CS2030 Programming Methodology
Semester 2 2021/2022

6 & 8 April 2022
Problem Set #10

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);
    }

    private 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

```
static 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`

(b) Suppose now we have another method

```
static 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`

(c) Suppose now we have yet another method

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

(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`.

(e) Calling `new A().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`

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);
```

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

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

3. Run the following program and observe which worker is running which task.

```
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

class B {
    static class Task extends RecursiveTask<Integer> {
        int count;
        Task(int count) {
            this.count = count;
        }

        public Integer compute() {
            System.out.println(Thread.currentThread().getName()
                    + " " + this.count);
            if (this.count == 4) {
                return this.count;
            }
            Task t = new Task(this.count + 1);
            t.fork();
            return t.join();
        }
    }

    public static void main(String[] args) {
        ForkJoinPool.commonPool().invoke(new Task(0));
    }
}
```

Suppose the program is invoked with a maximum of three additional workers. What can you observe about the behaviour of a worker when the task that it is running blocks at the call to `join`?

4. Given below is the classic recursive method to obtain the n^{th} term of the Fibonacci sequence $0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$ *without memoization*

```
static int fib(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```

   (a) Parallelize the above implementation by transforming the above to a recursive task and inherit from `java.util.concurrent.RecursiveTask`

   (b) Explore different variants and combinations of `fork`, `join` and `compute` invocations.