Université Paris Diderot **L3 Informatique** **Année 2016-2017**

Examen de session 2
de Compléments en Programmation Orientée Objet

21 juin 2017 – 15h30

— Durée : 2 heures 45 minutes
— Épreuve sans document autorisé, à l'exception d'une feuille A4 manuscrite recto-verso.
— Le sujet comporte 15 pages.
— Répondez au QCM directement sur la feuille, à détacher du sujet et à joindre à votre copie lors du rendu, en n'oubliant pas d'y inscrire vos nom, prénom et numéro d'étudiant. L'énoncé complet sera de toute façon déposé sur Moodle après l'épreuve.
— Avertissement : lisez bien chaque exercice en entier avant de répondre à ses premières questions.

I) Questions de cours

Notation du "QCM" :

— La notation est établie de telle sorte que la stratégie consistant à répondre au hasard de façon équiprobable ait une espérance de note égale à 0.
— Pour cette raison, évaluer une proposition de façon incorrecte retire des points. Ainsi, il vous est aussi donné la possibilité de ne pas évaluer la proposition.
— En pratique : évaluez chaque proposition en inscrivant "V"(rai), "F"(aux) ou rien (pour s'abstenir) dans la case correspondante.
— Points : 1 pour une réponse correcte, -1 pour une réponse incorrecte, 0 en cas d'abstention. La note d'un QCM ne descend pas en dessous de 0. Le total est multiplié par un coefficient.

Exercice 1 : QCM sur le cours

Évaluez les propositions ci-dessous. Répondez en accord avec le cours (notamment la version du langage Java concernée est la version 8).

1. F.Le mot-clé **protected** devant une déclaration permet de garantir que seules la classe contenant la déclaration ainsi que ses sous-classes ont accès à cette déclaration.
2. V.Dans une **enum**, on peut, au besoin, redéfinir les méthodes non-statiques de l'**enum** pour chacune des constantes de l'**enum**, comme dans l'exemple suivant :

```
1 enum Piece {
2     PILE {
3         @Override public void regarde() { System.out.println("PILE !"); }
4     },
5     FACE {
6         @Override public void regarde() { System.out.println("FACE !"); }
7     };
8     public void regarde() { System.out.println("Une piece... "); }
9 }
```

3. V.Chaque thread dispose de sa propre pile d'appels de méthodes.
4. F.Le compilateur Java vérifie que toute exception sera bien rattrapée dans un bloc **try** ... **catch** ....
5. F.Le mot-clé **abstract** peut apparaître dans la déclaration d'une variable locale.
6. V.Le mot-clé **final** peut apparaître dans la déclaration d'un attribut.
7. F.Le mot-clé **final** peut apparaître dans la déclaration d'une méthode d'interface.
8. V.Le qualificateur de visibilité **private** n'empêche pas l'accès, depuis une classe englobante, aux membres de ses classes imbriquées.
9. V.Le qualificateur de visibilité **private** n'empêche pas l'accès, depuis une classe imbriquée, aux membres de sa classe englobante.
10. F.Une classe locale peut accéder aux variables locales déclarées auparavant dans le même bloc sans restriction.
11. F.Si A<T> est une classe générique, quand je déclare une variable de type A<quelquechose>, alors quelquechose peut être n'importe quel type de données de Java.
12. F.Dans la classe A<T>, la déclaration suivante est correcte :

```
1 |   static A<T> fabrique(T param) { return new A<T>(param); }
```

(en supposant que le constructeur appelé existe effectivement)

II) Modélisation, classes et interfaces

Exercice 2 : Nombres

Nous voulons programmer une calculatrice implémentant les 4 opérations arithmétiques usuelles $(+, -, \times, \div)$ sur les nombres réels, tout en utilisant des représentations adaptées aux réels concrètement utilisés (des classes concrètes différentes pour, par exemple, les entiers, les rationnels et les réels approchés par nombres à virgule flottante).

