Contrôle final de compléments en POO

Vendredi 22 janvier

— Durée : 2h45
— Épreuve à réaliser sans document à l'exception d'une feuille A4 manuscrite recto-verso.
— Répondre aux exercices 6, 7 et 8 directement sur le sujet, à joindre au reste du rendu. Le sujet entier sera de toute façon mis à disposition sur Moodle après l'épreuve.
— Pour des raisons pratiques, ces 3 exercices sont à la fin du sujet. Vous pouvez évidemment travailler sur les exercices dans l'ordre qui vous arrange.
— Le barème (sur 22 points) est seulement donné à titre indicatif et pourra être modifié.

I) Choisir les bons elements

Exercice 1 : (2,5 pts.)

On donne une classe `DataStructure` qui comporte un tableau d'entiers.

```
1  public class DataStructure {
2      private int[] tab;
3
4      public DataStructure(int taille, int b) {
5          tab = new int[taille];
6          for (int i = 0; i < taille; i++)
7              tab[i] = (int) (Math.random() * b);
8      }
9
10     public void choisir(Predicate<Integer> p) {
11         for (int e : tab)
12             if (p.test(e)) System.out.println(e);
13     }
14 }
15
16 public class Test{
17     public static void main(String[] args) {
18         DataStructure data = new
               DataStructure(10, 20);
19         // insérer ici les instructions demandées
20     }
21 }
```

Sans modifier la classe `DataStructure`, remplir la méthode `main()` pour afficher :

1. tous les éléments du tableau de l'objet `data` ;
2. les éléments pairs du tableau de l'objet `data` ;
3. les éléments du tableau de l'objet `data` qui sont plus grands que 10.

Utiliser au moins une lambda-expression et au moins une classe locale ou anonyme.
Rappels :
— `Predicate<T>`, du *package* `java.util.function`, est une interface (fonctionnelle) contenant la seule méthode (abstraite) `boolean test(T e)`.
— L'opération `x % y` retourne le reste de la division euclidienne de l'entier `x` par l'entier `y`.

Correction : Par exemple :

```
1          data.choisir(x -> true);
2          data.choisir(x -> x % 2 == 0);
3          data.choisir(new Predicate<Integer>() {
4              @Override
5              public boolean test(Integer x) {
6                  return x >= 10;
7              }
8          });
```

II) Le type Soit

Il peut être utile pour une méthode de pouvoir retourner soit un élément de type `G`, soit un élément de `D` (par exemple : `G` est le type du résultat souhaité et `D` est un type des codes d'erreur.). Malheureusement, ces types "somme" n'existent pas "de base" dans Java. [1]

Pour contourner ce problème, on propose d'implémenter la classe `Soit<G, D>` [2] , dont les instances sont soit des instances de la sous-classe `Gauche<G, D>`, contenant un attribut de type `G`, soit des instances de la sous-classe `Droite<G, D>`, contenant un attribut de type `D`. La classe `Soit<G, D>`, elle-même, n'a pas d'attribut.

Exercice 2 : Écrire les classes `Soit<G, D>`, `Gauche<G, D>` et `Droite<G, D>` (4,5 pts.)

Contraintes architecturales : *(ignorez ce que vous ne savez pas faire !)*

— `Gauche<G, D>` et `Droite<G, D>` héritent de `Soit<G, D>`.

— `Soit<G, D>` n'est pas instanciable directement et ne peut pas être étendu par d'autres sous-classes que `Gauche<G, D>` et `Droite<G, D>`. *Astuce : ce sont des classes membres statiques de `Soit<G, D>` et le constructeur de `Soit<G, D>` est privé.*

— `Gauche<G, D>` et `Droite<G, D>` sont immuables. *Leurs instances sont non modifiables.*

Méthodes statiques définies dans `Soit<G, D>`, :

— les fabriques statiques `gauche(G g)` et `droite(D d)` permettant de créer des instances, respectivement de `Gauche<G, D>` et `Droite<G, D>`. Trouvez leurs signatures complètes ! *Indice : ces méthodes, étant statiques, ne connaissent pas les paramètres de type `G` et `D`.*

Méthodes d'instance (abstraites ou non) déclarées dans `Soit<G, D>` [3], :

— **boolean** `isGauche()` : retourne **true** si **this** est instance de `Gauche<G, D>`, **false** sinon.

