TP n° 11 bis : La récursion avec CompletableFuture
(Correction)

Les exercices qui suivent sont des questions qu'on peut se poser naturellement après avoir fait les exercices avec `ForkJoin` : peut-on faire la même chose avec `CompletableFuture` ?
La réponse est oui, mais cela demande de la réflexion (en pratique, `CompletableFuture` sera plus facile à utiliser dans des cas non récursifs).
Que cela ne tienne, essayons quand-même !

Quelques indications Pour vous aider sur la programmation de problèmes récursifs à l'aide de `CompletableFuture`, voici une traduction utilisant `CompletableFuture` du programme donné dans le cours qui calculait la suite de Fibonacci à l'aide de `ForJoinTask` :
On se donne déjà une fonction auxiliaire (explications données après) :

```
1  package up.cpoo.concurrent;
2  import java.util.concurrent.CompletableFuture;
3  import static java.util.concurrent.CompletableFuture.*;
4  import java.util.function.*;
5
6  public final class CompletableFutureTools {
7     private CompletableFutureTools() {}
8
9     /*
10     * Méthode manquante dans l'API CompletableFuture, pourtant très utile pour ce genre
11     * d'exercice. Elle prend en paramètre une fonction construisant un CompletableFuture
12     * et retourne un CompletableFuture qui devra retourner la même valeur que celui
13     * qui est retourné par la fonction...
14     * ... mais jamais supplyAndFlatten ne demande l'exécution de la tâche associée.
15     * Ainsi, elle ne sera exécutée que lorsqu'on appellera join.
16     */
17    public static <T> CompletableFuture<T> supplyAndFlatten(Supplier<CompletableFuture<T>> fn)
          {
18       return supplyAsync(fn).thenCompose(x -> x);
19       // (plus ou moins équivalent à :) return completedFuture(null).thenComposeAsync(x ->
             fn.get());
20    }
21
22 }
```

puis on programme Fibonacci :

```
1  import java.util.concurrent.CompletableFuture;
2  import static java.util.concurrent.CompletableFuture.*;
3  import static up.cpoo.concurrent.CompletableFutureTools.*;
4
5  public class FiboCF {
6
7     public static CompletableFuture<Integer> calculFibo(int n) {
8        if (n <= 1)
9           return completedFuture(1);
10       else {
11          CompletableFuture<Integer> f1 = supplyAndFlatten(() -> calculFibo(n - 1));
12          CompletableFuture<Integer> f2 = supplyAndFlatten(() -> calculFibo(n - 2));
13          return f1.thenCombine(f2, (x, y) -> x + y);
14       }
15    }
16
17    public static void main(String[] args) {
18       System.out.println(calculFibo(30).join());
19    }
20
21 }
```

L'idée c'est que la méthode « récursive » [1] retourne, au lieu de la valeur de type `T` à calculer, un `CompletableFuture<T>`, lui-même obtenu en composant les `CompletableFuture<T>` retournés par les appels « récursifs » (correspondant aux sous-tâches).

- Pour le cas de base, on utilise la méthode `static <T> CompletableFuture<T> completedFuture(T val)`, qui retourne un `CompletableFuture` déjà calculé dont la valeur est le paramètre passé à la méthode.
- Pour le cas « récursif », on veut éviter :
  - l'attente du résultat d'une sous-tâche (`get` ou `join`) au sein d'une tâche (il y avait de telles attentes avec `ForkJoinTask`, mais l'idée de `CompletableFuture` c'est justement de décomposer en tâches élémentaires qui calculent directement un résultat à partir des paramètres entrés, sans attendre de résultat d'une autre tâche).
    Le seul `join()` est appelé sur le résultat de l'appel initial (ligne 18, dans le `main`).
  - les « vrais » appels récursifs (qui provoqueraient des appels imbriqués non bornés de méthodes, limitant la répartition sur plusieurs *threads*, et risquant de faire déborder la pile d'exécution). Ainsi, on doit privilégier une récursion « indirecte », où les appels à `f` dans `f` n'apparaîtraient que dans le corps de lambda-expressions passées en paramètre à l'une des méthodes `xxxAsync` de `CompletableFuture`, pour être soumises au *thread pool* pour une exécution concurrente.

