CS2030 Lecture 9

The Art of Being Lazy

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

Semester 2 2019 / 2020

Lecture Outline

- Lazy evaluation in Java streams
- Designing our own infinite list
  - Via customized `get` method definitions
  - Via `Suppliers` and delayed data
- Implemented operations:
  - Data source: `generate` and `iterate`
  - Terminal: `forEach`
  - Stateless intermediate operation: `map` and `filter`
  - Stateful intermediate operation: `limit`
- Revisiting Java memory modeling, particularly variable capture and lambda closures

Lazy Evaluation in Streams

- Each intermediate operation results in a new stream
- Each new stream is an object representing the processing steps that have been specified up to that point in the pipeline
  - Chaining intermediate operations adds to the set of processing steps to perform on each stream element
  - The last stream object contains all processing steps to perform on each stream element
- An intermediate operation **does not** iterate elements and perform the processing steps
- A terminal operation initiates the stream pipeline operations
  - intermediate operations' processing steps are applied one stream element after another

Lazy Evaluation in Streams

- The following illustrates the movement of stream elements

```java
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
```

Lazy Evaluation in Streams

- Consider

```
Stream.iterate(1, x -> x + 1)
    .map(x -> x * 2)
    .limit(3)
    .forEach(System.out::println)
```

- Non-terminal operations `iterate`, `map` and `limit` each results in a new `Stream<T>`
- No operation on these non-terminals is performed until a terminal operation (in this case `forEach()`) is called
- A stream pipeline initiates with a terminal operation, and the upstream operations are applied

Implementation #1: Operation **generate**

- Consider the following:

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

interface IFL<T> {

    static <T> IFL<T> generate(Supplier<T> supplier) {
        return new IFL<T>() {
            public T get() {
                return supplier.get();
            }
        };
    }

    T get();
}
```

```
jshell> IFL<Integer> ifl = IFL.generate(() -> 1)
ifl ==> IFL@91161c7

jshell> ifl.get()
.. ==> 1

jshell> ifl.get()
.. ==> 1
```

- `generate()` creates an `IFL` object with specific `get()` method

Implement Operation **iterate**

```java
static <T> IFL<T> iterate(T seed, Function<T,T> next) {
    return new IFL<T>() {
        private T element = seed;
        private boolean firstElement = true;

        public T get() {
            if (firstElement) {
                firstElement = false;
            } else {
                element = next.apply(element);
            }
            return element;
        }
    };
}
```

- Notice that the implementation has a side-effect*

```
jshell> IFL<Integer> ifl = IFL.iterate(1, x -> x + 1)
ifl ==> IFL@59494225

jshell> ifl.get()
.. ==> 1

jshell> ifl.get()
.. ==> 2
```

* *We shall resolve this in implementation #2; let's focus on laziness first*

Implementation #1: Operation `forEach`

- `get` method should be encapsulated
- `forEach` is a terminal that repeatedly performs the `get()`
  - applies (`accept`) the `action` on each element

```java
public void forEach(Consumer<T> action) {
    T curr = get();
    while (true) {
        action.accept(curr);
        curr = get();
    }
}
```

- However, `forEach` should not be defined in the interface
  - Define an **abstract** class `IFLImpl` that implements `IFL`
  - Encapsulate `get` method within `IFLImpl`

Implementation #1: Operation **forEach**

```java
public interface IFL<T> {
    public static <T> IFL<T> generate(Supplier<T> supplier) {
        return IFLImpl.generate(supplier);
    }

    public static <T> IFL<T> iterate(T seed, Function<T, T> next) {
        return IFLImpl.iterate(seed, next);
    }

    public void forEach(Consumer<T> action);
}

abstract class IFLImpl<T> implements IFL<T> {
    static <T> IFLImpl<T> generate(Supplier<T> supplier) {
        return new IFLImpl<T>() {
        :
    }

    static <T> IFLImpl<T> iterate(T seed, Function<T, T> next) {
        return new IFLImpl<T>() {
        :
    }

    public void forEach(Consumer<T> action) {
        T curr = get();
        while (true) {
            action.accept(curr);
            curr = get();
        }
    }

    abstract T get();
}
```

```
jshell> IFL<Integer> ifl = IFL.iterate(1, x -> x + 1)
ifl ==> IFLImpl$1@6ddf90b0

jshell> ifl.forEach(System.out::println)
1
2
3
:
```

Infinite loop must be replaced eventually...

Implementation #1: Operation `map`

- The `map` operation takes in a function of type `Function<T,R>`

```java
public <R> IFL<R> map(Function<T, R> mapper) {
    return new IFLImpl<R>() {
        R get() {
            return mapper.apply(IFLImpl.this.get());
        }
    };
}
```

