Université Paris Cité

Compléments en Programmation Orientée Objet
TP n° 9 : Lambdas et *Multithreading* (primitives de synchronisation)

1 Lambdas

Exercice 1 : Un peu de Consumer

1. Ecrivez une classe `MaListe<E>` qui implémente `List<E>` par simple héritage de `LinkedList`
2. Définissez y la méthode `void pourChacun(Consumer<E> action)` ; son travail consistera à consommer successivement chaque élément de la liste.
3. Dans une classe `Test` construisez une liste contenant quelques entiers, et testez votre méthode `pourChacun` en lui fournissant en argument une lambda expression de sorte que le résultat affiche les éléments de la liste en retournant à la ligne à chaque fois ;
4. Vous pouvez obtenir le même résultat en transmettant la référence de la fonction `println` au lieu d'une lambda expression. Retrouvez la notation correspondante.
5. Montrez qu'en transmettant en argument à `pourChaque` une classe anonyme on peut faire en sorte que l'affichage numérote aussi les lignes.
6. Reformatez votre solution en déclarant une variable de type `Consumer` et transmettez cette variable à `pourChaque`
7. Définissez une classe `Personne` sa méthode de présentation `toString` et construisez une `MaListe` de quelques personnes.
8. On souhaite afficher cette liste de personne, avec la numérotation de ligne, en utilisant tel quel le `Consumer` de la question 5. Quelles modifications faut-il faire pour que son utilisation soit compatible.
9. Dans cette question, on va tourner un peu autour des types. On souhaite avoir un `Consumer` dont l'acceptation concerne une liste complète. C'est à dire qu'on pourra l'utiliser sous cette forme : `my_consumer.accept(maListe)`

   Ecrivez un tel `Consumer` dont le travail consiste à afficher les éléments de la liste ligne par ligne en les numérotant. Testez le. Vous devriez pouvoir l'écrire avec une lambda expression.

Exercice 2 : Autres interfaces

(cf. cours et `java.util.function`)
Ajoutez à `MaListe` les méthodes suivantes :

1. `List<E> filter(Predicate<E> pred)` : retourne une nouvelle liste consistant en les éléments de `this` qui satisfont le prédicat.
2. `<U> List<U> map(Function<E,U> f)` : retourne une liste dont les éléments sont tous les éléments de `this` auxquels on a appliqué la fonction `f`
3. `<U> U fold(U z, BiFunction<U, E, U> f)` : initialise un accumulateur a avec `z`, puis, pour chaque élément x de `this`, calcule $a =$ `f` (a, x) et finalement retourne a.

   Exemple : pour demander la somme d'une liste d'entiers : `l.fold(0, (a,x) -> a + x)`.
4. Écrivez et testez les appels permettant d'utiliser `fold` pour calculer le produit, puis le maximum d'une liste d'entiers.

Exercice 3 : Objets transformables

On donne l'interface suivante :

```
interface Transformable<T> {
    T getElement();
    void transform(UnaryOperator<T> trans);
}

```

Les instances de cette interface seront typiquement des objets avec un état (attribut) de type `T`, modifiable en passant des fonctions $T \to T$ à la méthode `transform`. Par exemple, avec un attribut de type `String`, pour lui concaténer la chaîne `"toto"` : `obj.transform(s -> s + "toto")` ; ou bien pour la passer en minuscules : `obj.transform(String::toLowerCase)`.

1. Écrivez la classe `EntierTransformable` qui implémente cette interface pour des entiers.
2. Écrivez un `main()` qui instancie un tel objet (en initialisant l'entier à 0), puis lui applique les opérations suivantes : multiplication par 2, ajout de 15, réinitialisation à 0...
3. Écrivez une classe `Additionneur` dont les objets peuvent être utilisés comme fonction $x \to x + n$ pour la classe `EntierTransformable`.
   En particulier le programme ci-dessous doit afficher 15 :

```
EntierTransformable x =  new EntierTransformable(12);
x.transform(new Additionneur(3));
System.out.println(x.getElement());

```

Exercice 4 : Curryfication

La *curryfication* est l'opération consistant à transformer une fonction de type $(T_1 \times T_2 \times T_3 \cdots \times T_n) \to R$ en $T_1 \to (T_2 \to (T_3 \to (\cdots \to R) \ldots))$.
L'intérêt est de permettre une application partielle en ne donnant que le(s) premier(s) argument(s), ce qui retournera une nouvelle fonction. Par exemple si l'addition curryfiée s'écrit $add = x \to (y \to (x + y))$, alors la valeur de $add(3)$ est la fonction $y \to (3 + y)$.

