

# CS2030 Lab 6

## Integer Streams

Qi Ji (qiji@u.nus.edu)

19th March 2019

# Recap on IntStream

```
int sum = IntStream
    .rangeClosed(1,10)           // factory
    .filter(x -> x % 2 == 0)     // intermediates
    .map(x -> x * 3)
    .sum();                      // terminal
```

# Recap on IntStream

- Java IntStreams can only be used once
  - After invoking an intermediate or terminal operation

# Example

```
IntStream stream = IntStream  
    .rangeClosed(1, 10)  
    .filter(x -> x % 2 == 0);
```

```
int sum1 = stream.map(x -> 3 * x).sum(); // ok  
int sum2 = stream.map(x -> x * x).sum();  
// this throws IllegalStateException
```

# Lazy Evaluation

```
int sum = IntStream
    .iterate(0, x -> x + 1) // nothing yet
    .filter(x -> x % 3 == 0) // still nothing
    .map(x -> x * 2) // nothing yet.....
    .take(10) // STILL nothing
    .sum(); // everything gets forced here
```

# Java names for functional interfaces

---

|                                      |                                |
|--------------------------------------|--------------------------------|
| <code>Function&lt;T,U&gt;</code>     | $T \rightarrow U$              |
| <code>UnaryOperator&lt;U&gt;</code>  | $U \rightarrow U$              |
| <code>BinaryOperator&lt;T&gt;</code> | $(T, T) \rightarrow T$         |
| <code>Predicate&lt;T&gt;</code>      | $T \rightarrow \text{boolean}$ |
| <code>Consumer&lt;T&gt;</code>       | $T \rightarrow \text{void}$    |
| <code>Supplier&lt;T&gt;</code>       | $() \rightarrow T$             |

---

# Common initial operations

```
IntStream.of(int... stuff)
// can pass int[] here ^
IntStream.range(int startIncl, int endExcl)
IntStream.iterate(int seed, IntUnaryOperator f)
// gives [a, f(a), f(f(a)), ... ]
IntStream.iterate(int seed,
    IntPredicate hasNext,
    IntUnaryOperator f)
// same as iterate(x, f).takeWhile(hasNext)
IntStream.generate(IntSupplier s)
```

# Common intermediate operations

```
.map(IntUnaryOperator mapper)  
    //  $x \rightarrow x + 5$ 
```

```
.filter(IntPredicate predicate)  
    //  $x \rightarrow x \% 2 == 1$ 
```

```
.skip(long n)
```

```
.limit(long maxSize)
```



# Common terminal operations

```
.count()      .sum()      .average()
.reduce(IntBinaryOperator op)
    // (x, y) -> x * y
.reduce(int identity, IntBinaryOperator op)
.anyMatch(IntPredicate predicate)
    // x -> x > 0
.allMatch(IntPredicate predicate)
.noneMatch(IntPredicate predicate)
.forEach(IntConsumer action)
    // x -> System.out.print(x)
```

# IntStream vs Stream<T>

- Java generics can only hold reference types (e.g. `Integer`)
- `IntStream` stores the primitive type `int`
- Most things in previous slides hold for `Stream<T>` also
- `IntStream` is basically the side-effect of badly-implemented generics in Java

# Level 1 (Perfect numbers)

*Given an integer  $n$ , determine whether it is a perfect number.*

**Definition.** A perfect number  $n$  is a positive integer that is equal to the sum of its proper divisors. A proper divisor of  $n$  is a factor of  $n$  in  $\{1, 2, \dots, n - 1\}$ .

The smallest perfect number is 6.

# Things to note for today's lab

- Correctness only
- Use streams to solve all levels
  - Can use `IntStream` or the generic `Stream<T>`.
- Levels are graded independently
  - submit all levels

# More notes

- Level 4 (variance)
  - `IntStream` has a `mapToDouble(...)` which returns a `DoubleStream`
- Levels 3 and 4 (and maybe others) – You might have to write a helper method to convert input into an `int[]`
  - don't have to use streams for the helper (although it's possible)
  - you also can change the method signature to accept `List<Integer>` instead

# More practice questions for those who find this lab too easy

(do the lab questions first)

Given positive integer  $n$

- find the  $n$ -th prime
- print first  $n$  integers containing the digit 1
- express  $n$  as prime factors
- find the  $n$ -th Fibonacci number
- check if  $n$  is a Fibonacci number