— `G getGaucheOr(G or)` : retourne l'élément encapsulé s'il est de type `G`, sinon retourne `or`.

— `G getGaucheOrElse(Supplier<G> orElse)` : retourne l'élément encapsulé s'il est de type `G`, sinon retourne la valeur de remplacement calculée par `orElse.get()`.

— `<V> V match(Function<G, V> fg, Function<D, V> fd)` : si l'instance courante est un `Gauche<G, D>` contenant `g`, retourne `fg.apply(g)`. Si l'instance courante est un `Droite<G, D>` contenant `d`, retourne `fd.apply(d)`.

Les méthodes abstraites de `Soit<G, D>` doivent, bien sûr, être redéfinies dans ses sous-classes.

Rappel : les interfaces fonctionnelles `Function<T, R>` et `Supplier<T>`, du *package* `java.util.function`, ont respectivement, comme unique méthode abstraite, la methode `R apply(T x)` et la méthode `T get()`.

Correction :

```
1  import java.util.function.Function;
2  import java.util.function.Supplier;
3
4  public abstract class Soit<G, D> {
5      private Soit() {
6      }
7
8      private static final class Gauche<G, D> extends Soit<G, D> {
9          private final G g;
10
11         private Gauche(G g) {
12             super();
13             this.g = g;
```

1. contrairement à OCaml, par exemple : `type ('g, 'd)soit = | Gauche of 'g | Droite of 'd`
2. inspirée de la classe `Either` du langage Scala
3. `isDroite()`, `getDroiteOr()` et `getDroiteOrElse()` sont définissables de la même façon en inversant `Gauche` et `Droite`, mais elles ne sont pas demandées dans cet exercice.

```
14          }
15
16          @Override
17          public boolean isGauche() {
18              return true;
19          }
20
21          @Override
22          public G getGaucheOrElse(Supplier<G> orElse) {
23              return g;
24          }
25
26          @Override
27          public <V> V match(Function<G, V> fg, Function<D, V> fd) {
28              return fg.apply(g);
29          }
30
31      }
32
33      private static final class Droite<G, D> extends Soit<G, D> {
34          private final D d;
35
36          private Droite(D d) {
37              super();
38              this.d = d;
39          }
40
41          @Override
42          public boolean isGauche() {
43              return false;
44          }
45
46          @Override
47          public G getGaucheOrElse(Supplier<G> orElse) {
48              return orElse.get();
49          }
50
51          @Override
52          public <V> V match(Function<G, V> fg, Function<D, V> fd) {
53              return fd.apply(d);
54          }
55
56      }
57
58      public static <G, D> Soit<G, D> gauche(G g) {
59          return new Gauche<>(g);
60      }
61
62      public static <G, D> Soit<G, D> droite(D d) {
63          return new Droite<>(d);
64      }
65
66      public abstract boolean isGauche();
67
68      public G getGaucheOr(G or) {
69          return getGaucheOrElse(() -> or);
70      }
71
72      public abstract G getGaucheOrElse(Supplier<G> orElse);
73
74      public abstract <V> V match(Function<G, V> fg, Function<D, V> fd);
75
76  }
```

III) Gestionnaire de téléchargements

Rappels pour la programmation concurrente :

L'interface `Runnable` : `interface Runnable { void run(); }`.

Dans la classe `Thread` :

— Constructeur `Thread()` : instancie un `Thread` dont la méthode `run()` ne fait rien.

— Constructeur `Thread(Runnable r)` : instancie un `Thread` dont la méthode `run()` appelle la méthode `r.run()`.

— `static void sleep(long millis) throws InterruptedException` : met le *thread* courant en pause pour au moins `millis` millisecondes.

— `void start()` : démarre un nouveau *thread*, associé à l'instance de `Thread` sur laquelle elle est appelée, en y exécutant la méthode `run()` de celle-ci.

— `void run()` : contient les instructions que le *thread* devra exécuter, à savoir : rien, à moins que l'instance de `Thread` ait été construite en passant une instance de `Runnable` au constructeur, auquel cas la méthode `run()` de cette instance est appelée.
À redéfinir quand on étend `Thread`.

Dans la classe `Object` :

— `void wait() throws InterruptedException` : met le *thread* courant en attente d'une notification sur le moniteur de l'objet. Typiquement utilisé dans une boucle **while**.

— `void notify()` : envoie une notification à un *thread* en attente sur le moniteur de l'objet.

