Contrôle final
de Compléments en Programmation Orientée Objet
(Correction)

6 janvier 2017 – 8h30

— Durée : 2 heures
— Épreuve sans document autorisé, à l'exception d'une feuille A4 manuscrite recto-verso.
— Le barème est indicatif (le total n'est même pas égal à 20). Il est donné seulement pour vous aider à évaluer le temps à passer sur chaque question.
— Attention, le sujet comporte 5 pages.

Notation des QCM : Une question à choix multiples consiste en un énoncé et une liste de propositions. Pour chaque proposition, il faudra cocher "Oui" ou "Non" ou bien laisser blanc.
— "Oui" pour une proposition vraie ou "Non" pour une proposition fausse donne 1 point.
— "Non" pour une proposition vraie ou "Oui" pour une proposition fausse retire 1 point.
— Toute autre possibilité compte 0 points.
Le score de la question (avant coefficient) est le maximum entre la somme des points et zéro.

Exercice 1 : Généralités (3 points)

1. Quelle est la version actuelle de Java SE ? (chiffre des unités) 8
2. Combien de types primitifs (types à valeur directe) existent dans Java ? 8
3. Pour deux variables de type référence `x` et `y`, si le test `x == y` retourne **false**, peut-on conclure en toute généralité, que...

Proposition	Oui	Non
... les deux objets représentent des entités "différentes" ?	☐	☒
... les deux références contiennent des adresses mémoire différentes ?	☒	☐
... les deux objets diffèrent par la valeur d'un champ (au moins) ?	☐	☒
... les deux variables désignent des objets différents ?	☒	☐

4. Parmi les langages suivants, lesquels sont des langages conçus pour être compilés en code-octet compatible avec la JVM ?

Proposition	Oui	Non	Proposition	Oui	Non	Proposition	Oui	Non
Python	☐	☒	HTML	☐	☒	C	☐	☒
Java	☒	☐	Kotlin	☒	☐	Scala	☒	☐

Exercice 2 : Redéfinition, surcharge (2 points)

(Répondre à chaque question par une seule phrase.)

1. Quels membres de classes (ou interfaces) peuvent-ils être redéfinis ?
   Correction : Toutes les méthodes non statiques héritables non finales... et rien d'autre.

2. Où peut-on redéfinir un membre ? Quelles sont les contraintes (signature/typage) ?
   Correction : Dans la classe qui en hérite. Contraintes : même signature (possibilité de retourner sous-type si le type de retour est un type référence), visibilité au moins aussi grande, les exceptions sous-contrôle déclarées dans une éventuelle clause **throws** doivent être un sous-ensemble de celles déclarées par la méthode héritée. (Pour avoir le point, il suffisait de dire "dans la classe qui en hérite, même signature".)

3. À quoi sert l'annotation facultative `@Override` ?
   Correction : À signaler au compilateur qu'une définition de méthode est en fait une redéfinition (il refusera de compiler si tel n'est pas le cas). L'intérêt : s'assurer que c'est en effet une redéfinition, car si ce n'était pas le cas, les instances de la classe qui contient cette définition ne se comporteraient pas de la façon qu'on attend (remplacement de la méthode héritée par sa redéfinition, via le mécanisme de la liaison dynamique).

4. Dans quelle situation parle-t-on de surcharge ?
   Correction : Dès que dans un contexte donné, il existe deux définitions visibles de méthodes (ou constructeurs) de même nom, mais avec des paramètres de types et/ou de nombre différent.

Exercice 3 : Redéfinition et surcharge, en pratique (3 points)

Qu'affiche le programme suivant :

```
1  class X {}
2  class Y extends X {}
3  class Z extends Y {}
4
5  class A {
6          void f(X y) { System.out.println("A et X"); }
7          void f(Y y) { System.out.println("A et Y"); }
8  }
9
10 class B extends A {
11         void f(X y) { System.out.println("B et X"); }
12         void f(Y y) { System.out.println("B et Y"); }
13 }
14
15 class C extends B {
16         void f(Y y) { System.out.println("C et Y"); }
17         void f(Z y) { System.out.println("C et Z"); }
18 }
19
20 class Test {
21     public static void main(String args[]) {
22         A a = new C();
23         Y y = new Y();
24         Z z = new Z();
25         a.f(y);      // affichage 1
26         a.f((X) y); // affichage 2
27         a.f(z);      // affichage 3, piège !
28     }
29 }
```

Questions à choix **unique** (cochez une case pour chaque affichage) :

Affichage 1 : ☐ A et X ☐ A et Y ☐ B et X ☐ B et Y ☒ C et Y ☐ C et Z
☐ ne compile pas, ☐ exception à l'exécution.

