Welcome to CS2030S Lab 7!

22 October 2021 [16A]

Please login to your pe node once it's 4pm.

PA2 Alternative Arrangements

- PA2 falls on the Friday of Week 12 (**05/11/2021**)
- Please let us know if you have to take tests immediately **before** or **after** your CS2030S lab (e.g. if you have lab from 1000-1200 and your test is at 1400 then it's not counted)
- We will book venues for you to take your test **if you need them** (PA2 will end around 10-15 minutes before the end of your lab slot so you can go elsewhere to take the test if you prefer)

Learning Objectives

Idea Behind this Week's Stream Exercises:

1. Understand the `IntStream` and `Stream` APIs
2. Apply what we have learnt with regards to stream operations

Rules:

1. No Recursion
2. No For Loops, No While Loops

What you are allowed:

1. Helper Classes, If Necessary
2. Helper Methods

With a stream, you specify what you want to have done, not how to do it. You leave scheduling of operations to the implementation.

The stream library can optimize the computation, e.g. using multiple threads for computing sums and counts and combining the results.

Instead of using imperative programming,

```
for (int i = 0; i < n; i++) {
    System.out.println(i);
}
```

we want to use declarative programming.

```
IntStream.range(0, n).
    forEach(System.out::println)
```

```
static IntStream range(int startInclusive, int endExclusive)
static IntStream rangeClosed(int startInclusive, int endInclusive)
```

Generate elements from `m` to `n-1`

```
jshell> IntStream.range(1,5).toArray()
$1 ==> int[4] { 1, 2, 3, 4 }
```

Generate elements from `m` to `n`

```
jshell> IntStream.rangeClosed(1,5).toArray()
$2 ==> int[5] { 1, 2, 3, 4, 5 }
```

```
static <T> Stream<T> of(T... values)
```

Create a stream whose elements are the given value(s).

This method has a *varargs* parameter, so you can construct a stream from any number of arguments.

```
jshell> Stream.of("gently", "down", "the", "stream").
    forEach(e → System.out.print(e + " "))
gently down the stream
```

```
Stream<T> limit(long maxSize)
```

Limit the number of elements to stream to `maxSize`

```
jshell> Stream.of("gently", "down", "the", "stream").
    limit(2).
    forEach(e → System.out.print(e + " "))
gently down
```

```
Stream<T> distinct()
```

Returns a stream with only distinct elements

```
jshell> Stream.of("merrily", "merrily", "merrily", "gently").
    distinct().toArray()
$5 ==> Object[2] { "merrily", "gently" }
```

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

Return a stream containing the results of applying `mapper` to the elements of this stream.

```
jshell> Stream.of("gently", "down", "the", "stream").
    map(String::toUpperCase).toArray()
$6 ⟹ Object[4] { "GENTLY", "DOWN", "THE", "STREAM" }
```

```
<R> Stream<R> flatMap(
    Function<? super T,? extends Stream<? extends R>> mapper)
```

🤯 Returns a stream obtained by concatenating the results of applying mapper to the elements of this stream. (Note that each result is a stream.)

```
jshell> IntStream.range(1, 3).
    flatMap(x → IntStream.range(1, 4)).toArray()
$7 ==> int[6] { 1, 2, 3, 1, 2, 3 }

// Intermediate Operations
// (1, 2).flatMap(x → (1, 2, 3))
// flatMap((1, 2, 3), (1, 2, 3))
// (1, 2, 3, 1, 2, 3)
```

Reductions (Terminal Operations)

Reduce the stream into a nonstream value.

Simple

```
long count()

Optional<T> max(
    Comparator<? super T> comparator)

Optional<T> findFirst()

boolean anyMatch(
    Predicate<? super T> predicate)
```

General 🤯

```
Optional<T> reduce(
    BinaryOperator<T> accumulator)

T reduce(
    T identity,
    BinaryOperator<T> accumulator)

<U> U reduce(
    U identity,
    BiFunction<U,? super T,U> accumulator,
    BinaryOperator<U> combiner)
```

Reduction Operations

```
jshell> IntStream.of(1,2,3,4).reduce((x, y) → x + y)
$8 ⟹ OptionalInt[10]
```

adds a **partial result** x with the **next value** y to yield a **new partial result**.

```
jshell> IntStream.of(1,2,3,4).reduce(5, (x, y) → x + y)
$9 ⟹ 15
```

Suppose you have a stream of objects and want to form the sum of some property, such as ***lengths in a stream of strings***.

Can you still use these 2 methods?

The `BinaryOperator<T>` is by definition `(T, T) → T` but now we have two types: The stream elements are `String`, and the accumulated result is an integer.

```
int result = 0;
for (String e: stream)
    result = accumulator.apply(result, e);
return result;
```

Reductions (Terminal Operations)

Reduce the stream into a nonstream value.

Simple

```
long count()

Optional<T> max(
    Comparator<? super T> comparator)

Optional<T> findFirst()

boolean anyMatch(
    Predicate<? super T> predicate)
```

General 🤯

```
Optional<T> reduce(
    BinaryOperator<T> accumulator)

T reduce(
    T identity,
    BinaryOperator<T> accumulator)

<U> U reduce(
    U identity,
    BiFunction<U,? super T,U> accumulator,
    BinaryOperator<U> combiner)
```

```
jshell> Stream.of("gently", "down", "the", "stream").
    reduce(0,
           (total, word) → total + word.length(),
           (total1, total2) → total1 + total2)
$10 ⟹ 19
```

`reduce` is not constrained to execute sequentially, i.e. can be applied to parallel streams.
If using parallel streams, the operation in `accumulator` must be *associative*. Otherwise, the results would be inconsistent.