Université Paris Cité **Programmation Orientée Objet**
L2 **Année 2024-2025**

TD n° 8 - Correction

Exceptions – Expressions lambda

Exercice 1

1. Quelle est la différence entre une exception *checked* et *unchecked* ?

   Correction : Une exception est *unchecked* précisément lorsqu'elle hérite de la classe `RuntimeException`. Les exceptions *unchecked* n'ont pas besoin d'être traitées explicitement, c'est-à-dire qu'il n'est pas nécessaire de les capturer ou de les déclarer avec `throws` (d'où leur nom). Elles correspondent aux erreurs causées par le programmeur, comme l'utilisation d'un pointeur `null`, ou l'accès à un élément hors des bornes d'un tableau. On peut éviter ces exceptions grâce à des vérifications d'usage (par exemple, vérifier qu'un objet n'est pas `null`).

   Les exceptions *checked* sont les classes qui héritent de `Exception` mais pas de `RuntimeException`. Elles correspondent aux erreurs pouvant se produire pendant une utilisant normale du programme, comme une erreur de communication avec un périphérique, la fin prématurée de la lecture d'un fichier, etc. Elles doivent être traitées explicitement, c'est-à-dire capturés avec une instruction `catch` ou bien passées à l'appelant avec un `throws`, sans quoi on a erreur à la compilation.

2. Déterminer si les classes suivantes sont *checked* ou *unchecked* :
   (a) `NullPointerException`
   (b) `ArrayIndexOutOfBoundsException`
   (c) `IOException`

   Correction : 2a et 2b héritent toutes les deux de `RuntimeException`, directement pour la première et indirectement pour la seconde (elle hérite de l'exception unchecked `IndexOutOfBoundsException`). Inversement, 2c est une héritière directe de la classe `Exception`, elle est donc une classe sœur (*sibling*) de `RuntimeException`. Par conséquent, elle est de type *checked*.

3. Quelles sont les conséquences de l'ajout d'une instruction `throws` dans la signature d'une méthode ? Quelle en est l'utilité ? Distinguez le cas où l'exception est *checked* de celui où elle est *unchecked*.

   Correction : L'ajout d'une instruction `throws` permet de déclarer les exceptions qui seront possiblement levées lors de l'exécution de la méthode. Dans le cas d'une exception *checked*, il revient aux méthodes appelantes de gérer les exceptions, soit en utilisant une instruction `throws` dans leur propre signature (l'exception est donc *passée au suivant*), soit en utilisant un bloc `try catch` pour s'en occuper directement. Dans le cas d'une exception *unchecked* (c'est-à-dire héritant de

`RuntimeException`), l'ajout d'une instruction `throws` est toujours possible mais n'entraîne aucune obligation de la part des méthodes appelantes.

4. Dans la syntaxe `try{1} catch(2) {3}`, à quelle condition le code en `3` est-il exécuté ?

   Correction : Le code en `3` sera exécuté si, et seulement si, le code en `1` provoque une exception du type spécifiée en `2`.

Exercice 2 On définit dans le code suivant trois exceptions et quatre méthodes :

```
class A extends Exception{}
class B extends Exception{}
class C extends B{}

public void a() throws A {throw new A();}
public void b() throws B {throw new B();}
public void c() throws B, C {throw new C();}
public void d() throws A, B, C {}
```

Qu'affichent les codes suivants, et quelles exceptions renvoient-ils ?

1.
```
try { a(); b(); }
catch (A e) { System.out.println("A"); }
catch (B e) { System.out.println("B"); }
```

   Correction : Affiche `"A"`, ne lève aucune exception.

2.
```
try { a(); b(); }
catch (B e) { System.out.println("B"); }
catch (A e) { System.out.println("A"); }
```

   Correction : Affiche `"A"`, ne lève aucune exception.

3.
```
try { b(); a(); }
catch (A e) { System.out.println("A"); }
catch (B e) { System.out.println("B"); }
```

   Correction : Affiche `"B"`, ne lève aucune exception.

