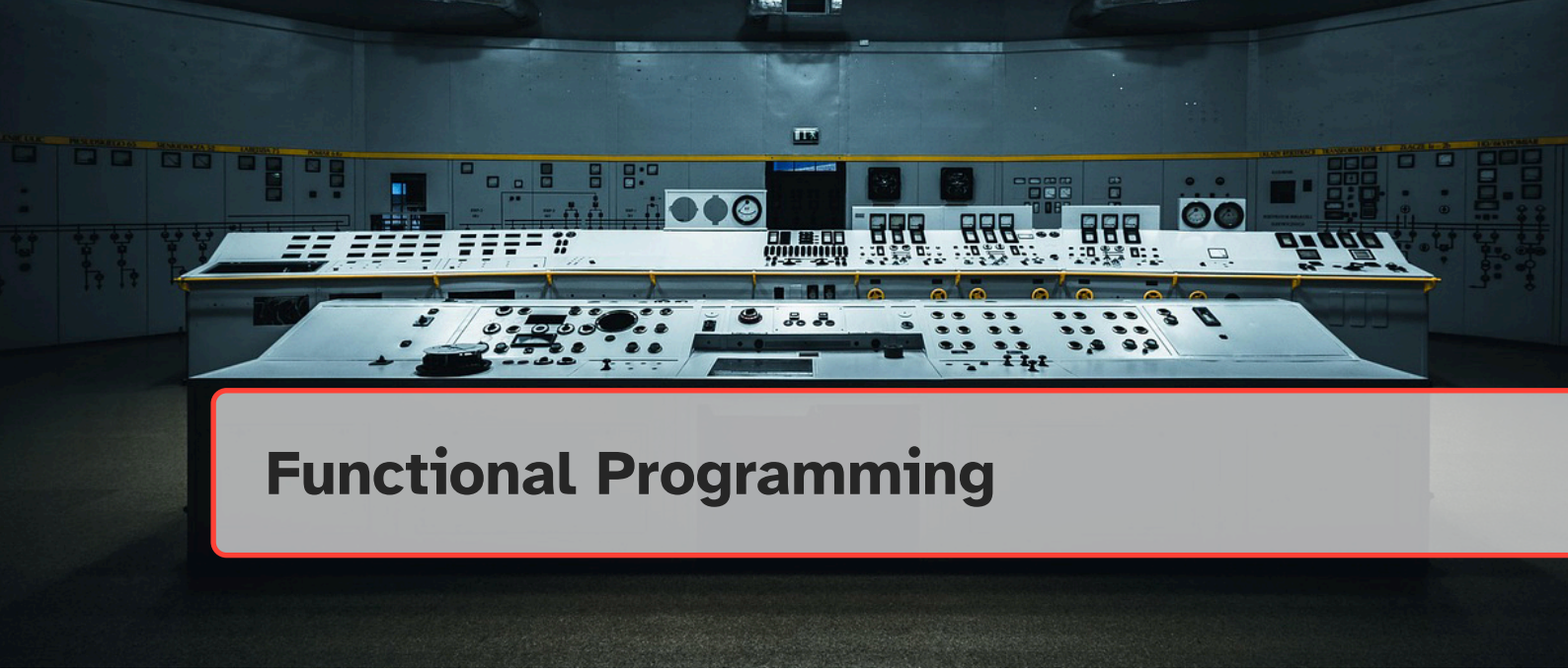


Functional Programming

Preliminary

The following questions are not going to be discussed. Answers is given without explanation. You are encouraged to discuss them on Ed.

1. Study the following method of finding the total money in our investment account if we close our account and after deducting a tax of 10%.

   Finding Invesment

   ```
   double totalMoney(Account acc) {
     if (acc == null || acc.isClosed()) {
       return 0.0;
     }
     double balance = acc.getBalance();
     double remaining = balance - (balance / 10);
     return remaining;
   }
   ```

   We want to rewrite the code in functional style. We will use the notation `X -> Y` to denote the type of a function where `X` is the type of the argument and `Y` is the type of the return value. Note that argument can be *implicit* (e.g., the target of invocation).

