CS2030 Lecture 8

Towards Declarative Programming — Optional and Stream

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

Semester 1 2021 / 2022

Lecture Outline and Learning Outcomes

- Appreciate the difference between imperative and **declarative** styles of programming
- Know how to use **Optional** to manage missing/null values
- Know how to use **Stream** to manage iteration
  - Able to create stream pipelines
  - Understand **lazy evaluation** in source/intermediate operations, and **eager evaluation** for terminal operations
  - Understand how lazy evaluation supports **infinite stream**
  - Able to define streams correctly to operate in both sequentially and parallel modes
  - Understand stream reduction using an associative accumulation function
  - Appreciate the overhead of parallelizing tasks

Imperative vs Declarative Programming

- Suppose we would like to sum up all even integers in an array

```
jshell> Integer[] ints = new Integer[]{1, 2, 3, 4, 5, 6, 7, 8, 9}
ints ==> Integer[9] { 1, 2, 3, 4, 5, 6, 7, 8, 9 }

jshell> List<Integer> list = List.<Integer>of(ints) // imperatively using List
list ==> [1, 2, 3, 4, 5, 6, 7, 8, 9]

jshell> int sum = 0
sum ==> 0

jshell> for (int i = 0; i < list.size(); i++) {
   ...> if (list.get(i) % 2 == 0) { sum = sum + list.get(i); } }

jshell> sum
sum ==> 20

jshell> ImList.<Integer>of(ints). // declaratively using ImList
   ...> filter(x -> x % 2 == 0).
   ...> reduce(0, (x, y) -> x + y)
$22 ==> 20
```

- **Imperative** programming specifies *how* to do a task, while **declarative** programming simply specifies *what* to do
- Abstract all list processing details into a *context* — `ImList`

`Optional`: Context for Missing/Invalid Values

- Consider the `createUnitCircle(Point p, Point q)` method discussed earlier
  - What do we return if $dist_{PQ}$ is not within $(0, 2]$?
  - ~~Exception? Assertion? `null`?~~ ... use `Optional`!

```
import java.util.Optional;
    static Optional<Circle> createUnitCircle(Point p, Point q) {
        double d = p.distanceTo(q);
        if (d > 0.0 && d <= 2.0) {
            Circle circle;
            // assigning circle with new Circle object
            return Optional.<Circle>of(circle);
        } else {
            return Optional.empty();
        }
    }
```

```
jshell> Circle.createUnitCircle(new Point(0, -1), new Point(0, 1))
$.. ==> Optional[Circle at (0.0, 0.0) with radius 1.0]

jshell> Circle.createUnitCircle(new Point(0, -1), new Point(0, 10))
$.. ==> Optional.empty
```

`Optional`: Context for Missing/Invalid Values

- Rather than let the client handle the **if**... **else** branching associated with managing missing or invalid values
  - abstract the details of handling such values into `Optional`

```
jshell> Circle.createUnitCircle(new Point(0.0, -1.0), new Point(0.0, 1.0)).
   ...> filter(x -> x.contains(new Point(0.5, 0.5))).
   ...> ifPresent(x -> System.out.println(x))
Circle at (0.0, 0.0) with radius 1.0

jshell> Circle.createUnitCircle(new Point(0.0, -1.0), new Point(0.0, 10.0)).
   ...> filter(x -> x.contains(new Point(0.5, 0.5))).
   ...> ifPresent(x -> System.out.println(x))

jshell> Circle.createUnitCircle(new Point(0.0, -1.0), new Point(0.0, 1.0)).
   ...> filter(x -> x.contains(new Point(1.0, 1.0))).
   ...> ifPresent(x -> System.out.println(x))
```

- Same method chain *elegantly* handles different situations
- *Tell Don't Ask*: avoid using `get()`, `isPresent()`, `isEmpty()`

Other useful methods in **Optional**

```
jshell> Optional.<Integer>of(1)
$.. ==> Optional[1]

jshell> Optional.<Integer>ofNullable(null)
$.. ==> Optional.empty

jshell> Optional.<Integer>empty()
$.. ==> Optional.empty

jshell> Optional.<Integer>of(1).filter(x -> x % 2 == 0)
$.. ==> Optional.empty

jshell> Optional.<Integer>of(1).filter(x -> x % 2 == 0).ifPresent(x -> System.out.println(x))

jshell> Optional.<Integer>of(1).filter(x -> x % 2 == 1).ifPresent(x -> System.out.println(x))
1

jshell> Optional.<Integer>of(1).filter(x -> x % 2 == 0).map(x -> x * 2)
$.. ==> Optional.empty

jshell> Optional.<Integer>of(1).filter(x -> x % 2 == 1).map(x -> x * 2)
$.. ==> Optional[2]

jshell> Optional.<Integer>of(1).filter(x -> x % 2 == 0).or(() -> Optional.of(2))
$.. ==> Optional[2]

jshell> Optional.<Integer>of(1).filter(x -> x % 2 == 0).orElse(2)
$.. ==> 2

jshell> Optional.<Integer>of(1).filter(x -> x % 2 == 1).orElse(2)
$.. ==> 1

jshell> int foo() { System.out.println(2); return 2; }
|  created method foo()

jshell> Optional.<Integer>of(1).filter(x -> x % 2 == 1).orElse(foo())
2
$.. ==> 1
```

`Stream`: Context for Iteration

- *Internal iteration* — a declarative approach

```
jshell> int sum = IntStream.range(1, 10). // primitive int stream
   ...> sum()
sum ==> 45

jshell> sum = Stream.<Integer>iterate(1, x -> x < 10, x -> x + 1). // generic
   ...> reduce(0, (x, y) -> x + y)
sum ==> 45
```

- `sum` is assigned with the result of a **stream pipeline**
- Literal meaning "loop through values 1 to 9, and sum them"
- No need to specify how to iterate through elements or use any *mutable* variables — no variable state, no surprises! 😃
- 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
- Stream elements within a stream *can only be consumed once*

Stream Pipeline

- A stream pipeline starts with a **data source**
  - `IntStream.range` (`Stream.<Integer>iterate`) creates a stream of `int` (`Integer`) elements respectively
- `sum` / `reduce` are **terminal operations**
  - reduces the stream of values into a single value
- Most stream pipelines contain **intermediate operations** that specify tasks to perform on a stream's elements

```
jshell> IntStream.range(1, 10).map(x -> x * 2).sum()
$.. ==> 90
```

- Source/intermediate operations return a new stream made up of processing steps specified up to that point in the pipeline, e.g.

```
jshell> Stream.<Integer>iterate(1, x -> x < 10, x -> x + 1).map(x -> x * 2)
$.. ==> java.util.stream.ReferencePipeline$3@3e6fa38a
```

Lazy Evaluation

- Source/intermediate operations use **lazy evaluation**
  - does not perform any operations on stream's elements until a terminal operation is called
- Terminal operation use **eager evaluation**
  - performs the requested operation as soon as it is called

```
jshell> IntStream foo(int n) {
   ...>     return IntStream.iterate(1, x -> x + 1)
   ...>         .limit(n)
   ...>         .peek(x -> System.out.println("limit: " + x))
   ...>         .filter(x -> x % 2 == 0)
   ...>         .peek(x -> System.out.println("filter: " + x))
   ...>         .map(x -> 2 * x)
   ...>         .peek(x -> System.out.println("map: " + x));
   ...> }
```

```
jshell> foo(5).sum()
limit: 1
limit: 2
filter: 2
map: 4
limit: 3
limit: 4
filter: 4
map: 8
limit: 5
$.. ==> 12
```

Infinite Stream

- Lazy evaluation allows us to work with infinite streams that represent an infinite number of elements
  - `iterate(T seed, Function<T,T> next)` produces a sequence starting with the first argument as a seed value
  - `generate(Supplier<T> supplier)` produces a sequence of the same value
- Intermediate operations, e.g. `limit`, can be used to restrict the total number of elements in the stream

```
jshell> boolean isPrime(int n) {
   ...> return n > 1 && IntStream.range(2, (int) Math.sqrt(n) + 1) // or (2,n)
   ...> .noneMatch(x -> n % x == 0); }
jshell> IntStream.iterate(2, x -> x + 1).
   ...> filter(x -> isPrime(x)).
   ...> limit(20). // find first 20 primes
   ...> forEach(x -> System.out.print(x + " "))
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
```