4.
```
try { c(); }
catch (B e) { System.out.println("B"); }
catch (C e) { System.out.println("C"); }
```

   Correction : Erreur à la compilation, le second bloc `catch` est inaccessible. Le type de l'argument du premier `catch` est toujours compatible avec le type des exceptions lançables par `c()` (`B`, `C`).

5.
```
try { c(); }
catch (C e) { System.out.println("C"); }
catch (B e) { System.out.println("B"); }
```

Correction : Affiche `"C"`, ne lève aucune exception.

6.
```
try { d(); }
catch (A e) { System.out.println("A"); }
catch (B e) { System.out.println("B"); }
finally { System.out.println("Finally");  }
```

Correction : Affiche `"Finally"`, ne lève aucune exception.

7.
```
try { a(); }
catch (A e) { System.out.println("A"); b(); }
finally { System.out.println("Finally"); }
```

Correction : Affiche `"A"` et `"Finally"`, puis lève l'exception `B`.

8.
```
try { a(); }
finally { System.out.println("Finally"); b(); }
```

Correction : Affiche `"Finally"` et lève l'exception `B`. L'exception `A` n'est jamais levée puisque le programme s'arrête sur l'exception `B` avant de pouvoir lever `A`. Citation de la documentation Java :

> "The `finally` block always executes when the `try` block exits. This ensures that the `finally` block is executed even if an unexpected exception occurs. [...] The runtime system always executes the statements within the `finally` block regardless of what happens within the `try` block. So it's the perfect place to perform cleanup."

```
try { a(); }
catch (A e) { System.out.println("A"); b(); }
finally { System.out.println("Finally"); c(); }
```

Correction : Affiche `"A"`, `"Finally"` et lève l'exception `C`. L'exception `A` est capturée par le `catch`. L'exception `B` n'est jamais levée, puisque le programme s'arrête sur l'exception `C` avant de pouvoir lever `B`.

9. **Exercice 3** On considère les classes suivantes :

```
public abstract class Figure {
    private double cX, cY;  // centre de la figure
    public Figure(double cX, double cY) {
        this.cX = cX;
        this.cY = cY;
    }
    public abstract boolean contient(Figure f);
    public abstract Figure intersection(Figure f);
}

public class Cercle extends Figure {
    private double r; // rayon
```

```
    //...
}

public class Rectangle extends Figure{
    private double l, h; // largeur, hauteur
    //...
}
```

1. Écrire des constructeurs pour `Cercle` et `Rectangle`. Dans chaque cas, le constructeur doit lever `IllegalArgumentException` lorsqu'approprié (par exemple un rayon négatif). Est-il nécessaire d'ajouter une instruction `throws` au constructeur ?
   Correction : Il n'est pas nécessaire d'ajouter une instruction `throws` puisque la classe `IllegalArgumentException` hérite de `RuntimeException`.

```
public Cercle(double cX, double cY, double r){
    super(cX, cY);
    if (r < 0){
        throw new IllegalArgumentException();
    }
    this.r = r;
}

public Rectangle(double x, double y, double l, double h)
   {
    super(x, y);
    if (l < 0 || h < 0){
        throw new IllegalArgumentException();
    }
    this.l = l;
    this.h = h;
}
```

2. À ce stade-ci, le code ne fonctionne pas car `Rectangle` et `Cercle` n'implémentent pas les méthodes abstraites héritées de `Figure`. Proposer une solution temporaire utilisant les exceptions.
   Correction : On peut par exemple lever une `UnsupportedOperationException`. Au lieu de déclarer ces méthodes dans `Figure` comme abstraites, on peut leur donner dans `Figure` les implémentations suivantes :

```
public boolean contient(Figure f) {
    throw new UnsupportedOperationException();
}

public Figure intersection(Figure f) {
    throw new UnsupportedOperationException();
}
```

   À charge pour les classes descendantes de réimplémenter ces deux méthodes.

