CS2030S Recitation

Problem Set 9

brian

2026-04-17

National University of Singapore

Recap: CompletableFuture

Recap: CompletableFuture (another monad)

- Useful for describing an asynchronous workflow cleanly
- Composition of asynchronous steps
- Context is whether the result as been *computed* or not
- `completedFuture` is your `of`
- `thenApply` is your `map`
- `thenCompose` is your `flatMap`

Recap: CompletableFuture

- `supplyAsync` = start async work that returns a value
- `runAsync` = start async work that returns no value
- `completedFuture` = make an already-finished future containing x
- `thenApply` = transform the result of a successful stage
- `thenCombine` = wait for 2 independent successful stages, then combine them
- `thenCompose` = use 1 result to start the next async stage
- `thenRun` = after success, run some side effect with no input or output
- `thenAccept` = after success, run a consumer
- `handle` = always run whether success or failure, return value
- `exceptionally` = if failure recover with fallback value, else keep same value
- `allOf` = complete when all futures complete
- `anyOf` = complete when the first future completes and take that result

Q1: Background

```
1  class A {
2    private final int x;
3    public A(int x) { this.x = x; }
4    private void sleep() {
5      System.out.println(Thread.currentThread().getName() + " " + x);
6      try { Thread.sleep(1000); } catch { System.out.prinln("interrupted"); }
7    }
8    public A incr() {
9      sleep();
10     return new A(this.x + 1);
11   }
12   public A decr() {
13     sleep();
14     if (x <= 0) { throw new IllegalStateException(); }
15     return new A(this.x - 1);
16   }
17   public String toString() {
18     return String.valueOf(this.x);
19   }
20 }
```

Q1a

Consider the following method `foo`:

```
1 static A foo(A a) {
2   return a.incr().decr();
3 }
```

Write the asynchronous version of the method `foo` such that

i) It returns a `CompletableFuture<A>`

ii) The body of the method is executed asynchronously, with both `incr` and `decr` running in the same thread

Q1a

- Brian thinks very hard with students
- Brian urges students to think harder if that's the only way of doing it

Q1a

```
1 static CompletableFuture<A> fooAsync(A a) {
2   return CompletableFuture
3       .supplyAsync(() -> a.incr().decr());
4 }
```

```
1 static CompletableFuture<A> fooAsync(A a) {
2   return CompletableFuture
3       .supplyAsync(() -> a.incr())
4       .thenApply(a -> a.decr());
5 }
```

Ok, now do the same thing but now `incr` and `decr` **may** run on different threads

Q1a

- Brian thinks hard again with students

```
1 static CompletableFuture<A> fooAsync(A a) {
2   return CompletableFuture
3       .supplyAsync(() -> a.incr())
4       .thenApplyAsync(a -> a.decr());
5 }
```

Note that this is not faster than before since `decr` must still wait for `incr` to finish.

This is faster tho (faster than if you just use `thenApply`)

```
1 CompletableFuture<A> cf = CompletableFuture.supplyAsync(() -> a.incr());
2 CompletableFuture<A> cf1 = cf.thenApplyAsync(x -> x.decr());
3 CompletableFuture<A> cf2 = cf.thenApplyAsync(x -> x.incr());
4 cf1.join();
5 cf2.join();
```

Q1b

Suppose another method `bar`

```
1 static A bar(A a) {
2   return a.incr();
3 }
```

Write the asynchronous version of the method `bar` (`barAsync`) such that it returns `CompletableFuture<A>` and the body of the method is executed asynchronously.

Q1b

- Brian thinks less hard because this is easier

Q1b

```
1 static CompletableFuture<A> barAsync(A a) {
2   return CompletableFuture.supplyAsync(() -> a.incr());
3 }
```

What would be the asynchronous equivalent of the invocation `bar(foo(new A(0)))`?

Q1b

- Brian thinks hard with students

Q1b

```
1 CompletableFuture<A> b = fooAsync(new A(0)).thenCompose(a -> barAsync(a));
```

Q1c

Suppose another method `baz`

```
1 static A baz(A a) {
2   if (x == 0) {
3     return new A(0);
4   }
5   return a.incr().decr();
6 }
```

Write the asynchronous version of `baz` called `bazAsync` so that it returns a `CompletableFuture<A>` and the body is executed asynchronously

Q1c

- Thinking hard

Q1c

```
1 static A bazAsync(A a) {
2   if (x == 0) {
3     return CompletableFuture.completedFuture(new A(0));
4   }
5   return CompletableFuture.supplyAsync(() -> a.incr().decr());
6 }
```

Q1d

Suppose another method `qux`

```
1  static A qux(A a) {
2    try {
3      a = a.decr();
4      a = a.decr();
5    } catch (IllegalStateException e) {
6      System.out.println("error");
7      a = new A(10);
8    }
9    return a.incr();
10 }
```

Write the asynchronous version of `qux`

Q1d

- Everyone thinking hard tgt

Q1d

```
1  static A quxAsync(A a) {
2    return CompletableFuture
3        .supplyAsync(() -> a.decr())
4        .thenApply(a -> a.decr())
5        .exceptionally(e -> {
6          System.out.println("error");
7          return new A(10);
8        })
9        .thenApply(x -> x.incr());
10 }
```