Affichage 2 : ☐ A et X ☐ A et Y ☒ B et X ☐ B et Y ☐ C et Y ☐ C et Z
☐ ne compile pas, ☐ exception à l'exécution.

Affichage 3 : ☐ A et X ☐ A et Y ☐ B et X ☐ B et Y ☒ C et Y ☐ C et Z
☐ ne compile pas, ☐ exception à l'exécution.

Correction : Explication du piège : a étant de type déclaré A, le compilateur détermine que la version de f à appeler sera une méthode f dont la signature existe dans les méthodes de la classe A. 2 candidats : f(X y) et f(Y y). Le paramètre d'appel y étant de type déclaré Y, c'est la seconde qui est choisie.
À l'exécution, la JVM cherchera donc, à partir de la classe de l'objet référencé par a (ici la classe C) une méthode de signature f(X y). Or elle trouve justement une telle méthode dans la classe C, qui affiche `C et Y`.

Exercice 4 : Exceptions (3 points)

On veut programmer une classe qui ressemble à la classe suivante :

```
1  class TrucQuiFaitMachin {
2    public boolean peutFaireMachin() {
3      /* teste une certaine condition garantissant que l'appel de faisMachin()
           retourne sans exception ; retourne true si elle est vraie, false sinon */
4    }
5    public void faisMachin() /* throws ou pas ? */{
6      if (!peutFaireMachin()) throw new MachinException();
7      /*
8        ... implémentation de l'action Machin ...
9      */
10   }
11 }
```

Questions :

1. Quelle est la différence entre une exception sous-contrôle et une exception hors-contrôle ? *(expliquez juste, en 2 phrases, ce que cela implique lors de la déclaration d'une méthode)*
   Correction : Une exception "sous-contrôle" est une exception qu'une déclaration de méthode se doit de lister à droite de sa clause **throws**, dès lors qu'elle est susceptible d'être provoquée, sans être rattrapé, lors de l'exécution de la méthode. Les autres exceptions sont dites "hors-contrôle" : peuvent être provoquées n'importe où sans ajouter de déclaration particulière.

2. Est-il souhaitable que l'exception MachinException soit sous contrôle ? Pour quelle raison ? Réécrivez la déclaration de la méthode faisMachin() en fonction de votre réponse, si nécessaire (**throws** ou pas ?).

   Toujours en fonction de votre réponse, écrivez la déclaration de MachinException (point crucial : quelle classe faut-il étendre ?).

   Correction : Il est plus logique que MachinException soit hors-contrôle.

   Explication : L'intérêt d'une exception sous-contrôle, c'est d'imposer de la prendre en compte dans un bloc **try catch** à un moment ou un autre du programme. C'est donc intéressant si la meilleure façon de traiter le cas correspondant à cette exception est via un tel bloc (dont le principal intérêt est la possibilité de traitement non local... mais dont l'exécution est moins efficace qu'une structure de contrôle classique).

   Dans le cas ici présent, l'API de la classe TrucQuiFaitMachin donne la possibilité d'éviter *localement* les comportements exceptionnels en testant la valeur de retour de la méthode peutFaireMachin(). Dans de telles conditions, appeler faisMachin() sans avoir testé au préalable peutFaireMachin() est une simple erreur de programmation, et l'exception MachinException sert alors juste à signaler une erreur à l'exécution causée par cette erreur de programmation (de la même façon que NullPointerException est provoquée quand on oublie de tester que l'expression sur laquelle est appelée une méthode n'est pas égale à **null**).

   Ainsi, l'utilisation normale de faisMachin() n'est pas à l'intrérieur d'un **try** { ... } **catch** (MachinException e){ ... } mais d'un **if** (peutFaireMachin()){ ... }, il est serait donc incohérent que MachinException soit sous-contrôle.

   Donc on ne met pas de clause **throws** à la méthode faisMachin().

   Par ailleurs, pour signaler à Java que MachinException est hors-contrôle, il aut qu'elle soit sous-classe de RuntimeException : **class** MachinException **extends** RuntimeException { ...}.