— `void notifyAll()` : envoie une notification à tous les *threads* en attente.

Ces 3 méthodes doivent être appelées dans un bloc synchronisé sur l'objet récepteur de l'appel.

Le contexte : Un gestionnaire de téléchargements est un logiciel permettant de gérer plusieurs téléchargements de fichiers en parallèle. Typiquement le nombre de téléchargements simultanés est borné (afin d'optimiser l'utilisation de la connexion physique). Les téléchargements qui ne sont pas démarrés immédiatement sont mis dans une file d'attente et seront démarrés dès que les conditions le permettent.

Pour les exercices, tout gestionnaire de téléchargement implémente l'interface suivante :

```
1  public interface DownloadManager {
2  // Soumet une tâche de téléchargement à effectuer dès que possible. Méthode non bloquante.
3      void submit(Download dl);
4  }
```

où les téléchargements manipulés sont instances de la classe suivante :

```
1  public class Download {
2  // Simule le téléchargement par une attente de durée aléatoire.
3      public void download() {
4          try { Thread.sleep((long) (10000 * Math.random())); }
5          catch (InterruptedException e) { System.out.println("Téléchargement interrompu."); }
6      }
7  }
```

Exercice 3 : Un seul téléchargement (1,5 pt.)

Écrivez une classe `SimpleDM implements DownloadManager`, qui gère un seul téléchargement en simultané.

Implémentation : la méthode `void submit(Download dl)` se contente de créer et démarrer un nouveau *thread* de téléchargement. Le travail de celui-ci (méthode `run()`) consiste à appeler `dl.download()` dans un bloc synchronisé sur l'instance de `SimpleDM`. On s'assure ainsi que le téléchargement précédent soit terminé pour commencer le suivant.

Remarque : on ne déclare pas de file d'attente. À la place, la file associée au moniteur de l'instance de `SimpleDM` est implicitement utilisée.

Correction :

```
1  public class SimpleDM implements DownloadManager {
2
3      @Override
4      public void submit(Download t) {
5          new Thread(() -> {
6              synchronized(SimpleDM.this) {
7                  t.download();
8              }
9          }).start();
10     }
11
12 }
```

Exercice 4 : Plusieurs téléchargements (2 pts.)

Écrivez une classe `MultipleDM` **implements** `DownloadManager` qui gère `n` téléchargements simultanés, où `n` est un paramètre entier du constructeur.

Comme à l'exercice précédent, tout téléchargement proposé est immédiatement confié à un nouveau *thread*. La différence, c'est que ce *thread* attend que le nombre de téléchargements en cours soit inférieur à `n` avant d'appeler `download()` et avertit les autres *threads* en attente quand son téléchargement est terminé (il faut un attribut **int** dans `MultipleDM` pour le décompte).

Attention au placement des blocs **synchronized** *! En particulier, on ne peut pas exécuter `download()` dans un bloc synchronisé sur l'instance de `MultipleDM` : en effet, n téléchargements doivent pouvoir s'exécuter en même temps sans se bloquer les uns les autres.*

Correction :

```
1  public class MultipleDM implements DownloadManager {
2      private final int parallelism;
3      private int runningThreads = 0;
4
5      public MultipleDM(int parallelism) {
6          this.parallelism = parallelism;
7      }
8
9      @Override
10     public void submit(Download t) {
11         new Thread(() -> {
12             synchronized (MultipleDM.this) {
13                 try {
14                     while (runningThreads >= parallelism)
15                         MultipleDM.this.wait();
16                     runningThreads++;
17                 } catch (InterruptedException e) {
18                     throw new RuntimeException("terminaison anormale", e);
19                 }
20             }
21             t.download();
22             synchronized (MultipleDM.this) {
23                 runningThreads--;
24                 MultipleDM.this.notifyAll();
25             }
26         }).start();
27     }
28 }
```

Exercice 5 : Plusieurs téléchargements, mais efficacement (2 pts.)

Maintenant, on programme une nouvelle implémentation de `DownloadManager`, appelée `PooledDM`. Dans `PooledDM`, au lieu de créer un nouveau *thread* à chaque téléchargement, on veut avoir en permanence `n` *threads*, créés et démarrés dès la construction de l'instance de `PooledDM`, qui prendront tour à tour les téléchargements demandés. [4]

Comme il n'y a plus de bijection entre les instances de `Download` et les instances de `Thread` qui les encapsulent, il n'est plus possible d'utiliser la file d'un moniteur pour stocker le travail en attente. À la place, `PooledDM` contient une file explicite d'instances de `Download` (un attribut).

