NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
FINAL ASSESSMENT FOR
Semester 2 AY2019/2020
PART 2 of 2

CS2030 Programming Methodology II

May 2020 Time Allowed 45 Minutes

INSTRUCTIONS TO CANDIDATES

1. This assessment is divided into two parts: Part 1 and Part 2.
2. This assessment paper contains 5 questions for Part 2.
3. Write all your answers in the answer boxes provided on Examplify.
4. The total marks for Part 2 is 40. Answer **ALL** questions.
5. This is a **OPEN BOOK** assessment. You are also free to refer to materials online.
6. All questions in this assessment paper use Java 11, unless otherwise specified.

No	Question	Marks
1	Optional	10
2	Header	5
3	Subtyping	8
4	Monad	9
5	Asynchornous	8
	Total	**40**

QUESTION 1

Question 1: Optional (10 points)

Study the method below.

```
String foo(String filename) {
  MyFile f = openFile(filename);
  if (f == null) {
    f = openFile("default.txt");
  }
  if (f != null) {
    Integer i = f.readNum();
    if (i < 10 && i >= 0) {
      return "The digit is " + i;
    }
  }
  return "Unable to read a single digit";
}
```

Rewrite the method below using a `Optional`. Your answer

- must consist of only a single return statement;
- must not use additional external classes or methods beyond those already used in the given code below;
- must not use `null` or the following `Optional`'s methods: `isEmpty`, `ifPresentOrElse`, `isPresent`, and `get`;
- must not contain `if`, `switch`, the ternary `? :` operators, or other branching logic besides those internally provided by `Optional` APIs.

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

A solution template is provided below:

```
String foo(String filename) {
  return Optional ..
    ;
}
```

You must only write the body of the method (including the keyword `return`) to obtain full marks.

Solution:

```
String foo(String filename) {
  return Optional.ofNullable(openFile(filename))
    .or(() -> Optional.ofNullable(openFile("default.txt")))
    .map(f -> f.readNum())
    .filter(i -> i < 10 && i >= 0)
    .map(i -> "The digit is " + i)
    .orElse("Unable to read a single digit");
}
```

The line `Optional.ofNullable(openFile(fileName)).or(() -> Optional.ofNullable(openFile(` is worth two marks. Many students did not manage to deal with opening the alternate file correctly, and many just ignored this.

For the rest of the operations `map`, `filter`, and `orElse` – each is worth two marks.

Question 1: Optional (10 points)

Study the method below. The method checks if a person with the given NRIC can enter a market based on the parity of the last digit of the NRIC and the current date.

```
boolean canEnter(NRIC nric) {
  if (nric == null) {
    throw new InvalidArgumentException();
  }
  Integer lastDigit = nric.lastDigit();
  if (lastDigit == null) {
    throw new InvalidArgumentException();
  }
  if (MyCalendar.currDate() % 2 == lastDigit % 2) {
    return true;
  } else {
    return false;
  }
}
```

Rewrite the method below using a `Optional`. Your answer

- must consist of only a single return statement;
- must not use additional external classes or methods beyond those already used in the given code below;
- must not use `null` or the following `Optional`'s methods: `isEmpty`, `ifPresentOrElse`, `isPresent`, and `get`;
- must not contain `if`, `switch`, the ternary `? :` operators, or other branching logic besides those internally provided by `Optional` APIs.

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

A solution template is provided below:

```
boolean canEnter(NRIC nric) {
  return Optional ..
    ;
}
```

You must only write the body of the method (including the keyword `return`) to obtain full marks.

Solution:

```
boolean canEnter(NRIC nric) {
  return Optional.ofNullable(nric)
    .map(nric -> nric.lastDigit())
    .map(lastDigit -> lastDigit % 2 == MyCalendar.currDate() % 2)
    .orElseThrow(() -> new InvalidArgumentException());
}
```

You are awarded 2 marks each for `Optional.ofNullable(nric)` and `map(x -> x.lastDigit())`; 3 marks each for `.map(x -> MyCalendar.currDate() % 2 == x % 2)` and `orElseThrow(() -> new InvalidArgumentException());`.

