CS2030 Programming Methodology
Semester 1 2018/2019

26 October 2018
Tutorial 7 Suggested Answers
Lazy Evaluation and Parallel Streams

1. Implement a class `LazyInt` that encapsulates a lazily evaluated Integer value. A `LazyInt` is specified by a `Supplier`, such that when the value of the `LazyInt` is needed, the `Supplier` will be evaluated to yield the value. Otherwise, the evaluation is delayed as much as possible.

   `LazyInt` supports the following operations:

   - `map` returns a `LazyInt` consisting of the results of applying the given function to the value of this `LazyInt`.
   - `flatMap` returns a `LazyInt` consisting of the results of replacing the value of this `LazyInt` with the value of a mapped `LazyInt` produced by applying the provided mapping function to the value.
   - `get` returns the value of `LazyInt`.

   As an example, the expression below will return `200`.

   ```
   new LazyInt(() -> 10)
       .map(x -> x * x)
       .flatMap(x -> new LazyInt(() -> x * 2))
       .get()
   ```

   Given the skeleton class with `import` statements omitted for brevity, complete the method bodies of `map` and `flatMap`.

   ```
   class LazyInt {
       Supplier<Integer> supplier;

       LazyInt(Supplier<Integer> supplier) {
           this.supplier = supplier;
       }

       int get() {
           return supplier.get();
       }

       LazyInt map(Function<? super Integer, Integer> mapper) {
           // To complete
       }

       LazyInt flatMap(Function<? super Integer, LazyInt> mapper) {
           // To complete
       }
   }
   ```

```
import java.util.function.Supplier;
import java.util.function.Function;

class LazyInt {
    Supplier<Integer> supplier;

    LazyInt(Supplier<Integer> supplier) {
        this.supplier = supplier;
    }

    int get() {
        return supplier.get();
    }

    LazyInt map(Function<? super Integer, Integer> mapper) {
       return new LazyInt(() -> mapper.apply(get()));
    }

    LazyInt flatMap(Function<? super Integer, LazyInt> mapper) {
       return new LazyInt(() -> mapper.apply(get()).get());
    }

    public static void main(String[] args) {
        System.out.println(new LazyInt(() -> 10)
                .map(x -> x * x)
                .flatMap(x -> new LazyInt(() -> x * 2))
                .get());
    }
}
```

Some common mistakes for method `flatMap`

- `return mapper.apply(get())` is no longer lazy since it forces the evaluation of `mapper` and `supplier`.
- `return new LazyInt(mapper.apply(get()).supplier` which is also not lazy

2. The following program fragment was demonstrated during an earlier lecture to illustrate the effect of lazy evaluation in streams.

```
int sum = IntStream
    .rangeClosed(1, 10)
    .filter(x -> {
        System.out.println("filter: " + x);
        return x % 2 == 0;
    })
    .map(x -> {
        System.out.println("map: " + x);
        return 2 * x;
    })
    .sum();

System.out.println(sum);
```

(a) What is the output of running the above sequential stream?
This was done in a previous lecture. The output demonstrates lazy evaluation, one stream element at a time.

```
filter: 1
filter: 2
map: 2
filter: 3
filter: 4
map: 4
filter: 5
filter: 6
map: 6
filter: 7
filter: 8
map: 8
filter: 9
filter: 10
map: 10
60
```

(b) Parallelize the stream by including the appropriate operation and observe the output. What conclusions can you make?
Include the intermediate operation `.parallel()` anywhere after the data source and terminal. One possible output the demonstrate parallelization is

```
filter: 7
filter: 4
filter: 1
filter: 9
filter: 8
filter: 10
filter: 3
filter: 2
filter: 6
map: 8
filter: 5
map: 10
map: 4
map: 6
map: 2
60
```

Clearly, for any stream element `filter` must occur before `map`.

(c) Include the expression `Thread.currentThread().getName()` into the `println` statement of the lambda associated with `filter` and `map`, and observe the output again. What does the expression do?

It returns the name of the current thread associated with the stream operation.

(d) Add the following to the start of the program

```
System.out.println(ForkJoinPool.commonPool().getParallelism());
```

You can control the level of parallelism by setting the system property that affects every `ForkJoinPool` creation in your program. You can do this by including the following flag when you run the program

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=10
```

The above example sets the level of parallelism to `10`.

How does the level of parallelism affect the output when the program is run?

