CS2030 Lecture 8

Java Streams and Functional Interfaces

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

Semester 1 2018 / 2019

Lecture Outline

- `IntStream` versus `Stream`
- Stateless versus stateful operations
- From `Stream` to `Collection`
- Single abstract method (SAM) and `FunctionalInterface`
  - `Comparator`
  - `Predicate`
  - `Consumer`
  - `Supplier`
  - `Function`
  - `BinaryOperator` / `Bifunction`
- Function composition
- Currying

Mapping Primitive Stream and `Stream`

- From `IntStream` to `Stream`

```
IntStream
    .rangeClosed(1, 3)
    .mapToObj(Circle::new) // c -> new Circle(c)
    .forEach(System.out::println);
```

- From `Stream` to `DoubleStream`

```
double maxArea = Stream
    .of(new Circle(5), new Circle(2))
    .mapToDouble(Circle::getArea) // c -> c.getArea()
    .max()
    .getAsDouble();

System.out.println(maxArea);
```


Stateless vs Stateful Operations

- Thus far, intermediate stream operations like `filter` and `map` are stateless, i.e. processing one stream element does not depend on other stream elements
- There are stateful intermediate operations that depend on the current state
- Example of stateful operations: `sorted` and `distinct`

```
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);
```

Stateless vs Stateful Operations

- Stream processing in stateful operations, e.g. `sorted`

```java
IntStream.of(7, 9, 5, 2, 8, 4)
    .map(x -> {
        System.out.println("Before: " + x);
        return x;
    })
    .sorted()
    .map(x -> {
        System.out.println("After: " + x);
        return x;
    })
    .forEach(System.out::println);
```

```
Before: 7
Before: 9
Before: 5
Before: 2
Before: 8
Before: 4
After: 2
2
After: 4
4
After: 5
5
After: 7
7
After: 8
8
After: 9
9
```


Stateless vs Stateful Operations

- Stream pipeline results are best maintained stateless.
- Example, testing primality of n

```java
boolean prime = IntStream
    .range(2, n)
    .filter(x -> n % x == 0)
    .count() == 0;
```

- What happens to the following?

```java
boolean prime = true;
IntStream
    .range(2, n)
    .filter(x -> n % x == 0)
    .forEach(x -> prime = false);
```

- Local variables referenced from a lambda expression must be **final** or **effectively final**

Stateless vs Stateful Operations

- Stream pipeline results may be nondeterministic or incorrect if the behavioral parameters to the stream operations are stateful
- A stateful lambda is one whose result depends on any state which might change during the execution of the stream pipeline

```java
MyBoolean prime = new MyBoolean(true);
IntStream
    .range(2, n)
    .filter(x -> n % x == 0)
    .forEach(x -> prime.flag = false);
```

- Although the above does not generate a compilation error, it is nonetheless attempting to access mutable state

From Stream to Collection and *vice-versa*

- `Collection`'s `stream()` produces a stream from a collection
- `Stream`'s `collect()` is a terminal operation that collects stream elements into say, a `List`

```java
Circle circles[] = {
    new Circle(1), new Circle(2), new Circle(3)};

List<Circle> listOfCircles = Arrays.asList(circles);
listOfCircles
    .stream()
    .filter(c -> c.getArea() < 20)
    .collect(Collectors.toList());

System.out.println(listOfCircles);
```

`flatMap` operation

- Using `map`, every stream element is mapped into exactly one other stream element
- `flatMap` transforms each stream element into a stream of other elements (either zero or more)
  - Takes in a function that produces another stream and "flattens" the stream

```java
List<String> stringList = Arrays.asList(
        "live", "long", "and", "prosper");

stringList.stream()
    .forEach(System.out::println);

stringList.stream()
    .flatMap(x -> x.chars().boxed())
    .forEach(System.out::println);
```