```
jshell> IFL<Integer> ifl = IFL.iterate(1, x -> x + 1).map(x -> x * 2)
ifl ==> IFLImpl$1@548e7350

jshell> ifl.forEach(System.out::println)
2
4
6
:
```

- In the argument of `mapper.apply(...)`
  - `IFLImpl.this` is the captured reference to the enclosing `IFLImpl` class

Implementation #1: Operation `filter`

- The outcome of a filter could either be an element of the stream or nothing — use `Optional`
  - Modify method `get` to return `Optional<T>`
  - Modify all current methods to work with `Optional<T>`
  - Make liberal use of methods in `Optional` class:
    - Avoid explicit handling of presence/absence of elements
    - Avoid using `Optional`'s `get`

```
public IFL<T> filter(Predicate<T> predicate) {
    return new IFLImpl<T>() {
        Optional<T> get() {
            return IFLImpl.this.get().filter(predicate);

        }
    };
}
```

Intermediate Operation `limit`

- Unlike `map` and `filter` which are *stateless* operations, `limit` is a *stateful* operation
  - A state has to be maintained
- Need a way to indicate to terminal operations when a stream has no more elements
- Would returning `Optional.empty()` as in `filter` work?
  - `Optional.empty()` is an indication to terminal operations not to process the current element, but to continue with the next element
  - `limit` is different in that terminal operation should terminate stream processing when the limit is reached
- *Ponder over it... you're on your own now...*

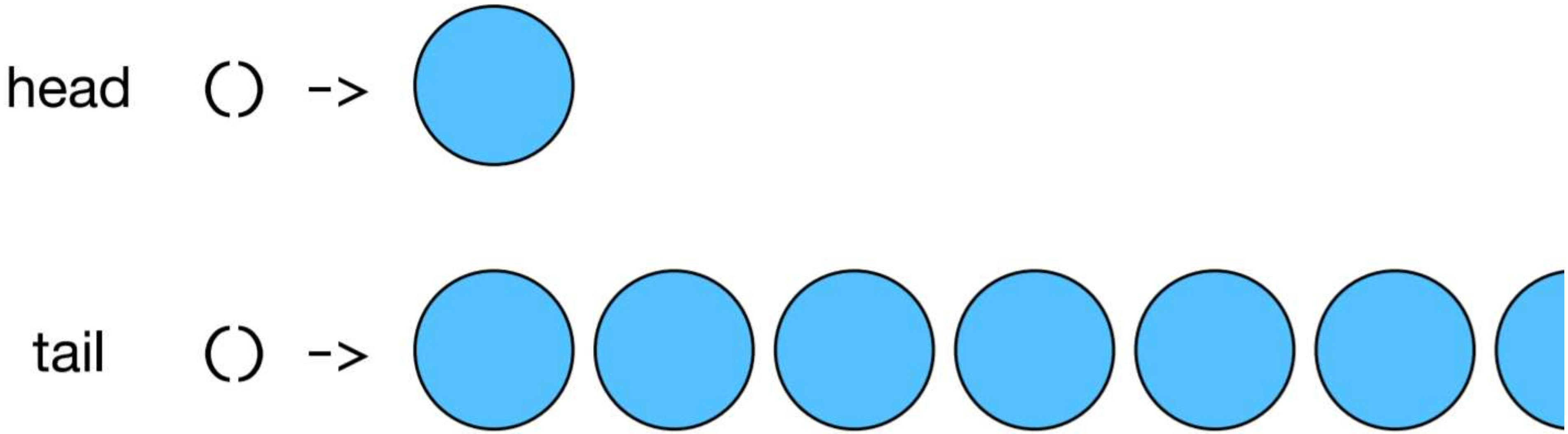
Delayed Data

- A lambda expression not only stores the function to invoke, but also data from the enclosing environment

```java
class DelayedData {
    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();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        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);
    }
}
```

Towards An Immutable Implementation

- What defines an infinite list?
  - The head value
  - The tail list
  - Whether the list is empty



Towards An Immutable Implementation

- Different `IFL`s representing different operations will have their own customized operations on the head and tail

```
class IFLImpl<T> implements IFL<T> {
    private final Supplier<T> head;
    private final Supplier<IFLImp<T>> tail;

    private IFLImpl(Supplier<T> head, Supplier<IFLImpl<T>> tail) {
        this.head = head;
        this.tail = tail;
    }

    boolean isEmpty() {
        return false;
    }
}
```

- Notice the use of `Suppliers` for delayed data

Implementation #2: Data Source Operations

```java
class IFLImpl<T> implements IFL<T> {
    private final Supplier<T> head;
    private final Supplier<IFLImpl<T>> tail;

    private IFLImpl(Supplier<T> head, Supplier<IFLImpl<T>> tail) {
        this.head = head;
        this.tail = tail;
    }

    static <T> IFLImpl<T> generate(Supplier<T> supplier) {
        Supplier<T> newHead = supplier;
        Supplier<IFLImpl<T>> newTail = () -> IFLImpl.generate(supplier);
        return new IFLImpl<T>(newHead, newTail);
    }

    static <T> IFLImpl<T> iterate(T seed, Function<T,T> next) {
        Supplier<T> newHead = () -> seed;
        Supplier<IFLImpl<T>> newTail = () -> IFLImpl.iterate(next.apply(seed), next)
        return new IFLImpl<T>(newHead, newTail);
    }
```

Implementation #2: Operation **forEach**

