CS2030 Lecture 9

Java Streams and Functional Interfaces

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

Semester 2 2018 / 2019

Lecture Outline

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

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);
```

Stateless vs Stateful Operations

- Wherever possible, maintain stateless stream pipeline results
- Example, testing primality of n
- What happens to the following?

```
boolean isPrime(int n) {
    boolean prime = true;
    IntStream
        .range(2, n)
        .filter(x -> n % x == 0)
        .forEach(x -> prime = false);
    return prime;
}
```

ERROR

- Local variables referenced from a lambda expression must be
  - **final**, i.e. explicitly declared `final`, or
  - **effectively final**, value of variable or parameter is never changed after initialization

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 pipeline

```
class MyBoolean {
    boolean flag = true;
}
boolean isPrime(int n) {
    MyBoolean prime = new MyBoolean();
    IntStream
        .range(2, n)
        .filter(x -> n % x == 0)
        .forEach(x -> prime.flag = false);
    return prime.flag;
}
```

HAHAHAHAHAHAHAH

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

Mapping Primitive Stream and `Stream`

- From `IntStream` to `Stream`

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

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

- From `Stream` to `DoubleStream`

```
DoubleStream areas = Stream
    .of(new Circle(5), new Circle(2))
    .mapToDouble(Circle::getArea); // c -> c.getArea()

double totalArea = areas.sum();
System.out.println(totalArea);
```

From `Collection` to `Stream` and *vice-versa*

- Given an array of `Circle` objects

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

- To produce, a stream of `Circle` objects, one can use

```
Stream<Circle> circleStream = Stream.of(circles);
```

- In the case of a list of circles,

```
List<Circle> listOfCircles = Arrays.asList(circles);
```

  a `Collection`'s `stream()` produces a stream from a collection

```
Stream<Circle> circleStream = listOfCircles.stream();
```

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

```
circleStream
    .filter(c -> c.getArea() < 20)
    .collect(Collectors.toList());
```

`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) by taking in a function that produces another stream and "flattens" it
  - Example, maximum disc coverage problem

```java
Stream<Optional<Circle>> unitCircles = Stream
    .of(points)
    .flatMap(i -> {
        Stream<Point> pts2 = Arrays.stream(points);
        return pts2.map(j -> createUnitCircle(i, j));
    });

return unitCircles
    .flatMap(Optional::stream) // .filter(x -> x.isPresent())
    .map(x -> findCoverage(x, points)) // use x.get() instead
    .reduce(0, (x, y) -> Math.max(x, y));
```

Single Abstact Method Revisited

- 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

`@FunctionalInterface` and Higher Order Functions

- Higher Order Functions: functions that take in other functions
- Example, `Predicate<T>` with method `boolean test(T t)`

```java
List<Circle> select(Predicate<Circle> pred, Circle... circles) {
    List<Circle> outList = new ArrayList<>();
    for (Circle circle : circles) {
        if (pred.test(circle)) {
            outList.add(circle);
        }
    }
    return outList;
}
```

- Or declaratively using streams

```java
List<Circle> select(Predicate<Circle> pred, Circle... circles) {
    return Stream
        .of(circles)
        .filter(pred)
        .collect(Collectors.toList());
}
```

`@FunctionalInterface` and Higher Order Functions

- Finding circles with even radius:

```
jshell> Circle[] circles =
    new Circle[]{new Circle(1), new Circle(2), new Circle(3)}
jshell> Predicate<Circle> pred = x -> x.getRadius() % 2 == 0
jshell> select(pred, circles).stream().forEach(System.out::println
Circle with radius: 2.0
```

- Finding circles with radius > 10:

```
jshell> Predicate<Circle> pred = x -> x.getArea() > 10
jshell> select(pred, circles).stream().forEach(System.out::println
Circle with radius: 2.0
Circle with radius: 3.0
```

- Adherence to the **Principle of Abstraction**:
Each significant piece of functionality in a program should be implemented in just one place in the source code

@FunctionalInterface and Higher Order Functions

- But `getArea()` could be a method from the superclass `Shape`

```
jshell> Predicate<Shape> pred = x -> x.getArea() > 10
jshell> select(pred, circles).stream().forEach(System.out::println)
|  Error:
|  incompatible types: java.util.function.Predicate<Shape> cannot be converted
|  to java.util.function.Predicate<Circle>
|  select(pred, circles).stream().forEach(System.out::println)
|         ^--^
```

- Since `Circle` extends `Shape`, can either perform circle tests or shape tests on the `Circle` object
  - As such, the test predicate could also be a super-class of `Circle`
  - Need to change the parameter type to