1. Écrivez une méthode qui prend une fonction binaire de type $(T \times U) \to R$ et retourne sa version curryfiée, de type $T \to (U \to R))$ (comment ces types se traduisent-ils à l'aide des interfaces de `java.util.function` ?).
2. Écrivez la méthode inverse.
3. Même questions pour les fonctions ternaires. Avant de vous lancer dans cette question, remarquez qu'il n'existe pas d'interface java modélisant les fonctions ternaires et qu'il faudra donc définir une interface adaptée (exemple : `interface TriFunction<T, U, V, R> { R apply(T,U,V) ;}`).

Exercice 5 : Optionnels fonctionnels (suite)

On souhaite maintenant compléter l'API de `Optionnel` de l'exercice 3 de TP 7 en fournissant des méthodes permettant d'exécuter du code conditionnellement en fonction de l'état (vide ou non vide) de l'optionnel.
Concrètement, on demande les méthodes suivantes (« `???` » = trouvez la bonne interface fonctionnelle.) :

- `Optionnel<T> filtre(Predicate<T> cond)` : retourne l'optionnel lui-même s'il contient une valeur et qu'elle satisfait le prédicat `cond` ; retourne `Optionnel.vide()` sinon.
- `void siPresent(??? f)` : si l'optionnel contient une valeur `v`, alors exécute `f` avec le paramètre `v`. Si l'optionnel est vide, ne fait rien.
- `<U> Optionnel<U> map(??? f)` : si l'optionnel contient une valeur `v`, alors retourne un optionnel contenant le résultat de `f` appliqué à `v`. Sinon retourne l'optionnel vide.

2 Accès en compétition et *thread-safety*

Exercice 6 : Accès en compétition

Les classes suivantes interdisent-elles les accès en compétition au contenu de leurs instances ?
Attention : pour cet exercice, on considère que le « contenu », c'est aussi bien les attributs que les attributs des attributs, et ainsi de suite.
Rappel : 2 accès à une même variable partagée sont en compétition si au moins l'un est en écriture et il n'y a pas de relation arrivé-avant entre les deux accès.

```java
public final class Ressource {
    public static class Data { public int x; }
    public final Data content;
    public Ressource(int x) {
        content = new Data();
        content.x = x;
    }
}

public final class Ressource2 {
    private String content;
    private boolean pris = false;

    private synchronized void lock() throws InterruptedException {
        while(pris) wait();
        pris = true;
    }

    private synchronized void unlock() {
        pris = false;
        notify();
    }

    public void set(String s) throws InterruptedException {
        lock();
        try { content = s; }
        finally { unlock(); }
    }

    public String get() throws InterruptedException {
        lock();
        try { return content; }
        finally { unlock(); }
    }
}

public final class Ressource3 {
    public static class Data {
        public final int x;
        public Data(int x) { this.x = x; }
    }
    public volatile Data content;

    public Ressource3(int x) { content = new Data(x); }
}
```

Exercice 7 : *Thread-safe ?*

Une classe est *thread-safe* si sa spécification reste vraie dans un contexte d'utilisation multi-thread. Quelles classes parmi les suivantes sont *thread-safe* pour la spécification : « à tout moment, la valeur retournée par le getteur est égale au nombre d'appels à `incremente` déjà entièrement exécutés » ?

```java
public final class Compteur {
    private int i=0;
    public synchronized void incremente() { i++; }
    public synchronized int get() { return i; }
}

```

```java
public final class Compteur2 {
    private volatile int i = 0;
    public void incremente() { i++; }
    public int get() { return i; }
}

public final class Compteur3 {
    private int i=0;
    public synchronized void incremente() { i++; }
    public int get() { return i; }
}
```

3 Synchronisation et moniteurs

Exercice 8 : Compteurs

On considère la classe `Compteur`, que nous voulons tester et améliorer :

```java
public class Compteur {
    private int compte = 0;
    public int getCompte() { return compte; }
    public void incrementer() { compte++; }
    public void decrementer() { compte--; }
}
```