Single Abstact Method

- To facilitate lambda abstractions and method references, single abstract methods, or SAMs, are utilized
- Java's functional interface is an attempt to provide SAMs:
  - There is only one abstract method, although
  - Other abstract methods (like `toString`) are allowed if they are implemented by `java.lang.Object`
  - Functional interfaces also comprise some default methods (for the purpose of function composition)
- Only one abstract method so that the compiler can infer which method body the lambda expression implements
- Such an interface is more commonly known as a SAM interface

`Predicate` Functional Interface

- Example: `Stream`'s `filter` method is declared as:

```java
Stream<T> filter(Predicate <? super T> predicate)
```

- Only stream elements matching the given predicate is returned
- Abstract method in `Predicate<T>`:

```java
boolean test(T t)
```

- Sample usage using anonymous class:

```java
Circle[] circles = {new Circle(1), new Circle(2), new Circle(3)};
Stream.of(circles)
    .filter(new Predicate<Circle>() {
        public boolean test(Circle c) {
            return c.getArea() < 20;
        }
    })
    .forEach(System.out::println);
```


`Predicate` Functional Interface

- More examples:

```java
Stream.of(circles)
    .filter(c -> c.getArea() < 20)
    .forEach(System.out::println);


Stream.of(circles)
    .filter(new Predicate<Shape>() {
        public boolean test(Shape s) {
            return s.getID() % 2 == 0;
        }
    })
    .forEach(System.out::println);
```

- `Predicate` is a *consumer*...

`Consumer` Functional Interface

- `Stream<T>`'s `forEach` method is declared as:
  `void forEach(Consumer <? super T> action)`
- Accepts a single input and returns nothing
- Abstract method in `Consumer<T>`:
  `void accept(T t)`
- Sample usage using anonymous class:

```
Circle[] circles = {new Circle(1), new Circle(2), new Circle(3)};
Stream.of(circles)
    .forEach(new Consumer<Circle>() {
        @Override
        public void accept(Circle c) {
            System.out.println(c.getArea());
        }
    });
```


t

`Consumer` Functional Interface

- More examples:

```
Stream.of(circles)
    .forEach(c -> System.out.println(c.getArea()));

Stream.of(circles)
    .forEach(Circle::printArea);

Stream.of(circles)
    .forEach(new Consumer<Shape>() {
        @Override
        public void accept(Shape s) {
            System.out.println("Shape #" +
                s.getID() + ": " + s);
        }
    });
```

- *Consumer super...*

`Supplier` Functional Interface

- `Stream<T>`'s generic `generate` method is declared as:
  `static <T> Stream<T> generate(Supplier<? extends T> s)`
- A supplier of results
- Abstract method in `Supplier<T>`: `T get()`
- Sample usage using anonymous class:

```
Stream
    .generate(new Supplier<Circle>() {
        @Override
        public Circle get() {
            return new Circle(2.0);
        }
    })
    .limit(5)
    .forEach(System.out::println);
```


`Supplier` Functional Interface

- Other examples:

```
Stream.generate(() -> new Circle(2.0))
    .limit(5)
    .forEach(System.out::println);

List<Circle> circles = Stream
    .generate(new Supplier<UnitCircle>() {
        @Override
        public UnitCircle get() {
            return new UnitCircle();
        }
    })
    .limit(5)
    .collect(Collectors.toList());
 System.out.println(circles);
```

- *Supplier (producer) extends...*

`Function` Functional Interface

- `Stream<T>`'s generic `map` method is declared as:

  `<R> Stream<R> map(Function<? super T, ? extends R> mapper)`
- Accepts one type `T` argument and produces a type `R` result
- Abstract method in `Function<T,R>`:

  `R apply(T t)`
- Sample usage using anonymous class:

```java
Circle[] circles = {new Circle(1), new Circle(2), new Circle(3)};
Stream.of(circles)
    .map(new Function<Circle, Double>() {
        @Override
        public Double apply(Circle c) {
            return c.getArea();
        }
    })
    .forEach(System.out::println);
```


