CS2030 AY18/19 SEM 2

WEEK 11 | 05 APRIL 19

TA GAN CHIN YAO

DISCLAIMER

Slides are made by me, unofficial, optional

Available to download at bit.ly/cs2030_gan

Slides (if any) will be uploaded on Friday weekly

Q1.

1. Implement a class `LazyInt` that encapsulates a lazily evaluated Integer value. A `LazyInt` is specified by a `Supplier`, such that when the value of the `LazyInt` is needed, the `Supplier` will be evaluated to yield the value. Otherwise, the evaluation is delayed as much as possible.

   `LazyInt` supports the following operations:

   - `map` returns a `LazyInt` consisting of the results of applying the given function to the value of this `LazyInt`.
   - `flatMap` returns a `LazyInt` consisting of the results of replacing the value of this `LazyInt` with the value of a mapped `LazyInt` produced by applying the provided mapping function to the value.
   - `get` returns the value of `LazyInt`.

   As an example, the expression below will return `200`.

```
new LazyInt(() -> 10)
    .map(x -> x * x)
    .flatMap(x -> new LazyInt(() -> x * 2))
    .get()
```

Q1.

Given the skeleton class with `import` statements omitted for brevity, complete the method bodies of `map` and `flatMap`.

```
class LazyInt {
    Supplier<Integer> supplier;

    LazyInt(Supplier<Integer> supplier) {
        this.supplier = supplier;
    }

    int get() {
        return supplier.get();
    }

    LazyInt map(Function<? super Integer, Integer> mapper) {
        // To complete
    }

    LazyInt flatMap(Function<? super Integer, LazyInt> mapper) {
        // To complete
    }
}
```

LazyInt here is basically a Supplier that can map()

Q1.

Ans

```
class LazyInt {
    Supplier<Integer> supplier;

    LazyInt(Supplier<Integer> supplier) {
        this.supplier = supplier;
    }

    int get() {
        return supplier.get();    returns an integer
    }

    LazyInt map(Function<? super Integer, Integer> mapper) {
       return new LazyInt(() -> mapper.apply(get()));
    }

    LazyInt flatMap(Function<? super Integer, LazyInt> mapper) {
       return new LazyInt(() -> mapper.apply(get()).get());
    }

    public static void main(String[] args) {
        System.out.println(new LazyInt(() -> 10)
                .map(x -> x * x)
                .flatMap(x -> new LazyInt(() -> x * 2))
                .get());
    }
}
```

Some common mistakes for method flatMap:

- return mapper.apply(get()) is no longer lazy since it forces the evaluation of mapper and supplier.
- return new LazyInt(mapper.apply(get()).supplier which is also not lazy

Q2.

2. Study the following implementation of an infinite list.

```
public interface IFL<T> {

    public 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;
            };

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

    public <R> IFL<R> map(Function<T, R> mapper);
    public void forEach(Consumer<T> action);
}
```

Q2.

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

    abstract Optional<T> get();
}
```

Q2. a.

(a) Modify the iterate method such that it now supports a condition to stop iterating.

IFL<Integer> if = IFL.iterate(0, i -> i < 2, i -> i + 1);

```
public 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;
        };

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

Original

```
public static <T> IFL<T> iterate(T seed, Predicate<T> p, 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);
                if(p.test(element)) {
                    return Optional.of(element);
                } else {
                    return Optional.empty();
                }
            }
    };

}
```

Ans

Q2. b.

(b) Suppose we call

$$\texttt{IFL.iterate(0, i -> i < 2, } \overbrace{\texttt{i -> i + 1}}^{e}\texttt{).map(f).map(g).forEach(c)}$$

where `f` and `g` are lambda expressions of type `Function` and `c` is a lambda expression of type `Consumer`. Let `e` be the lambda expression `i -> i + 1` passed to `iterate`. Write down the sequence of which the lambda expressions `e`, `f`, `g`, and `c` that are evaluated. Verify your answer. `f g c e f g c e`

```
IFL.iterate(0, x -> x < 2, x -> {
        System.out.print("e ");
        return x + 1;
        })
    .map(x -> {
        System.out.print("f ");
        return x + 2;
        })
    .map(x -> {
        System.out.print("g ");
        return x * 2;
        })
    .forEach(x -> System.out.print("c "));
```

Output:

f g c e f g c e

TA GAN CHIN YAO

Q2. c.

(c) Define method `concat` takes in two IFL objects, `ifl1` and `ifl2`, and creates a new `IFL` whose elements are all the elements of the first list `ifl1` followed by all the elements of the second list `ifl2`.

```
public static <T> LazyList<T> concat(LazyList<T> l1, LazyList<T> l2)
```

The elements in newly concatenated list must be lazily evaluated as well. For example, in

```
IFL<Integer> ifl1 = IFL.iterate(0, i -> i < 2, i -> i + 1);
IFL<Integer> ifl2 = IFL.iterate(5, i -> i < 8, i -> i + 2);
IFL<Integer> ifl3 = IFL.concat(ifl1, ifl2);
ifl3.forEach(x -> System.out.print(x + " "));
```

0, 1
5, 7

Being a lazy-evaluated, nothing is evaluated when `ifl3` is created. Thus, `concat` should not result in an infinite loop even if the list `ifl1` infinitely long. The elements are only evaluated when terminal operator `forEach` is called. In the example above, `0 1 5 7` will be printed.