```
public void forEach(Consumer<T> consumer) {
    IFLImpl<T> curr = this;
    while (true) {
        consumer.accept(curr.head.get());
        curr = curr.tail.get();
    }
}
```

```
jshell> IFL.iterate(1, x -> x + 1)
```

- `IFL.iterate` returns an `IFLImpl` object and `forEach` method invoked; `curr` refers to itself
- `curr.head.get()` invokes the head supplier to get an element
- `curr.tail.get()` invokes the tail supplier to call `iterate`
  - Returns a new `IFLImpl` object with next element (new seed value captured), and the same tail supplier with new seed
- `curr` refers to this new `IFLImpl` object

Implementation #2: Operation **`forEach`**

- Unlike the previous implementation where states get updated e.g. `element` in the `IFL` object returned by `iterate`), each `IFLImpl` object is now immutable
  - `head` and `tail` suppliers are declared **`private final`**

```
jshell> IFL<Integer> ifl = IFL.iterate(1, x -> x + 1)
ifl ==> IFLImpl@6a41eaa2

jshell> ifl.forEach(System.out::println)
1
2
3
⋮

jshell> ifl.forEach(System.out::println)
1
2
3
⋮
```

Implementation #2: Operation `map`

```
public <R> IFLImpl<R> map(Function<T,R> mapper) {
    Supplier<R> newHead = () -> mapper.apply(head.get());
    Supplier<IFLImpl<R>> newTail = () -> tail.get().map(mapper);
    return new IFLImpl<R>(newHead, newTail);
}
```

- `map` returns a new `IFLImpl` object with the head supplier that
  - is triggered by a terminal operation (e.g. `forEach`) at the end of the stream pipeline
  - performs a `head.get()` to get the element upstream
  - apply the `mapper` function to obtain the mapped element
- `head.get()` initiates the upstream call (via subsequent `head.get()` until the data source if reached
- The element is generated by the data source and returned (and operated on) progressively downstream

Implementation #2: Operation **filter**

```java
class IFLImpl<T> implements IFL<T> {
    private final Supplier<Optional<T>> head;
    private final Supplier<IFLImpl<T>> tail;

    private IFLImpl(Supplier<Optional<T>> head, Supplier<IFLImpl<T>> tail) {
        this.head = head;
        this.tail = tail;
    }

    static <T> IFLImpl<T> generate(Supplier<T> supplier) {
        Supplier<Optional<T>> newHead = () -> Optional.of(supplier.get());
        Supplier<IFLImpl<T>> newTail = () -> IFLImpl.generate(supplier);
        return new IFLImpl<T>(newHead, newTail);
    }

    static <T> IFLImpl<T> iterate(T seed, Function<T,T> next) {
        Supplier<Optional<T>> newHead = () -> Optional.of(seed);
        Supplier<IFLImpl<T>> newTail = () -> IFLImpl.iterate(next.apply(seed), next);
        return new IFLImpl<T>(newHead, newTail);
    }

    public void forEach(Consumer<T> action) {
        IFLImpl<T> curr = this;
        while (true) {
            curr.head.get().ifPresent(action);
            curr = curr.tail.get();
        }
    }

    public <R> IFLImpl<R> map(Function<T,R> mapper) {
        Supplier<Optional<R>> newHead = () -> head.get().map(mapper);
        Supplier<IFLImpl<R>> newTail = () -> tail.get().map(mapper);
        return new IFLImpl<R>(newHead, newTail);
    }
```

Implementation #2: Operation `filter`

```
public IFLImpl<T> filter(Predicate<T> predicate) {
    Supplier<Optional<T>> newHead = () -> head.get().filter(predicate);
    Supplier<IFLImpl<T>> newTail = () -> tail.get().filter(predicate);
    return new IFLImpl<T>(newHead, newTail);
}
```

- Notice that the head supplier makes use of `Optional`'s `filter` method:
  - *If a value is present, and the value matches the given predicate, returns an Optional describing the value, otherwise returns an empty Optional*
- Not be confused with the tail supplier's `filter` method which is implemented in `IFLImpl`

Implementation #2: Operation `limit`

- As for the stateful `limit` operation, the previous implementation maintained a state
- However, in the spirit of immutability, cannot maintain state
- *Hint:*
  - An `EmptyList` is a `IFL` (or is a `IFLImpl`?)
    - Rather than create a new `IFLImpl` everytime, `limit` should create a new `EmptyList` when appropriate
    - The **`boolean`** method `isEmpty()` returns **`true`** so that terminal operations know when to stop the stream pipeline operation
- *Once again, the rest is up to you...*

Lecture Summary

- Appreciate the notion of lazy evaluation and compare it with eager evaluation
- Understand how the Java memory model supports the mechanism behind
  - inner classes and closures
  - delayed data using `Supplier`'s `get()` method
- A note about our implementations so far
  - For simplicity, type wildcards are ignored
  - Note that some of these operations are incomplete and subjected to fine-tuning