`Function` Functional Interface

- More examples:

```java
Stream.of(circles)
    .map(c -> c.getArea() * c.getRadius() * 4.0 / 3)
    .forEach(c ->
            System.out.println("Volume: " + c));

List<Number> listOfIDs = Stream
    .of(circles)
    .map(new Function<Shape, Number>() {
        @Override
        public Number apply(Shape s) {
            return s.getID();
        }
    })
    .collect(Collectors.toList());
System.out.println(listOfIDs);
```


`BinaryOperator` Functional Interface

- `Stream<T>`'s single-argument `reduce` method is declared as:

  `Optional<T> reduce(BinaryOperator<T> accumulator)`
- `BinaryOperator<T>` extends `BiFunction<T,T,T>`
- `BiFunction` accepts two arguments and produces a result
- Abstract method in `BiFunction<T,U,R>`:

  `R apply(T t, U u)`
- Sample usage in object-oriented programming:

```java
Circle[] circles = {new Circle(1), new Circle(2), new Circle(3)};
Circle newCircle = Stream.of(circles)
    .reduce(new BinaryOperator<Circle>() {
        @Override
        public Circle apply(Circle c1, Circle c2) {
            return new Circle(c1.getRadius() + c2.getRadius());
        }
    })
    .get();
System.out.println(newCircle);
```


`BinaryOperator` Functional Interface

- More examples:

```java
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)
  - If a reduction value exists, `get()` returns the value
  - Otherwise, `NoSuchElementException` is thrown
  - `Optional` provides a `ifPresent` method that performs the given action with the value if it is present, but otherwise does nothing

Function Composition

- Function composition of the form: $(g \circ f)(x) = g(f(x))$
- Example:

```
Function<String, Integer> f = str -> str.length();
Function<Integer, Circle> g = x -> new Circle(x);
```

- `Function<T,R>` has a default `andThen` method:

```
default <V> Function<T,V> andThen(
        Function<? super R, ? extends V> after)
```

- E.g. `System.out.println(f.andThen(g).apply("abc"));`
- `Function<T,R>` has an alternative default `compose` method:

```
default <V> Function<V,R> compose(
        Function<? super V, ? extends T> before)
```

- E.g. `System.out.println(g.compose(f).apply("abc"));`

`BiFunction` Revisited

- Consider the following:

```
BiFunction<Integer, Integer, Integer> f;
f = (x, y) -> x + y;
System.out.println(f.apply(1, 2));
```

- Can we achieve the same with `Function<T,R>` instead?

```
Function<Integer, Function<Integer,Integer>> g = new Function<>()
    @Override
    public Function<Integer,Integer> apply(Integer x) {
        Function<Integer,Integer> f = new Function<>() {
            @Override
            public Integer apply(Integer y) {
                return x + y;
            }
        };
        return f;
    }
};
System.out.println(g.apply(1).apply(2));
```


Currying

- Indeed, tne lambda expression `(x, y) -> x + y` can indeed by re-expressed as `x -> y -> x + y`

```
Function<Integer, Function<Integer,Integer>> g;

g = x -> y -> x + y;
System.out.println(g.apply(1).apply(2));
```

- This is known as **currying** which gives us a way to handle lambdas of arbitrary number of arguments
- `g` returns a lambda of type `Function<Integer, Integer>`, and we can make use of it to say, increment:

```
Function<Integer, Integer> inc = g.apply(1);

System.out.println(inc.apply(10));
```


Lecture Summary

- Be familiar with the user of object `Stream`
- Know the difference between stateless and stateful operations
- Know how to obtain a collection from a stream
- Appreciate the difference between `map` and `flatMap`
- Understand how Java Functional Interface can be used for single abstract method for handling lambda expressions
- Know the common functional interfaces and situations where they are used
- Appreciate function composition and currying to manage more complex lambdas