Pour cela, nous avons les interfaces suivantes :

```
1  public interface Reel {
2      Reel plus(Reel autre);
3      Reel moins(Reel autre);
4      Reel fois(Reel autre);
5      Reel divisePar(Reel autre);
6      double getValeurApprochee();
7  }
8
9  public interface Rationnel extends Reel {
10     long getNumerateur();
11     long getDenominateur();
12 }
13
14 public interface Entier extends Rationnel {
15     long getValeurEntiere();
16 }
```

Nous programmerons les implémentations (classes) suivantes :

- ReelApproche : implémente Reel en utilisant une représentation approchée qui se contente de stocker un attribut **double** ;
- Fraction : implémente Rationnel en représentant les nombres rationnels en tant que fraction de deux entiers (numérateur et dénominateur), représentés par deux attributs **long** ;
- EntierLong : implémente Entier en représentant les entiers par une valeur de type **long** (correspondant à un unique attribut).

Consigne : toutes les implémentations sont *immuables*. C'est-à-dire que leurs instances ne sont plus modifiables une fois construites. Ainsi, les opérations doivent généralement retourner de *nouveaux* objets.

À faire :

1. Dessinez le graphe de sous-typage contenant les 6 types introduits par l'énoncés, en notant bien ce qui est classe et ce qui est interface.

   Quelle est la particularité de ce graphe (regardez notamment l'héritage de classes) ?

   > **Correction :** // TODO : insérer tikz
   > Particularité : il n'y a pas d'héritage de classe (formulations équivalentes acceptées).

2. Avant d'écrire les classes, regardez si certaines méthodes des interfaces ne peuvent pas déjà être implémentées directement dans les interfaces (méthodes **default**), notamment les méthodes héritées de super-interfaces.

   Si c'est le cas, réécrivez les interfaces que vous voulez compléter ou modifier.

   > **Correction :** En effet, certaines valeurs retournées par les méthodes héritées peuvent être directement déduites des valeurs retournées par les nouvelles méthodes.
   >
   > ```
   > 1  public interface Rationnel extends Reel {
   > ```

```
    long getNumerateur();
    long getDenominateur();

    default getValeurApprochee() {
        return (0. + getNumerateur())/getDenominateur();
    }
}

public interface Entier extends Rationnel {
    long getValeurEntiere();

    default getNumerateur() {
        return getValeurEntiere();
    }

    default getDenominateur() {
        return 1L;
    }
}
```

3. Écrivez les 3 classes d'implémentation. Pensez notamment :
   — aux attributs (non modifiables) et aux constructeurs ;
   — aux méthodes **public** String toString() (retournant la représentation en chaîne la plus usuelle pour chaque type de nombres) et **public boolean** equals(Object other) (rappel : $a/b = c/d$ si et seulement si $ad = bc$) ;
   — aux méthodes requises par les interfaces (rappel : $a/b + c/d = (ad + bc)/(bd)$).
   Notez que le type de retour, aussi bien que celui du paramètre des opérations est Reel, et qu'il faut donc prendre en compte la possibilité d'avoir un paramètre d'une autre classe que celle de **this** et toujours se demander quelle classe instancier pour construire le résultat (exemple : quel type représente le mieux la somme d'un entier et d'un rationnel ?).
   Essayez le plus possible d'éviter les répétitions de code.

Correction :

```
class ReelApproche implements Reel {
    private final double valeur;
    public ReelApproche(double valeur) { this.valeur = valeur; }
    @Override public double getValeurApprochee() { return valeur; }

    // fonction statique auxiliaire pour éviter les répétitions de code
    static Reel plusImpl(Reel a, Reel b)  {
        return new ReelApproche(a.getValeurApprochee() + b.getValeurApprochee());
    }

    @Override public Reel plus(Reel autre) { return plusImpl(this, autre); }

    // similairement pour autres opérations
}

class Fraction implements Rationnel {
    private final long numerateur, denominateur;
    public Fraction(long numerateur, long denominateur) {
        if (denominateur == 0L) throw new IllegalArgumentException("dénominateur nul")
        this.numerateur = numerateur;
```

```
21          this.denominateur = denominateur;
22      }
23      @Override public long getNumerateur() { return numerateur; }
24      @Override public long getDenominateur() { return denominateur; }
25
26      // fonction statique auxiliaire pour éviter les répétitions de code
27      static Reel plusImpl(Rationnel a, Rationnel b)  {
28          return new Fraction(a.getNumerateur() * b.getDenominateur()
29                  + a.getDenominateur() * b.getNumerateur(),
30              a.getDenominateur() * b.getDenominateur())
31      }
32
33      @Override public Reel plus(Reel autre) {
34          return (autre instanceof Rationnel)
35              ?plusImpl(this, (Rationnel) autre)
36              :ReelApproche.plusImpl(this, autre)
37      }
38
39      // similairement pour autres opérations
40  }
41
42
43  class EntierLong implements Entier {
44      private final long valeur;
45      public Fraction(long valur) { this.valeur = valeur; }
46      @Override public long getValeurEntiere() { return valeur; }
47      @Override public Reel plus(Reel autre) {
48          if (autre instanceof Entier) {
49              Entier ent = (Entier) autre;
50              return new EntierLong(valeur + ent.getValeurEntiere());
51          }
52          else if (autre instanceof Rationnel)
53              return Fraction.plusImpl(this, (Rationnel) autre);
54          else return ReelApproche.plusImpl(this, autre);
55      }
56
57      // similairement pour autres opérations
58  }
```

