un type primitif ; il vit donc dans une hiérarchie de types séparés des types référence.

9. V 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.

   Correction : Le nombre de *threads* n'est limité que par les paramètres du noyau du système d'exploitation et la mémoire disponible.
   Le fait de pouvoir en exécuter plus que le nombre de cœurs est une fonctionnalité importante de cette abstraction. Les *threads* sont alors exécutés en temps partagé (l'ordonnanceur du noyau veille à les faire progresser tour à tour).

10. F `ArrayList<Integer>` est sous-type de `ArrayList<Object>`.

    Correction : Non, les types génériques de Java sont invariants. Si le paramètre est différent, alors le type est incomparable (dans la relation "sous-type de").

11. V `ArrayList<Integer>` est sous-type de `List<Integer>`.

    Correction : Oui car `ArrayList` implémente `List` et les 2 types génériques ont le même paramètre.

12. F L'objet référencé par un attribut `private` de la classe `C` ne peut être lu que depuis `C`.

    Correction : Non, une copie de la référence peut très bien exister quelque part ailleurs (autre objet, variable locale dans méthode d'autre classe.). C'est le phénomène d'*aliasing*.

13. [V] Ajouter `final` à un attribut `private` peut empêcher le programme de compiler.

    Correction : En effet, cela se produirait dans tout programme où cet attribut serait mis à jour dans une des méthodes.

14. [F] Ajouter `final` à une méthode `private` peut empêcher le programme de compiler.

    Correction : Non : pour une méthode, `final` sert juste à empêcher les redéfinitions. Or une méthode `private` ne peut pas être redéfinie.
    En fait, pour une méthode `private`, le qualificateur `final` est juste redondant.

15. [V] Pour une méthode, `abstract` et `final` sont incompatibles.

    Correction : Cette combinaison est en tout cas absurde : une méthode `abstract` ne sert à rien si elle ne peut pas être redéfinie dans une sous-classe.
    Notamment, la classe ne pourrait pas avoir d'instances, même indirectes.
    Par choix des concepteurs du compilateur de Java, cette combinaison absurde est interdite.

16. [F] L'*upcasting* doit toujours être demandé explicitement (notation `(TypeCible)expression`).

    Correction : La « conversion » vers un supertype est généralement automatique, car considérée "sans risque". C'est cela qui permet le polymorphisme par sous-typage.

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.

Correction :

```
1  public class NewDataTuple {
2      private final ReadWriteProperty<Type1> data1Property;
3      public ReadWriteProperty<Type1> getData1Property() { return data1Property; }
4      private final ReadWriteProperty<Type2> data2Property;
5      public ReadWriteProperty<Type2> getData2Property() { return data2Property; }
6
7      public NewDataTuple(Type1 v1, Type2 v2) {
8          data1Property = new SimpleProperty<>(v1);
9          data2Property = new SimpleProperty<>(v2);
10     }
11
12     public Type1 getData1() { return data1Property.get(); }
13     public void setData1(Type1 value) { data1Property.set(value); }
14
15     public Type2 getData2() { return data2Property.get(); }
16     public void setData2(Type2 value) { data2Property.set(value); }
17 }
```

Remarque : dans ce corrigé, on en fait un peu plus que demandé dans la question, afin d'avoir une classe qui compile vraiment et qui soit cohérente.

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)

   Correction :

   ```
   1  public class LoggedProperty<T> implements ReadWriteProperty<T> {
   2      private T value;
   3      public final String name;
   4      public LoggedProperty(String name, T initial) { value = initial; this.name = name; }
   5      @Override public T get() { return value; }
   6      @Override public void set(T newVal) {
   7          System.out.println("Propriété " + name + " mise à jour de " + value + " vers " +
                  newVal +".");
   8          value = newVal;
   9      }
   10 }
   ```

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`.

   Correction :

   ```
   1  public class SynchronizedProperty<T> implements ReadWriteProperty<T> {
   2      private T value;
   3      public SynchronizedProperty(T initial) { value = initial; }
   4      @Override synchronized public T get() { return value; }
   5      @Override synchronized public void set(T newVal) { value = newVal; }
   6  }
   ```

   Remarque : s'il est clair que 2 exécutions simultanée des accesseurs est impossible, il

est moins évident que l'initialisation faite dans le constructeur soit visible lors d'un appel à `get` non précédé d'un appel à `set`.
Cela est pourtant bien le cas (à condition que la référence à l'instance de `SynchronizedProperty` construite soit correctement publiée). En effet, soit cet appel à `get` est dans le même *thread* que l'appel au constructeur, auquel cas, dans l'ordre du programme, on est après le retour du constructeur, donc après l'initialisation de l'attribut, auquel cas la visibilité est assurée ; soit on est dans un autre *thread*, auquel il a fallu passer une référence à l'objet construit à l'autre *thread*, ce qui s'est fait après le retour du constructeur (ordre du programme), et l'autre *thread* l'a forcément reçue *après* (relation arrivé-avant due à publication correcte), mais `avant` (ordre du programme) d'appeler `get`.

