CS2030 Programming Methodology
Semester 1 2023/2024

8 & 9 November 2023
Problem Set #10 Suggested Answers
Asynchronous Programming

1. Study the given class `A` below, which uses the methods `incr` and `decr` to imitate slow computations.

```
class A {
    private final int x;

    A() {
        this(0);
    }

    A(int x) {
        this.x = x;
    }

    void sleep() {
        System.out.println(Thread.currentThread().getName() + " " + x);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("interrupted");
        }
    }

    A incr() {
        sleep();
        return new A(this.x + 1);
    }

    A decr() {
        sleep();
        if (x < 0) {
            throw new IllegalStateException();
        }
        return new A(this.x - 1);
    }

    public String toString() {
        return "" + x;
    }
}
```

(a) Suppose we have a method

```
A foo(A a) {
    return a.incr().decr();
}
```

Convert the method `foo` above to a method that returns a `CompletableFuture` so that the body of the method is executed asynchronously. Try different variations by using

i. `supplyAsync` only;
ii. `supplyAsync` and `thenApply`;
iii. `supplyAsync` and `thenApplyAsync`

Demonstrate how you would retrieve the result of the computation.

See also: `thenRun`, `thenAccept`, `runAsync`

Answer:

i.
```
CompletableFuture<A> foo(A a) {
    return CompletableFuture.<A>supplyAsync(() -> a.incr().decr());
}
```

ii.
```
// same as foo above
CompletableFuture<A> foo(A a) {
    return CompletableFuture.<A>supplyAsync(() -> a.incr())
        .thenApply(x -> x.decr());
}
```

iii.
```
// decr() could be run in another thread
CompletableFuture<A> foo(A a) {
    return CompletableFuture.<A>supplyAsync(() -> a.incr())
        .thenApplyAsync(x -> x.decr());
}
```

To wait for the result,

```
CompletableFuture<A> a = foo(new A());
// do something else
a.join();
```

(b) Suppose now we have another method

```
A bar(A a) {
    return a.incr();
}
```

which we would like to invoke using `bar(foo(new A()))`. Convert the computation within `bar` to run asynchronously as well. `bar` should now return a `CompletableFuture`. In addition, show the equivalent of calling `bar(foo(new A()))` in an asynchronous fashion, using the method `thenCompose`.

See also: `thenCombine`

Without modifying `bar`*, we use* `thenApply`

```
CompletableFuture<A> b = foo(new A()).thenApply(x -> bar(x));
System.out.println(b.join());
```

By modifying `bar` *to return* `CompletableFuture<A>`*, we use* `thenCompose`

```
CompletableFuture<A> bar(A a) {
    return CompletableFuture.<A>supplyAsync(() -> a.incr());
}

CompletableFuture<A> b = foo(new A()).thenCompose(x -> bar(x));
System.out.println(b.join());
```

(c) Suppose now we have yet another method

```
A baz(A a, int x) {
    if (x == 0) {
        return new A(0);
    } else {
        return a.incr().decr();
    }
}
```

Convert the computation within `baz` in the `else` clause to run asynchronously. `baz` should now return a `CompletableFuture`. You may find the method `completedFuture` useful.

```
CompletableFuture<A> baz(A a, int x) {
    if (x == 0) {
        return CompletableFuture.<A>completedFuture(new A(0));
    } else {
        return CompletableFuture.<A>supplyAsync(() -> a.incr().decr());
    }
}

CompletableFuture<A> c = baz(new A(), 1);
System.out.println(c.join());
```

Note that `CompletableFuture` is a monad:

- `completedFuture` is equivalent to `of`,
- `thenCompose` is `flatMap`, and
- `thenApply` is `map`.

(d) Let's now call `foo`, `bar`, `baz` asynchronously. We would like to output the string `"done!"` when *all* three method calls complete. Show how you can use the `allOf()` method to achieve this behavior.

See also: `anyOf`, `runAfterBoth`, `runAfterEither`.

```
CompletableFuture<Void> all = CompletableFuture.<Void>allOf(
        foo(new A()),
        bar(new A()),
        baz(new A(), 1));
all.join();
System.out.println("done!");
```

(e) Calling `new A().decr().decr()` would cause an exception to be thrown, even when it is done asynchronously. Show how you would use the `handle()` method to gracefully handle exceptions thrown (such as printing them out) within a chain of `CompletableFuture` calls.
See also: `whenComplete` and `exceptionally`

```
CompletableFuture<A> exc = CompletableFuture
    .<A>supplyAsync(() -> new A().decr().decr())
    .handle((result, exception) -> {
        if (result == null) {
            System.out.println("ERROR: " + exception);
            return new A();
        } else {
            return result;
        }
    });

System.out.println(exc.join());
```

2. Modify the following sequences of code such that `f`, `g`, `h` and `i` are now invoked asynchronously, via `CompletableFuture`. Assume that `a` has been initialized as

```
A a = new A();
```

(a)
```
B b = f(a);
C c = g(b);
D d = h(c);
```

```
CompletableFuture<D> cf = CompletableFuture
    .<B>supplyAsync(() -> f(a))
    .thenApply(b -> g(b))
    .thenApply(c -> h(c));
D d = cf.join();
```

(b)
```
B b = f(a);
C c = g(b);
h(c); // no return value
```

```
CompletableFuture<Void> cf = CompletableFuture
    .<B>supplyAsync(() -> f(a))
    .thenApply(b -> g(b))
    .thenAccept(c -> h(c));
cf.join();
```

(c)
```
B b = f(a);
C c = g(b);
D d = h(b);
E e = i(c, d);
```

```
CompletableFuture<B> cfb = CompletableFuture
  .<B>supplyAsync(() -> f(a));
CompletableFuture<C> cfc = cfb
  .thenApply(b -> g(b));
CompletableFuture<D> cfd = cfb
  .thenApplyAsync(b -> h(b));
CompletableFuture<E> cfe = cfc
  .thenCombine(cfd, (c, d) -> i(c, d));
E e = cfe.join();
```