Exercice 4 *Expressions lambda.*

1. On considère l'interface suivante, qui permet d'implémenter des prédicats sur les entiers :

```
public interface Predicate {
  public boolean test(int n);
}
```

Donner l'implémentation d'une méthode

`static int[] filter(Predicate pred, int... tab)`

Placée dans une classe quelconque, cette méthode devra renvoyer un nouveau tableau contenant la suite des éléments de `tab` vérifiant le prédicat `pred`, dans l'ordre où ils apparaissent dans `tab`.

Correction :

```
public static int[] filter(Predicate pred, int... tab) {
  int nbr = 0;
  for (int n :tab) {
    nbr += (pred.test(n)) ? 1 : 0;
  }
  int i = 0;
  int[] res = new int[nbr];
  for (int n :tab) {
    if (pred.test(n)) {
      res[i++] = n;
    }
  }
  return res;
}
```

Dans le code suivant, trouver l'expression lambda qui, en remplacement du commentaire, permettra de filtrer la suite des entiers donnée :

```
int[] f = filter(/* */, -1, -1, 0, 1, 2);
```

(a) en ne conservant que les éléments impairs ;

Correction : `n % 2 == 1` ne fonctionne pas car `-1 % 2` retourne `-1`.

```
n -> n % 2 != 0
```

(b) en ne conservant que les éléments strictement positifs.

Correction :

```
n -> n > 0
```

2. Soit l'interface suivante, permettant de spécifier un ordre sur les entiers :

```
interface Comparator {
  boolean lessThan(int n, int m);
}
```

Dans une classe quelconque, on suppose implémentée la méthode suivante, permettant de trier sur place les éléments d'un tableau `tab` suivant l'ordre spécifié par `cmp` :

```
public static void sort(Comparator cmp, int[] tab)
```

Dans le code suivant, trouver l'expression lambda qui, en remplacement du commentaire, permettra de trier les éléments de `tab` :

```
int[] tab = { 10, 30, 5, 0, -2, 100, -9 };
sort(/* */, tab);
```

(a) par ordre croissant, (b) par ordre décroissant, (c) par ordre croissant en valeur absolue.

Correction :

```
(n, m) -> n < m
```

```
(n, m) -> m > n
```

```
(n, m) -> Math.abs(n) < Math.abs(m)
```

Exercice 5 On propose d'implémenter un chronomètre qui déclenche un ou plusieurs évènements (`ActionEvent`) à intervalles réguliers en utilisant la classe `Timer` de la bibliothèque `Swing`. Par exemple, on pourrait utiliser un chronomètre dans une animation comme déclencheur pour dessiner les scènes. Pour configurer un chronomètre, il faut créer une instance de `Timer` en lui passant en argument un ou plusieurs `ActionListener`. On démarre le chronomètre en utilisant la méthode `start`.

À titre d'exemple, le code suivant crée et démarre un chronomètre anonyme qui déclenche un événement une fois par seconde. Dans le constructeur de `Timer`, le deuxième argument permet d'enregistrer un écouteur d'action qui exécutera du code chaque fois qu'un `ActionEvent` est levé par le chronomètre.

```
int delay = 1000; // délai en millisecondes
ActionListener listener = new ActionListener() {
    public void actionPerformed(ActionEvent evt) { /* */ }
};
new Timer(delay, listener).start();
```

1. À l'aide d'une classe anonyme, créer un chronomètre qui affiche l'heure courante à chaque seconde (utiliser la méthode statique `LocalTime.now()` de `java.time.LocalTime`).

   Correction :

```
ActionListener ecouteur = new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    System.out.println(LocalTime.now());
  }
};
Timer t1 = new Timer(1000, ecouteur);
```

Complément d'information. Pour que le code fonctionne correctement en pratique, il faut aussi ajouter une instruction `new JFrame().setVisible(true)`. Cette instruction provoque le démarrage du *Event Dispatching Thread* sur lequel Swing exécute les événements.

