TP n° 11 bis : La récursion avec CompletableFuture

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) :

```java
package up.cpoo.concurrent;
import java.util.concurrent.CompletableFuture;
import static java.util.concurrent.CompletableFuture.*;
import java.util.function.*;

public final class CompletableFutureTools {
   private CompletableFutureTools() {}

   /*
    * Méthode manquante dans l'API CompletableFuture, pourtant très utile pour ce genre
    * d'exercice. Elle prend en paramètre une fonction construisant un CompletableFuture
    * et retourne un CompletableFuture qui devra retourner la même valeur que celui
    * qui est retourné par la fonction...
    * ... mais jamais supplyAndFlatten ne demande l'exécution de la tâche associée.
    * Ainsi, elle ne sera exécutée que lorsqu'on appellera join.
    */
   public static <T> CompletableFuture<T> supplyAndFlatten(Supplier<CompletableFuture<T>> fn)
       {
      return supplyAsync(fn).thenCompose(x -> x);
      // (plus ou moins équivalent à :) return completedFuture(null).thenComposeAsync(x ->
          fn.get());
   }

}
```

puis on programme Fibonacci :

```java
import java.util.concurrent.CompletableFuture;
import static java.util.concurrent.CompletableFuture.*;
import static up.cpoo.concurrent.CompletableFutureTools.*;

public class FiboCF {

   public static CompletableFuture<Integer> calculFibo(int n) {
      if (n <= 1)
         return completedFuture(1);
      else {
         CompletableFuture<Integer> f1 = supplyAndFlatten(() -> calculFibo(n - 1));
         CompletableFuture<Integer> f2 = supplyAndFlatten(() -> calculFibo(n - 2));
         return f1.thenCombine(f2, (x, y) -> x + y);
      }
   }

   public static void main(String[] args) {
      System.out.println(calculFibo(30).join());
   }

}
```

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 ?

Exercice 2 : Tri fusion

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

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).