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.

Correction :

```
1  public final class ImmutableProperty<T> implements ReadProperty<T> {
2      private final T value;
3      public ImmutableProperty(T initial) { value = initial; }
4      @Override public T get() { return value; }
5  }
```

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
```

Correction :

```
1  import java.util.function.Supplier;
2
3  // en supposant que l'initialiseur n'ait pas le droit de retourner null
4  // (sinon on peut ajouter un attribut booléen pour savoir si on a déjà  initialisé)
5  public final class LazyProperty<T> implements ReadProperty<T> {
6      private T value;
7      private final Supplier<T> initializer;
8      public LazyProperty(Supplier<T> initializer) { this.initializer = initializer; }
9      @Override public T get() {
10         if (value == null) {
11             value = initializer.get();
12         }
13         return value;
14     }
15 }
```

Remarque : cette classe n'est pas *thread safe*. On peut ajouter `synchronized` à sa méthode pour qu'elle le soit, mais on perd en performance. Il existe pourtant un mécanisme permettant de garder aussi une performance correcte : recherchez « *double-check idiom* ».

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 :

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

Correction :

```
public final class SumProperty implements ReadProperty<Double> {
    private final ReadProperty<Double> ope1;
    private final ReadProperty<Double> ope2;

    public SumProperty(ReadProperty<Double> ope1, ReadProperty<Double> ope2
    ) {
        this.ope1 = ope1;
        this.ope2 = ope2;
    }

    @Override public Double get() {
        return ope1.get() + ope2.get();
    }
}
```

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.

Correction :

```
public final class Rectangle {
    public final ReadWriteProperty<Double> lengthProperty;
    public final ReadWriteProperty<Double> widthProperty;
    public final ReadProperty<Double> surface;
    public final ReadProperty<Double> perimeter;

    public Rectangle(double length, double width) {
        lengthProperty = new SimpleProperty<>(length);
        widthProperty = new SimpleProperty<>(width);
        surface = new ProductProperty(lengthProperty, widthProperty);
        perimeter = new ScaleProperty(new SumProperty(lengthProperty, widthProperty), 2);
    }
}
```

avec

```
public final class ScaleProperty implements ReadProperty<Double> {
    private final ReadProperty<Double> ope;
    private final double scale;

    public ScaleProperty(ReadProperty<Double> ope, double scale) {
        this.ope = ope;
        this.scale = scale;
    }

    @Override public Double get() {
        return ope.get() * scale;
    }
}
```

Remarque : au lieu de définir et utiliser `ScaleProperty` on peut aussi faire en utilisant `ProductProperty` et en lui passant une `ImmutableProperty` de valeur 2.

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.

Correction :

```
public final class ImmutableRectangle {
    public final ReadProperty<Double> lengthProperty;
    public final ReadProperty<Double> widthProperty;
    public final ReadProperty<Double> surface;
    public final ReadProperty<Double> perimeter;

    public ImmutableRectangle(double length, double width) {
        lengthProperty = new ImmutableProperty<>(length);
        widthProperty = new ImmutableProperty<>(width);
        surface = new LazyProperty<>(() -> length * width);
        perimeter = new LazyProperty<>(() -> 2 * (length + width));
    }
}
```

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` :

```
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`.

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

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

```java
public final class ImageMono {
    public final int height, width;
    public final double[][] pixels;
    public ImageMono(int height, int width) {
        this.height = height; this.width = width; pixels = new double[height][width];
    }
}
```

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.

Correction :

```java
import java.util.concurrent.ForkJoinTask;
import java.util.List;
import java.util.LinkedList;

public class ImageTools {
    private ImageTools() {}

    private static void convertLine(ImageRGB src, ImageMono dst, int width, int lineNum) {
        for (int j = 0; j < width; j++) {
            PixelRGB pix = src.pixels[lineNum][j];
            dst.pixels[lineNum][j] = (pix.rValue + pix.gValue + pix.bValue) / 3.0;
```

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.

```
12          }
13      }
14
15      public static ImageMono toMonochrome(ImageRGB src) {
16          ImageMono dst = new ImageMono(src.height, src.width);
17          for (int i = 0; i < src.height; i++)
18              convertLine(src, dst, src.width, i);
19          return dst;
20      }
21
22      public static ImageMono toMonochromeConcurrent(ImageRGB src) {
23          ImageMono dst = new ImageMono(src.height, src.width);
24          List<ForkJoinTask<?>> jobs = new LinkedList<>();
25          for (int i = 0; i < src.height; i++) {
26              final int line = i; // juste pour ne pas passer une variable modifiable au lambda
27              ForkJoinTask<?> job = ForkJoinTask.adapt(() -> convertLine(src, dst, src.width, line));
28              jobs.add(job.fork()); // en fait, fork retourne this; on pourrait juste faire
                    jobs.add(job); job.fork();
29          }
30          for (ForkJoinTask job: jobs) job.join();
31          return dst;
32      }
33
34  }
```