4. Après avoir écrit ces classes, quel intérêt voyez-vous à la structure d'héritage utilisée dans cet exercice (la particularité que vous avez remarquée à la première question) ?

   Correction : Un intérêt est de permettre des représentations (les attributs) complètement indépendantes (les classes, qui n'héritent pas les unes des autres) pour les éléments d'un ensemble mathématique et ceux de son complémentaire dans un sur-ensemble alors que, pour autant, on garde un sous-typage (les interfaces) cohérent avec la relation d'inclusion ensembliste.

5. Serait-ce une bonne idée d'ajouter le mot-clé **final** au début des déclarations des 3 classes d'implémentation ? (Qu'est-ce qu'on y perdrait, qu'est-ce qu'on y gagnerait ?)

   Correction : Plutôt une bonne idée, car en déclarant une classe **final**, on empêche d'en hériter. Ainsi on force à préserver l'architecture particulière de cette modélisation (pour le cas où on travaillerait avec d'autres développeurs, ou pour le cas où on reprendrait le travail sur ce projet après une longue pause et où on aurait oublié les choix architecturaux).
   Évidemment, il y a "les défauts des avantages" : on perd forcément en souplesse, car il n'y

a plus moyen d'hériter du code de ces classe pour implémenter des objets plus compliqués. Très probablement, cependant, cela n'en vaut pas la peine.

III) Gestion des erreurs

On suppose que les classes A, B et C sont déjà écrites et disposent toutes de méthodes publiques non statiques appelées f(), g(), h(), ... (à vous de les adapter ou d'en ajouter si vous en avez besoin pour écrire des exemples ou contre-exemples dans la suite).

On veut programmer une méthode qui fait un calcul et retourne le résultat en fonction de ses paramètres (jusque là, tout est banal).

Pour schématiser, cette méthode se présente à peu près ainsi :

```
1  B calculeUnBdepuisUnA(A a) {
2      C c = calculeUnCdepuisUnA(a);
3      B b = calculeUnBdepuisUnC(c);
4      return b;
5  }
```

Seulement, on sait que dans certains cas, sur certaines valeurs de c, la deuxième partie du calcul (calculeUnBdepuisUnC()) ne peut pas aboutir. Ces cas correspondent à des cas d'erreurs.

Ainsi, on dispose d'une méthode **boolean** estBonC(C c) (supposée déjà écrite dans la même classe que calculeUnBdepuisUnA()) qui retourne **true** si son paramètre permet l'exécution de l'appel calculeUnBdepuisUnC(c) sans erreur et **false** sinon.

Disposer d'une telle méthode, c'est bien. Mais la façon de prendre en compte ces cas d'erreurs quand ils sont détectés n'est pas évidente.

Dans cette partie du sujet, vous explorerez plusieurs techniques pour prendre en compte les erreurs détectées grâce à l'appel à la méthode estBonC(), puis vous les comparerez.

Consigne générale : gardez vos réponses courtes. Quand on ne demande pas juste du code Java, en général une ou deux phrases en Français suffisent pour répondre.

Exercice 3 : Une possibilité, la nullité

1. Modifiez calculeUnBdepuisUnA() qui retourne **null** dans les cas où il y aurait eu une erreur dans la version d'origine (et n'exécute pas calculeUnBdepuisUnC(c) dans ces cas).