  Le problème est que ces méthodes retournent toutes un résultat enrobé dans un `CompletableFuture` (sauf `thenCompose[Async]` [2])... on obtient alors un `CompletableFuture<CompletableFuture<T>>`... qu'il faut « aplatir » vers un `CompletableFuture<T>`, d'où la méthode auxiliaire `supplyAndFlatten` proposée. [3] [4]

Exercice 1 : Échauffement : récursion infinie

Comparez les programmes suivants :

```
1  public class InfiniteRecursion {
2     public static void incr(int n) {
3        System.out.println(n);
4        incr(1 + n);
5     }
6
7     public static void main(String[] args) {
8        incr(0);
9     }
10 }
```

et

```
1  import static java.util.concurrent.CompletableFuture.*;
2
3  import java.util.concurrent.CompletableFuture;
4
5  public class InfiniteRecursionCF {
6     public static CompletableFuture<Void> incr(int n) {
7        System.out.println(n);
8        return completedFuture(n+1).thenComposeAsync(InfiniteRecursionCF::incr);
9     }
10
```

1. C'est une fausse récursion, voir remarques plus loin.
2. Ces 2 méthodes sont à `CompletableFuture` ce que `flatMap` est à `Stream`.
3. Il aurait été bien pratique d'avoir une telle méthode dans l'API `CompletableFuture` du JDK !
4. La variante proposée en commentaire utilise directement `thenComposeAsync`, qui retourne un résultat déjà « aplati », mais cette méthode doit être appelée sur un `CompletableFuture` existant ; nous lui en fournissons donc un « factice » (`completedFuture(null)`).

```
11      public static void main(String[] args) {
12          incr(0).join();
13      }
14  }
```

Que se passera-t-il quand vous exécuterez le premier ? Et le second ? Pourquoi ?
Au passage, l'exécution du second est-elle concurrente ou séquentielle ?

Correction : Le premier programme va boucler un certain nombre de fois jusqu'à quitter sur un `StackOverFlowError`.
Le second va juste boucler indéfiniment (et compter jusqu'à ce qu'on arrête le programme).
Dans le premier les appels de fonction sont réellement récursifs, ce qui veut dire qu'un nouveau *frame* est créé et ajouté à la pile à chaque appel. Or la pile a une capacité bornée, ce qui explique le dépassement de pile (et l'erreur associée).
Dans le second, il n'y a pas de vrai appel récursif. Les `CompletableFuture` créés sont envoyés au *thread pool* et s'exécutent aussi vite que de nouveaux sont envoyés (normal puisqu'il y a une nouvelle tâche ajoutée par tâche exécutée), si bien que non seulement la pile d'aucun *thread* ne grossit, mais de plus, même les files d'attente du *thread pool* restent aussi stables.
Par cette astuce-là, il est toujours possible d'éviter le débordement de pile, par contre, la mémoire utilisée (dans le tas) n'est pas toujours forcément bornée (notamment quand la récursivité n'est pas terminale, cf. cours de programmation fonctionnelle).
Remarquons pour finir que le programme avec `CompletableFuture` a une exécution essentiellement séquentielle : chaque exécution de `incr` a besoin que la précédente soit terminée [a] pour démarrer. Il s'agit donc d'une utilisation de `CompletableFuture` détournée de l'usage initialement prévu.

a. Si on excepte le retour de méthode lui-même, ce qui n'est pas grand chose.

Exercice 2 : Tri fusion

Refaire l'exercice 1 du TP 11 en utilisant `CompletableFuture` à la place de `ForkJoinTask`.