Many students incorrectly follow the order of throwing exceptions and wrote:

```
return Optional.ofNullable(nric)
    .orElseThrow(() -> new InvalidArgumentException())
    .map(x -> x.lastDigit())
    .orElseThrow(() -> new InvalidArgumentException());
    .map(x -> MyCalendar.currDate() % 2 == x % 2)
```

But the type does not match since orElseThrow on an Optional<T> returns T so we cannot continue to chain anymore.

To get around this, some students wrote:

```
return Optional.ofNullable(Optional.ofNullable(Optional.ofNullable(nric)
    .orElseThrow(() -> new InvalidArgumentException()))
    .map(x -> x.lastDigit())
    .orElseThrow(() -> new InvalidArgumentException()))
    .map(x -> MyCalendar.currDate() % 2 == x % 2)
```

to match the type, but this is way too complicated. The key idea behind Optional is that we can safely chain method calls without worrying about nulls so we only need to handle exception at the end. I take off one mark for this type of solution.

Another correct solution is the following:

```
return Optional.ofNullable(nric)
    .map(x -> x.lastDigit())
    .map(x -> x%2)
    .orElseThrow(() -> new InvalidArgumentException())
    == MyCalendar.currDate()%2;
```

Question 1: Optional (10 points)

Study the method below:

```
Optional<Internship> match(Resume r) {
  if (r == null) {
    return Optional.empty();
  }
  Optional<ArrayList<String>> optList = r.getListOfLanguages();
  List<String> list;
  if (optList.isEmpty()) {
    list = new ArrayList<String>();
  } else {
    list = optList.get();
  }
  if (list.contains("Java")) {
    return Optional.ofNullable(findInternship(list));
  } else {
    return Optional.empty();
  }
}
```

Rewrite the method using `Optional` 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`, `Optional`'s `isEmpty()`, `isPresent()`, `ifPresentOrElse`, and `get()` method;
- it does not contain `if`, `switch`, the ternary `? :` operators, or other branching logic besides those internally provided by `Optional` 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.

A solution template is provided below:

```
Optional<Internship> match(Resume r) {
  return Optional...
    ;
}
```

You must only write the body of the method (including the keyword `return`) to obtain full marks.

Solution:

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

Calling `Optional.ofNullable` and `filter` correctly would get you two marks each. `flatMap` and `map` would get you three marks each. You will get partial marks for calling the wrong methods (`map` instead of `flatMap` etc).

Note that, for this question, we do not have to create a dummy list, if the original list is `null`, since `Optional` would take care of that special case for us. Minus 1 point if you fail to realize that.

QUESTION 2

Question 2: Header (5 points)

Frustrated by the limitations of Java's Stream API, Ah Beng sent a proposal to the Java Executive Committee (JEC) to propose adding a new method to Java's Stream API called `merge`. The method `merge` works as follows. Suppose we call `s1.merge(s2, lambda)`, where `s1` and `s2` are streams. The method `merge` would return a new stream `s3`. The first element of `s3` is the result of applying the lambda expression `lambda` to the first element of `s1` and `s2`. The second element of `s3` is the result of applying `lambda` to the second element of `s1` and `s2`, and so on.

The example below shows how Ah Beng intended the `merge` method to be use.

```
jshell> Stream<Integer> s1 = Stream.of(1, 2, 3)
jshell> Stream<String> s2 = Stream.of("hello", "world")
jshell> BiFunction<Number, Object, String> lambda = (x, y) -> x + ":" + y
jshell> Stream<Object> s3 = s1.merge(s2, lambda)
jshell> s3.toArray()
$.. ==> ["1:hello", "2:world", "3:null"]
```

Ah Beng does not have to implement `merge`, but he has to provide the JEC with the method header for the API, specifying the type parameters (if necessary), the return type, the name, and the type of each parameter.

To convince the JEC that he knows what he is doing, Ah Beng has to come up with the most flexible method header to cater to different usage scenarios. Since you have taken CS2030, Ah Beng came to you for help.

Write down what you think the method header for the new `merge` method for `Stream<T>` should be.

Solution: This question assesses if you know how to properly design an API using generics and PECS principles.

The answer should look something like this:

```
<S,R> Stream<R> merge(Stream<? super S>, BiFunction<? super T, ? super S,
? extends R> lambda)
```

Two type parameters should be involved for this method to be as general as possible – we wish to merge a `Stream<T>` and a `Stream<S>` into a `Stream<R>`. (This operation is sometimes known as *zip*).