   (Note : si on avait répondu que sous-contrôle c'était mieux, pour être cohérent, il fallait ajouter **throws** MachinException à la déclaration de faisMachin() et avoir MachinException **extends** Exception.)

3. Écrivez un main() qui crée un objet de la classe TrucQuiFaitMachin et appelle la méthode faisMachin() sur cet objet en prenant les précautions nécessaires (le programme ne doit pas se quitter sur une exception non rattrapée).

Les précautions à prendre ne seront pas les mêmes en fonction de votre réponse à la question précédente. Répondez de façon cohérente pour avoir les points !

Correction : Pour la version hors-contrôle :

```
1 public static void main(String[] args)  {
2     TrucQuiFaitMachin tqfm = new TrucQuiFaitMachin( ... );
3     ...
4     if (tqfm.peutFaireMachin()) tqfm.faisMachin();
5     else { ... }
6     ...
7 }
```

Pour la version sous-contrôle :

```
1 public static void main(String[] args)  {
2     TrucQuiFaitMachin tqfm = new TrucQuiFaitMachin( ... );
3     ...
4     try { tqfm.faisMachin(); }
5     catch (MachinException e) { ... }
6     ...
7 }
```

4. *(bonus)* Si la méthode peutFaireMachin() ne fait que tester l'état interne de l'objet, pouvez-vous dire quelle exception déjà déclarée dans l'API Java doit être utilisée de préférence à la place de MachinException() ?

   Correction : L'exception (hors-contrôle) IllegalStateException sert exactement à cela : dire qu'un appel de méthode est actuellement impossible à cause de l'état interne d'un objet.

Exercice 5 : Échange synchronisé (5 points)

Ci-dessous, l'implémentation d'une classe PaireEchangeable dont les objets sont des paires d'entiers "échangeables" : on peut échanger la valeur gauche et la valeur droite.

On donne aussi un client Client pour cette classe et un programme de test Test.

```
1  package threads;
2
3  class PaireEchangeable {
4    private int gauche;
5    private int droite;
6
7    public PaireEchangeable(int gauche, int
         droite) {
8      this.gauche = gauche;
9      this.droite = droite;
10   }
11
12   public int getGauche() {
13     return gauche;
14   }
15
16   public int getDroite() {
17     return droite;
18   }
19
20   public void echange() {
21     int tmp = gauche;
22     gauche = droite;
23     droite = tmp;
24   }
25 }
26
27 class Client extends Thread {
28   PaireEchangeable paire;
29
30   public Client(String name, PaireEchangeable
         paire) {
31     super(name);
32     this.paire = paire;
33   }
34
35   @Override
36   public void run() {
37     int g = paire.getGauche();
38     int d = paire.getDroite();
39     System.out.println(getName() + ": (" + g +
           ", " + d + ")");
40     paire.echange();
41     g = paire.getGauche();
```

```
42      d = paire.getDroite();
43      System.out.println(getName() + ": (" + g +
            ", " + d + ")");
44    }
45  }
46
47  public class Test {
48    public static void main(String[] args) {
49      PaireEchangeable p = new
            PaireEchangeable(0, 1);
50      Thread t1 = new Client("Thread1", p);
51      Thread t2 = new Client("Thread2", p);
52      t1.start(); t2.start();
53      try { t1.join(); t2.join(); }
54      catch (InterruptedException e) { }
55      System.out.println("À la fin : (" +
            p.getGauche() + ", " + p.getDroite() +
            ")");
56    }
57  }
```

On considère que la classe PaireEchangeable a un comportement correct si

— quand la méthode echange() n'est pas en cours d'exécution, les valeurs de gauche et droite sont les 2 entiers donnés lors de l'initialisation (dans le bon sens ou inversés)

— les positions sont inversées si et seulement si la méthode echange() a été exécutée un nombre impair de fois.

Questions :

1. Comment s'appelle le phénomène qui fait qu'un même programme multi-thread (avec les mêmes entrées) peut admettre plusieurs exécutions différentes ? *(4 mots maximum !)*
   Correction : Les interférences de threads.

2. D'après les critères de correction donnés plus haut, quel doit être l'affichage [1] final d'une exécution correcte de ce programme ? *(1 ligne !)*
   Correction : `À la fin : (0, 1)` (en effet : il doit y avoir eu 2 échanges une fois que tout s'est exécuté)

3. La classe PaireEchangeable peut-elle (au moins pour certaines exécutions) se comporter de façon correcte dans cet exemple ? Si oui, décrivez une exécution correcte. [2]
   Correction : Exemple d'exécution correcte : toute exécution où la méthode echange() ne serait exécutée que par un seul thread à la fois (donc les instructions (t1, 21-23) ne doivent pas s'entrelacer avec les instructions (t2, 21-23)). Remarque : les autres instructions n'ont aucun impact sur la correction de l'exécution. Il ne servait à rien de les mentionner dans la description.