Q2. c.

Ans:

```
public static <T> IFL<T> concat(IFL<T> list1, IFL<T> list2) {
    return new IFLImpl<T>() {
        IFLImpl<T> list = (IFLImpl<T>)list1;
        Optional<T> get() {
            Optional<T> element = list.get();
            if (!element.isPresent()) {
                list = (IFLImpl<T>)list2;
                element = list.get();
            }
            return element;
        }
    };

}
```

Q3.

3. The following depicts a classic tail-recursive implementation for finding the sum of values of n (given by $\sum_{i=0}^{n} i$) for $n \geq 0$.

```
static long sum(long n, long result) {
    if (n == 0) {
        return result;
    } else {
        return sum(n - 1, n + result);
    }
}
```

https://stackoverflow.com/questions/33923/what-is-tail-recursion

Java is not true Functional Programming Language. Java compiler does not optimize for tail recursion. Hence, StackOverflowError can occur. Languages that optimize for tail recursion will NOT encounter StackOverflow

In particular, the implementation above is considered **tail-recursive** because the recursive function is at the tail end of the method, i.e. no computation is done after the recursive call returns. As an example, `sum(100, 0)` gives `5050`.

However, this recursive implementation causes a `java.lang.StackOverflowError` error for large values such as `sum(100000, 0)`.

Although the tail-recursive implementation can be simply re-written in an iterative form using loops, we desire to capture the original intent of the tail-recursive implementation using delayed evaluation via the `Supplier` functional interface.

Q3.

We represent each recursive computation as a Compute<T> object. A Compute<T> object can be either:

- a recursive case, represented by a Recursive<T> object, that can be recursed, or
- a base case, represented by a Base<T> object, that can be evaluated to a value of type T.

As such, we can rewrite the above sum method as

```
static Compute<Long> sum(long n, long s) {
    if (n == 0) {
        return new Base<>(() -> s);
    } else {
        return new Recursive<>(() -> sum(n - 1, n + s));
    }
}
```

and evaluate the sum of n terms via the summer method below:

```
static long summer(long n) {
    Compute<Long> result = sum(n, 0);

    while (result.isRecursive()) {
        result = result.recurse();
    }

    return result.evaluate();
}
```

```
static long sum(long n, long result) {
    if (n == 0) {
        return result;
    } else {
        return sum(n - 1, n + result);
    }
}
```

Original

Q3. a.

(a) Complete the program by writing the Compute, Base and Recursive classes.

```
public interface Compute<T> {
    public boolean isRecursive();

    public Compute<T> recurse();

    public T evaluate();
}
```

```
public class Base<T> implements Compute<T> {
    private Supplier<T> supplier;

    public Base(Supplier<T> supplier) {
        this.supplier = supplier;
    }

    public boolean isRecursive() {
        return false;
    }

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

    public Compute<T> recurse() {
        throw new IllegalStateException("Invalid recursive call in base case");
    }

}
```

Q3. a.

```
public class Recursive<T> implements Compute<T> {
    private Supplier<Compute<T>> supplier;

    public Recursive(Supplier<Compute<T>> supplier) {
        this.supplier = supplier;
    }

    public boolean isRecursive() {
        return true;
    }

    public Compute<T> recurse() {
        return supplier.get();
    }

    public T evaluate() {
        throw new IllegalStateException("Invalid evaluation in recursive case");
    }
}
```

Q3. b.

(b) By making use of a suitable client class Main, show how the "tail-recursive" implementation is invoked

```
class Main {
    static long summer(long n) {
        Compute<Long> result = sum(n, 0);

        while (result.isRecursive()) {
            result = result.recurse();
        }

        return result.evaluate();
    }

    static Compute<Long> sum(long n, long s) {
        if (n == 0) {
            return new Base<>(() -> s);
        } else {
            return new Recursive<>(() -> sum(n - 1, n + s));
        }
    }

    public static void main(String[] args) {
        System.out.println(summer(new Scanner(System.in).nextLong()));
    }
}
```

Q3. c.

(c) Redefine the Main class so that it now computes the factorial of n recursively.

```
class Main {
    static long factorial(long n) {
        Compute<Long> result = fact(n, 1);

        while (result.isRecursive()) {
            result = result.recurse();
        }

        return result.evaluate();
    }

    static Compute<Long> fact(long n, long s) {
        if (n == 0) {
            return new Base<>(() -> s);
        } else {
            return new Recursive<>(() -> fact(n - 1, n * s));
        }
    }

    public static void main(String[] args) {
        System.out.println(factorial(new Scanner(System.in).nextLong()));
    }
}
```

Side note: What did we realise about part b and c? The implementations are the same, with the exception that one is + one is *. We can actually further abstract them into a Function if we want

SUMMARY CONCEPTS

- Use Supplier, map, flatmap, Consumer, forEach, iterate, Optional

QUESTIONS?