Learning objective: compose futures to avoid blocking as much as possible.

Exercise 1: future composition prediction

What is the expected output of the following future composition?

```java
System.out.println("Launching computation on " + Thread.currentThread());
CompletableFuture.supplyAsync(() -> {
    System.out.println("Producing 5 on " + Thread.currentThread());
    Thread.sleep(1000);
    return 5;
})
    .thenApply(res -> {
        System.out.println("Adding 2 on " + Thread.currentThread());
        return res + 2;
    })
    .thenCombine(CompletableFuture.supplyAsync(() -> {
        System.out.println("Producing 8 on " + Thread.currentThread());
        return 8;
    }), (res1, res2) -> {
        var res = res1 * res2;
        System.out.println("Multiplying " + res + " on " + Thread.currentThread());
        return res;
    });
System.out.println("Finish launch on " + Thread.currentThread());
```

Exercise 2: composition with sequence and join

Assume variables x, y and z are initialized, and functions f, g, h and F take some non-negligible amount of time to compute. Propose an asynchronous execution that avoids any blocking operation for the following computation: F(g(f(x)) + y) + h(z)).

Exercise 3: composition with condition

Similarly, propose an asynchronous execution that avoids any blocking operation for the following computation: first, we compute f(x); if the result is positive, we compute g(f(x)), otherwise, h(f(x)).

Exercise 4: composition with fork

Similarly, propose an asynchronous execution that avoids any blocking operation for the following computation while minimizing the amount of computation: g(f(x)) + h(f(x)).

Exercise 5: rewriting allOf

We want to implement the behavior of allOf from class CompletableFuture that takes multiple completable futures as arguments. From a collection of runnables, build a completable future that represents their completion by relying on runAfterBoth.

Exercise 6: rewriting anyOf

Same question with anyOf and runAfterEither.