4. Cette exécution correcte est-elle unique ? Si non, donnez un autre scénario correct.
   Correction : Elle n'est pas unique : qu'on fasse (t1, 21-23) puis (t2, 21-23) ou bien le contraire, le résultat est le même.

5. Le comportement de la classe PaireEchangeable est-il garanti correct dans cet exemple ?
   — Si oui, prouvez-le.
   — Si non, donnez un exemple d'exécution incorrecte. Puis proposez une façon (simple), à l'aide des primitives de synchronisation de Java, de garantir que le comportement de PaireEchangeable sera toujours correct. [3]

   Correction : Il n'est pas correct. On peut trouver un exemple en entrelaçant les lignes 21 à 23. Exemple : (t1, 21), (t2, 21), (t1, 22), (t2, 22), (t1, 23), (t2, 23). Résultat : dans les 2 threads, on copie d'abord la variable gauche dans une variable temporaire (2 exemplaires car les variables locales sont locales à un thread), puis on met (2 fois) la variable droite dans la variable gauche, puis (2 fois) la valeur sauvegardée dans la variable gauche. Au final, c'est comme si on avait fait un seul échange. Affichage final : `À la fin : (1, 0)`.

1. La méthode String getName() de la classe Thread retourne le nom du thread **this**, c.-à-d. celui qu'on lui a passé lors de sa construction.

2. Décrivez seulement les instructions exécutées sur les threads Thread1 et Thread2. Vous pouvez écrire une exécution linéarisée par une séquence de la forme suivante : *(Thread2, l.52), (Thread1, l.3), (Thread1, l.4), ...*

3. Écrivez seulement les lignes que vous souhaitez modifier ou ajouter, en précisant leurs numéros.

Une façon d'empêcher cela est d'empêcher l'exécution simultanée de deux échanges. À cet effet, on peut ajouter le mot-clé **synchronized** dans la signature de la méthode echange(). Attention : cela ne sera suffisant que tant qu'on n'ajoute pas une autre méthode qui modifie les attributs gauche et droite, auquel cas il faudra aussi qu'elle soit **synchronized**.

Exercice 6 : Délégation (4 points)

On donne les interfaces suivantes :

```
1  interface Commerce<T> {
2    int vend(T bien); // retourne le montant de la vente du bien en paramètre
3  }
4
5  interface Artisanat<T> {
6    T fabrique(); // fabrique un bien de type T
7  }
```

Par définition, une Boulangerie est un établissement d'artisanat qui fait le commerce du pain qu'il produit. On va l'implémenter en utilisant la capacité de vente d'un Vendeur (une Personne implémentant Commerce<Pain>) et de production d'un Boulanger (implémentant Artisanat<Pain>).

Questions :

1. Écrivez les classes [4] Pain, Boulangerie, Personne, Boulanger, Vendeur.

   Spécification :

   — Boulangerie implémente Commerce<Pain> et Artisanat<Pain> par délégation des méthodes de ces interfaces, respectivement, à un Vendeur et à un Boulanger. Ceux-ci peuvent être fournis à la boulangerie via son constructeur ou bien affectés après la construction ;

   — une Personne a un nom ;

   — le Pain a un prix de vente ;

   — le Vendeur vend() le pain à son prix de vente (la méthode se contente de retourner le prix de vente de son argument) ;

   — le Boulanger fabrique (instancie) des pains tous au même prix.

