
CS2030 Lecture 8

Towards Declarative Programming Using Streams

Henry Chia (hchia@comp.nus.edu.sg)

Semester 2 2019 / 2020

Lecture Outline

- Declarative versus imperative programming
- Internal versus external iteration
- Stream concepts using `IntStream` and `Stream<T>`
 - Stream elements and pipelines
 - Intermediate and terminal operations
 - Stateless vs stateful operations
 - Lazy and eager evaluations
 - `map` and `flatMap`
 - Reduction
 - Infinite streams

External Iteration

- Another example of handling context — that of looping
- An imperative loop specifies **how** to loop and sum

```
int sum = 0;  
for (int x = 1; x <= 10; x++) {  
    sum += x;  
}
```

- Realize the variables `i` and `sum` *mutates* at each iteration
- Errors could be introduced when
 - `sum` is initialized wrongly before the loop
 - looping variable `x` is initialized wrongly
 - loop condition is wrong
 - increment of `x` is wrong
 - aggregation of `sum` is wrong

Internal Iteration

- A *declarative* approach that specifies **what** to do

```
int sum = IntStream  
    .rangeClosed(1, 10)  
    .sum();
```

- sum is assigned with the result of a **stream pipeline**
- Literal meaning “for the range 1 through 10, sum them”
- A **stream** is a sequence of elements on which tasks are performed; the stream pipeline moves the stream’s elements through a sequence of tasks
- No need to specify how to iterate through elements or use any *mutable* variables — no variable state, no problem 😊
- IntStream handles all the iteration details

Streams and Pipelines

- A stream pipeline starts with a **data source**
- Static method `IntStream.rangeClosed(1, 10)` creates an `IntStream` containing the ordered sequence `1, 2, ..., 9, 10`
 - `range(1, 10)` produces the ordered sequence `1, 2, ..., 8, 9`
- Instance method `sum` is the processing step, or **reduction**
 - it reduces the stream of values into a single value
 - Other reductions include `count`, `min`, `max`, `average`

```
long count = IntStream
    .rangeClosed(1, 10)
    .count();
```

```
OptionalInt max = IntStream
    .rangeClosed(1, 10)
    .max();
System.out.println(max.getAsInt())
```

- Reduction is a **terminal operation** that produce a result

Intermediate Operations

- Most stream pipelines contain **intermediate operations** that specify tasks to perform on a stream's elements
- An intermediate operation results in a new stream comprising processing steps specified up to that point in the pipeline

- Example, map:

```
int sum = IntStream.rangeClosed(1, 10)
    .map(x -> x * 2)
    .sum();
```

- Example, flatMap

```
int sum = IntStream.rangeClosed(1, 5)
    .flatMap(x -> IntStream.rangeClosed(1, x))
    .sum();
```

- Notice the “flattening” effect in flatMap

Intermediate Operations

- ❑ Intermediate operations use **lazy evaluation**
- ❑ Does not perform any operations on stream's elements until a terminal operation is called. Using filtering as an example,
 - Select elements that match a condition, or **predicate**

```
int sum = IntStream.rangeClosed(1, 10)
    .filter(x -> x % 2 == 0)
    .map(x -> 2 * x)
    .sum();
```
 - `filter` receives a lambda that takes in one parameter and returns a **boolean** result
 - if true, the element is included in the resulting stream
- ❑ Terminal operation use **eager evaluation**, i.e. perform the requested operation when they are called

Movement of Stream Elements

- The following illustrates the movement of stream elements:

```
int sum = IntStream
    .rangeClosed(1, 10)
    .filter(
        x -> {
            System.out.println("filter: " + x);
            return x % 2 == 0;
        })
    .map(
        x -> {
            System.out.println("map: " + x);
            return 2 * x;
        })
    .sum();
System.out.println(sum);
```

```
filter: 1
filter: 2
map: 2
filter: 3
filter: 4
map: 4
filter: 5
filter: 6
map: 6
filter: 7
filter: 8
map: 8
filter: 9
filter: 10
map: 10
sum is 60
```

- Stream elements within a stream can only be consumed once
- Cannot iterate through a stream multiple times

The last stream object contains all processing steps to perform on each stream element

Method References

- A lambda that simply calls another method can be replaced with just that method's name, e.g. in the `forEach` terminal

```
IntStream
    .rangeClosed(1, 10)
    .forEach(x -> System.out.println(x));
```