We look for several components in your answer:

- **Type parameter (1 mark)**: Your answer should declare the type parameter `<S,R>`. You can of course use other symbols, but if you include `T` you get 0.5 marks only. If you declare only one parameter, you get 0.5 mark only. Some students declare this method as static, in which case, the type parameter declaration must include `T` to get full marks.
- **Return type (1 mark)**: The return type should be `Stream<R>`. If you return `Stream<T>` you get 0 marks.
- **Type of first argument (1 mark)**: The first argument should be `Stream<S>` or `Stream<? extends S>`. You can use another name of the type argument but it cannot be `T` and cannot be the same as the return type. If you write `Stream<? extends T>` however, at least you showed that you know PECS, and you get 0.5 marks. If you write `Stream<?`

`super S>`, then you confused between producer and consumer, and you get 0.5 marks only.

- **Type of the second argument (2 marks)**: If you get the order of type parameters correct, you get 0.5 marks. To get full marks, you need to apply PECS correctly.

Question 2: Header (5 points)

Frustrated by the limitations of Java's Stream API, Ah Lian sent a proposal to the Java Executive Committee (JEC) to propose adding a new method to Java's Stream API called `doubleReduce`. The method `doubleReduce` is an extension of the `reduce` method, and it performs a reduction on the elements of the calling stream, using the provided `identity` and `accumulator`, and is equivalent to:

```
U result = identity;
for (T i : this stream)
  for (T j : this stream)
    result = accumulator.apply(result, i, j)
return result;
```

The example below shows how Ah Lian intended the `doubleReduce` method to be use.

```
jshell> Stream.of(1,2,3).doubleReduce("", s -> (x,y) -> s + x + ":" + y + " ");
$.. ==> "1:1 1:2 1:3 2:1 2:2 2:3 3:1 3:2 3:3 "
```

Ah Lian does not have to implement `doubleReduce`, but he has to provide the JEC with the method header for the API, specifying the the type parameters (if necessary), the return type, the name, and the type of each parameter.

To convince the JEC that he knows what he is doing, Ah Lian has to come up with the most flexible method header to cater to different usage scenarios. Since you have taken CS2030, Ah Lian came to you for help.

Write down what you think the method header for the new `doubleReduce` method for `Stream<T>` should be.

Solution: The answer should look something like this:

```
<R> R doubleReduce(R identity, Function<? super R, ? extends BiFunction<?
super T, ? super T, ? extends R>> accumulator)
```

First, some students are confused by the accumulator which takes in three arguments in the pseudocode, vs. the curried lambda expression given in how `doubleReduce` should be used. The pseudocode is just what it is – it is not the actual implementation of `doubleReduce`. (note: this stream is not even real Java syntax)

There are several components we look for in your answer.

- **Return type (0.5 marks):** The return type should be a general generic type `R`. You can, of course, use other letters (many of you used `U`). If you return `T`, however, you get 0.
- **Type parameter declaration (0.5 marks):** You should declare the type parameter `<R>` (and it should be consistent with the return type). If you include `T` in type parameter declaration, it is wrong and you will get 0 for this component.
- **First argument (1 marks):** The first argument should be of type `R`. If your first argument is of type `T`, you get 0.5 marks. If it is `String`, then you get 0 – your method would not be general enough!

- **Second argument (3 marks):** This should be a curried version of `Function<R, BiFunction<T, T, R>>`. One mark is allocated to the correct ordering of the types to `Function` and `BiFunction`. Another one to the use of `Function` and `BiFunction` (Use of `TriFunction` is accepted here too). The final mark is allocated to the use of PECS – so writing `Function<R,BiFunction<` would only get you two marks, to get full marks, you need at least: `Function<U, BiFunction<? super T, ? super T, U>>`.

Question 2: Header (5 points)

Frustrated by the limitations of Java's Stream API, Ah Kow sent a proposal to the Java Executive Committee (JEC) to propose adding a new method to Java's Stream API called `nestedMap`. The method `nestedMap` is an extension of the `map` method, and it applies a given lambda expression on the elements of the calling stream.

