Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2022-2023**

TD – Séance n°8

Exceptions – Expressions Lambda

Exercice 1 Répondre aux questions suivantes :

1. Quelle est la différence entre une excpetion "checked" et une exception "unchecked" ?
2. De laquelle de ces deux classes hérite une `NullPointerException` ? une `ArrayIndexOutOfBoundsException` ? une `IOException` ?
3. Quelle conséquences (quelles nécessités ailleurs dans le code) peut-il y avoir à ajouter `throws` à la signature d'une méthode ? Séparez le cas où l'exception est "checked" et "unchecked".
4. Dans la syntaxe `try {...1...} catch(...2...) {...3...}`, à quelle condition le code en `3` est-il exécuté en cas de levée d'exception en `1` ?

Exercice 2 On définit trois nouvelles exceptions :

```
class CallF extends Exception {}
class CallG extends Exception {}
class CallH extends CallF{}
```

ainsi que les méthodes :

```
public void f() throws CallF {throw new CallF();}
public void g() throws CallG {throw new CallG();}
public void h() throws CallF, CallH {throw new CallH();}
public void skip() throws CallF, CallG, CallH {}
```

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

1.
```
try { f(); g(); }
catch (CallF e) { System.out.println("Catch f()"); }
catch (CallG e) { System.out.println("Catch g()"); }
```

2.
```
try { f(); g(); }
catch (CallG e) { System.out.println("Catch g()"); }
catch (CallF e) { System.out.println("Catch f()"); }
```

3.
```
try { g(); f(); }
catch (CallF e) { System.out.println("Catch f()"); }
catch (CallG e) { System.out.println("Catch g()"); }
```

4.
```
try { h(); }
catch (CallF e) { System.out.println("Catch f()"); }
catch (CallH e) { System.out.println("Catch h()"); }
```

5.
```
try { h(); }
catch (CallH e) { System.out.println("Catch h()"); }
catch (CallF e) { System.out.println("Catch f()"); }
```

6.
```
try { skip(); }
catch (CallF e) { System.out.println("Catch f()"); }
catch (CallG e) { System.out.println("Catch g()"); }
finally { System.out.println("Finally");  }
```

7.
```
try { f(); }
catch (CallF e) { System.out.println("Catch f()"); g(); }
finally { System.out.println("Finally"); }
```

8.
```
try { f(); }
finally { System.out.println("Finally"); g(); }
```

Exercice 3 On considère la classe :

```
public class Universite {
  private int nbEtu;  // nombre d'étudiants inscrits à l'université
  private int capALL; // capacité en arts, lettres, langues
  private int capSHS; // capacité en sciences humaines et sociales
  private int capSTS; // capacité en sciences, technologies et santé
}
```

1. Écrire un constructeur `public Universite(int nbEtu, int capALL, int capSHS, int capSTS)`. Ce constructeur lèvera une exception de type `IllegalArgumentException` (qui est une sous-classe de `RuntimeException`) lorsqu'il le faudra (pour des nombres d'étudiants négatifs par exemple, ou si `nbEtu` est supérieur à la somme des capacités d'accueil).
2. Écrire une méthode `restructuration(int capALL, int capSHS, int capSTS)` qui change les capacités d'accueil de l'université dans chaque pôle de disciplines. Créer une exception `TropPeuDCapaciteException` héritant de `RuntimeException` qui sera levée si la nouvelle capacité totale ne suffit pas pour le nombre d'étudiants.
3. Créer une nouvelle exception `DirectiveMinisterielleException` héritant de `Exception` et modifier la méthode `restructuration` pour qu'elle lève cette exception si les capacités sont trop déséquilibrées (i.e. si un pôle est plus grand que la somme des autres). Le message associé à l'exception devra décrire quel pôle est disproportionné par rapport aux autres.

