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

Contrôle final
de Compléments en Programmation Orientée Objet

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) ☐
2. Combien de types primitifs (types à valeur directe) existent dans Java ? ☐
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 ?
2. Où peut-on redéfinir un membre ? Quelles sont les contraintes (signature/typage) ?
3. À quoi sert l'annotation facultative `@Override` ?
4. Dans quelle situation parle-t-on de surcharge ?

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.

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)*
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 ?).
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 !
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() ?

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 !)*
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 !)*
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]
4. Cette exécution correcte est-elle unique ? Si non, donnez un autre scénario correct.
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]

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
```

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.

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

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.
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.
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.

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.