   Correction :

   ```
   1  B calculeUnBdepuisUnA(A a) {
   2      C c = calculeUnCdepuisUnA(a);
   3      return estBon(c)?calculeUnBdepuisUnC(c):null;
   4  }
   ```

2. Écrivez un exemple d'utilisation, où vous devez appeler la méthode calculeUnBdepuisUnA(), puis appeler la méthode f() (de la classe B) sur le résultat.

   Correction :

   ```
   1      A a = new A(...);
   2      B b = calculeUnBdepuisUnA(a);
   3      if (b != null) System.out.println("J'ai calculé " + b + " et je lui applique f() :" +
              b.f());
   4      else System.out.println("Il y a eu une erreur lors du calcul de b. Donc je ne lui applique
              pas f()");
   ```

3. Quel est l'inconvénient, à l'usage, d'une méthode ainsi modifiée (écrivez un exemple de mauvais usage de cette méthode) ?

Correction : L'inconvénient, c'est qu'il est très facile d'oublier que la valeur calculée (pas forcément utilisée tout de suite) peut contenir **null**. Donc on risque par inadvertance d'appeler une méthode dessus en oubliant de vérifier qu'elle ne avut pas **null**.

Exemple :

```
A a = new A(...);
B b = calculeUnBdepuisUnA(a);
// et beaucoup plus tard...
System.out.println("J'ai calculé " + b + " et je lui applique f() sans vérifier :" + b.f());
    // ici, NullPointerException potentiel !
```

Exercice 4 : Une autre possibilité, les exceptions

1. Modifiez la version originale de calculeUnBdepuisUnA() en faisant quitter la méthode sur une exception dans les cas où il y aurait eu une erreur dans la version d'origine. Déclarez et utilisez votre propre exception qui hérite de RuntimeException.

Correction :

```
class MauvaisCException extends RuntimeException {}

B calculeUnBdepuisUnA(A a) { // notez : pas de throws (exception hors contrôle)
    C c = calculeUnCdepuisUnA(a);
    if (estBon(c)) return calculeUnBdepuisUnC(c);
    else throw new MauvaisCException();
}
```

2. Écrivez un exemple d'utilisation où on appelle la méthode calculeUnBdepuisUnA(), puis la méthode f() (de la classe B) sur le résultat, en prenant toutes les précautions nécessaires.

Correction :

```
A a = new A(...);
try {
    B b = calculeUnBdepuisUnA(a);
    System.out.println("J'ai calculé " + b + " et je lui applique f() :" + b.f());
} catch (MauvaisCException e) {
    System.out.println("Il y a eu une erreur lors du calcul de b. Donc je ne lui applique
        pas f()");
}
```

3. Quel est l'inconvénient de cette version de la méthode ? (écrivez un exemple d'usage de cette méthode qui exhibe l'inconvénient) Est-ce que cet inconvénient est préférable à celui de la technique utilisant la valeur **null** et pourquoi ?

Correction : L'inconvénient, c'est qu'il est très facile d'oublier que la méthode peut quitter sur une exception et donc oublier le **try catch**. Du coup, le programme risque de se quitter complètement.

Exemple :

```
public static void main(String[] args) {
    f1(); // sans try catch et paf ! on quitte sur MauvaisCException
}
void f1() { f2() } // sans try catch
```

```
5 void f2() { f3() } // sans try catch
6 void f4() {
7     A a = new A(...);
8     B b = calculeUnBdepuisUnA(a); // sans try catch
9 }
```

Quand on programme le main() ou f1(), on ne sait plus forcément que f2(), ou encore moins f3() appelé par f2(), peut lancer une exception. L'erreur de programmation est vite venue (cela dit, ça peut ne pas être une erreur de programmation, s'il s'agit d'une erreur identifiée comme fatale, non récupérable).

Ce genre de "plantage" est moins grave que le NullPointerException de l'exercice précédent, car il se produit dès que l'erreur est détectée (et l'information apparaît lors du "plantage"). Le NullPointerException, quant à lui, n'apparaît que bien plus tard, quand on essaye d'utiliser le résultat. Un tel programme est bien plus difficile à déboguer.

4. Quelles seraient les différences (avantages et/ou inconvénients) si votre exception héritait directement de Exception au lieu de RuntimeException ?