```
$ java -Djava.util.concurrent.ForkJoinPool.common.parallelism=4 SomeClass
4
filter: 9 ForkJoinPool.commonPool-worker-2
filter: 7 main
filter: 2 ForkJoinPool.commonPool-worker-4
filter: 3 ForkJoinPool.commonPool-worker-1
filter: 10 ForkJoinPool.commonPool-worker-2
filter: 5 ForkJoinPool.commonPool-worker-1
filter: 4 ForkJoinPool.commonPool-worker-1
filter: 6 ForkJoinPool.commonPool-worker-3
map: 2 ForkJoinPool.commonPool-worker-4
filter: 1 ForkJoinPool.commonPool-worker-4
map: 4 ForkJoinPool.commonPool-worker-1
map: 10 ForkJoinPool.commonPool-worker-2
filter: 8 ForkJoinPool.commonPool-worker-4
map: 6 ForkJoinPool.commonPool-worker-3
map: 8 ForkJoinPool.commonPool-worker-4
60

$ java -Djava.util.concurrent.ForkJoinPool.common.parallelism=0 SomeClass
1
filter: 3 ForkJoinPool.commonPool-worker-1
filter: 6 main
filter: 4 ForkJoinPool.commonPool-worker-1
map: 6 main
filter: 7 main
map: 4 ForkJoinPool.commonPool-worker-1
filter: 5 ForkJoinPool.commonPool-worker-1
filter: 1 ForkJoinPool.commonPool-worker-1
filter: 2 ForkJoinPool.commonPool-worker-1
map: 2 ForkJoinPool.commonPool-worker-1
filter: 9 main
filter: 8 ForkJoinPool.commonPool-worker-1
filter: 10 main
map: 8 ForkJoinPool.commonPool-worker-1
map: 10 main
60
```

3. The following program fragment counts the number of primes between $2,000,000$ and $3,000,000$.

```
long count = IntStream
        .range(2_000_000, 3_000_000)
        .filter(x -> isPrime(x))
        .count();

System.out.println(count);
```

(a) Include the `isPrime` method and run the program. Without having to parallelize the stream, try to obtain the output after a reasonable amount of waiting time.
Use the following definition of `isPrime`

```
static boolean isPrime(int n) {
    return IntStream
        .rangeClosed(2, (int) Math.sqrt(n))
        .parallel()
        .noneMatch(x -> n % x == 0);
}
```

(b) You can measure the time it takes to run the program by using the `Instant.now` and `Duration.between` methods. Find out how long your program takes to find all 67883 primes.

```
import java.time.Instant;
import java.time.Duration;
    ...
    public static void main(String[] args) {
        Instant start = Instant.now();
        long count = IntStream
            .range(2_000_000, 3_000_000)
            .filter(x -> isPrime(x))
            .count();
        Instant stop = Instant.now();

        System.out.println(count);
        System.out.println(Duration.between(start, stop).toMillis() + "ms");
    }
```

(c) Parallelize the stream and observe how long it takes.
Students should witness a significant speed-up.

(d) Notice that the `isPrime` method can be parallelized since determining primes are independent from one another. Find out how much time it takes to run the program. What can you conclude?
Parallelizing the `isPrime` method slows down the computation, possibly making it even slower than the sequential version. There is a overhead involved in parallelization and hence simple tasks should be done sequentially to avoid the overhead.

4. What is the outcome of the following stream pipeline?

```
Stream.of(1, 2, 3, 4)
    .reduce(0, (result, x) -> result * 2 + x);
```

What happens if we parallelize the stream? Explain.

Running the stream sequentially gives 26 since the pipeline evaluates

```
((((0 * 2 + 1) * 2 + 2) * 2 + 3) * 2 + 4)
```

A possible parallel run (with output from each `reduce` operation) gives 18.

```
0 * 2 + 4 = 4 : ForkJoinPool.commonPool-worker-370
0 * 2 + 3 = 3 : main
0 * 2 + 2 = 2 : ForkJoinPool.commonPool-worker-441
0 * 2 + 1 = 1 : ForkJoinPool.commonPool-worker-299
3 * 2 + 4 = 10 : main
1 * 2 + 2 = 4 : ForkJoinPool.commonPool-worker-299
4 * 2 + 10 = 18 : ForkJoinPool.commonPool-worker-299
18
```

Notice that reduction with the identity value `0` is happens for all four stream elements. This is followed by reducing `(1, 2)` to give `4`, and reducing `(3, 4)` to give `10`. Finally, reducing `(4, 10)` gives `18`.

The above stream cannot be parallelized because `2 * result + x` is not associative, i.e. the order of reduction matters.