CS2030 Programming Methodology
Semester 2 2018/2019

20 March – 22 March 2019
Tutorial 6 Suggested Guidance
Java Primitive Streams

1. To approximate the value of π, one can sum up the first n terms of the following series:

$$\frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \ldots$$

You are given the following stream implementation,

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

double approxPI(int n) {
    int sign = 1;

    double ans = IntStream
        .rangeClosed(1, n)
        .mapToDouble(x -> {
                double term = 4.0 * sign / (2 * x - 1);
                sign = sign * -1;
                return term;
            })
        .sum();
    return ans;
}
```

Identify the error(s) and provide an alternative functioning stream implementation. Do not use any methods in `java.lang.Math`.

```
int sign(int n) {
    return IntStream
        .rangeClosed(1, n)
        .reduce(-1, (x, y) -> x * -1);
}

double approxPI(int n) {
    return IntStream
        .rangeClosed(1, n)
        .mapToDouble(x -> sign(x) * 4.0 / (2 * x - 1))
        .sum();
}

approxPI(100_000);
```

2. Using Java `Stream`, write a method `omega` with signature `LongStream omega(int n)` that takes in an `int n` and returns a `LongStream` containing the first n omega numbers.

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

```
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 primeFactors(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 -> primeFactors(x).count());
}

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

3. The sum of squares of a series of numbers can be implemented as follows:

```
int sumSq(int... list) {
    int sum = 0;
    for (int value : list) {
        sum += sq(value);
    }
    return sum;
}
```

```
int sq(int x) {
    return x * x;
}
```

On the other hand, to find the sum of absolute values of a given series will require implementing the following:

```
int sumAbs(int... list) {
    int sum = 0;
    for (int value : list) {
        sum += abs(value);
    }
    return sum;
}
```

```
int abs(int x) {
    return x > 0 ? x : -x;
}
```

Notice that `sumSq` and `sumAbs` methods are almost identical apart from the function application of each element of the list. By adhering to the *principle of abstraction*, demonstrate how we can replace them with a single method `sum` that takes in the list of elements as well as the function to be applied on each element.

Hint: Make use of `IntUnaryOperator`.

```
import java.util.function.IntUnaryOperator

int sum(IntUnaryOperator func, int... list) {
    int sum = 0;
    for (int value : list) {
        sum += func.applyAsInt(value);
    }
    return sum;
}

sum(x -> x * x, 1, -2, 3)

sum(x -> x > 0 ? x : -x, 1, -2, 3)
```

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

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

```
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.

```
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`.

5. By now, we are familiar with the `IntUnaryOperator` which takes one integer as argument and returns another integer result. As an example,

```
IntUnaryOperator f = x -> x + 1;
f.applyAsInt(3);
```

(a) Make use of `IntBinaryOperator` to evaluate $g(x, y) = x + y$.

```
IntBinaryOperator g = (x, y) -> x + y;
g.applyAsInt(3, 4);
```

(b) **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 `IntFuction` and `IntUnaryOperator` functional interfaces can achieve the curried function evaluation of two arguments.

```
IntFunction<IntUnaryOperator> h = x -> y -> x + y;
h.apply(3).applyAsInt(4);
```

If the lambda above looks intriguing, one can replace the lambda with anonymous inner classes instead to make sense of the scope of the variables `x` and `y`.

```
IntFunction<IntUnaryOperator> h = new IntFunction<IntUnaryOperator>() {
    public IntUnaryOperator apply(int x) {
        return new IntUnaryOperator() {
            public int applyAsInt(int y) {
                return x + y;
            }
        };
    }
}
```

(c) Implement a curried version of $p(x, y, z) = x + y + z$

```
IntFunction<IntFunction<IntUnaryOperator>> p = x -> y -> z -> x + y + z;
p.apply(3).apply(4).applyAsInt(4);
```