CS2030 Lecture 9

Infinite List
— *A tale of two list implementations*

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

Semester 1 2018 / 2019

Lecture Outline

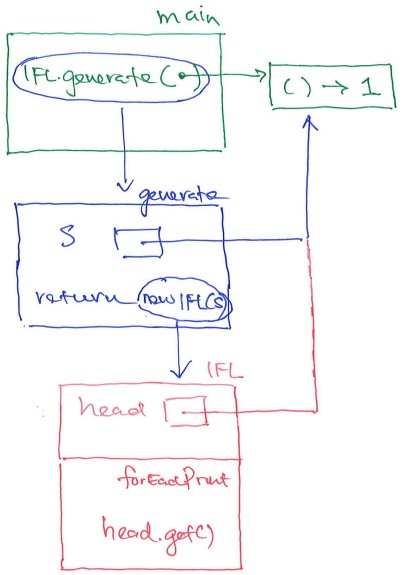
- Designing an infinite list
  - Using delayed data via suppliers
  - Using inner classes

Infinite List of One Element?

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

class IFL<T> {
    private Supplier<T> head;

    IFL(Supplier<T> s) {
        this.head = s;
    }

    static <U> IFL<U> generate (Supplier<U> s) {
        return new IFL<U>(s);
    }

    void forEachPrint() {
        while (true) {
            System.out.println(head.get());
        }
    }

    public static void main(String[] args) {
        IFL.generate(() -> 1)
            .forEachPrint();
    }
}
```

- Is this really an infinite list.. or one element accessed infinitely?

Infinite List of One Element?

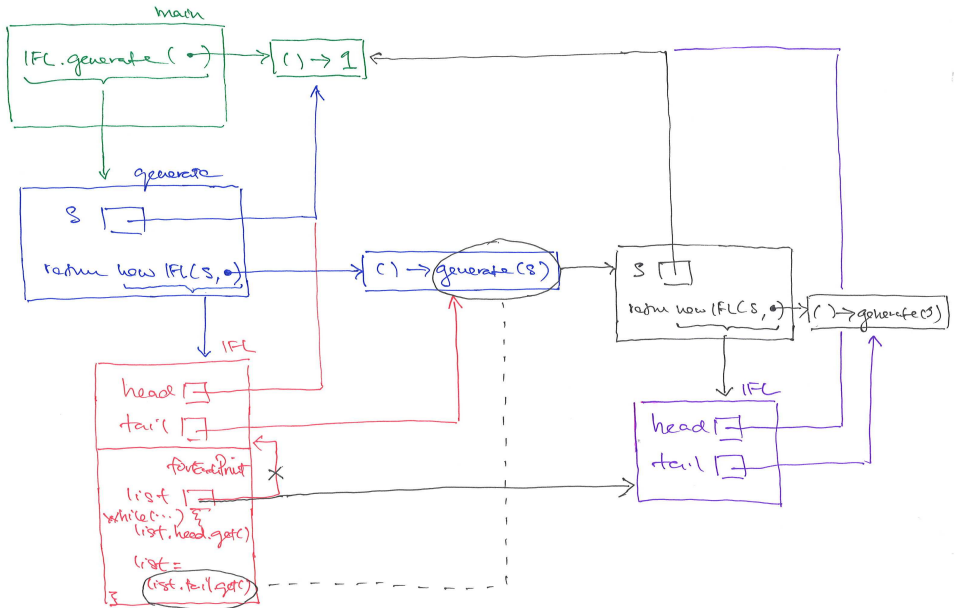


Infinite List of One Element

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

class IFL<T> {
    private Supplier<T> head;
    private Supplier<IFL<T>> tail;

    IFL(Supplier<T> s,
            Supplier<IFL<T>> next) {
        this.head = s;
        this.tail = next;
    }

    static <U> IFL<U> generate(
            Supplier<U> s) {
        return new IFL<U>(s,
                () -> generate(s));
    }

    void forEachPrint() {
        IFL<T> list = this;
        while (true) {
            System.out.println(
                    list.head.get());
            list = list.tail.get();
        }
    }

    public static void main(String[] args) {
        IFL.generate(() -> 1)
            .forEachPrint();
    }
}
```

- Define `IFL<T>` list as a `T` head, followed by a `IFL<T>` tail
- `list.tail.get()` generates the next `IFL` instance for access