Q1e

Consider the sniper below

```
1 fooAsync(new A(0));
2 barAsync(new A(0));
3 bazAsync(new A(0), 1);
4 System.out.println("done.");
```

Since everything is async, “done” is printed almost immediately. Revise the code so that it only prints when all 3 methods are done (but we must still allow the 3 methods to be async)

Q1e

Q1e

```
1 CompletableFuture<void> all = CompletableFuture.allOf(
2   fooAsync(new A(0)),
3   barAsync(new A(0)),
4   bazAsync(new A(0), 1)
5 )
6 .thenRun(() -> System.out.prinln("done."));
```

Q1f

```
1 CompletableFuture<A> cf1 = fooAsync(new A(0));
2 CompletableFuture<A> cf2 = fooAsync(new A(1));
3 cf1.join();
4 System.out.println(cf2.join().incr());
```

The code watied for both calls to finish before incrementing then printing the value inside `A`.

Revise the code so that it increments and prints the value inside `A` as soon as one of `cf1` or `cf2` finishes, while allowing them to run concurrently.

Q1f

Q1f

```
1 CompletableFuture.anyOf(fooAsync(new A(0)), fooAsync(new A(1)))
2     .thenApply(x -> ((A) x).incr()) // cast because anyOf returns CF<Object>
3     .thenAccept(System.out::println)
4     .join();
```

Alternative answer

```
1 fooAsync(new A(0))
2     .applyToEither(fooAsync(new A(1)), x -> x.incr())
3     .thenAccept(System.out::println)
4     .join();
```

Q2I

Assume `zero`, `addOne`, and `sum` all also call `slowTask` once in their body, and `slowTask` takes 1 second to execute.

How long does this take to execute

```
1 CompletableFuture<Integer> cf1 = CompletableFuture.supplyAsync(() -> zero());
2 CompletableFuture<Integer> cf2 = cf1.thenApplyAsync(x -> addOne(x));
3 CompletableFuture<Integer> cf3 = cf1.thenApplyAsync(x -> addOne(x));
4 cf2.thenCombineAsync(cf3, (x, y) -> sum(x, y)).join();
```

Q2I

Q2I

- `cf1` takes 1 second to execute
- `cf2` and `cf3` can execute in parallel (because of `thenApplyAsync`) so takes 1
- the sum depends on the result of `cf2` and `cf3` so it has to wait and also takes 1 second
- Answer: $1 + 1 + 1 = 3$ seconds

Assume `zero`, `addOne`, and `sum` all also call `slowTask` once in their body, and `slowTask` takes 1 second to execute.

How long does this take to execute

```
1 CompletableFuture<Integer> cf1 = CompletableFuture.supplyAsync(() -> zero());
2 CompletableFuture<Integer> cf2 = cf1.thenApply(x -> addOne(x));
3 CompletableFuture<Integer> cf3 = cf1.thenApply(x -> addOne(x));
4 cf2.thenCombineAsync(cf3, (x, y) -> sum(x, y)).join();
```

Q2II

Q2II

- `cf1` takes 1 second
- `cf2` and `cf3` both need to wait for the same thread that did `cf1`
  - so whichever goes first, the second one must wait before using the same thread, therefore 2 seconds
- the sum waits for both to end and takes 1 second
- Answer: $1 + 2 + 1 = 4$ seconds

Assume `zero`, `addOne`, and `sum` all also call `slowTask` once in their body, and `slowTask` takes 1 second to execute.

How long does this take to execute

```
1 CompletableFuture<Integer> cf1 = CompletableFuture.supplyAsync(() -> zero());
2 CompletableFuture<Integer> cf2 = cf1.thenApplyAsync(x -> addOne(x));
3 CompletableFuture<Integer> cf3 = cf1.thenApply(x -> addOne(x));
4 cf2.thenCombine(cf3, (x, y) -> sum(x, y)).join();
```

Q2III

Q2III

- Note that `CompletableFuture` keeps a stack of the lambda expressions to call next
  - so `cf3` is executed first before `cf2`
- `cf1` takes 1 second
- `cf3` waits for `cf1` then uses the same thread taking 1 second
- `cf2` then waits for `cf3` then uses the same thread taking 1 second
- the sum waits for both to end and takes 1 second
- Answer: $1 + 1 + 1 + 1 = 4$ seconds

Assume `zero`, `addOne`, and `sum` all also call `slowTask` once in their body, and `slowTask` takes 1 second to execute.

How long does this take to execute

```
1 CompletableFuture<Integer> cf1 = CompletableFuture.supplyAsync(() -> zero());
2 CompletableFuture<Integer> cf2 = cf1.thenApply(x -> addOne(x));
3 CompletableFuture<Integer> cf3 = cf1.thenApplyAsync(x -> addOne(x));
4 cf2.thenCombine(cf3, (x, y) -> sum(x, y)).join();
```

Q2IV

Q2IV

- The stack now calls async first then non-async
- `cf1` takes 1 second
- `cf3` is done in another thread after `cf1` finishes
- `cf2` is done in the same thread as `cf1`
- Together they contribute 1 second
- the sum waits for both to end and takes 1 second
- Answer: $1 + 1 + 1 = 3$ seconds

Goodbye 🥲

Thanks everyone

It’s been fun while it lasted ♥

All the best for exams

Download slides here: