CS2030 Lecture 9

Java Streams

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

Semester 1 2023 / 2024

Lecture Outline and Learning Outcomes

□ Know how to create **stream** pipelines for *internal* iteration

- Know the difference between primitive and generic streams

□ Understand **lazy evaluation** in source/intermediate operations, and **eager evaluation** for terminal operations

□ Appreciate how lazy evaluation supports **infinite stream**

□ Able to implement a basic lazy context by encapsulating a `Supplier` functional interface for *delayed data*

□ Appreciate that streams should be *inherently parallelizable*

□ Know how to write correct streams that are non-interfering and stateless with no side effects

External Iteration

- An external iteration is defined *imperatively*
  - e.g. sum of all integers in the closed interval $[1, 10]$

```
jshell> int sum = 0
sum ==> 0

jshell> for (int x = 1; x <= 10; x = x + 1) {
   ...>     sum = sum + x;
   ...> }

jshell> sum
sum ==> 55
```

- 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: **`Stream`**

□ Internal iteration is defined *declaratively*

  – e.g. using a primitive integer stream

```
jshell> int sum = IntStream.rangeClosed(1, 10).
   ...> sum()
sum ==> 55
```

□ Literal meaning “loop through values 1 to 10, 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; stream elements move through a sequence of tasks in the stream pipeline

□ Result is obtained at the end of stream processing

Stream Pipeline

- A stream pipeline comprises
  - a **data source** (e.g. `IntStream::rangeClosed`) to start the stream
  - some **intermediate operations** (e.g. `IntStream::map`) that specify tasks to perform on a stream's elements

    ```
    jshell> IntStream stream = IntStream.rangeClosed(1, 10).
       ...> map(x -> x * 2)
    stream ==> java.util.stream.IntPipeline$Head@12edcd21
    ```

  - a **terminal operation** (e.g. `IntStream::sum`) that *reduces* the stream elements into a single value

    ```
    jshell> stream.sum()
    $.. ==> 110
    ```

- Each source/intermediate operation returns a new stream of processing steps specified up to that point in the pipeline
- Stream elements within a stream *can only be consumed once*

Exercise: Primality Test

- Given the following external iteration:

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

- Complete the following internal iteration:

```
boolean isPrime(int n) {
    return n > 1 && IntStream...



}
```

Reducing a Stream to a Value

- Iterate through `IntStream` elements and reduce to an **`int`**

```
int reduce(int identity, IntBinaryOperator op)
```

- `IntBinaryOperator` with single abstract method:

```
int applyAsInt(int left, int right)
```

```
jshell> IntStream.rangeClosed(1, 10).
   ...> reduce(0, (x,y) -> x + y)
$.. ==> 55
```

```
jshell> IntStream.rangeClosed(1, 10).
   ...> reduce(1, (x,y) -> x * y)
$.. ==> 3628800
```

- Alternative one argument `reduce` that returns `OptionalInt`

```
OptionalInt reduce(IntBinaryOperator op)
```

```
jshell> IntStream.range(1, 10).reduce((x, y) -> x * y)
$.. ==> OptionalInt[362880]
```

```
jshell> IntStream.range(1, 1).reduce((x, y) -> x * y)
$.. ==> OptionalInt.empty
```

flatMap Method 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 tries to map each stream element into one other stream

```
jshell> IntStream.rangeClosed(1, 3).
   ...> map(x -> IntStream.rangeClosed(x,3).map(y -> x * y))
|  Error:
|  incompatible types: bad return type in lambda expression
|      java.util.stream.IntStream cannot be converted to int
|  map(x -> IntStream.rangeClosed(x,3).map(y -> x * y))
|           ^----------------------------------------^
```

□ 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> IntStream.rangeClosed(1, 3).
   ...> flatMap(x -> IntStream.rangeClosed(x,3).map(y -> x * y)).
   ...> forEach(x -> System.out.print(x + " "))
1 2 3 4 6 9
```

Generic **Stream<T>**

- Stream<T> is a stream over reference-typed objects with data sources: of, iterate and generate

```
jshell> int sum = Stream.<Integer>iterate(1, x -> x <= 10, x -> x + 1).
   ...> reduce(0, (x,y) -> x + y) // note: reduce(T, BinaryOperator<T>)
sum ==> 55
```

- boxed() wraps stream elements in its wrapper type

```
jshell> Stream<Integer> stream = IntStream.rangeClosed(1, 10).boxed()
stream ==> java.util.stream.IntPipeline$1@5010be6

jshell> List<Integer> list = stream.toList()
list ==> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- mapToObj converts from primitive to generic stream

```
jshell> IntStream.rangeClosed(1, 10).mapToObj(x -> "<" + x + ">").
   ...> toList()
$.. ==> [<1>, <2>, <3>, <4>, <5>, <6>, <7>, <8>, <9>, <10>]
```

- Stream::toList() converts generic stream to generic list
- List::stream() converts generic list to generic stream

Lazy Evaluation in Streams

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

```
jshell> Stream.<Integer>iterate(1, x -> x + 1).
   ...> limit(5).
   ...> peek(x -> System.out.println("limit: " + x)).
   ...> filter(x -> x % 2 == 0).
   ...> peek(x -> System.out.println("filter: " + x)).
   ...> map(x -> x * 2).
   ...> peek(x -> System.out.println("map: " + x)).
   ...> reduce(0, (x, y) -> {
   ...>     System.out.println("reduce: " + x + " + " + y);
   ...>     return x + y;
   ...> })
```

```
limit: 1
limit: 2
filter: 2
map: 4
reduce: 0 + 4
limit: 3
limit: 4
filter: 4
map: 8
reduce: 4 + 8
limit: 5
$.. ==> 12
```

Infinite Stream

- Lazy evaluation allows us to work with infinite streams that represent an infinite number of elements
  - `Stream<T>::generate(Supplier<T> supplier)` produces an infinite sequence of values generated by `supplier`
  - `Stream<T>::iterate(T seed, UnaryOperator<T> next)` produces an infinite sequence by repeatedly applying the function `next` starting with the `seed` value
- Intermediate operations, e.g. `limit`, can be used to restrict the total number of elements in the stream

```
jshell> Stream.<Integer>iterate(1, x -> x + 1).
   ...> filter(x -> x % 2 == 1).
   ...> limit(20). // find first 20 odd numbers
   ...> forEach(x -> System.out.print(x + " "))
1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
```

Lazy Class

□ To understand how lazy evaluation works, define a **Lazy** class

```java
import java.util.function.Supplier;

class Lazy<T> {
    private final Supplier<T> supplier;

    private Lazy(Supplier<T> supplier) {
        this.supplier = supplier;
    }

    static <T> Lazy<T> of(Supplier<T> supplier) {
        return new Lazy<T>(supplier);
    }

    static <T> Lazy<T> of(T t) {
        return new Lazy<T>(() -> t);
    }

    public T get() {
        return supplier.get();
    }
}
```

```
jshell> int foo() {
   ...>    System.out.println("foo");
   ...>    return -1;
   ...> }
|  created method foo()

jshell> Lazy<Integer> lazy = Lazy.of(foo())
foo
$.. ==> Lazy@ae45eb6

jshell> lazy.get()
$.. ==> -1

jshell> lazy = Lazy.<Integer>of(() -> foo())
$.. ==> Lazy@6f7fd0e6

jshell> lazy.get()
foo
$.. ==> -1
```

□ `Lazy.of(foo())` evaluates `foo` method *eagerly*

□ `Lazy.of(() -> foo())` evaluates `foo` *lazily*, i.e. only when `get()` is invoked sometime later

Mapping a Lazy Value

- Define `map` that returns a new `Lazy`

```
<R> Lazy<R> map(Function<? super T, ? extends R> mapper) {
    Supplier<R> supplier = () -> mapper.apply(this.get());
    return Lazy.<R>of(supplier);
}
```

```
jshell> Lazy<Integer> i = Lazy.<String>of(() -> "abc").
   ...>     map(x -> { System.out.println("map1"); return x.length(); }).
   ...>     map(x -> { System.out.println("map2"); return x * 2; })
i ==> Lazy@51565ec2 // map is not evaluated until a get()

jshell> i.get() // map is lazily evaluated :)
map1
map2
$.. ==> 6
```

- What is wrong with the following implementation of `map`?

```
<R> Lazy<R> map(Function<? super T, ? extends R> mapper) {
    R r = mapper.apply(this.get());
    return Lazy.<R>of(() -> r);
}
```

Inherently Parallelizable Stream

- A stream pipeline should be *inherently parallelizable*
  - intermediate operations can operate on elements in parallel
  - reduction uses a divide-and-conquer strategy
- `parallel()` operation switches the stream pipeline to parallel
  - invoke anywhere between the data source and terminal
  - `sequential()` switches off parallel operation
- `reduce` method uses an *associative accumulation function*, e.g.

```
T reduce(T identity, BinaryOperator<T> acc) // BiFunction<T,T,T>
```

```
jshell> DoubleStream.of(1.0, 2.0, 3.0, 4.0).parallel().
   ...> reduce(1.0, (x, y) -> x * y) // multiply is associative
$.. ==> 24.0
```

```
jshell> DoubleStream.of(1.0, 2.0, 3.0, 4.0).parallel().
reduce(24.0, (x, y) -> x / y) // divide is not associative
$.. ==> 1.5 // should be 1.0?
```

Example: Parallelism in Streams

□ Parallelizing the seach for primes

```
jshell> Runtime.getRuntime().availableProcessors()
$.. ==> 8

jshell> ForkJoinPool.commonPool().getParallelism()
$.. ==> 7

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
```

□ Avoid parallelizing trivial tasks, e.g. parallelizing `isPrime`

– creates more work in terms of parallelizing overhead

– worthwhile only if the parallel task is complex enough

Correctness of Streams

□ To ensure correct execution, stream operations

- must not interfere with stream data

```
jshell> List<String> list = new ArrayList<String>(
   ...>    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 (`map` vs `distinct`) with no side effects

```
jshell> List<Integer> list = List.of(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<Integer>()
result ==> []

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