Flat Mapping in Stream

- How about nested loops?

```
for (x = 1; x <= 3; x++)
   for (y = x; y <= 3; y++)
       System.out.println((x * y) + " "); // output is 1 2 3 4 6 9
```

- `map` maps a stream element into one other stream element

```
jshell> Stream.<Integer>of(1, 2, 3).
   ...> map(x -> Stream.<Integer>iterate(x, y -> y <= 3, y -> y + 1).
   ...>    map(y -> x * y)).
   ...> forEach(x -> System.out.println(x))
java.util.stream.ReferencePipeline$3@ae45eb6
java.util.stream.ReferencePipeline$3@59f99ea
java.util.stream.ReferencePipeline$3@27efef64
```

- `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 then *flattens* it

```
jshell> Stream.<Integer>of(1, 2, 3).
   ...> flatMap(x -> Stream.<Integer>iterate(x, y -> y <= 3, y -> y + 1).
   ...>    map(y -> x * y)).
   ...> forEach(x -> System.out.print(x + " "))
1 2 3 4 6 9
```

Handling Parallelism in Streams

- Parallelizing the seach for primes

```
jshell> import java.time.*
jshell> long numOfPrimes(int from, int to) {
   ...>     Instant start = Instant.now(); // start timing
   ...>     long howMany = IntStream.rangeClosed(from, to)
   ...>         .parallel().filter(x -> isPrime(x)).count();
   ...>     Instant stop = Instant.now(); // end timing
   ...>     System.out.println("Duration: " + Duration.between(start, stop).toMillis() + "ms");
   ...>     return howMany;
   ...> }
jshell> numOfPrimes(2_000_000, 3_000_000)
Duration: 239ms
$.. ==> 67883
```

- `parallel()` operation switches the stream pipeline to parallel
  - invoke anywhere between the data source and terminal
  - `sequential()` switches off parallel operation
- Avoid parallelizing trivial tasks, e.g. `isPrime`
  - creates more work in terms of parallelizing overhead
  - worthwhile only if the task is complex enough

Correctness of (Parallel) Streams

- To ensure correct execution, stream operations
  - must not interfere with stream data

```
jshell> List<String> list = new ArrayList<>(List.of("abc","def","xyz"))
list ==> [abc, def, xyz]

jshell> list.stream().peek(str -> { if (str.equals("xyz")) list.add("pqr"); }).
   ...> forEach(x -> {})
   |  Exception java.util.ConcurrentModificationException
   |  ...
```

  - preferably stateless (*cf.* `sorted` and `limit` which are stateful) with no side effects

```
jshell> List<Integer> list = Arrays.asList(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)
list ==> [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

jshell> List<Integer> result = new ArrayList<>()
result ==> []

jshell> list.stream().parallel().
   ...> filter(x -> isPrime(x)).
   ...> forEach(x -> result.add(x)) // what is result?
```

    - use `forEachOrdered` or `.collect(Collectors.toList())`, or
    - replace `ArrayList` with `CopyOnWriteArrayList`

Associative Accumulation Function

- `Stream`'s three-argument `reduce` method:

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

- Rules to follow when parallelizing
  - `combiner.apply(identity, i)` must be equal to `i`