Correction : Si MauvaisCException héritait directement de Exception, se serait alors une exception "sous contrôle", pour laquelle, le compilateur Java vérifue qu'elle est bien rattrapée ou, à défaut, signalée par une clause **throws**. Un oubli comme dans l'exemple précédent est donc exclus.

La contrepartie, c'est la "pollution syntaxique" : toutes les méthodes dans lesquelles on ne souhaite pas rattraper l'exception récupèrent cette fameuse clause **throws** MauvaisCException. Cela peut devenir encombrant si on programme une méthode qui utilise plusieurs composants, chacun avec son propre lot d'exceptions sous contrôle.

Exercice 5 : Avec exceptions, cas particulier

Imaginez le cas particulier où le précalcul (la méthode calculeUnCdepuisUnA()) ne fait rien. Dans ce cas, tester la validité du résultat intermédiaire c revient au même que de tester la validité du paramètre a (en effet, dans ce cas, A = C et a = c).

1. Réécrivez votre exemple d'utilisation (question 2), dans ce cas particulier, de telle sorte à garantir que calculeUnBdepuisUnA() n'est appelé que si on sait qu'il n'y aura pas d'erreur (donc ne lancera jamais d'exception). Simplifiez autant que possible cet appel.

Correction :

```
1     A a = new A(...);
2     if (estBonC(a)) { // j'ai le droit si A et C sont la même classe
3         B b = calculeUnBdepuisUnA(a);
4         System.out.println("J'ai calculé " + b + " et je lui applique f() :" + b.f());
5     } else {
6         System.out.println("Il y a eu une erreur lors du calcul de b. Donc je ne lui applique
              pas f()");
7     }
```

2. Du coup, dans ce cas, quel est maintenant l'avantage à utiliser une exception qui étend RuntimeException plutôt que directement Exception ?

Correction : Dans ce cas particulier, on se débarrasse des **try catch** (on évite tout lancement d'exception : ce qui est une optimisation, du point de vue performance). Si l'exception était sous contrôle, on serait quand-même obligé de mettre ce bloc **try catch** devenu inutile et encombrant, si on souhaite éviter un **throws** (encore plus encombrant).

Exercice 6 : Encore une autre possibilité, les optionnels

Java 8 fournit la classe Optional<T> dont les instances sont des conteneurs pour zéro ou un élément de type T : ainsi, un tel objet contient soit rien, soit un autre objet (jamais **null**).

Voici un extrait de cette classe :

```
1  public class Optional<T> {
2
3      // retourne un optionnel vide
4      public static <T> Optional<T> empty() { ... }
5
6      // retourne un optionnel contenant la valeur spécifiée (non nulle, sinon quitte
           sur exception)
7      public static <T> Optional<T> of(T value) { ... }
8
9      // retourne vrai si cet optionnel contient une valeur, faux sinon
10     public boolean isPresent() { ... }
11
12     // retourne le contenu de l'optionnel (si non vide), sinon quitte sur
           NoSuchElementException
13     public T get() { ... }
14
15     ...
16
17 }
```

1. Modifiez la méthode calculeUnBdepuisUnA() de telle sorte qu'elle retourne un optionnel vide, dans les cas où il y aurait eu une erreur dans la version d'origine, ou un optionnel contenant le résultat, dans les autres cas. Pensez à modifier le type de retour de la méthode.

Correction :

```
1  Optional<B> calculeUnBdepuisUnA(A a) {
2      C c = calculeUnCdepuisUnA(a);
3      return estBon(c)?Optional.of(calculeUnBdepuisUnC(c)):Optional.empty();
4  }
```

2. Écrivez un exemple d'utilisation, qui appelle d'abord la méthode calculeUnBdepuisUnA(), puis appelle la méthode f() (de la classe B) sur le résultat (en prenant les précautions nécessaires).

Correction :

```
1      A a = new A(...);
2      Optional<B> b = calculeUnBdepuisUnA(a);
3      if (b.isPresent()) System.out.println("J'ai calculé " + b.get() + " et je lui applique f()
           :" + b.get().f());
4      else System.out.println("Il y a eu une erreur lors du calcul de b. Donc je ne lui applique
           pas f()");
```

3. Quels avantages et inconvénients pouvez-vous voir à cette technique ? (par rapport à utiliser la valeur **null**, et par rapport à la technique des exceptions)

Correction : L'intérêt, c'est que le résultat de la méthode n'est pas de type B mais Optional<B>, et donc on ne peut pas appeler directement dessus les méthodes de la classe B (il faudra d'abord faire get()). Il n'y a donc aucun risque de planter sur NullPointerException.
Il est vrai que la méthode get() provoque une exception si elle est appelée sur un optionnel vide, mais c'est une erreur de programmation qu'on fait moins facilement, puisque quand on écrit get(), on sait qu'on a affaire à un Optional. Normalement, cela devrait mettre la "puce à l'oreille".

