CS2030 Programming Methodology

Semester 2 2018/2019

27 March – 29 March 2019

Tutorial 7 Suggested Answers

Java Streams and Functional Interfaces

1. Given the following class `A`.

   ```
   class A {
       int field;
       void method() {
           Function<Integer, Integer> func = x -> field + x;
       }
   }
   ```

   Model the execution of the program fragment:

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

   In particular, focus on the use of the *stack* and *heap* memory.

   - `A` is a class so the instance field `field` would be on the heap.
   - `func` is a local variable, so it would go on the stack.
   - `func` refers to the lambda expression, which is internally implemented as an anonymous class, so it refers to an object on the heap.
   - Finally, `x` is an argument to the lambda expression, so it is not stored anywhere.

2. Suppose we have the following lambda expression of type `Function<String, Integer>`:

   ```
   str -> str.indexOf(' ')
   ```

   (a) Write a `main` method to test the usage of the lambda expression above.

   ```
   public static void main(String[] args) {
       Function<String, Integer> f = str -> str.indexOf(' ');
       System.out.println(f.apply("hello world"));
   }
   ```

   (b) Java implements lambda expressions as anonymous classes. Write the equivalent anonymous class for the lambda expression above.

   ```
   Function<String, Integer> f = new Function<>() {
       public Integer apply(String str) {
           return str.indexOf(' ');
       }
   };
   System.out.println(f.apply("hello world"));
   ```

3. Complete the method `and` that takes in two `Predicate` objects `p1` and `p2` and returns a new `Predicate` object that evaluates to `true` if and only if both `p1` and `p2` evaluate to `true`.

```
Predicate<T> and(Predicate<T> p1, Predicate<T> p2) {
```

- Using lambda:

  ```
  return x -> p1.test(x) && p2.test(x);
  ```

- Using anonymous class:

  ```
  return new Predicate<T>() {
      public boolean test(T x) {
          return p1.test(x) && p2.test(x);
      }
  }
  ```

- The following is wrong:

  ```
  return p1.test(x) && p2.test(x);
  ```

  It *eagerly* evaluates the predicates and returns a boolean.

4. Write a method product that takes in two `List` objects `list1` and `list2`, and produce a `Stream` containing elements combining each element from `list1` with every element from `list2` using a `BiFunction`. This operation is similar to a Cartesian product.

```
public static <T,U,R> Stream<R> product(List<? extends T> list1,
        List<? extends U> list2,
        BiFunction<? super T, ? super U, R> func)
```

For example, the following program fragment

```
List<Integer> list1 = new ArrayList<>();
List<Integer> list2 = new ArrayList<>();

Collections.addAll(list1, 1, 2, 3, 4);
Collections.addAll(list2, 10, 20);

product(list1, list2, (str1, str2) -> str1 + str2)
    .forEach(System.out::println);
```

gives the output

```
11
21
12
```

```
22
13
23
14
24
```

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
import java.util.stream.Stream;
import java.util.function.BiFunction;

class Product {

    public static <T, U, R> Stream<R> product(
            List<? extends T> list1,
            List<? extends U> list2,
            BiFunction<? super T, ? super U, ? extends R> func) {

        return list1.stream()
            .flatMap(x ->
                    list2.stream()
                    .map(y -> func.apply(x,y)));
            }

    public static void main(String[] args) {
        List<Integer> list1 = new ArrayList<>();
        List<Integer> list2 = new ArrayList<>();

        Collections.addAll(list1, 1, 2, 3, 4);
        Collections.addAll(list2, 10, 20);

        product(list1, list2, (str1, str2) -> str1 + str2)
            .forEach(System.out::println);

    }
}
```

5. Write a method that returns the first n Fibonacci numbers as a `Stream<BigInteger>`. The `BigInteger` class is used to avoid overflow.

   For instance, the first 10 Fibonacci numbers are $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$.

   Hint: It would be useful to write a new `Pair` class that keeps two items around in the stream.

```
Stream<BigInteger> fibonacci(int n) {
    return Stream.iterate(
                new Pair<>(BigInteger.ZERO, BigInteger.ONE),
                pr -> new Pair<>(pr.second, pr.first.add(pr.second)))
            .map(pr -> pr.second).limit(n);
}
```