CS2030 Lecture 10

Infinite List

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

Semester 2 2018 / 2019

Lecture Outline

- Designing an infinite list to mimic Java streams
  - Each non-terminal stream operation generates an infinite list `IFL` of generic type `T`
    - Data source: `generation`
    - Data source: `iterate`
    - Stateless intermediate operation: `map`
    - Stateful intermediate operation: `limit`
    - Terminal: `forEach`
  - Use of anonymous inner classes with customized implementations of method `get()`
  - Mental modeling with extended Java OOP constructs
  - For simplicity, type wildcards are ignored

Implement Operation `generate`

- Consider the following example:

```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();
            }
        };
    }

    public T get();
}
```

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

jshell> ifl.get()
$4 ==> 1

jshell> ifl.get()
$5 ==> 1
```

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

Implement Operation `generate`

Your notes...

Implement Operation `iterate`

- `iterate` is less trivial and requires the seed, as well as the function to generate the next element, to be stored

```
static <T> IFL<T> iterate(T seed, Function<T, T> next) {
    return new IFL<T>() {
        T element = seed;

        public T get() {
            element = next.apply(element);
            return element;
        }
    };
}
```

- Sample run

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

jshell> ifl.get()
$11 ==> 2

jshell> ifl.get()
$12 ==> 3
```

- Notice that the first element is missing?

Implement Operation `iterate`

- The fix is to maintain an instance variable of type `Function<T,T>` such that
  - the first application of the function returns the first element, i.e. the seed
  - subsequent application then applies the iterate function on the element

```
static <T> IFL<T> iterate(T seed, Function<T, T> next) {
    return new IFL<T>() {
        T element = seed;
        Function<T, T> func = x -> {
            func = next;
            return element;
        };

        public T get() {
            element = func.apply(element);
            return element;
        }
    };
}
```

Implement Operation `iterate`

Your notes...

Implement Operation `map`

- The `map` operation takes an element of type `T` and returns a mapped element of type `R`
- The function definition looks something like this

```
public <R> IFL<R> map(Function<T, R> mapper) {
    return new IFL<R>() {      has a get method that does some mapping
        public R get() {
            return mapper.apply(...);
        }
    };
}
```

- However, such a method should not be defined in the interface
- Avoid the use of **`default`** methods in interfaces
- The trick is to define an **`abstract`** class that implements the interface and create objects of this abstract class

Implement Operation map

```
interface IFL<T> {

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

    static <T> IFL<T> iterate(T seed, Function<T, T> next) {
        return new IFLImpl<T>() {
            T element = seed;
            Function<T, T> func = x -> {
                func = next;
                return element;
            };

            public T get() {
                element = func.apply(element);
                return element;
            }
        };
    }

    public T get();
    public <R> IFL<R> map(Function<T, R> mapper);
}
```

Implement Operation **map**

- The abstract class `IFLImpl` is defined as follows:

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

- In the argument of `mapper.apply(IFLImpl.this.get())`
  - `IFLImpl.this` refers to the enclosing `IFLImpl` class scope

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

jshell> ifl.get()
$8 ==> 2

jshell> ifl.get()
$9 ==> 4
```

Implement Operation `map`

Your notes...

Implement Operation **forEach**

- `forEach` is a terminal that repeatedly performs a `get()` on the previous operation and applies (`accept`) the `action` on each element

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

```
jshell> IFL<Integer> ifl = IFL.iterate(1, x -> x + 1).map(x -> x * 2)
ifl ==> IFLImpl$1@6ddf90b0

jshell> ifl.forEach(System.out::println)
2
4
6
8
⋮
```

- Retain the infinite loop in the `forEach` method for now

Implement Operation **`forEach`**

Your notes...

Implement Operation `limit`

- `limit` is a stateful operation, so we need to maintain a state
- As long as within the limit, keep executing `get()` on the previous operation
- When the limit is up, what should we return?
  - The most ideal would be to return `Optional.empty()`
  - So the `get()` method of `limit` should have a return type of `Optional<T>`
    - This means that the abstract method `get()` should be redefined as

      ```
      public Optional<T> get();
      ```

    - All method implementations should conform to this new specification!

Implement Operation `limit`

- Modifications to infinite list interface

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

static <T> IFL<T> iterate(T seed, Function<T, T> next) {
    return new IFLImpl<T>() {
        T element = seed;
        Function<T, T> func = x -> {
            func = next;
            return element;
        };

        public Optional<T> get() {
            element = func.apply(element);
            return Optional.of(element);
        }
    };
}
```

Implement Operation `limit`

- Modifications to infinite list implementor

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

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

Implement Operation `limit`

```java
public IFL<T> limit(long n) {
    if (n < 0) {
        throw new IllegalArgumentException("" + n);
    } else {
        return new IFLImpl<T>() {
            long remaining = n;
            public Optional<T> get() {
                if (remaining == 0) {
                    return Optional.empty();
                } else {
                    --remaining;
                    return IFLImpl.this.get();
                }
            }
        };
    }
}
```

```
jshell> IFL.iterate(1, x -> x + 1).map(x -> x * 2).limit(3).forEach(System.out::println)
2
4
6
```

Implement Operation `limit`

Your notes...

Lazy Evaluation Revisited

- Consider

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

- The non-terminal operations `iterate`, `map` and `limit` each results in a new infinite list of type `T`
- Notice that no operation on these non-terminals is performed until a terminal operation (in this case `forEach()`) is called
- So a stream pipeline initiates with a terminal operation, and the upstream operations are applied

Other Noteworthy Implementation

```java
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));
    }
```

```java
    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
- Makes use of suppliers to generate the head and tail
- `list.tail.get()` generates the next `IFL` instance for access

Other Noteworthy Implementation

```java
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`

Other Noteworthy Implementation

- 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;
}
```

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

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

Other Noteworthy Implementation

- 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() { }
```

- Work out the rest of the operations and determine how *lazy* eacn operation can be

Lecture Summary

- Understand and model the mechanism behind delayed data and invocation in the spirit of `Supplier`'s `get()` method
- Understand and model the mechanism involving inner classes and scoping rules in the context of inner classes
- Appreciate the lazy evaluation behind our implementation
- Consider other possible implementations and the pros and cons of each