Ah Kow intended

```
s.nestedMap(lambda)
```

to be equivalent to:

```
s.map(i -> s.flatMap(j, (i,j) -> lambda.apply(i,j)));
```

Ah Kow does not have to implement `nestedMap`, but he has to provide the JEC with the method header for the API, specifying the type parameters (if necessary), the return type, the name, and the type of each parameter.

To convince the JEC that he knows what he is doing, Ah Kow has to come up with the most flexible method header to cater to different usage scenarios. Since you have taken CS2030, Ah Kow came to you for help.

Write down what you think the method header for the new `nestedMap` method for `Stream<T>` should be.

Solution: There is a correction during the final, in which the equivalent expression is corrected to:

```
s.map(i -> s.flatMap(j -> lambda.apply(i,j)));
```

The header for the method `nestedMap` should look like this:

```
<R> Stream<Stream<R>> nestedMap(BiFunction<? super T, ? super T, ? extends S
```

There are several components we look for in your answer when we grade:

Return type (1.5 marks): The return type should be the same as what is being returned by `map` – which is a `Stream` of something. Any answer that returns a non-`Stream` will receive 0 marks. But, what does this `Stream` contains? Each element in `s` is mapped to the output from `flatMap`, which is another stream. So, the return type should be a `Stream` of `Stream`.

If you answer looks something like `Stream<R>`, you will get 0.8 marks at most. If your return type has type parameter `T`, then it is not general enough since `map` should allow transformation of `T` to another type – you will get penalized. If your return type contains wildcards, then you are not using wildcards correctly and will be penalized.

Parameter (3 marks): Regardless of which version of equivalent expression you looked at, it should be clear that `lambda` is a `BiFunction` that takes in two parameters `i` and `j`, both of type `T`. You will get 1 mark at most if you used `Function` as the parameter of `nestedMap`.

What should this `BiFunction` returns? Note that the output from `lambda` is the output of the lambda expression passed into `flatMap`, and `flatMap` takes in a lambda expression that returns a `Stream`. As such, this `BiFunction` should return a stream `Stream<R>`.

Writing `BiFunction<T,T,Stream<R>>` would get you 2 marks. To get the full marks, your answer should use bounded wildcards following PECS principle.

Type Parameter (0.5 marks): Finally, don't forget to declare the type parameter `<R>`. If you include `T` in the type parameter declaration, you will not get any marks for this component.

QUESTION 3

Question 3: Subtyping (8 points)

Suppose we have Java classes `A1`, `A2`, .. `A5`, with the following class hierarchy:

`A5 <: A4 <: A3 <: A2 <: A1`

Consider the following method call

```
scanThis( Stream.of(1).map(x -> new A3()) );
```

Ignoring what `scanThis` does to the argument, what are the possible valid types for the argument of `scanThis` so that the statement above compiles without any warning or error?

Write down, one per line, up to 10 possible valid types (and only the valid types) of the argument `scanThis`.

Note that this question will be graded by a bot – it is important to write only one type per line. Do not include any extra text.

Solution:

```
Object
Stream<? extends A1>
Stream<? extends A2>
Stream<? extends A3>
Stream<? extends Object>
Stream<? super A1>
Stream<? super A2>
Stream<? super A3>
Stream<? super A4>
Stream<? super A5>
Stream<? super Object>
Stream<?>
Stream<A1>
Stream<A2>
Stream<A3>
Stream<Object>
```

This question assesses if you understand the notions of type conversion, subtyping, and variance of types. You get 0.9 marks for each correct answer, -0.4 for each wrong answer. We asked for up to 10, but you only need 9 to get full marks. Some students only wrote down the type parameters, you get a -1 mark penalty since the type is incomplete. We inserted the corresponding type for you before we passed it to the grader. Anything extra, such as variable name or function name (which is not part of the type), will get -1.

Let's look at the answers. The obvious one is `Stream<A3>` since `map` takes in a lambda that returns `A3`. If `scanThis` takes in `Stream<A3>` then it is an exact match with no type conversion.

`scanThis` can be defined with any type that is a supertype of `Stream<A3>`, however. Recall that we can assign a variable of a subtype to its supertype without the need to cast. What are the supertypes of `Stream<A3>`? `Object`, of course, since it is a supertype of every type. `Stream<?>` of course, since it is a supertype of any parameterized type of `Stream`.

