

TP n° 11 : ForkJoin, CompletableFuture, Flow (Correction)

I) Problèmes récurifs avec ForkJoin

Exercice 1 : Tri fusion

Le code suivant présente une implémentation du tri fusion en Java :

```
1 import java.util.*;
2
3 public class TriFusion {
4
5     protected static <E extends Comparable<? super E>> List<E> fusion(List<E> l1, List<E> l2) {
6         // ArrayList plutôt que LinkedList (pour get en temps constant)
7         List<E> l3 = new ArrayList<>(l1.size() + l2.size());
8         ListIterator<E> it1 = l1.listIterator(), it2 = l2.listIterator();
9         while (it1.hasNext() || it2.hasNext())
10             if (it1.hasNext()
11                 && (!it2.hasNext()
12                     || l1.get(it1.nextIndex()).compareTo(l2.get(it2.nextIndex())) < 0))
13                 l3.add(it1.next());
14             else l3.add(it2.next());
15         return l3;
16     }
17
18     public static <E extends Comparable<? super E>> List<E> triMonoThread(List<E> l) {
19         if (l.size() <= 1) return l;
20         else {
21             int pivot = Math.floorDiv(l.size(), 2);
22             List<E> l1 = triMonoThread(l.subList(0, pivot));
23             List<E> l2 = triMonoThread(l.subList(pivot, l.size()));
24             return fusion(l1, l2);
25         }
26     }
27
28     public static void main(String[] args) {
29         // doit afficher : [1, 2, 12, 81, 99, 122, 122, 234, 2134]
30         System.out.println(triMonoThread(Arrays.asList(234, 2134, 1, 122, 122, 2, 99, 12, 81)));
31     }
32 }
```

Programmez une version concurrente de ce code à l'aide de [ForkJoinTask](#).

Correction : Correction en utilisant des adaptateurs de lambda (alternative : étendre [RecursiveTask](#) comme dans la correction de l'exercice d'après).

```
1 import java.util.Arrays;
2 import java.util.List;
3 import java.util.concurrent.ForkJoinTask;
4 import static java.util.concurrent.ForkJoinTask.adapt;
5
6 public class TriFusionFJ {
7
8     private static <E extends Comparable<? super E>> List<E> triForkJoin(List<E> l) {
9         if (l.size() <= 1)
10             return l;
11         else {
12             // instrumentons, juste pour montrer ce qui se passe :
13             System.out.println(
14                 "Liste de taille " + l.size() + " séparée dans le thread " +
15                 Thread.currentThread().getName());
16             int pivot = Math.floorDiv(l.size(), 2);
17             ForkJoinTask<List<E>> f1 = adapt(() -> triForkJoin(l.subList(0,
```

```

17         pivot))).fork();
18         ForkJoinTask<List<E>> f2 = adapt(() -> triForkJoin(l.subList(pivot,
19         l.size()))).fork();
20         List<E> result = TriFusion.fusion(f2.join(), f1.join());
21         // instrumentons aussi après les join :
22         System.out.println(
23             "Liste de taille " + l.size() + " fusionnée dans le thread " +
24             Thread.currentThread().getName());
25         return result;
26     }
27 }
28
29 public static void main(String[] args) {
30     List<Integer> l = Arrays.asList(8, 4, 7, 1, 2, 9, 4, 3, 5, 7);
31     System.out.println("liste triée : " + triForkJoin(l));
32 }

```

Remarque : ici on “fork” simplement les tâches sans préciser de `ForkJoinPool`. Cela veut dire que c’est le *pool* par défaut qui sera utilisé.

Exercice 2 : Factorisation d’entiers

But/prétexte de l’exercice : écrire une méthode qui factorise les nombres entiers en facteurs premiers en suivant l’algorithme récursif suivant :

`factorize(n)` :

- entrée : n , le nombre à factoriser
 - on calcule : $m = \lfloor \sqrt{n} \rfloor$ (arrondi vers l’entier en dessous de la racine carrée de n)
 - on part de m et on décroît jusqu’à trouver d le premier diviseur de n inférieur ou égal à m .
 - on appelle `factorize(d)` et `factorize(n/d)`
 - on retourne l’union des 2 ensembles obtenus ci-dessus.
1. Implémentez cet algorithme à l’aide d’une méthode faisant des appels récursifs sur le même *thread* (faites comme si vous n’aviez jamais entendu parler de *threads*). N’utilisez pas des **int** mais des **long**.