- Using method reference

```
IntStream
    .rangeClosed(1, 10)
    .forEach(System.out::println);
```

- Types of method references:

Syntax: `Object :: methodName`

- reference to a static method
- reference to an instance method
- reference to a constructor

Stateless vs Stateful Operations

- Intermediate stream operations like `filter` and `map` are **stateless**, i.e. processing one stream element does not depend on other stream elements
- There are, however, **stateful** intermediate operations that depend on the current state
- E.g. stateful operations: `sorted`, `limit`, `distinct`, etc.

```
IntStream
```

```
.of(7, 9, 5, 2, 8, 4, 1, 6, 10, 3)  
.sorted()  
.forEach(System.out::println);
```

```
IntStream
```

```
.of(1, 1, 1, 0, 0, 0, 1, 0, 0, 1)  
.distinct()  
.forEach(System.out::println);
```

Boolean Terminal Operations

- Useful terminal operations that return a **boolean** result
 - **noneMatch** returns **true** if none of the elements pass the given predicate
 - **allMatch** returns **true** if every element passes the given predicate
 - **anyMatch** returns **true** if at least one element passes the given predicate
- Primality checking: external vs internal iteration

```
static boolean isPrime(int n) {  
    for (x = 2; x < n; x++) {  
        if (n % x == 0) {  
            return false;  
        }  
    }  
    return true;  
}
```

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

User-defined Reductions

- Using IntStream's reduce method
- Terminal operations are specific implementations of reduce
- For example, using reduce in place of sum

```
IntStream  
    .of(values)  
    .reduce(0, (x, y) -> x + y)
```

- First argument to reduce is the operation's identity value
- Second argument is the lambda that receives two `int` values, adds them and returns the result; in the above
 - ▷ First calculation uses identity value 0 as left operand
 - ▷ Subsequent calculations uses the result of the prior calculation as the left operand
 - ▷ If stream is empty, the identity value is returned

Infinite Stream

- Lazy evaluation allows us to work with infinite streams that represent an infinite number of elements
- Since streams are lazy until a terminal operation is performed, intermediate operations can be used to restrict the total number of elements in the stream
- `iterate` generates an ordered sequence starting using the first argument as a seed value
- Example, to find the first 500 primes

```
IntStream
```

```
.iterate(2, x -> x + 1)  
.filter(x -> isPrime(x))  
.limit(500)  
.forEach(System.out::println);
```

From IntStream to **Stream<T>**

- From IntStream to Stream

```
Stream<Circle> circles = IntStream
    .rangeClosed(1, 3)
    .mapToObj(Circle::new); // c -> new Circle(c)

circles.forEach(System.out::println);
```

- There are equivalent intermediate operations in Stream<T>
- Functional interfaces that stream operations take in:
 - Predicate<T> used in filter(Predicate <? **super** T> predicate)
 - Consumer<T> used in forEach(Consumer <? **super** T> consumer)
 - Supplier<T> used in generate(Supplier<? **extends** T> s)
 - Function<T,R> used in
map(Function<? **super** T, ? **extends** R> mapper)
 - and more...

Stream<T>'s reduce Operator

- Stream<T>'s two-argument reduce method is declared as:
`T reduce(T identity, BinaryOperator<T> accumulator)`

```
Circle[] circles = {new Circle(1), new Circle(2), new Circle(3)}  
Circle newCircle = Stream  
    .of(circles)  
    .reduce(new Circle(0),  
        (c1, c2) -> new Circle(c1.getRadius() + c2.getRadius()))
```
- Overloaded single-argument reduce method:
`Optional<T> reduce(BinaryOperator<T> accumulator)`

```
Stream.of(circles)  
    .reduce((c1, c2) -> new Circle(c1.getRadius()  
        + c2.getRadius()))  
    .ifPresent(System.out::println);
```
- reduce returns an Optional<T> which may have a value, or is empty (e.g. reduction on an empty stream)

Lecture Summary

- ❑ Understand the use of functions as cross-barrier state manipulator, as well as facilitating the abstraction principle
- ❑ Appreciate the declarative style of programming
- ❑ Understand how Java Functional Interface with a single abstract method can be used in stream operations
- ❑ Appreciate how lazy evaluations are used for intermediate operations, eager evaluation for terminal operations
- ❑ Know how to define reductions for use in a stream pipeline
- ❑ Appreciate how lazy evaluations support infinite streams