   We will use `Maybe` to help us with conditionals. Recap the relevant method descriptor in `Maybe`.

   - `Maybe<T> filter(BooleanCondition<? super T> cond)`
   - `<U> Maybe<U> map(Transformer<? super T, ? extends U> func)`
   - `<U> Maybe<U> flatMap(Transformer<? super T, ? extends Maybe<? extends U>> func)`
   - `<U extends T> orElse(U val)`

   (a) What is the type of the method `acc.isClosed()`?
   (b) Which method in `Maybe` can we use `acc.isClosed()`?
   (c) What is the type of the method `acc.getBalance()`?
   (d) Which method in `Maybe` can we use `acc.getBalance()`?
   (e) What is the "type" of the operation `balance - (balance / 10)`?
   (f) Which method in `Maybe` can we use the operation?
   (g) Rewrite the method `totalMoney` to use functional style with `Maybe`.

Discussion

The following questions are problem sets to be discussed during recitation. You are expected to attempt them before coming to the recitation. You may be randomly called to present your answer during recitation. You are encouraged to discuss them on Ed.

1. Study the method below where `Maybe<T>` is our version of `Optional<T>` [1].

   Finding Internship

   ```
   Maybe<Internship> match(Resume r) {
     if (r == null) {
       return Maybe.none();
     }

     Maybe<List<String>> optList = r.getListOfLanguages();
     List<String> list;

     if (optList.equals(Maybe.none())) {
       list = List.of();
     } else {
       list = optList.get(); // cannot call
     }

     if (list.contains("Java")) {
       return Maybe.of(findInternship(list));
     } else {
       return Maybe.none();
     }
   }
   ```

   Rewrite the method match using functional style using `Maybe<T>`. Recap that functional style is a style such that

   - it consists of only a single return statement;
   - it does not contain any selection statement/expression logic besides those internally provided by the `Maybe` APIs;
   - it does not contain repetition/recursion logic besides those internally provided by the `Maybe` and/or `Stream` APIs;
   - it does not use additional classes/methods beyond those already used in the given code.

   Additionally, you should not use `null` or `Maybe::get`.

2. Consider the following interface `Producer<T>` from the notes. It is declared with `@FunctionalInterface` [2] annotation.

   Producer

   ```
   @FunctionalInterface
   public interface Producer<T> {
     T produce();
   }
   ```

   Further consider the following class `A`.

   A

   ```
   public class A {
     private int x;
   ```

```
  public A(int x) {
    this.x = x;
  }

  public int get() {
    // Line A
    return this.x;
  }
}
```

Draw the content of the stack and the heap at `// Line A` when we execute the following code snippet in `main`.

Code Snippet

```
A a = new A(5);
Producer<Integer> p = () -> a.get();
p.produce();
```

Note: Ignore the input parameter to `main`. Additionally, represent anonymous function with the compile-time type name as well as the lambda expression for the type name.

Exercise

The following questions are to be discussed on Ed as no answer will be given. ***Use only concepts that you have learnt****. You are expected to attempt them before the next lecture as the concepts may be used and/or mentioned in the lecture. The concept may also be tested for assessment.*

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

   Sum

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

   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)` returns 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 `Producer` functional interface.

   We represent each recursive computation as a `Compute<T>` object. A `Compute<T>` object can be either one of the following.

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

   Assume that `Producer<T>` from `cs2030s.fp` package is already provided.

(a) Implement the interface `Compute<T>` with a three methods.
   - `T evaluate()`
   - `Compute<T> step()`
   - `boolean isRecursive()`

(b) Implement the concrete class `Base<T>` that is a subtype of `Compute<T>`.

(c) Implement the concrete class `Recursive<T>` that is a subtype of `Compute<T>`.

(d) Rewrite the method `sum` to return `Compute<Long>` instead.

(e) Show how the tail-recursive implementation is updated into a non-recursive implementation.

Bibliography

[1] Oracle, "Java 25 API: Optional." [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Optional.html

[2] Oracle, "Annotation: FunctionalInterface." [Online]. Available: https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/lang/FunctionalInterface.html