1. À cet effet, on se donne la classe `CompteurTest` ci-dessous :

```java
public class CompteurTest {
    private final Compteur compteur = new Compteur();

    public void incrementerTest() {
        compteur.incrementer();
        System.out.println(compteur.getCompte() + " obtenu après incrémentation");
    }

    public void decrementerTest() {
        compteur.decrementer();
        System.out.println(compteur.getCompte() + " obtenu après décrémentation");
    }
}

```

   Écrivez un `main` qui lance sur une seule et même instance de la classe `CompteurTest` des appels à `incrementerTest` et `decrementerTest` depuis des *threads* différents. Pour vous entraîner à utiliser plusieurs syntaxes, lancez en parallèle :
   - une décrémentation à partir d'une classe locale, dérivée de `Thread` ;
   - une décrémentation à partir d'une implémentation anonyme de `Runnable` ;
   - une incrémentation à partir d'une lambda-expression obtenue par lambda-abstraction (syntaxe `args -> result`) ;
   - une incrémentation à partir d'une lambda-expression obtenue par référence de méthode (syntaxe `context::methodName`).

2. On souhaite maintenant qu'il soit garanti, même dans un contexte *multi-thread*, que la valeur de `compte` (telle que retournée par `getCompte`) soit toujours égale au nombre d'exécutions d'`incrementer` moins le nombre d'exécutions de `decrementer` ayant terminé avant le retour de `getCompte` (rappel : l'incrémentation `compte++` et la décrémentation `compte--` ne sont pas des opérations atomiques).

   Obtenez cette garantie en ajoutant le mot-clé **`synchronized`** aux endroits adéquats dans la classe `Compteur`.

3. Est-ce que les modifications de la question précédente assurent que `incrementerTest` et `decrementerTest` affichent bien la valeur du compteur obtenue après, respectivement, l'appel à `incrementer` ou à `decrementer` fait dans chacunes des deux méthodes de test ?

   Comment modifier `CompteurTest` pour que ce soit bien le cas ?

4. On veut ajouter à la classe `Compteur` la propriété supplémentaire suivante : « `compte` n'est jamais négatif ». Celle-ci peut être obtenue en rendant l'appel à `decrementer` bloquant quand `compte` n'est pas strictement positif. Modifiez la classe `CompteurTest` en introduisant les `wait()` et `notify()` nécessaires.

Exercice 9 :

On considère le programme suivant (fichier `Producer.java` sur moodle) :

```java
import java.util.Scanner;

public final class Producer {
    private long v = 0;
    private final Thread thread = new Thread(this::produce);
    public Thread getThread() { return thread; }
    private void produce() {
        while(true) { v++; }
    }
    public synchronized long getValue() { return v; }
}

final class Main {
    public static void main(String[] args) {
        // Démarrage du thread producteur
        Producer p = new Producer();
        p.getThread().start();
        // Lecture des valeurs produites
        try (var scanner = new Scanner(System.in)) {
            while (!scanner.nextLine().equals("q"))
                System.out.println("main = " + p.getValue());
        }
        // Interruption du thread producteur
        Thread.State state = p.getThread().getState();
        System.out.println( "state before interrupt " + state.name());
        p.getThread().interrupt();
        do {
            state = p.getThread().getState();
            System.out.println(state.name());
        }
        while (p.getThread().isInterrupted() && state != Thread.State.TERMINATED);
    }
}
```

Le *thread* du producteur produit une suite de valeurs `long`, qui peuvent être interprétées comme les mesures d'une quantité physique obtenues à l'aide d'un capteur. Le capteur fournit une suite continue de valeurs, peu importe si on veut ou non lire la valeur courante.

Le *thread* principal lit, dans une boucle, des lignes sur l'entrée standard et affiche la valeur courante du capteur tant que la ligne lue n'est pas le caractère `q`.
Dès que l'utilisateur entre le caractère `q` sur le terminal, on essaye d'arrêter le *thread* producteur.

1. Qu'observez-vous en exécutant le programme ? Est-ce que la tentative d'interruption réussit ?
2. Modifiez le code de `produce` de telle sorte que l'interruption demandée dans la méthode `main` interrompe bien le *thread* producteur (et que l'état de ce *thread* devienne bien, mais peut-être pas immédiatement, `TERMINATED`).
3. Comment pourrait-on rendre l'incrémentation dans `produce` atomique ? Est-ce que ajouter `volatile` à la déclaration de `v` suffit ? Sinon, quelle est la solution ? (Attention à ne pas rendre le programme inutilisable...)
4. Argumentez sur l'utilité de prendre une telle précaution.