Next, we invoke the covariant rule, which says that if `A3 <: T`, then `Stream<A3> <: Stream<? extends T>`. So, we have `Stream<? extends A3>`, `Stream<? extends A2>`, `Stream<? extends A1>`, and `Stream<? extends Object>`.

We also have the contravariant rule, which says that if `S <: A3`, then `Stream<A3> <: Stream<? super S>`. So we also have `Stream<? super A3>`, `Stream<? super A4>`, and `Stream<? super A5>`.

That's already 10 possibilities, more than enough to get you full marks. But wait! There are more!

Recall that `map` takes in `Function<? super T, ? extends U>` and returns `Stream<A3>`. So `scanThis` could be taking in `Stream<A2>`, `Stream<A1>` and `Stream<Object>` as well. Through type inference, `A2`, `A1`, or `Object` would be resolved to `U`, and `x -> new A2()` would still type check to `Function<? super T, ? extends U>`.

So, that's three more.

But wait, there are even more! Now that you see the expression can return `Stream<Object>`, `scanThis` should be able to take in supertype of `Stream<Object>`, not just `Stream<A3>`. What are the supertypes of `Stream<Object>`? It includes `Stream<? super A1>`, `Stream<? super A2>`, `Stream<? super Object>`, and `Stream<? super X>`, for `X` such as `Integer`, `String`, etc. as well! So there are many possibilities that you can include.

Common wrong answers are:

```
Stream<? extends A4>
Stream<? extends A5>
Stream<A4>
Stream<A5>
```

Note that for Question 3, we graded both your original answer and our modified answer (e.g., by inserting the missing Stream/Optional) and take the max of the two.

Question 3: Subtyping (8 points)

Suppose we have Java classes `A1`, `A2`, `A3`, and interfaces `I`, with the following subtype relationships:

`A3 <: A2 <: A1`

`A2 <: I`

Consider the following method call

```
process( Optional.of(1).map(x -> new A2()) );
```

Ignoring what `process` does to the argument, what are the possible valid types for the argument of `process` so that the statement above compiles without any warning or error?

Write down, one per line, up to 10 possible valid types (and only the valid types) of the argument `process`.

Note that this question will be graded by a bot – it is important to write only one type per line. Do not include any extra text.

Solution:

```
Object
Optional<? extends A1>
Optional<? extends A2>
Optional<? extends I>
Optional<? extends Object>
Optional<? super Object>
Optional<? super I>
Optional<? super A1>
Optional<? super A2>
Optional<? super A3>
Optional<?>
Optional<A1>
Optional<A2>
Optional<I>
Optional<Object>
```

This question assesses if you understand the notions of type conversion, subtyping, and variance of types. You get 0.9 marks for each correct answer, -0.4 for each wrong answer. We asked for up to 10, but you only need 9 to get full marks. Some students only wrote down the type parameters, you get a -1 mark penalty since the type is incomplete. We inserted the corresponding type for you before we passed it to the grader. Anything extra, such as variable name or function name (which is not part of the type), will get -1.

Let's look at the answers. The obvious one is `Optional<A2>` since `map` takes in a lambda that returns `A2`. If `process` takes in `Optional<A2>` then it is an exact match no type conversion.

`process` can be defined with any type that is a supertype of `Optional<A2>`, however. Recall that we can assign a variable of a subtype to its supertype without the need to cast. What are the supertypes of `Optional<A2>`? `Object`, of course, since it is a supertype of every type. `Optional<?>`, of course, since it is a supertype of any parameterized type of `Optional`.

Next, we invoke the covariant rule, which says that if `A2 <: T`, then `Optional<A2> <: Optional<? extends T>`. So, we have `Optional<? extends A2>`, `Optional<? extends A1>`, `Optional<? extends Object>`, and `Optional<? extends I>`.

We also have the contravariant rule, which says that if `S <: A2`, then `Stream<A2> <: Stream<? super S>`. So we also have `Optional<? super A2>` and `Optional<? super A3>`.

That's already 9 possibilities to get you full marks. But wait! There are more!

Recall that map takes in `Function<? super T, ? extends U>` and returns `Optional<U>`. So `process` could be taking in `Optional<A1>` and `Optional<Object>` as well. Through type inference, `A1` or `Object` would be resolved to `U`, and `x -> new A2()` would still type check to `Function<? super T, ? extends U>`.

