

# CS2030 AY18/19

## SEM 2

WEEK 9 | 22 MARCH 19  
TA GAN CHIN YAO

# DISCLAIMER

Slides are made by me, **unofficial, optional**  
Available to download at **[bit.ly/cs2030\\_gan](https://bit.ly/cs2030_gan)**  
Slides (if any) will be uploaded on  
Friday weekly

**Q1.**

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

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

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;
}
```

Essentially:

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

    double ans = IntStream
        .rangeClosed(1, n)
        .mapToDouble(
            new IntToDoubleFunction() {
                @Override
                public double applyAsDouble(int 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`.

Q1.

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

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

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;
}
```

Compile error.

- `int sign` is local variable, hence it is variable captured inside the anonymous inner class.
- `Sign` must be `final` or `effectively final`.
- You can access `sign`, i.e. `int k = sign;` but you cannot change `sign` (since `sign` is `effectively final`)

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

Q1.

One quick way to solve

```
class A {  
    int sign = 1; // move sign to instance variable  
    double approxPI(int n) {  
        // make sure you reset the value of sign  
        // since sign will be mutated  
        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;  
    }  
}
```

Another way provided by the prof:

```
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);
```

## Q2.

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.

Note: 1 is not prime numbers

Prime numbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 ...

Prime factors of:

- 1: nothing, as prime is  $> 1$
- 2: 2
- 3: 3
- 4:  $2 * 2$
- 5: 5
- 6:  $3 * 2$
- 7: 7
- 8:  $2 * 2 * 2$
- 9:  $3 * 3$
- 10:  $5 * 2$
- Omega of 10 is 2 (5, 2)
- Omega of 7 is 1 (7)

Q2.

```
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)
```

This is just one way provided by the prof.

There are many other ways.

e.g. this is also correct

```
LongStream omega(int n) {  
    return LongStream  
        .range(1, n + 1)  
        .map(x -> primeFactors((int) x).count());  
}
```



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`.*

# Ans:

```
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)
```

**IntUnaryOperator API: interface with SAM**

**Note:**

We use IntUnaryOperator becomes it is natural, since we need to take in an int and returns an int, and IntUnaryOperator do just that.

We can also use Function<Integer, Integer> or UnaryOperator<Integer> if we wish, and it will still work. You have to change func.applyAsInt to corresponding methods in Function or UnaryOperator

Modifier and Type	Method and Description
int	<b>applyAsInt</b> (int operand) Applies this operator to the given operand.

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  $g(10)$ .

Ans:

```
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);
```

This is creating an object of an anonymous inner class that extends Func

Note: We cannot use lambda here since Func is abstract class, not interface

If Func was an interface:

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

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

## Q4. b.

OK: Works

```
abstract class Func {
    abstract public int apply(int x);

    public Func compose(Func g) {
        Func self = this;
        return new Func() {
            public int apply(int x) {
                return self.apply(g.apply(x));
            }
        };
    }
}
```

Not ok: Infinite recursion

```
abstract class Func {
    abstract public int apply(int x);
    Func self = this;

    public Func compose(Func g) {
        return new Func() {
            public int apply(int x) {
                return self.apply(g.apply(x));
            }
        };
    }
}
```

## Q5. a.

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  $(x, y) \rightarrow x + y$  can be translated to  $x \rightarrow y \rightarrow x + y$ .

Show how the use of `IntFunction` 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);`

### API:

@FunctionalInterface

public interface `IntFunction<R>`

`R`

`apply(int value)`

Applies this function to the given argument.

@FunctionalInterface

public interface `IntUnaryOperator`

`int`

`applyAsInt(int operand)`

Applies this operator to the given operand.

## Q5. b.

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);`

API:

```
@FunctionalInterface
public interface IntFunction<R>
R
```

```
apply(int value)
Applies this function to the given argument.
```

```
@FunctionalInterface
public interface IntUnaryOperator
int
```

```
applyAsInt(int operand)
Applies this operator to the given operand.
```



## Extra practice

Correct:

a. `Function<Function<Integer, String>, String> obj = x -> "hi";`

Wrong:

a. `Function<Function<Integer, String>, String> obj  
= x -> "hi" -> "hi2";`

Correct:

b. `Function<Integer, Function<Function<Integer, String>, String>> obj  
= x -> y -> "hi";`

Wrong:

b. `Function<Integer, Function<Function<Integer, String>, String>> obj  
= x -> y -> z -> "hi" -> "hi2";`

Correct:

c. `Function<IntUnaryOperator, Function<IntFunction<String>, IntUnaryOperator>>  
obj = x -> y -> z -> 3;`

# SUMMARY CONCEPTS

- `lambda`
- Variable capture
- How to use `IntUnaryOperator`
- How to use `IntFunction`
- `IntStream`, `LongStream`, `Stream`
- Currying

**QUESTIONS?**