Infinite List of One Element

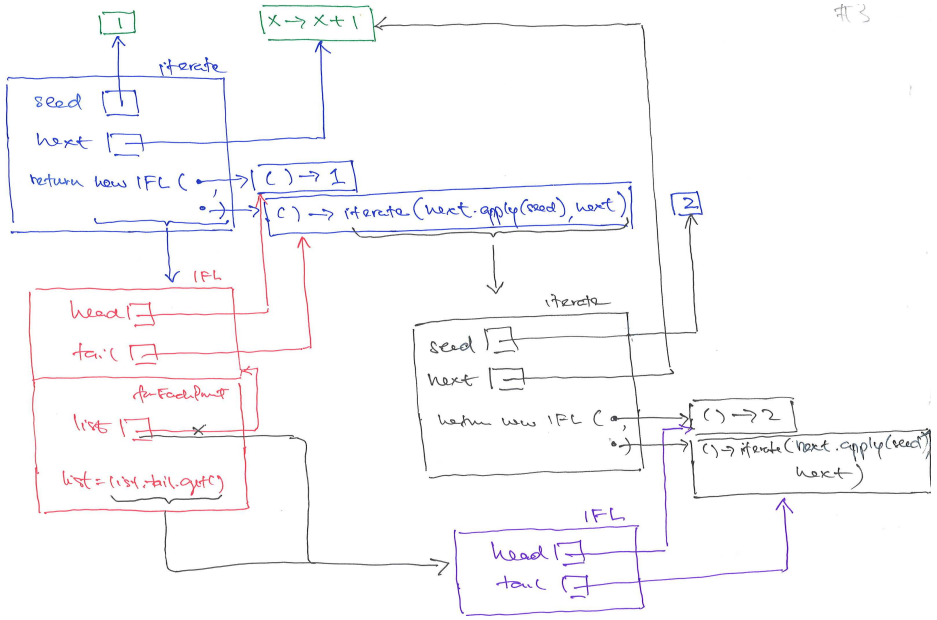


Iterating an Infinite List

```
import java.util.function.Function;

    static <U> IFL<U> iterate (U seed, Function<U,U> next) {
        return new IFL<U>(() -> seed,
                () -> iterate(next.apply(seed), next));
    }

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

- For `iterate(next.apply(seed), next)`,
  - A new `seed` value is passed to each `iterate` method via `next.apply(seed)`
  - Each `iterate` method generates an `IFL` using a new lambda `() -> seed` as the `head`
- `head.get()` gives different values depending on the lambda associated with `head`

Iterating an Infinite List



Defining a Terminal Operation

```
import java.util.function.Consumer;

    void forEach(Consumer<T> consumer) {
        IFL<T> list = this;
        while (true) {
            consumer.accept(list.head.get());
            list = list.tail.get();
        }
    }

        IFL.iterate(1, x -> x + 1)
            .forEach(x -> {
                    System.out.println(x);
                    (new Scanner(System.in)).nextLine();
            });
```


Limiting an Infinite List

```
    IFL<T> limit(int n) {
        if (n > 1) {
            return new IFL<T>(head, () -> tail.get().limit(n - 1));
        } else {
            return new IFL<T>(head, () -> new EmptyList<T>());
        }
    }

    boolean isEmpty() {
        return false;
    }

        IFL.iterate(1, x -> x + 1)
            .limit(3)
            .forEach(System.out::println);
```

- Need to take care of the case of an empty list

Limiting an Infinite List

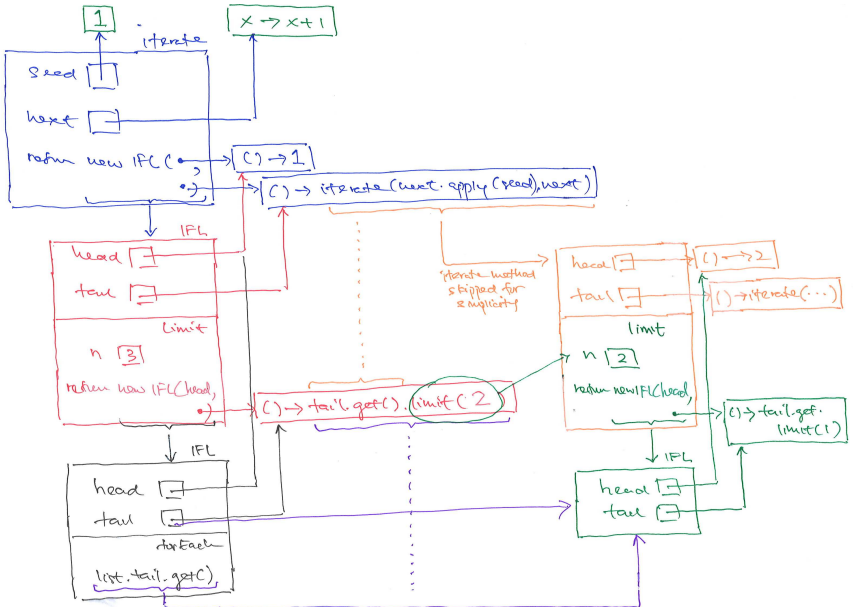
- One way is to make `EmptyList` a sub-class of `IFL`