So, that's two more.

But wait, there are even more! Now that you see the expression can return `Optional<Object>`, `process` should be able to take in supertype of `Optional<Object>`, not just `Optional<A2>`. What are the supertypes of `Optional<Object>`? It includes `Optional<? super A1>`, `Optional<? super Object>`, and `Optional<? super X>`, for `X` such as `Integer`, `String`, etc. as well! So there are many possibilities that you can include.

Common wrong answers are:

```
Optional<? extends A3>
Optional<A3>
```

Note that for Question 3, we graded both your original answer and our modified answer (e.g., by inserting the missing Stream/Optional) and take the max of the two.

Question 3: Subtyping (8 points)

Suppose we have Java classes `A`, `B`, `C1`, and `C2`, with the following subtype relationships:

`C1 <: B <: A`

`C2 <: B <: A`

Consider the following method call

```
doIt( IntStream.of(1).mapToObj(x -> new B()) );
```

Ignoring what `doIt` does to the argument, what are the possible valid types for the argument of `doIt` so that the statement above compiles without any warning or error?

Write down, one per line, up to 10 possible valid types (and only the valid types) of the argument `doIt`.

Note that this question will be graded by a bot – it is important to write only one type per line. Do not include any extra text.

Solution:

```
Object
Stream<? extends A>
Stream<? extends B>
Stream<? extends Object>
Stream<? super Object>
Stream<? super A>
Stream<? super B>
Stream<? super C1>
Stream<? super C2>
Stream<?>
Stream<A>
Stream<B>
Stream<Object>
```

This question assesses if you understand the notions of type conversion, subtyping, and variance of types. You get 0.9 marks for each correct answer, -0.4 for each wrong answer. We asked for up to 10, but you only need 9 to get full marks. Some students only wrote down the type parameters, such answers received a -1 mark penalty since the type is incomplete. We inserted the corresponding type for you before we passed it to the grader. Anything extra, such as variable name or function name (which is not part of the type), will get -1. For this question, many students wrote `IntStream<..>` instead of `Stream`. But `IntStream` is not a generic type. You get -0.5 point off.

Let's look at the answers. The obvious one is `Stream<B>` since `mapToObj` returns `new B()`. If `doIt` takes in `Stream<B>` then it is an exact match with no type conversion.

`doIt` can be defined with any type that is a supertype of `Stream<B>`, however. Recall that we can assign a variable of a subtype to its supertype without the need to cast. What are the supertypes of `Stream<B>`? `Object`, of course, since it is a supertype of every type. `Stream<?>` of course, since it is a supertype of any parameterized type of `Stream`.

Next, we invoke the covariant rule, which says that if `S <: T`, then `Stream<S> <: Stream<? extends T>`. So, we have `Stream<? extends B>`, `Stream<? extends A>`, and `Stream<? extends Object>`.

We also have the contravariant rule, which says that if `S <: T`, then `Stream<T> <: Stream<? super S>`. So we also have `Stream<? super B>`, `Stream<? super C1>` and `Stream<? super C2>`.

That's already 9 possibilities to get you full marks. But wait! There are more!

Recall that `mapToObj` takes in `IntFunction<? extends U>` but returns `Stream<U>`. So `doIt` could be taking in `Stream<A>` and `Stream<Object>` as well. Through type inference, A or Object would be U, and `x -> new B()` would still type check to `IntFunction<? extends U>`.

So, that's two more.

But wait, there are more! Now that you see the expression can return `Stream<Object>`, `doIt` should be able to take in supertype of `Stream<Object>`, not just `Stream<B>`. What are the supertypes of `Stream<Object>`? It includes `Stream<? super A>`, `Stream<? super Object>`, and `Stream<? super X>`, for `X` such as `Integer`, `String`, etc. as well! So there are many possibilities that you can include.

Common wrong answers are:

```
Stream<? extends C1>
Stream<? extends C2>
Stream<C1>
Stream<C2>
```

Note that for Question 3, we graded both your original answer and our modified answer (e.g., after inserting the missing Stream/Optional) and take the max of the two.