```
List<Circle> select(Predicate<? super Circle> pred,
        Circle... circles)
```

`@FunctionalInterface` and Consumers

- `Predicate<T>` used as a consumer, e.g. in

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

- Likewise, `Consumer<T>` with `accept` method

  `void accept(T t)`

- Example usage:

```
void doSomething(Consumer<? super Circle> action, Circle... circle
    for (Circle circle : circles) {
        action.accept(circle);
    }
}

Circle[] circles =
    new Circle[]{new Circle(1), new Circle(2), new Circle(3)}
doSomething(System.out::println, circles)
```

- Or simply, `Stream.of(circles).forEach(System.out::println)`

@FunctionalInterface and Suppliers (Producers)

- `Supplier<T>` with abstract method `T get()`
- A supplier of results that takes no argument

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

List<Circle> getCircles(Supplier<Circle> supplier, int n) {
    List<Circle> outList = new ArrayList<>();

    for (int i = 0; i < n; i++) {
        outList.add(supplier.get());
    }
    return outList;
}
```

- Support different ways of generating `Circle` objects

```
Supplier<Circle> supplier = () -> new Circle(Math.random())
List<Circle> circles = getCircles(supplier, 5)
circles.stream().forEach(System.out::println)
```

`@FunctionalInterface` and Suppliers (Producers)

- Suppose we have `getShapes` instead

```java
List<Shape> getShapes(Supplier<Shape> supplier, int n) {
    List<Shape> outList = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        outList.add(supplier.get());
    }
    return outList;
}
```

- Since circles are shapes, a supplier of circles is possible

```
jshell> Supplier<Circle> supplier = () -> new Circle(Math.random())
jshell> List<Shape> shapes = getShapes(supplier, 5);
|  Error:
|  incompatible types: java.util.function.Supplier<Circle> cannot be converted
|  to java.util.function.Supplier<Shape>
|  List<Shape> shapes = getShapes(supplier, 5);
|                                 ^------^
```

- Need to change the parameter type to

```java
List<Shape> getShapes(Supplier<? extends Shape> supplier, int n)
```

@FunctionalInterface and Suppliers (Producers)

- Using `Supplier<T>` as delayed data

```
class DelayedData {
    static Scanner sc = new Scanner(System.in);
    private int index;
    private int input;

    public DelayedData(int index, int input) {
        this.index = index;
        this.input = input;
    }

    public String toString() {
        return index + " : " + input;
    }

    public static void main(String[] args) {
        DelayedData[] data = new DelayedData[5];
        for (int i = 0; i < data.length; i++) {
            data[i] = new DelayedData(i, sc.nextInt());
        }
        Stream
            .of(data)
            .filter(x -> x.index % 2 == 0)
            .forEach(System.out::println);
    }
}
```

@FunctionalInterface and Suppliers (Producers)

- Reading input only when needed

```
class DelayedData {
    static Scanner sc = new Scanner(System.in);
    private int index;
    private Supplier<Integer> input;

    public DelayedData(int index, Supplier<Integer> input) {
        this.index = index;
        this.input = input;
    }

    public String toString() {
        return index + " : " + input.get();    reads here
    }

    public static void main(String[] args) {
        DelayedData[] data = new DelayedData[5];
        for (int i = 0; i < data.length; i++) {
            data[i] = new DelayedData(i, () -> sc.nextInt()); no reading yet
        }
        Stream
            .of(data)
            .filter(x -> x.index % 2 == 0)
            .forEach(System.out::println);
    }
}
```

Function/UnaryOperator @FunctionalInterface

- `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)`
- `UnaryOperator<T>` extends `Function<T,T>`

```
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);

Stream.of(circles)
    .map(Circle::getArea)
    .forEach(System.out::println)
```

BiFunction/BinaryOperator @FunctionalInterface

- BiFunction accepts two arguments and produces a result
- Abstract method in BiFunction<T,U,R>:

  `R apply(T t, U u)`
- BinaryOperator<T> extends BiFunction<T,T,T>
- Stream<T>'s single-argument reduce method is declared as:

  `T reduce(T identity, BinaryOperator<T> accumulator)`
- Sample usage:

```java
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()))
    .get();

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

`BinaryOperator` Functional Interface

- Overloaded `reduce` method:

  `Optional<T> reduce(BinaryOperator<T> accumulator)`
- Sample usage:

```
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"));`

Function With Multiple Arguments

- Consider the following:

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

Currying

- Indeed, the lambda expression `(x, y) -> x + y` can be re-expressed as `x -> y -> x + y`

```
Function<Integer, UnaryOperator<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 `UnaryOperator<Integer>`, and we can make use of it to say, increment:

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

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

Lecture Summary

- Distinguish the difference between stateless and stateful operations
- Be familiar with the user of object `Stream`
- 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