Correction :

```
1   import java.util.Arrays;
2   import java.util.List;
3   import java.util.concurrent.CompletableFuture;
4   import static java.util.concurrent.CompletableFuture.completedFuture;
5   import static up.cpoo.concurrent.CompletableFutureTools.supplyAndFlatten;
6
7   public class TriFusionCF {
8
9       public static <E extends Comparable<? super E>> CompletableFuture<List<E>>
            trierAvecCompletableFuture(
10            List<E> list) {
11          // cas de base : liste à 0 ou 1 élément, on ne change rien
12          if (list.size() <= 1)
13              return completedFuture(list);
14          /*
15           * cas récursif : on encadre les appels récursifs dans thenComposeAsync (via la
                fonction
16           * auxilliaire supplyNotNow) afin que la décomposition en sous-tâches soit
                elle-même
17           * une tâche parallélisable
18           *
19           * (alternative, revenant au même: enlever supplyNotNow du premier niveau, mais
                placer
```

```
20          * les appels "récursifs" chacun dans son propre appel à supplyNotNow avant de
                combiner
21          * leurs résultats respectifs)
22          */
23         else
24            return supplyAndFlatten(() -> {
25               int pivot = Math.floorDiv(list.size(), 2);
26               // instrumentons, juste pour montrer ce qui se passe :
27               System.out.println("Liste de taille " + list.size() + " séparée dans le
                     thread "
28                     + Thread.currentThread().getName());
29               List<E> l1 = list.subList(0, pivot);
30               List<E> l2 = list.subList(pivot, list.size());
31               return
                     trierAvecCompletableFuture(l1).thenCombine(trierAvecCompletableFuture(l2),
                     (ll1, ll2) -> {
32                  // on instrumente aussi ici :
33                  System.out.println("Fusion de listes de tailles " + ll1.size() + " et
                        " + ll2.size()
34                           + " dans le thread " + Thread.currentThread().getName());
35                  return TriFusion.fusion(ll1, ll2);
36               });
37            });
38
39      }
40
41      public static void main(String[] args) {
42         List<Integer> l = Arrays.asList(8, 4, 7, 1, 2, 9, 4, 3, 5, 7);
43         System.out.println("liste triée : " + trierAvecCompletableFuture(l).join());
44      }
45
46   }
```

Ici non plus, on ne précise pas le *pool* utilisé. Ce sera donc, de même, le *pool* par défaut.

Exercice 3 : Factorisation d'entiers

Refaire l'exercice 2 du TP 11 en utilisant `CompletableFuture` à la place de `ForkJoinTask`.

Indication : il est possible d'utiliser la méthode `supplyAndFlatten`, mais il y a une écriture plus succincte commençant par un appel à `supplyAsync` (utilisant bien sûr, inévitablement, `thenCompose` ou `thenComposeAsync` un peu plus loin).

Correction :

```
1  import java.util.HashSet;
2  import java.util.Set;
3  import java.util.concurrent.CompletableFuture;
4  import static java.util.concurrent.CompletableFuture.*;
5
6  import java.util.Collections;
7
8  public class FactorisationCF {
9     private static Set<Long> fusion(Set<Long> h1, Set<Long> h2) {
10       HashSet<Long> ret = new HashSet<>();
11       ret.addAll(h1);
12       ret.addAll(h2);
13       return ret;
14    }
15
16    private static long largestDivisor(long x) {
17       long m = (long) Math.sqrt(x);
18       while (x % m != 0)
19          m--;
```

```
20          return m;
21      }
22
23      public static CompletableFuture<Set<Long>> factorise(long x) {
24          /* Remarque: les 2 appels à factorise sont dans une lambda. L'exécution de la
25           * tâche décrite par le corps de l'appel courant se soldera simplement par
26           * l'exécution des 2 appels récursifs, lesquels se contentent chacun d'alimenter
27           * le thread pool avec une nouvelle tâche asynchrone chacune. Aucun vrai calcul
28           * décrit dans les appels récursifs ne sera effectué avant que l'exécution
29           * des nouvelles tâches ne soit demandée.
30           * */
31
32          return supplyAsync(() -> largestDivisor(x))
33                .thenCompose(m -> (m == 1) ? completedFuture(Collections.singleton(x))
34                      : factorise(m).thenCombine(factorise(x / m),
                              FactorisationCF::fusion));
35      }
36
37      public static void main(String[] args) {
38          System.out.println(factorise(1730884069530000l).join());
39      }
40  }
```