Correction :

```

1  import java.util.Set;
2  import java.util.TreeSet;
3
4  public class Factorisation {
5      public static Set<Long> factorise(long number){
6          var divisor = (long)Math.sqrt(number);
7          while(number % divisor!=0){
8              divisor--;
9          }
10         if(divisor==1){
11             var selfSingleton = new TreeSet<Long>();
12             selfSingleton.add(number);
13             return selfSingleton;
14         }
15         else
16         {
17             var factors = factorise(divisor);
18             factors.addAll(factorise(number/divisor));
19             return factors;
20         }
21     }
22 }

```

2. Réécrivez cette méthode pour que les tâches soient des `ForkJoinTask` qu'on envoie sur un `ForkJoinPool` de taille fixée. Pour cela, implémenter une classe `Factorisation` qui **extends** la classe `RecursiveTask<>`.

Testez sur des entiers pour lesquels vous savez qu'il y a beaucoup de facteurs. Testez par exemple sur l'entier `1730884069530000L` (notez le `L` : on travaille sur des **long**).

Correction : Ici on étend `RecursiveTask` pour créer des tâches. On aurait aussi pu adapter des lambdas, comme à l'exercice précédent.

```
1  import java.util.Set;
2  import java.util.TreeSet; // plus sympa que HashSet car ça trie les éléments
3  import java.util.concurrent.ForkJoinPool;
4  import java.util.concurrent.RecursiveTask;
5
6  public class FactorisationFJ extends RecursiveTask<Set<Long>> {
7      // serialVersionUID requis parce que RecursiveTask implémente Serializable
8      private static final long serialVersionUID = 4302593939061433100L;
9      private final long number;
10
11     public FactorisationFJ(long number) {
12         this.number = number;
13     }
14
15     @Override
16     protected Set<Long> compute() {
17         long divisor = (long) Math.sqrt(number);
18         while (number % divisor != 0) {
19             divisor--;
20         }
21         if (divisor == 1) {
22             var selfSingleton = new TreeSet<Long>();
23             selfSingleton.add(number);
24             return selfSingleton;
25         } else {
26             FactorisationFJ f1 = new FactorisationFJ(divisor);
27             f1.fork();
28             FactorisationFJ f2 = new FactorisationFJ(number / divisor);
29             Set<Long> h = f2.compute();
30             /*
31              * on pourrait aussi forker pour plus de granularité dans
32              * l'ordonnancement. Quoi qu'il en soit, ça ne devrait pas changer
33              * le degré de concurrence.
34              */
35             h.addAll(f1.join());
36             /*
37              * Remarque: on récupère un ensemble mutable créé dans un autre
38              * thread. Heureusement, le join assure qu'il n'y a pas d'accès
39              * en compétition au contenu de cet ensemble (ce qui se passe
40              * dans la sous-tâche "arrive-avant" ce qu'on fait ici).
41              *
42              * On pourrait décider de travailler sur des ensembles immuables pour
43              * "assurer ses arrières", mais le résultat est moins optimal
44              * (notamment, la fusion se fait forcément dans une structure
45              * mutable, qu'il faudrait ensuite transformer en ensemble immuable
46              * via une copie... ).
47              */
48             return h;
49         }
50     }
51
52     public static void main(String[] args) {
53         System.out.println((new ForkJoinPool(5)).invoke(new
54             FactorisationFJ(1730884069530000L)));
55     }
56 }
```

II) Traitements de flux de données avec **Flow**

Exercice 3 : Données épidémiologiques

La pandémie de Covid-19 faisant rage, la clé de l'organisation de la réponse tient à la bonne interprétation des données statistiques disponibles.

À cette fin, les experts ont mis au point divers indicateurs, qui souvent doivent être calculés à partir des données brutes au fur et à mesure qu'elles arrivent.

Parmi les données brutes disponibles, on peut trouver les suivantes (par jour) :

- Le nombre de tests pratiqués.
- Le nombre de nouveaux cas détectés.
- Le nombre de nouvelles hospitalisations.
- Le nombre de sorties de l'hôpital.

Les indicateurs dérivés jugés pertinents sont :

- Le nombre total de cas détectés.
- La moyenne de chacun des indicateurs bruts sur les 7 derniers jours ¹.
- Le taux de reproduction effectif (le fameux R). Il s'agit du nombre de personnes contaminées par un porteur en moyenne.

En considérant qu'il se passe 2 semaines en moyenne entre le moment où l'on est contaminé et celui où l'on contamine quelqu'un d'autre, on peut approximer R effectif en divisant le nombre de nouveaux cas du jour par celui de 14 jours avant (on peut aussi lisser comme à la question précédente).

- Le nombre de patients hospitalisés à un instant donné.
- Le taux d'incidence (c'est à dire pourcentage de positifs parmi les tests réalisés).

À faire : écrire un simulateur, utilisant l'API **Flow**, qui imprime les indicateurs dérivés au fur et à mesure que les données brutes quotidiennes arrivent (implémentez seulement quelques exemples, puis passez à la suite du TP !).