Cette file sera une instance de la classe `LinkedBlockingQueue<E>`, une implémentation *thread-safe* de l'interface `Queue<E>`. Cette classe contient en outre les méthodes :

- `boolean offer(E e)` : insère un élément `e` et débloque les *threads* en attente sur `take()` ;
- et `E take()` : récupère (et supprime) le premier élément de la file. La méthode `E take() throws InterruptedException` est bloquante quand on l'appelle sur une file vide. Elle peut lancer `InterruptedException` si l'attente est interrompue.

Le travail d'un *thread* de téléchargement consistera ainsi en une boucle infinie qui attend qu'il y ait un téléchargement dans la file, le sort de la file et l'exécute.

Étant donnée la spécification des méthodes `take()` et `offer()`, demandez-vous s'il est nécessaire d'ajouter d'autres formes de synchronisation.

Correction :

```java
import java.util.concurrent.LinkedBlockingQueue;
import java.util.stream.Stream;

public class PooledDM implements DownloadManager {
    class DownloadThread extends Thread {
        @Override
        public void run() {
            while (!interrupted()) {
                try {
                        waitingQueue.take().download();
                } catch (InterruptedException e) {
                    throw new RuntimeException("terminaison anormale", e);
                }
            }
        }
    }

    public PooledDM(int nThreads) {
        waitingQueue = new LinkedBlockingQueue<>();
        for (int i = 0; i < nThreads; i++) new DownloadThread().start();
    }

    private final LinkedBlockingQueue<Download> waitingQueue;

    @Override
    public void submit(Download t) {
        waitingQueue.offer(t);
    }
}
```

4. Remarque : le principe de `PooledDM` est similaire à celui de la classe `ThreadPoolExecutor` vue en cours. On ne demande donc pas d'utiliser cette dernière, mais d'en reproduire le fonctionnement (si vous ne vous rappelez pas, ce n'est pas grave : l'énoncé donne suffisamment d'informations).

IV) Questionnaire

Notation des exercices 6 et 7 (QCM) :

- *La notation est établie de telle sorte que la stratégie consistant à répondre au hasard de façon équiprobable ait une espérance de note proche de* 0.
- *Pour cette raison, répondre de façon incorrecte à une proposition retire des points...*
- *... mais* **vous avez le droit de ne pas répondre** *! (dans ce cas, pas de pénalité)*
- *En pratique, vous évaluerez chaque proposition en inscrivant "V"(rai), "F"(aux) ou rien (pour s'abstenir) dans la case correspondante.*
- *Pour chaque exercice, un coefficient de difficulté sera déterminé pour homogénéiser la difficulté entre les exercices.*
- *La note d'un exercice de QCM est donnée par l'algorithme suivant :*

```
p = nombre de bonnes réponses
p = p - nombre de mauvaises réponses
p = p × coefficient de difficulté
p = max(p, 0) // pas de note négative
p = min(p, nombre de propositions) // on ne dépasse pas le barème de l'exercice
retourner p
```

Notation de l'exercice 8 : *chaque question admet 1 bonne réponse parmi 3. Ainsi, vous devez juste cocher la bonne réponse et ne rien remplir si vous ne savez pas. Pour chaque question :*

- *on a +1 pt. si seule la bonne réponse est cochée*
- *on a -0,5 pt. si seule une mauvaise réponse est cochée*
- *on a 0 pt. si le nombre de réponses cochées est différent de 1.*

Comme pour les QCM, un coefficient de difficulté s'appliquera, et la note sera tronquée pour être positive et inférieure au barème

Exercice 6 : Généralités et types en Java (3 pts.)

1. [V] Un même code-octet JVM est exécutable sur plusieurs plateformes physiques (x86, PPC, ARM, ...)
2. [F] Le code source doit être compilé en code-octet avant chaque exécution.
3. [V] Le contenu d'une variable est généralement un mot de 32 bits (parfois 64).
4. [F] Les objets sont stockés dans la pile.
5. [F] Quand une fonction reçoit un objet en paramètre, elle accède à une copie de l'objet.
6. [F] Le type d'une expression est calculé à l'exécution.
7. [V] Certaines vérifications de type ont lieu à l'exécution.
8. [V] En Java, si `A` et `B` sont des types référence, `A` est sous-type de `B` si et seulement si toutes les instances de `A` sont aussi des instances de `B`.
9. [F] Tous les types de Java sont sous-types de `Object`.
10. [F] Quand on "*cast*" (transtype) une expression d'un type référence vers un autre, dans certains cas, Java doit, à l'exécution, modifier l'objet référencé pour le convertir.
11. [V] Pour faire un *downcasting*, on doit demander explicitement le transtypage.
12. [F] Un transtypage de valeur primitive se traduit toujours par une instruction spécifique dans le code-octet.