   Correction :

```
1  public class Pain {
2          public final int prixVente;
3
4          public Pain(int prixVente) {
5                  this.prixVente = prixVente;
6          }
7  }
8
9  public class Personne {
10         public final String nom;
11
12         public Personne(String nom) {
13                 this.nom = nom;
14         }
15 }
16
17 public class Boulanger extends Personne implements Artisanat<Pain> {
18         @Override
19         public Pain fabrique() {
20                 return new Pain(1);
```

4. Pour faire court : laissez les attributs publics et n'écrivez pas de getteurs et setteurs, sauf quand ces méthodes sont demandées pour implémenter une interface.

```
21          }
22
23          public Boulanger(String nom) {
24                  super(nom);
25          }
26  }
27
28  public class Vendeur extends Personne implements Commerce<Pain> {
29          public Vendeur(String nom) {
30                  super(nom);
31          }
32
33          @Override
34          public int vend(Pain bien) {
35                  return bien.prixVente;
36          }
37  }
38
39  public class Boulangerie implements Artisanat<Pain>, Commerce<Pain> {
40          public Boulanger boulanger;
41          public Vendeur vendeur;
42
43          @Override
44          public Pain fabrique() {
45                  return boulanger.fabrique();
46          }
47
48          @Override
49          public int vend(Pain bien) {
50                  return vendeur.vend(bien);
51          }
52
53          public Boulangerie(Boulanger boulanger, Vendeur vendeur) {
54                  this.boulanger = boulanger;
55                  this.vendeur = vendeur;
56          }
57  }
```

2. Écrivez une classe Test avec un main() qui instancie une boulangerie (et son personnel), lui fait fabriquer des pains, puis les vendre.

   Correction :

```
1  public class TestBoulangerie {
2          public static void main(String[] args) {
3                  Boulangerie b = new Boulangerie(new Boulanger("Fred"), new
                       Vendeur("Jeannette"));
4                  System.out.println(b.vend(b.fabrique()));
5          }
6  }
```

Exercice 7 : Génériques (3 points)

On reprend les classes de l'exercice précédent. On définit l'interface :

```
1  interface Bien { int getPrix(); }
```

(et on a maintenant **class** Pain **implements** Bien { ... })

Questions :

1. Modifiez les interfaces Commerce<T> et Artisanat<T> pour forcer leur paramètre T à être sous-type de Bien.

   Correction :

```
1  interface Commerce<T extends Bien> {
2    int vend(T bien); // retourne le montant de la vente du bien en paramètre
3  }
4
5  interface Artisanat<T extends Bien> {
6    T fabrique(); // fabrique un bien de type T
7  }
```

2. Modifiez la classe Vendeur pour la rendre générique : on peut désormais instancier cette classe pour créer des vendeurs vendant d'autres Biens que du Pain.

   Correction :

```
1  public class Vendeur<T extends Bien> extends Personne implements Commerce<T> {
2          public Vendeur(String nom) {
3                  super(nom);
4          }
5
6          @Override
7          public int vend(T bien) {
8                  return bien.getPrix();
9          }
10 }
```

3. Supposons que nous voulions créer, de la même façon, une version générique de Boulanger, que nous appellerions Artisan<T> (Boulanger se comportant comme Artisan<Pain>).

   Quelle difficulté allons-nous rencontrer en écrivant la méthode T fabrique() (qui fabrique et retourne des instances de T) ?

   Proposez une façon de contourner ce problème.

   Correction : La difficulté ici, c'est que la méthode T fabrique() est supposée retourner une nouvelle instance de T. Or dans une classe générique, il n'y a aucune façon d'instancier un nouvel objet du type du paramètre (ici T) directement avec **new** T(...) (la valeur de T n'est pas connue à la compilation, or la recherche de constructeur est gérée par le mécanisme de liaison statique, c'est-à-dire que le compilateur est censé déjà trouver quel constructeur appeler, ce qui est impossible ici).

   La façon typique de contourner cette limitation, c'est de fournir une "fabrique" de biens de type T au constructeur de la classe Artisan<T> (c'est à dire un objet qui a une méthode qui retourne des nouvelles instances du type demandé). On peut pour cela utiliser l'interface Supplier<T> fournie par Java 8 (ou bien déclarer une interface ou une classe personnalisée). Exemple :

```
1  public class Artisan<T extends Bien> extends Personne implements Artisanat<T> {
2          private Supplier<T> factory;
3
4          @Override
5          public T fabrique() {
6                  return factory.get();
7          }
8
9          public Artisan(String nom, Supplier<T> fabrique) {
10                 super(nom);
11                 this.factory = fabrique;
12         }
13 }
```

Après, on pourrait réécrire, par exemple, la classe Boulanger :

```
1 public class Boulanger extends Artisan<Pain> {
2         public Boulanger(String nom) {
3                 super(nom, () -> new Pain(1)); // pour du pain à 1 euro
4         }
5 }
```

ou bien instancier directement un artisan qui fait du pain : **new** Artisan<Pain>("Marcel", ()-> **new** Pain(1)).