2. Peut-on refaire la question précédente en utilisant une expression lambda ?
   Correction : Oui : l'interface `ActionListener` est une *interface fonctionnelle* car elle n'a qu'une seule méthode abstraite (la méthode `actionPerformed`). On peut maintenant créer notre `Timer` beaucoup plus facilement.

```
new Timer(1000,
evt -> System.out.println(LocalTime.now())
).start();
```

3. On veut créer un chronomètre qui incrémente un compteur interne toutes les deux secondes. Peut-on utiliser une expression lambda dans ce cas ? Comment peut-on récupérer la valeur courante du compteur ?
   Correction : On doit créer une classe implémentant `ActionListener` qui incrémente un compteur interne. Comme une expression lambda ne permet pas de créer un objet muni d'attributs, il faut utiliser une classe (ici, locale et anonyme).

```
ActionListener compteMoutons = new ActionListener() {
  private int count = 0;
  public void actionPerformed(ActionEvent evt) {
    System.out.println(++count);
  }
};
new Timer(2000, compteMoutons).start();
```

Noter qu'il est impossible avec cette solution de récupérer la valeur du compteur, `ActionListener` ne contient aucune méthode permettant cette lecture. Il faudrait par exemple définir une extension de `ActionListener` contenant cette méthode supplémentaire, et créer une instance de classe anonyme implémentant cette extension.

Exercice 6 Toute liste d'entiers en Java (par exemple une `ArrayList<Integer>`) est manipulable à l'aide d'une reférence de type `List<Integer>`, l'interface contenant les méthodes de gestion telles que `get`, `add`, etc.

Proposer une manière de créer, à partir d'une liste d'entiers d'implémentation quelconque, une liste encore manipulable via `List<Integer>` (c.-à-d. une "vraie"

liste), contenant les mêmes éléments, mais en lecture seule : toute tentative de modification de la copie devra être impossible.

Correction : On peut brider toutes les méthodes d'ajout, de remplacement et de suppression d'une copie en `ArrayList` de la liste initiale :

```
List<Integer> l = new LinkedList<>();
l.add(42);

List<Integer> lRO = new ArrayList<Integer>(l){
  private void te(){
    throw new UnsupportedOperationException(); }
  private boolean tb(){ te(); return false; }
  private Integer tn(){ te(); return null;  }

  public boolean add(Integer n){ return tb(); }
  public void add(int i, Integer n){ te(); }
  public boolean addAll
  (int index, Collection<? extends Integer> c){
    return tb();
  }
  /* etc. :
  public boolean addAll(Collection<? extends Integer> c)
  public void clear()
  public Integer remove(int index)
  public boolean remove(Object o)
  public boolean removeAll(Collection<?> c)
  public boolean removeIf(Predicate<? super Integer> filter)
  protected void removeRange(int fromIndex, int toIndex)
  public boolean retainAll(Collection<?> c)
  public Integer set(int index, Integer element)
  */
};
```

Cette solution présente deux inconvénients :

1. Elle est dépendante du fait que le constructeur `ArrayList(Collection<? extends E> c)` n'invoque aucune de ces méthodes : la collection `c` est convertie en tableau, et une copie de ce tableau devient le support de la liste construite. Le constructeur de même signature de `LinkedList`, en revanche, invoque `addAll`, et son usage ici déclencherait une exception.
2. Elle est entièrement dépendant de l'API de `List`, et rien ne garantit que de nouvelles méthodes ne seront pas ajoutées à cette interface dans les futures versions de Java.

Une solution plus sûre est de se servir de la méthode statique `copyOf` de `List`, spécifiée comme renvoyant une liste non modifiable contenant les mêmes éléments qu'une collection donnée. La seule contrainte est que cette collection ne contienne aucune référence nulle (sous peine de `NullPointerException`) :

```
  List<Integer> l = new ArrayList<>();
  l.add(42);
```

```
List<Integer> lRO = List.copyOf(l);
ListIterator<Integer> it = lRO.listIterator();
it.next();
it.remove();
/*
Exception in thread "main" java.lang.
    UnsupportedOperationException
at java.base/java.util.ImmutableCollections.uoe(
    ImmutableCollections.java:142)
at java.base/java.util.ImmutableCollections$ListItr.remove(
    ImmutableCollections.java:380)
at ...
*/
```