# CS2030 Programming Methodology

Semester 1 2019/2020

11 October 2019

Problem Set #6 Suggested Guidance

**Lambda and Streams**

1. Write a method `omega` with signature `IntStream omega(int n)` that takes in an `int n` and returns a `IntStream` containing the first $n$ omega numbers.

   The $i^{\text{th}}$ omega number is the number of distinct prime factors for the number $i$. The first 10 omega numbers are $0, 1, 1, 1, 1, 2, 1, 1, 1, 2$.

   The `isPrime` method is given below:

   ```
   boolean isPrime(int n) {
       return IntStream
           .range(2, n)
           .noneMatch(x -> n%x == 0);
   }
   ```

   *We use* `LongStream` *in order to work with large integer values.*

   ```
   import java.util.stream.IntStream;
   import java.util.stream.LongStream;

   boolean isPrime(int n) {
       return IntStream
           .range(2, n)
           .noneMatch(x -> n%x == 0);
   }

   IntStream primeFactorsOf(int x) {
       return factors(x)
           .filter(d -> isPrime(d));
   }

   IntStream factors(int x) {
       return IntStream
           .rangeClosed(2, x)
           .filter(d -> x % d == 0);
   }

   LongStream omega(int n) {
       return IntStream
           .range(1, n + 1)
           .mapToLong(x -> primeFactorsOf(x).count());
   }

   omega(10).forEach(System.out::println)
   ```

2. Write a method that returns the first $n$ Fibonacci numbers as a `Stream<Integer>`.

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

   *Hint*: Write an additional `Pair` class that keeps two items around in the stream

   *We use the* `BigInteger` *class to avoid overflow.*

```
class Pair<T> {
    T first;
    T second;

    Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }
}

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);
}
```

3. 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<String> list2 = new ArrayList<>();

Collections.addAll(list1, 1, 2, 3, 4);
Collections.addAll(list2, "A", "B");

product(list1, list2, (str1, str2) -> str1 + str2)
    .reduce("", (x, y) -> x + y + " ")
```

   gives the output

```
1A 1B 2A 2B 3A 3B 4A 4B
```

```java
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)));
}
```

4. You are given two functions $f(x) = 2 \times x$ and $g(x) = 2 + x$.

   (a) By creating an abstract class `Func` with a public abstract method `apply`, evaluate $f(10)$ and $g(10)$.

```java
abstract class Func {
    abstract int apply(int a);
}

Func f = new Func() {
    int apply(int x) {
        return 2 * x;
    }};

Func g = new Func() {
    int apply(int x) {
        return 2 + x;
    }};

f.apply(10);
g.apply(10);
```

*We cannot use a lambda here since* `Func` *is not a functional interface.*

```java
interface Func {
    int apply(int a);
}

Func f = x -> 2 * x;
Func g = x -> 2 + x;
f.apply(10);
g.apply(10);
```

(b) The composition of two functions is given by $f \circ g(x) = f(g(x))$. As an example, $f \circ g(10) = f(2+10) = (2+10) * 2 = 24$. Extend the abstract class in question 4a so as to support composition, i.e. `f.compose(g).apply(10)` will give 24.

```
abstract class Func {
    abstract int apply(int a);

    Func compose(Func g) {
        return new Func() {
            public int apply(int x) {
                return Func.this.apply(g.apply(x)); // <-- take note!
            }
        };
    }
}

Func f = new Func() {
    int apply(int x) {
        return 2 * x;
    }};

Func g = new Func() {
    int apply(int x) {
        return 2 + x;
    }};

f.compose(g).apply(10);
```

*What happens if we replace the statement* `return Func.this.apply(g.apply(x))` *with* `return this.apply(g.apply(x))` *instead? The* `apply` *method will recursive call itself! The* `this` *in* `Func.this` *is known as a "qualified this" and it refers not to it's own object, but the enclosing object. Here, the enclosing object's* `apply` *method is the one that returns* `2 * x`*.*

(c) Now re-implement question 4b as a functional interface `Func<T,R>`

```
@FunctionalInterface
interface Func<T,R> {
    R apply(T a);

    default <V> Func<V,R> compose(Func<? super V, ? extends T> g) {
        return x -> Func.this.apply(g.apply(x));
    }
}

Func<String, Integer> f = x -> x.length();
Func<Integer, String> g = x -> x + "";

g.compose(f).apply("this") +
g.compose(f).apply("is") +
g.compose(f).apply("fun!!!")
```

5. **Currying** is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument, $g(x, y) = h(x)(y)$. Using the context of lambdas in Java, the lambda expression `(x, y) -> x + y` can be translated to `x -> y -> x + y`.

   Show how the use of appropriate functional interfaces can achieve the curried function evaluation of two arguments.

   *Hint: If the lambda above looks intriguing, try replacing the lambda with anonymous inner classes instead to make sense of the scope of the variables* `x` *and* `y`*.*

```
jshell> Function<Integer,Function<Integer,Integer>> h = x -> y -> x + y
h ==> $Lambda$14/13326370@4b9e13df

jshell> h.apply(3).apply(4)
$3 ==> 7
```

   *Notice that we now have the ability to implement partial functions, such as this one that increments by one*

```
jshell> Function<Integer,Integer> inc = h.apply(1)
inc ==> $Lambda$15/1196765369@1d057a39

jshell> inc.apply(10)
$4 ==> 11
```

   *Try to implement a curried version of* $p(x, y, z) = x + y + z$