IV) Concurrence

On essaye de trouver une technique pratique et généralisable pour partager des informations entre plusieurs threads. Ses informations sont a priori codées dans des types définis par des classes déjà écrites, qu'on aimerait pouvoir utiliser telles quelles.

À cet effet, on veut une classe générique qui encapsule la donnée partagée et synchronise les accès à cette donnée.

Exercice 7 : Premier essai naïf

Nous proposons la "boîte" synchronisée suivante :

```
1 class BoiteSynchro<T> {
2     private T contenu;
3
4     public BoiteSynchro(T contenu) { this.contenu = contenu; }
5     public synchronized T getContenu() { return this.contenu; }
6     public synchronized void setContenu(T contenu) { this.contenu = contenu; }
7 }
```

Montrez sur l'exemple suivant, en exhibant une exécution fautive, que cette technique ne marche pas : il est possible d'arriver à un état intrinsèquement incohérent, où la boîte contiendrait une paire $(0, 0)$ ou $(1, 1)$.

Pour exhiber une exécution : donnez la séquence des numéros de ligne exécutés, en précisant quel thread exécute la ligne. Vous pouvez vous restreindre à la sous-séquence intéressante.

```
1  class PaireEchangeable {
2    private int gauche;
3    private int droite;
4
5    public PaireEchangeable(int gauche, int
         droite) {
6      this.gauche = gauche;
7      this.droite = droite;
8    }
9
10   public int getGauche() {
11     return gauche;
12   }
13
14   public int getDroite() {
15     return droite;
16   }
17
18   public void echange() {
19     int tmp = gauche;
20     gauche = droite;
21     droite = tmp;
22   }
23 }
24
25 class Client extends Thread {
26   private final
27       BoiteSynchro<PaireEchangeable> paire;
28
   public Client(String name,
       BoiteSynchro<PaireEchangeable paire) {
29   super(name);
30   this.paire = paire;
31 }
32
33 @Override
34 public void run() {
35   p = paire.getContenu();
36   p.echange();
37 }
38 }
39
40 public class Test {
41   public static void main(String[] args) {
42     BoiteSynchro<PaireEchangeable> p = new
           BoiteSynchro<>(new
           PaireEchangeable(0, 1));
43     Thread t1 = new Client("Thread1", p);
44     Thread t2 = new Client("Thread2", p);
45     t1.start(); t2.start();
46     try { t1.join(); t2.join(); }
47     catch (InterruptedException e) { }
48   }
49 }
```

Correction : À un moment donné, la méthode echange() peut s'exécuter deux fois sur le même objet sur deux threads différents (`Thread1` et `Thread2`).

L'enchaînement suivant est possible :

```
Thread1, l19       -- ici, le tmp de Thread 1 vaut 0
Thread2, l19       -- ici, le tmp de Thread 2 vaut 0
Thread2  l20       -- ici, gauche vaut 1
Thread2  l21       -- ici, droite vaut 0
Thread1  l20       -- ici, gauche vaut 0
Thread1  l21       -- ici, droite vaut 0
```

À la fin, les deux attributs de la paire valent 0.

Exercice 8 : Immuabilité

Le comportement fautif exhibé à l'exercice précédent est largement dû au fait qu'une fois qu'on obtient une référence vers la boîte, il est possible d'appeler dessus une méthode qui la modifie, plusieurs fois simultanément, et que ces appels simultanés ne sont pas synchronisés.

Une façon d'éviter cette situation consiste à limiter l'utilisation de BoiteSynchro<T> pour les cas où T est une classe immuable (c'est-à-dire, dont les instances ont un état non modifiable).