```
class EmptyList<T> extends IFL<T> {

    EmptyList() { }

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

- And include the empty constructor in `IFL` class

```
    protected IFL() { }
```


Limiting an Infinite List



Filtering an Infinite List

```
import java.util.function.Predicate;

    private Predicate<T> headPredicate;

    IFL(Supplier<T> s, Supplier<IFL<T>> next, Predicate<T> predicate)
        this.head = s;
        this.tail = next;
        this.headPredicate = predicate;
    }

    IFL<T> filter(Predicate<T> predicate) {
        return new IFL<T>(head,
                () -> tail.get().filter(predicate), predicate);
    }
```

- `filter` method creates a lambda to filter new instances of `IFL`
- `predicate` needs to be passed to subsequent `IFL`s for filtering

Filtering an Infinite List

```
    boolean isHeadFiltered() {
        if (headPredicate != null) {
            return !headPredicate.test(head.get());
        } else {
            return false;
        }
    }

    void forEach(Consumer<T> consumer) {
        IFL<T> list = this;
        while (!list.isEmpty()) {
            if (!isHeadFiltered()) {
                consumer.accept(list.head.get());
            }
            list = list.tail.get();
        }
    }

        IFL.iterate(1, x -> x + 1)
            .filter(x -> x % 2 == 0)
            .forEach(System.out::println);
```

- `predicate` storedin `headPredicate` so as to use it in terminals to determine if the corresponding `IFL` gets filtered

Delayed Data via Suppliers and Method Calls

- Consider the following

```
IFL.iterate(1, x -> x + 1)
    .filter(x -> x % 2 == 0)
    .limit(3)
    .forEach(System.out::println);
```

- When the terminal operation (e.g. `forEach`) is reached and the `head` processed, subsequent `tail.get()`s initates the upstream movement back to the data source
- The next element is generated for the downstream process of applying stream operations via successive method calls
- However, lambdas are created when moving downstream in preparation for the next upstream movement
- Since these lambdas are not processed this until later, is there a way to improve on this semi-lazy approach?

Using Anonymous Inner Classes

- What defines an infinite list?
  - Whether the list is empty
  - The head value if it is not empty
  - The tail list if it is not empty
- Different `IFL`s representing different operations will have their own customized implementations for the above
- Start with an abstract `IFL` class

```
abstract class IFL<T> {
    abstract boolean isEmpty();
    abstract T head();
    abstract IFL<T> tail();
}
```

Data source: **generation**

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

    static <T> IFL<T> generate(Supplier<T> s) {
        return new IFL<T>() {
            boolean isEmpty() {
                return false;
            }
            T head() {
                return s.get();
            }
            IFL<T> tail() {
                return IFL.generate(s);
            }
        };
    }
```

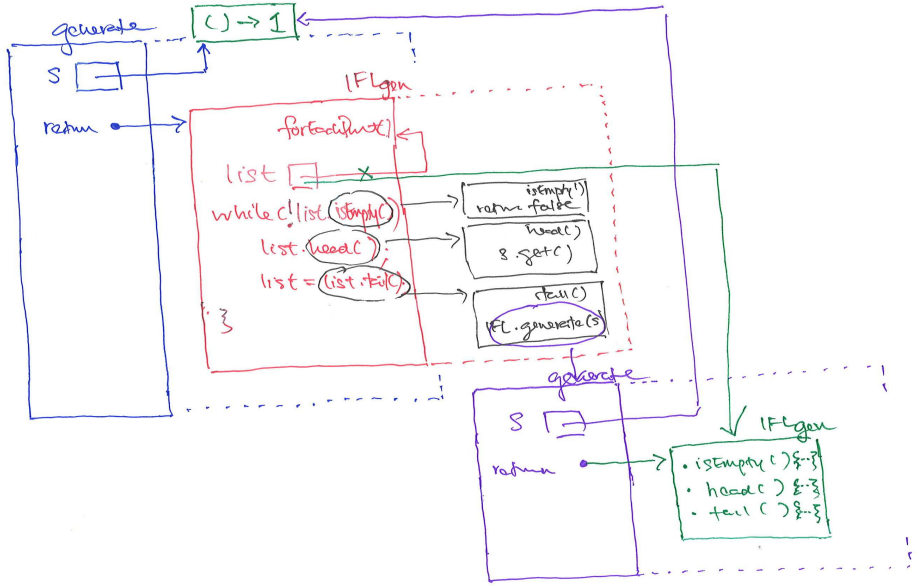
- The `generate` method returns a new concrete implementation of the abstract `IFL` class with the methods `isEmpty()`, `head()` and `tail()` defined

Data source: **generation**