- Les données brutes seront soumises (méthode **submit**) à une instance de **SubmissionPublisher** différente pour chaque série de données. L'alimentation de ces *publishers* se fera soit depuis des tableaux de données, soit depuis l'entrée standard.
- Les indicateurs seront calculés dans des **Flow.Processor**², abonnés aux flux de données brutes (méthode **subscribe**).
- L'affichage sera réalisé dans des **Flow.Subscriber** abonnés aux indicateurs.
- Tous ces objets seront instanciés, connectés entre eux (**subscribe**) et alimentés (**submit**) dans le **main** de votre programme.
- Instrumentez votre code pour que tous les appels à **onNext** affichent le nom du *thread* courant, ce qui vous permettra de constater le parallélisme effectif.

Attention

- Certains indicateurs dépendent de plusieurs flux de données brutes, qu'il faudra lire de façon synchronisée depuis plusieurs **Publisher**. Pour cela, vous pouvez instancier, étendre ou composer la classe **FanInProcessor** donnée ci-dessous (ou la modifier selon vos besoins).

1. En effet, les statistiques rapportées le weekend sont incomplètes et généralement reportées sur le début de semaine suivante. La moyenne glissante sur 7 jours permet de gommer ces variations artificielles.

2. Vous pouvez étendre la classe **AbstractProcessor** montrée en cours, par exemple, ou bien travailler directement à partir de **SubmissionPublisher**.

- `FanInProcessor` est complètement inutile si l'indicateur implémenté n'utilise qu'une seule série de données brutes! Ne l'utilisez pas dans ce cas!

La classe `FanInProcessor` :

```
1 import java.util.concurrent.Flow.*;
2 import java.util.concurrent.SubmissionPublisher;
3 import java.util.function.*;
4
5 public class FanInProcessor<U, V, R> implements Publisher<R>, AutoCloseable {
6     public class FanInSubscriber<T> implements Subscriber<T> {
7         private final Consumer<T> store;
8         private Subscription subscription;
9
10        public FanInSubscriber(Consumer<T> store) {
11            this.store = store;
12        }
13
14        @Override
15        public void onSubscribe(Subscription subscription) {
16            this.subscription = subscription;
17            subscription.request(1);
18        }
19
20        @Override
21        public void onNext(T message) {
22            synchronized (FanInProcessor.this) {
23                store.accept(message);
24                tryProcessNext();
25            }
26        }
27
28        @Override
29        public void onError(Throwable arg0) {
30        }
31
32        @Override
33        public void onComplete() {
34        }
35    }
36
37    public final FanInSubscriber<U> leftInput = new FanInSubscriber<>(message -> {
38        lastLeft = message;
39    });
40
41    public final FanInSubscriber<V> rightInput = new FanInSubscriber<>(message -> {
42        lastRight = message;
43    });
44
45    private U lastLeft; // null quand dernière donnée disponible déjà consommée
46    private V lastRight; // null quand dernière donnée disponible déjà consommée
47    private final SubmissionPublisher<R> output = new SubmissionPublisher<>();
48    private final BiFunction<U, V, R> transform;
49
50    @Override
51    public void subscribe(Subscriber<? super R> subscriber) {
52        output.subscribe(subscriber);
53    }
54
55    @Override
56    public void close() {
57        output.close();
58    }
59
60    private void tryProcessNext() {
61        if (lastLeft == null || lastRight == null)
62            return; // la source en avance s'arrête là et ne fait rien de particulier
63        output.submit(transform.apply(lastLeft, lastRight));
64    }
```

```
65     lastLeft = null;
66     lastRight = null;
67     leftInput.subscription.request(1);
68     rightInput.subscription.request(1);
69 }
70
71 public FanInProcessor(BiFunction<U, V, R> transform) {
72     this.transform = transform;
73 }
74
75 }
```

Exemple d'utilisation :

```
1 import java.util.concurrent.SubmissionPublisher;
2 import static java.util.stream.Stream.of;
3 import static java.util.concurrent.ForkJoinPool.commonPool;
4 import static java.util.concurrent.TimeUnit.MILLISECONDS;
5
6 public class FanInTest {
7     public static void main(String[] args) throws InterruptedException {
8         try (var source1 = new SubmissionPublisher<Integer>();
9             var source2 = new SubmissionPublisher<Integer>();
10            var multiplicator = new FanInProcessor<Integer, Integer, Integer>((a, b) -> a *
11                b)) {
12             // 1. on construit l'application
13             source1.subscribe(multiplicator.leftInput);
14             source2.subscribe(multiplicator.rightInput);
15             multiplicator.subscribe(new PrintSubscriber<>());
16             // 2. on l'alimente avec des données
17             of(34, 1234, 123, 12, 98, 93).forEach(source1::submit); // première série de données
18             of(49, 34, 313, 132, 898, 293).forEach(source2::submit); // deuxième série de données
19             // 3. on attend que ça se termine
20             commonPool().awaitTermination(1000, MILLISECONDS);
21         } // les ressources sont fermées en sortant du try-with-resource
22     }
23 }
```