1. Écrivez une version immuable de PaireEchangeable (les attributs deviennent **final**, et la signature de la méthode echange() devient **public** PaireEchangeable echange() : cette méthode retourne désormais un nouvel objet, qui est une paire identique à **this** mais avec ses éléments échangés) ;

   Correction :

```
1  public class PaireEchangeable {
2          private final int gauche, droite;
3
4          public PaireEchangeable(int gauche, int droite) {
5              this.gauche = gauche;
6              this.droite  = droite;
7          }
8
9          public int getGauche() { return gauche; }
10
11         public int getDroite() { return droite; }
12
13         public PaireEchangeable echange() { return new PaireEchangeable(droite, gauche); }
14
15         public String toString() { return "(" + gauche + ", " + droite + ")"; }
16 }
```

2. Réécrivez la classe Client afin qu'elle utilise correctement la nouvelle version de PaireEchangeable.

   Correction :

```
1  public class Client extends Thread {
2      BoiteSynchro<PaireEchangeable> paire;
3
4      public Client(String name, BoiteSynchro<PaireEchangeable> paire) {
5          super(name);
```

```
6           this.paire = paire;
7       }
8
9       @Override
10      public void run() {
11          PaireEchangeable p = paire.getContenu();
12          p = p.echange();
13          paire.setContenu(p);
14      }
15  }
```

3. Montrez que des comportements fautifs sont malheureusement encore possibles. Cette fois-ci, on ne pourra plus obtenir un état incohérent (comme $(0, 0)$), mais il est encore possible qu'un des échanges demandés soit "oublié", de telle sorte qu'à la fin, la boîte contienne $(1, 0)$ au lieu de $(0, 1)$, alors qu'un nombre pair d'échanges a été demandé.
   Afin de pouvoir décrire ce comportement, numérotez les lignes de vos nouvelles classes.

Correction :

```
Thread1, l11    -- localement Thread1 a une paire non inversée
Thread2, l11    -- localement Thread1 a une paire non inversée
Thread1, l12    -- localement Thread1 a une paire inversée 1 fois
Thread2, l12    -- localement Thread2 a une paire inversée 1 fois
Thread1, l13    -- Thread1 copie sa paire inversée dans la boite
Thread2, l13    -- Thread2 copie sa paire inversée dans la boite
```

À la fin, la boîte contient une paire inversée juste une fois.

Exercice 9 : Lambdas

Le problème de la version précédente, c'est qu'un échange est une opération non atomique : il faut d'abord lire la paire, puis faire l'échange et enfin écrire la paire modifiée. Entre ces trois sous-opérations, d'autres opérations peuvent s'intercaler.

Pour résoudre ce problème, il faut pouvoir pouvoir demander l'échange par un unique appel de méthode synchronisée sur la boîte. Mais il est hors de question d'ajouter à la classe BoiteSynchro<T> une méthode ne servant qu'à échanger les deux éléments d'une paire : en effet cette classe est générique, et le type T n'est pas nécessairement PaireEchangeable. [1]

La solution : créer une méthode synchronisée servant à appliquer une opération quelconque, passée en paramètre, au contenu de la boîte et à remplacer ce contenu par le résultat de l'opération.

```
1  class BoiteSynchro<T> {
2      ...
3      public synchronized void appliqueOperation(UnaryOperator<T> op) { /* à compléter */ }
4      ...
5
6  }
```

Rappel : l'interface UnaryOperator<T> sert à représenter des fonctions de T vers T. Voici un extrait de sa déclaration :

1. De fait, de multiples opérations sont envisageables sur le contenu de la boîte et il n'y a aucun moyen de prévoir à l'avance lesquelles ce sera. Il est donc impossible d'ajouter des méthodes spécifiques pour chaque opération possible sur le type T dans la classe BoiteSynchro<T>.

```
1 interface UnaryOperator<T> {
2     /* Applique cette fonction à l'argument donné */
3     T apply(T t)
4 }
```

À faire :

1. Complétez la méthode appliqueOperation() ci-dessus.

Correction :

```
1         public synchronized void appliqueOperation(UnaryOperator<T> op) {
2                 contenu = op.apply(contenu);
3         }
```

2. Modifiez la classe Client pour quelle utilise appliqueOperation() à la place de getContenu() et setContenu().

Correction : On modifie la méthode run() :

```
1           @Override
2           public void run() {
3             paire.appliqueOperation(PaireEchangeable::echange); // passage de référence
                  de méthode
4           }
```