QUESTION 4

Question 4: Monad (9 points)

Consider the class `IntMonad` below, which encapsultes a single `int` value. The implementation of `flatMap` is incomplete.

```
class IntMonad {
  private int v;

  private IntMonad(int v) {
    this.v = v;
  }

  static IntMonad of(int v) {
    return new IntMonad(v);
  }

  IntMonad flatMap(Function<Integer, IntMonad> map) {
    ..
  }
}
```

Now, consider the following three versions of `flatMap`.

```
// (a)
return IntMonad.of(this.v);

// (b)
return map.apply(2 * this.v);

// (c)
return map.apply(map.apply(this.v).v);
```

Let's represent the three laws of Monad with letter `L`, `R`, and `A`:

- `L`: Left Identity
- `R`: Right Identity
- `A`: Associative

Which of the above implementation of `flatMap` would cause `IntMonad` to violate the Laws of Monad?

Fill in the blank with the letter (or letters) representing the laws that a given `flatMap` implementation violates. Fill in the blank with the string `none` if no law is violated. For instance, if a given implementation violates the Right Identity and the Associative law, fill in the blank with the string `RA` or `AR`.

Note that this question will be graded by a bot. So, filling in with any other text, such as "R, A", "Right Identity and Associative", "none, because ..", will lead to the answer being marked as wrong even if the intention of the answer is correct.

Solution: For this question, we want to assess if students have developed the right intuition about the operations and the rules of monads.

You get 1 mark for correctly including a law, and 1 mark for correctly excluding a law.

(a) L If all `flatMap` does is `IntMap.apply(this.v)`, it is returning the object with the original `v` and ignore the `map` function completely.

Left identity says that `IntMonad.of(i).flatMap(f)` must be the same as `f.apply(i)`. So, as long as `f` is not `x->x`, the left identity law is violated.

The right identity law says that `monad.flatMap(x -> IntMonad.of(x))` must equals to `monad`. Since this version of `flatMap` does nothing, the right identity law holds.

The associative law says that:

`monad.flatMap(g).flatMap(f)`

must be the same as

`monad.flatMap(x -> g.apply(x).flatMap(f))`.

Since `flatMap` does nothing, the left-hand side is the same as `monad`, and the right-hand side is also the same as `monad`. Regardless of what `f` and `g` are (since they are ignored). So, only left identity law is violated.

(b) LR

Here `flatMap` does not faithfully apply the function to `v`, but it applies it to `2v` instead.

Left identity says that `IntMonad.of(i).flatMap(f)` must be the same as `f.apply(i)`. So, the left identify law is violated since the left-hand side applies `f` to `2*i`, the right-hand side applies `f` to `i`.

The right identity law says that monad.flatMap(x -> IntMonad.of(x)) must equals to `monad`. The left-hand side would lead to a monad containing 2*v, which is different from the right hand side. The right identity law is violated.

The associative law says that

`monad.flatMap(g).flatMap(f)`

must be the same as

`monad.flatMap(x -> g.apply(x).flatMap(f))`.

Suppose `monad` contains a number `v` to begin with, then

`monad.flatMap(g).flatMap(f)` would lead to (intuitively) `f(2g(2v))`, while the right-hand side would also lead to `f(2g(2v))`. The associate law holds.

(c) LA

This version of flatMap, intuitively, returns `f(f(x))`.

Left identity says that `IntMonad.of(i).flatMap(f)` must be the same as `f.apply(i)`. So, the left identify law is violated since left-hand side returns `f(f(x))` and the right-hand side, `f(x)`.

The right identity law says that

`monad.flatMap(x -> IntMonad.of(x))`

must equals to `monad`. Note that here the function `f` does not change `x`, just putting `x` into a `IntMonad`, so apply `f` twice is the same as not apply `f`. So the right identity law holds.

The associative law says that

`monad.flatMap(g).flatMap(f)`

must be the same as

`monad.flatMap(x -> g.apply(x).flatMap(f))`.

`monad.flatMap(g).flatMap(f)` would lead to (intuitively) `f(f(g(g(x))`, while the right-hand side leads to `f(f(g(f(f(g(x))))))`. So the associative law is violated.