4. Des restrictions budgétaires frappent toutes les universités et il est absolument nécessaire d'avoir une méthode `restrictionBudgetaire` dans la classe `Universite` pour faciliter la vie des bureaucrates décidant des coupes. La restriction budgétaire consiste ici à réduire la capacité de chaque pôle d'une nouvelle valeur prise aléatoirement entre 0 et 50% de l'ancienne valeur. On a commencé par écrire le code suivant :

```
public void restrictionBudgetaire()
    throws DirectiveMinisterielleException {
  Random rd = new Random();
  int nCapALL = capALL - rd.nextInt((int) (0.5 * capALL));
  int nCapSHS = capSHS - rd.nextInt((int) (0.5 * capSHS));
  int nCapSTS = capSTS - rd.nextInt((int) (0.5 * capSTS));
  this.restructuration(nCapALL, nCapSHS, nCapSTS);
}
```

   On a également ajouté à la classe `Universite` la méthode :

```
public void reduction(int nb) {
  nbEtu -= nb;
}
```

   La méthode `restrictionBudgetaire` peut renvoyer des exceptions de deux types : `DirectiveMinisterielleException` et `TropPeuDCapaciteException`. Encadrer l'appel à `restructuration` par un `try...catch(...)...` de façon à :
   — Réduire de 10 le nombre d'étudiants inscrits en cas de levée d'une `TropPeuDCapaciteException`.
   — Tirer de nouvelles capacités au hasard pour chaque pôle en cas de `DirectiveMinisterielleException`.
   — Appeler de nouveau `restructuration` dans les deux cas.
   On ne ratrappera pas les exceptions levées dans ce deuxième appel à restructuration.

5. On souhaiterait, lorsqu'une restructuration échoue en raison d'un trop grand nombre d'étudiants connaître le nombre d'étudiants en surplus pour les expulser. Comment pouvons nous modifier `TropPeuDCapaciteException` ?

Exercice 4 *Chronomètre*

Un *chronomètre* est un objet de type `Timer` qui déclenche un ou plusieurs "événements d'actionOverride" `ActionEvent` à des intervalles spécifiés. Par exemple, on utilise un chronomètre dans une animation comme déclencheur pour dessiner les scènes.

Pour configurer un chronomètre, il faut créer un objet de type `Timer` (qui implique l'enregistrement d'un ou plusieurs "écouteurs d'action" `ActionListener`), et démarrer le chronomètre en utilisant sa méthode `start`. Par exemple, le code suivant crée et démarre un chronomètre qui déclenche un événement d'action une fois par seconde (comme spécifié par le premier argument `delay` du constructeur de `Timer`). Le deuxième argument du constructeur de `Timer` spécifie un écouteur d'action pour recevoir les événements d'action (`ActionEvent`) du chronomètre.

```
int delay = 1000; // milliseconds
ActionListener taskPerformer = new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        // ... Perform a task ...
    }
};
new Timer(delay, taskPerformer).start();
```

Une fois le chronomètre démarré, il attend une durée qui est égal au délai avant de déclencher son premier événement (`ActionEvent evt`) sur le(s) écouteur(s) enregistré(s). Après ce premier événement, il continue à déclencher des événements chaque fois que le délai entre les événements s'écoule, jusqu'à ce qu'il soit arrêté.

1. Créer un chronomètre `Timer t1` qui affiche l'heure courante à l'écran chaque seconde. Pour obtenir l'heure courante, utiliser la méthode statique `LocalTime.now()` trouvée dans `java.time.LocalTime`. Il faut créer un objet `ActionListener afficheHeure` par l'entremise d'une classe anonyme de `ActionListener` en précisant sa seule méthode abstraite `void actionPerformed(ActionEvent evt)` selon l'exemple ci-dessus.
2. Observer que l'interface `ActionListener` est en fait une *interface fonctionnelle*, car elle n'a qu'une seule méthode abstraite, la méthode `actionPerformed`. Refaire question 1 en utilisant une expression lambda au lieu d'une classe anonyme.
3. Créer un autre chronomètre `Timer t2` qui compte les entiers $1, 2, 3, \ldots$ chaque deux secondes. Il faut créer un autre objet `ActionListener compterMoutons`. Est-ce qu'on peut utiliser une expression lambda dans ce cas ?