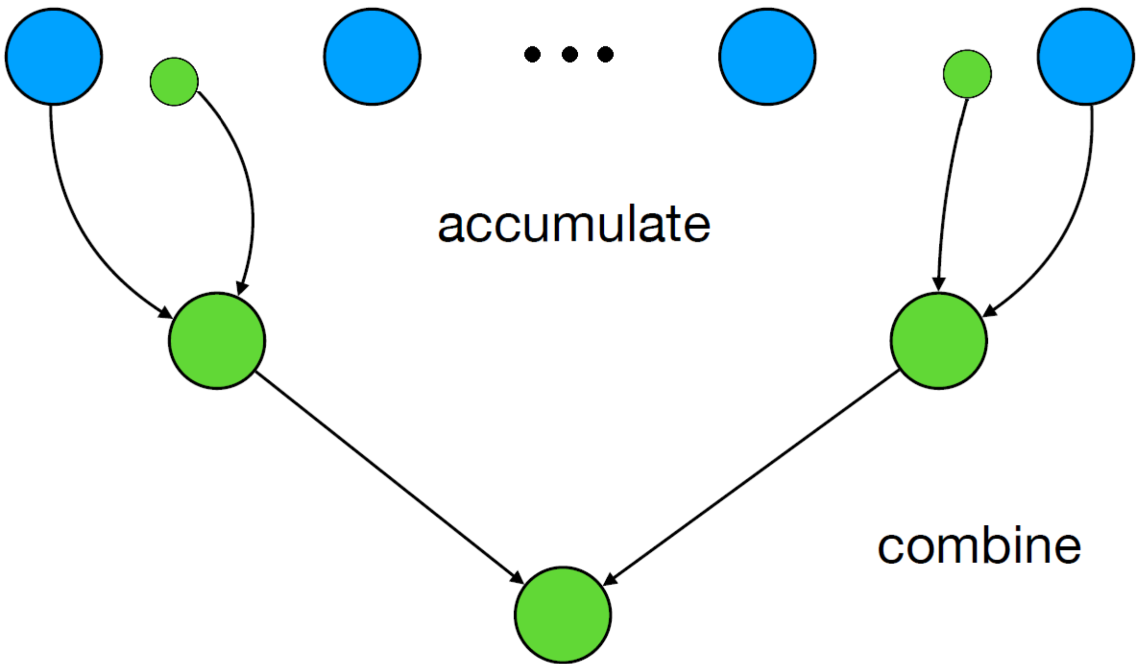
    ```
    .reduce(1, (x,y) -> x + y, (x,y) -> x + y) // wrong!
    ```

  - `combiner` and `accumulator` must be *associative*, i.e. order of application does not matter, e.g. $(1/2)/3 \neq 1/(2/3)$

    ```
    .reduce(1.0, (x,y) -> x / y, (x,y) -> x / y) // wrong!
    ```

  - `combiner` and `accumulator` must be *compatible*, i.e. `combiner.apply(u, accumulator.apply(identity, t))` must be equal to `accumulator.apply(u, t)`

Associative Accumulation Function



- accumulator (`(T,U) -> U`) accumulates a stream element `T` with identity or outcome of another combine `U`
- combiner (`(U,U) -> U`) combines any identity `U` or outcome of another combine `U`

Accumulator and Combiner

- Effects of the accumulator and combiner in parallel streams

```
jshell> String name() {
   ...>     return Thread.currentThread().getName();
   ...> }
|  created method name()

jshell> Stream.of(1, 2, 3, 4, 5).
   ...>    parallel().
   ...>    filter(x -> {
   ...>       System.out.println("filter:  " + x + " "  + name());
   ...>       return x % 2 == 1; }).
   ...>    reduce(0,
   ...>       (x, y) -> {
   ...>          System.out.println("accumulate: " + x + " + " + y + " " + name());
   ...>          return x + y; },
   ...>       (x, y) -> {
   ...>          System.out.println("combine: " + x + " + " + y + " " + name());
   ...>          return x + y; })
filter:  5 ForkJoinPool.commonPool-worker-1
filter:  4 ForkJoinPool.commonPool-worker-3
filter:  1 ForkJoinPool.commonPool-worker-3
filter:  3 main
filter:  2 ForkJoinPool.commonPool-worker-2
accumulate: 0 + 5 ForkJoinPool.commonPool-worker-1 // accumulate with identity
accumulate: 0 + 1 ForkJoinPool.commonPool-worker-3
combine: 1 + 0 ForkJoinPool.commonPool-worker-3 // combine with identity
accumulate: 0 + 3 main
combine: 0 + 5 ForkJoinPool.commonPool-worker-1 //
combine: 3 + 5 ForkJoinPool.commonPool-worker-1 // combine outcomes from two combines
combine: 1 + 8 ForkJoinPool.commonPool-worker-1
$.. ==> 9
```