Exercice 7 : Programmation orientée objet en Java (4,5 pts.)

1. Lesquelles de ces techniques permettent de réutiliser/adapter des fonctionnalités déjà programmées sans les réécrire ?

   [F] encapsulation [V] composition [V] polymorphisme par sous-typage
   [V] généricité [V] héritage

2. Les entités suivantes peuvent être membres d'une classe :

   [V] attribut [V] méthode [V] classe
   [V] interface [V] constructeur

3. [F] On peut déclarer une classe non imbriquée avec la visibilité **private**.
4. [F] Quand, dans une classe, on définit une méthode de même nom qu'une méthode héritée, il y a nécessairement masquage ou redéfinition de cette dernière.
5. [V] Le type que définit une interface est sous-type de `Object`.
6. [V] Le programme suivant compile :

```
1 class A { final boolean a = 0; }
2 class B extends A { final boolean a = 1;}
```

7. [F] Un **enum** peut hériter d'une classe arbitraire (en utilisant le mot-clé **extends**).
8. [V] Un **enum** peut avoir plusieurs supertypes directs.
9. [V] Dans une classe `B`, membre statique de `A`, (et en dehors de tout éventuel type imbriqué `C` défini à l'intérieur de `B`), **this** (seul) désigne toujours une instance de `B`.
10. [V] Dans une classe `B`, membre non statique de `A`, (et en dehors de tout éventuel type imbriqué `C` défini à l'intérieur de `B`), **this** (seul) désigne toujours une instance de `B`.

Exercice 8 : Exécution (2 pts.)

Pour le programme Java ci-dessous :

```
1  public class D {
2      public String toString() { return "D"; }
3  }
4
5  public class A extends D {
6      public String toString() { return "A"; }
7  }
8
9  public class B extends D {
10     public String toString() { return "B"; }
11
12     public void f(B b) {
13         System.out.println("B.f(B)");
14     }
15
16     public void f(C c) {
17         System.out.println("B.f(C)");
18     }
19 }
20
21 public class C extends B {
22     public String toString() { return "C"; }
23
24     public void f(A a) {
25         System.out.println("C.f(A)");
26     }
27
28     public void f(B b) {
29         System.out.println("C.f(B)");
30     }
31
32     public static void main(String [] args) {
33         A a = new A();
34         B b = new B();
35         C c = new C();
36         B bc = new C();
37         D[] tab= new D[5];
38         tab[1] = a;
39         tab[2] = b;
40         tab[3] = c;
41         tab[4] = bc;
42     }
43 }
```

L	Si on ajoute la ligne L dans `main()`, alors le programme...		
`b.f(c);`	[F] ne compile pas	[F] affiche `B.f(B)`	[V] affiche `B.f(C)`
`b.f(bc);`	[F] ne compile pas	[V] affiche `B.f(B)`	[F] affiche `B.f(C)`
`c.f(b);`	[F] ne compile pas	[F] affiche `B.f(C)`	[V] affiche `C.f(B)`
`c.f(c);`	[F] ne compile pas	[V] affiche `B.f(C)`	[F] affiche `C.f(B)`
`bc.f(c);`	[F] ne compile pas	[V] affiche `B.f(C)`	[F] affiche `C.f(B)`
`bc.f(a);`	[V] ne compile pas	[F] affiche `C.f(A)`	[F] affiche `C.f(B)`

Si on ajoute la ligne `for (D r : tab) { System.out.print(r+";"); }` dans `main()`, le programme affiche : [F] `null;D;D;D;D;` [F] `null;A;B;C;B;` [V] `null;A;B;C;C;`

Si on ajoute la ligne `for (D r : tab) { System.out.print(r.toString()+ ";"); }` dans `main()`, le programme :

[V] lance une `NullPointerException` [F] affiche `null;A;B;C;B;` [F] affiche `null;A;B;C;C;`