CS2030S Recitation: Problem Set #10 — Wk 12, Sem 1, 2022/23

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

```java
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);
  }

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

   (a) Suppose we have a method

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

   Convert the method `foo` above to a method that returns `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`

Suggested Guide:

i. Possible code

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

ii. Possible code

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

iii. Possible code

```
// decr() could be run in another thread
static CompletableFuture<A> foo(A a) {
  return CompletableFuture.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

```
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`

Suggested Guide:

```
static CompletableFuture<A> bar(A a) {
  return CompletableFuture.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

```
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.

Suggested Guide:

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

CompletableFuture<A> c = baz1(new A(), 1);
System.out.println(c.join());
```

Now that `CompletableFuture` is a monad:

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

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

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

Suggested Guide:

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

(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 (*e.g., such as printing them out*) within a chain of `CompletableFuture` calls.

See also: `whenComplete`, `exceptionally`

Suggested Guide:

```
CompletableFuture<A> exc = CompletableFuture
    .supplyAsync(() -> new A().decr().decr())
    .handle((result, exception) -> {
      if (exception != 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

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

(a) Problem #A

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

Suggested Guide:

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

(b) Problem #B

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

Suggested Guide:

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

(c) Problem #C

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

Suggested Guide:

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

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

```java
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?

> **Suggested Guide:**
>
> You should observe that there exists a worker running Task i that also picks up Task j ($j > i$). Since Task i blocks at `join()`, this means that the worker does not sit idling waiting at `join()` but puts the blocking task aside and picks up (*i.e.*, *steals*) another task to execute.

4. Given below is the classic recursive method to obtain the nth term of the Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, . . . *without memoization*

```java
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`.

Suggested Guide:

```java
import java.util.concurrent.RecursiveTask;
class Fib extends RecursiveTask<Integer> {
  final int n;

  Fib(int n) {
    this.n = n;
  }

  @Override
  protected Integer compute() {
    if (n <= 1) {
      return n;
    }
    Fib f1 = new Fib(n - 1);
    Fib f2 = new Fib(n - 2);
    // try different variants here...
  }
}
```

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

Suggested Guide:

i. Variant #1

```java
f1.fork();
return f2.compute() + f1.join();
```

This is the same as lecture example.

ii. Variant #2

```java
f1.fork();
return f2.join() + f1.compute();
```

This works, but ***slow***, since in Java subexpressions are evaluated left to right (*i.e., for* `A + B`, `A` *is evaluated first before* `B`*, by the away this has nothing to do with associativity*). So `f1.join()` needs to wait for `f1.fork()` to complete before `f2.compute()` can be evaluated. Compare this with 4(b)i where `f2.compute()` proceeds while `f1.fork()` is running.

iii. Variant #3

```java
return f1.compute() + f2.compute();
```

This is ***sequentially recursive***. Not much different from 4(b)ii, but still slightly faster as there is no overhead involved in forking and joining. Everything is done by the main thread.

iv. Variant #4

```
f1.fork();
f2.fork();
return f2.join() + f1.join();
```

Apart from the first recursion, main thread delegates all other work to worker threads in the common pool.

v. Variant #5

```
f1.fork();
f2.fork();
return f1.join() + f2.join();
```

Looks the same as 4(b)iv, but 4(b)iv still preferred as it follows the convention of joins to be returned innermost first. Since a thread forks tasks to the front of its own double-ended queue, the last task forked should be the one that is joined when the thread becomes idle; tasks at the back of the dequeue are stolen by other idle worker threads.

vi. Other non-functional combinations

```
return f1.join() + f2.join();    // A
return f1.fork() + f2.fork();    // B
return f1.compute() + f2.fork(); // C
return f1.fork() + f2.join();    // D
```

A `fork()` must be followed by a `join()` to get the result back. None of the options that uses fork also join back the result. The only option that gives us the correct result is `A`. Note that it computes the Fibonacci number ***sequentially***.

You can use the version on the next page to test the performance.

```java
import java.util.concurrent.RecursiveTask;
import java.time.Instant;
import java.time.Duration;

class Fib extends RecursiveTask<Integer> {
  final int n;

  Fib(int n) {
    this.n = n;
  }

  private void waitOneSec() {
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) { }
  }

  @Override
  protected Integer compute() {
    System.out.println(Thread.currentThread().getName()
    + " : " + n);
    waitOneSec();

    if (n <= 1) {
      return n;
    }

    Fib f1 = new Fib(n - 1);
    Fib f2 = new Fib(n - 2);

    // try different variants here...
  }

  public static void main(String[] args) {
    Instant start = Instant.now();
    System.out.println(new Fib(5).compute());
    Instant end = Instant.now();

    System.out.println(Duration.between(start, end).
   toMillis());
  }
}
```