1. **Adapted from 2019/20 CS2030 Final**

   Study the method below:

```
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 using `Maybe<T>` such that

   - it consists of only a single return statement;
   - it does not use additional external classes or methods beyond those already used in the given code below;
   - must not use `null` or `get`;
   - it does not contain `if`, `switch`, the ternary `? :` operators, or other branching logic besides those internally provided by `Maybe` APIs.

   Note that the specification and implementation details of the external classes `Resume` and `Internship` used in the method are not required to answer this question.

   Comments:

```
Maybe<Internship> match(Resume r) {
  return Maybe.of(r)
    .flatMap(x -> x.getListOfLanguages())
    .filter(l -> l.contains("Java"))
    .map(l -> findInternship(l));
}
```

2. Consider the interface `Producer<T>` from the notes

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

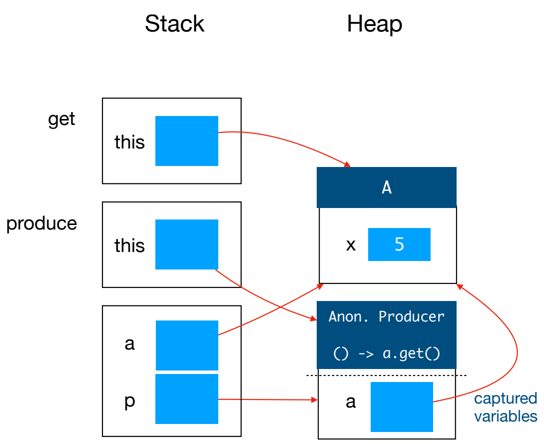
along with the class `A` below:

```
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 call the following:

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

Comments:



Past Year Questions

3. **Recitation 7 AY21/22 Sem 2.**

The following code 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);
  }
}
```

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 `Producer` functional interface.

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 `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();
}
```

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

Comments:

```
interface Compute<T> {
  boolean isRecursive();
  Compute<T> recurse();
  T evaluate();
}

class Base<T> implements Compute<T> {
  private Producer<T> producer;

  public Base(Producer<T> producer) {
    this.producer = producer;
  }

  @Override
  public boolean isRecursive() {
    return false;
  }

  @Override
  public T evaluate() {
    return producer.produce();
  }

  @Override
```

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

}

class Recursive<T> implements Compute<T> {
  private Producer<Compute<T>> producer;

  public Recursive(Producer<Compute<T>> producer) {
    this.producer = producer;
  }

  @Override
  public boolean isRecursive() {
    return true;
  }

  @Override
  public Compute<T> recurse() {
    return producer.produce();
  }

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

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

Comments:

```java
import java.util.Scanner;

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) {
    Scanner sc = new Scanner(System.in);
    System.out.println(summer(sc.nextLong()));
```

```
    sc.close();
  }
}
```

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

Comments:

```
import java.util.Scanner;

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

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    Compute<Long> result = factorial(sc.nextLong(), 1);
    while (result.isRecursive()) {
      result = result.recurse();
    }
    System.out.println(result.evaluate());
    sc.close();
  }
}
```