```
    void forEachPrint() {
        IFL<T> list = this;
        while (!list.isEmpty()) {
            System.out.println(list.head());
            list = list.tail();
        }

        IFL.generate(() -> 1)
            .forEachPrint();
```

- `forEachPrint()` is called from within the first concrete $\texttt{IFL}_{gen}$
- `list = list.tail()` references a new concrete instance $\texttt{IFL}_{gen}$ of the abstract `IFL`
- Subsequent `isEmpty()`, `head()` and `tail()` is called from within this new instance
- All terminals behave in much the same way

Data source: **generation**



Intermediate Operation: **limit**

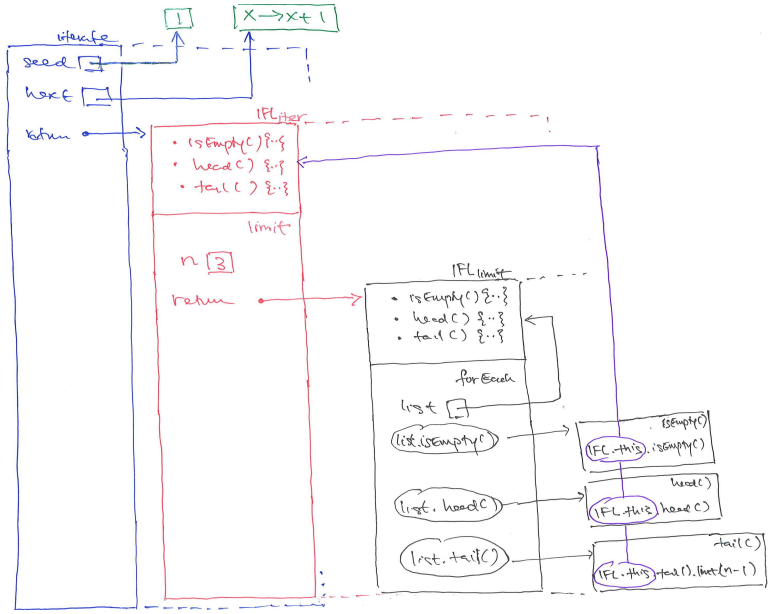
```
    IFL<T> limit(int n) {
        return new IFL<T>() {
            boolean isEmpty() {
                if (n > 0)
                    return IFL.this.isEmpty();
                else
                    return true;
            }
            T head() {
                return IFL.this.head();
            }
            IFL<T> tail() {
                return IFL.this.tail().limit(n- 1);
            }
        };
    }
```

- `IFL.this` refers to the enclosing `IFL` class scope
- `isEmpty()` methods looks at the current `n` to decide if it can declare emptiness; otherwise checks with upstream operation
- `IFL.this.tail().limit(n - 1)` applies the subsequent limit to the `tail` generated by the upstream operation

Intermediate Operation: `limit`

```
static <T> IFL<T> iterate(T seed, Function<T, T> next) {
    return new IFL<T>() {
        boolean isEmpty() {
            return false;
        }
        T head() {
            return seed;
        }
        IFL<T> tail() {
            return IFL.iterate(next.apply(seed), next);
        }
    };
}

void forEach(Consumer<T> consumer) {
    IFL<T> list = this;
    while (!list.isEmpty()) {
        consumer.accept(list.head());
        list = list.tail();
    }
}

    IFL.iterate(1, x -> x + 1)
        .limit(3)
        .forEach(System.out::println);
```


Intermediate Operation: `limit`



Intermediate Operation: `filter`

```
IFL<T> filter(Predicate<T> p) {
    return new IFL<T>() {
        IFL<T> list = IFL.this;
        boolean isEmpty() {
            while (!list.isEmpty()) {
                if (p.test(list.head())) {
                    return false;
                } else {
                    list = list.tail();
                }
            }
            return true;
        }
        T head() {
            return list.head();
        }
        IFL<T> tail() {
            return list.tail().filter(p);
        }
    };
}
```


Lecture Summary

- Understand the mechanism behind delayed data and invocation using `Suppliers`
- Understand the mechanism involving inner anonymous classes and scoping rules in the context of inner classes
- Appreciate how we have transited from a semi-lazy implementation to a strictly-lazy one
- There are still other subtle issues like **caching** that might need to be addressed
  - How to minimize access to the same element of the stream, i.e. method `head()` should execute a `head.get()` on the data source only once, and subsequent calls to `head()` should return a cached value