Exercice 5 On va créer quelques types simples de structures de données dans cet exercice. On implémentera pour cela l'interface `BoiteAEntiers`. Il existe une interface similaire en Java appelée `Collection` et implémentée notamment par `ArrayList`, `LinkedList` et `HashMap`, mais l'on ne l'utilisera pas ici.

```
interface BoiteAEntiers {
  int lire(int index);

  void ajouter(int valeur);

  void vider();

  int nombreDElements();

  default void inserer(int index, int valeur) {
    throw new UnsupportedOperationException("La méthode
                          inserer n'est pas définie.");}

  default int retirer(int index) {
    throw new UnsupportedOperationException("La méthode
                          retirer n'est pas définie.");}
}
```

1. Ecrire une classe `TableauFixeEnLecture` implémentant l'interface `BoiteAEntiers` avec un constructeur prenant un tableau d'entiers en entrée et le copiant. On n'est pas censé pouvoir modifier les éléments du tableau dans cette classe (tableau en lecture seule) : implémenter les méthodes de l'interface en conséquence. On créera une nouvelle exception `EcritureInterditeException` héritant de `LectureEcritureException`, héritant elle-même de `UnsupportedOperationException`.
2. Ecrire une classe `TableauFixeEnEcriture` au comportement inverse : la classe possède un tableau en attribut dans lequel on peut écrire, mais sans qu'on puisse lire les éléments que l'on a écrit dedans. Créer une exception `LectureException`. (on implémentera `inserer` mais pas `retirer` pour cette classe).
3. Ecrire une classe `ListeDEntier` utilisant une `LinkedList` en interne qui implémente toutes les opérations de `BoiteAEntier`.
4. On ajoute une méthode `ajoutRisque(Object o)` à l'interface `BoiteAEntier`. Que faut-il faire pour ne pas avoir à modifier les classes implémentant l'interface ? Cette méthode doit permettre d'ajouter un objet dont on n'est pas sûr a priori que ce soit un entier. Son emploi étant risqué, elle pourra lever l'exception `IllegalArgumentException`.

Exercice 6 (Facultatif) On souhaite implémenter une structure de données du type dictionnaire (map en anglais), c'est à dire un ensemble de valeurs identifiées par leur clé, comme dans un annuaire. À chaque clé peut être associée au plus une seule valeur. Par exemple, un dictionnaire dont les clés sont des `String` et les valeurs des `int` peut servir à stocker les scores de joueurs en lignes identifiés par leur pseudonymes. On souhaite disposer de méthodes `set` et `get` tel que :

```
Dictionnaire d = new Dictionnaire();
d.set("Pierre",5); //ajoute une entrée dans le dictionnaire
d.set("Marie",10); //pareil
//affiche 5
System.out.println("Le score de Pierre est " + d.get("Pierre"));
//Comme l'entrée Marie existe déjà, set la modifie
d.set("Marie",d.get("Marie")+2); //modifie l'entrée Marie
//affiche 12
System.out.println("Le score de Marie est " + d.get("Marie"));
int s = d.get("inconnu"); //lève une exception
```

Une manière d'implémenter un dictionnaire est de le représenter comme une liste de couples (clé,valeur).

1. Écrire les champs et le constructeur de la classe Dictionnaire. On écrira une classe interne pour représenter une paire clé/valeur et une `LinkedList` pour représenter le dictionnaire
2. Écrire les méthodes `set` et `get`. On rappelle que `set` doit créer une nouvelle entrée s'il n'existe pas déjà de valeur pour la clé donnée, mais doit modifier l'existante dans le cas contraire. Si l'on essaie de lire la valeur d'une clé qui n'est pas dans le dictionnaire, `get` devra lever une exception `ValeurNonTrouveeException` dont on écrira la classe (interne).
3. La bibliothèque java comporte une interface `Iterator`, qui permet de parcourir une structure de données. `Iterator<String>` représente le parcours d'une structure contenant des chaînes de caractères, et `Iterator<Integer>` d'une structure contenant des Integer. On notera que :
   — Les interfaces `Iterator<String>` et `Iterator<Integer>` définissent toutes les deux une méthode `boolean hasNext()` qui indique si le parcours est terminé ou non.
   — `Iterator<String>` et `Iterator<Integer>` définissent une méthode `String next()` qui renvoie le prochain élément, ou lève l'exception `NotSuchElementException` si le parcours est terminé.

   On peut les utiliser dans une boucle while par exemple, pour parcourir des structures variées (listes, tableaux, arbres, etc ...) :

   ```
   //on suppose Iterator<String> it initialisé
   while(it.hasNext())
     System.out.println("Clé suivante : " + it.next());
   ```

   Écrire les méthodes `Iterator<String> parcoursCles()` et `Iterator<Integer> parcoursValeurs()` qui renvoient un itérateur qui permet de parcourir respectivement les clés et les valeurs. On pourra utiliser des classes internes anonymes.