avec

```
1 import java.util.concurrent.Flow.*;
2
3 public class PrintSubscriber<T> implements Subscriber<T> {
4     private Subscription subscription;
5
6     @Override
7     public void onSubscribe(Subscription subscription) {
8         this.subscription = subscription;
9         subscription.request(1);
10    }
11
12    @Override
13    public void onNext(T message) {
14        subscription.request(1);
15        System.out.println(message);
16    }
17
18    @Override
19    public void onError(Throwable arg0) {
20    }
21
22    @Override
23    public void onComplete() {
24    }
25 }
```

III) Calculs concurrents simples avec `CompletableFuture`

Exercice 4 :

Ci-dessous, un certain nombre de calculs arithmétiques :

1. 12
2. $2 * 5$
3. $(3 + 2) * 9$
4. $(x + 5)$, où x est lu dans l'entrée standard dans une tâche concurrente
5. $(3 + 2) * (3 + 2)$, sans recalculer le terme qui se répète.

Écrivez ces calculs sous forme de tâches successives décrites par des `CompletableFuture`, avec pour règle : effectuer une seule opération par tâche.

Exemple pour $6 + 7$:

```
1  var e1 = completedFuture(6);
2  var e2 = completedFuture(7);
3  var e3 = e1.thenCombine(e2, (a,b) -> a + b);
4  System.out.println(e3.join()); // affiche 13
```

(Ne pas oublier le `import static java.util.concurrent.CompletableFuture.*;` initial!)

Faites d'abord en sorte que le résultat soit imprimé dans `main`, puis modifiez ensuite vos programmes pour que le `println` soit aussi exécuté dans une tâche concurrente.

Enfin ajoutez `System.out.println(Thread.currentThread().getName());` dans chaque tâche concurrente pour savoir dans quel *thread* elle s'exécute.

Conseil : n'hésitez pas à vous référer à la documentation de `CompletableFuture`³.

Correction :

```
1  import static java.util.concurrent.CompletableFuture.*;
2  import java.util.Scanner;
3  import java.util.concurrent.CompletableFuture;
4  import java.util.ArrayList;
5
6  public class CalculsCF {
7      public static void main(String[] args) throws InterruptedException {
8          try (var s = new Scanner(System.in)) {
9              /*
10               * On aura besoin d'un scanner plus tard. On l'introduit ici car on veut que
11               * s.close() soit exécuté : seulement quand la tâche concurrente qui
12               * l'utilise est terminée, et dans tous les cas (fin sur exception ou non).
13               */
14
15              var tasks = new ArrayList<CompletableFuture<?>>();
16
17              // 1.
18              tasks.add(completedFuture(12));
19              // 2.
20              tasks.add(completedFuture(2)
21                  .thenCombine(
22                      completedFuture(5),
23                      (a, b) -> a * b));
24              // 3.
25              tasks.add(completedFuture(3)
26                  .thenCombine(
27                      completedFuture(2),
```

3. <https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/CompletableFuture.html>

```
28         (a, b) -> a + b)
29         .thenCombine(
30             completedFuture(9),
31             (a, b) -> a * b));
32     // 4.
33     tasks.add(supplyAsync(s::nextInt)
34         .thenCombine(
35             completedFuture(5),
36             (a, b) -> a + b));
37     // 5.
38     var x = completedFuture(3)
39         .thenCombine(
40             completedFuture(2),
41             (a, b) -> a + b);
42     tasks.add(x.thenCombine(x, (a, b) -> a * b));
43
44     // soit on join chaque tâche et on l'affiche :
45     // for (var t: tasks) System.out.println(t.join());
46
47     // soit on ajoute l'affichage comme tâche concurrente (et il faut quand-même
48     // join-er pour ne pas quitter le programme avant que tout ne soit terminé) :
49     var printedTasks = new ArrayList<CompletableFuture<?>>();
50     for (var t: tasks) printedTasks.add(t.thenAccept(System.out::println));
51     for (var t: printedTasks) t.join();
52     //
53     // On aurait aussi pu répéter les thenAccept(System.out::println) dans
54     // la définition de chaque tâche, et se contenter d'un join général à la fin.
55     // Par contre, on ne peut pas faire les join tout de suite (sans liste) sans
56     // perdre le parallélisme.
57 }
58
59
60 // instrumentons (commenter si on ne veut plus l'instrumentation)
61 private static void printThread() {
62     System.out.println(Thread.currentThread().getName());
63 }
64
65 private static <T> CompletableFuture<T> completedFuture(T value) {
66     printThread();
67     return CompletableFuture.completedFuture(value);
68 }
69 }
```