Question 4: Monad (9 points)

Consider the class `IntMonad` below, which encapsultes a single `int` value. The implementation of `flatMap` is incomplete.

```
class IntMonad {
  private int v;

  private IntMonad(int v) {
    this.v = v;
  }

  static IntMonad of(int v) {
    return new IntMonad(v);
  }

  IntMonad flatMap(Function<Integer, IntMonad> map) {
    ..
  }
}
```

Now, consider the following three versions of `flatMap`.

```
// (a)
return IntMonad.of(this.v);

// (b)
return map.apply(Math.max(0, this.v));

// (c)
return IntMonad.of(2 * map.apply(this.v).v);
```

Let's represents of the three laws of Monad with letter `L`, `R`, and `A`:

- `L`: Left Identity
- `R`: Right Identity
- `A`: Associative

Which of the above implementation of `flatMap` would cause `IntMonad` to violate the Laws of Monad?

Fill in the blank with the letter (or letters) representing the laws that a given `flatMap` implementation violates. Fill in the blank with the string `none` if no law is violated. For instance, if a given implementation violates the Right Identity and the Associative law, fill in the blank with the string `RA` or `AR`.

Note that this question will be graded by a bot. So, filling in with any other text, such as "R, A", "Right Identity and Associative", "none, because ..", will lead to the answer being marked as wrong even if the intention of the answer is correct.

Solution: For this question, we want to assess if students have developed the right intuition about the operations and the rules of monads.

You get 1 mark for correctly including a law, and 1 mark for correctly excluding a law.

(a) L

If all `flatMap` does is `IntMap.apply(this.v)`, it is returning the object with the original `v` and ignore the `map` function completely.

Left identity says that `IntMonad.of(i).flatMap(f)` must be the same as `f.apply(i)`. So, as long as `f` is not `x->x`, the left identity law is violated.

The right identity law says that `monad.flatMap(x -> IntMonad.of(x))` must equals to `monad`. Since this version of `flatMap` does nothing, the right identity law holds.

The associative law says that:

`monad.flatMap(g).flatMap(f)`

must be the same as

`monad.flatMap(x -> g.apply(x).flatMap(f))`.

Since `flatMap` does nothing, the left-hand side is the same as `monad`, and the right-hand side is also the same as `monad`. Regardless of what `f` and `g` are (since they are ignored). So, only left identity law is violated.

(b) LR

Here `flatMap` does not faithfully apply the function to `v`, but it applies it to `max(0,v)` instead. We know that if `flatMap` applies the method to `v`, then all the law holds. So, we only need to check for the cases where `v` is a negative number.

Left identity says that `IntMonad.of(i).flatMap(f)` must be the same as `f.apply(i)`. So, the left identify law is violated if `i` is negative, since the left-hand side applies `f` to 0, the right-hand side applies `f` to i.

The right identity law says that `monad.flatMap(x -> IntMonad.of(x))` must equals to `monad`. If `monad` contains a negative number, calling it with this version of `flatMap` would turn it into a monad containing a 0. So the right identity law is violated too.

The associative law says that

`monad.flatMap(g).flatMap(f)`

must be the same as

`monad.flatMap(x -> g.apply(x).flatMap(f))`.

If monad contains a negative number `v` to begin with, then `monad.flatMap(g).flatMap(f)` would lead to (intuitively) `f(g(0))`, while the right-hand side also leads to `f(g(0))`. So the associative law holds.

(c) LRA

This version of flatMap, intuitively, returns `2f(x)`.

Left identity says that `IntMonad.of(i).flatMap(f)` must be the same as `f.apply(i)`. So, the left identify law is violated since left-hand side returns `2f(x)` and the right-hand side, `f(x)`.

The right identity law says that

`monad.flatMap(x -> IntMonad.of(x))`

must equals to `monad`. The right identity law is violated too, since after `flatMap` the value inside would be doubled for the left-hand side.

The associative law says that

`monad.flatMap(g).flatMap(f)`

must be the same as

`monad.flatMap(x -> g.apply(x).flatMap(f))`.

`monad.flatMap(g).flatMap(f)` would lead to (intuitively) `2f(2g(x))`, while the right-hand side leads to `2(2f(g(x)))`. So the associative law is also violated.

Question 4: Monad (9 points)

Consider the class `IntMonad` below, which encapsultes a single `int` value. The implementation of `flatMap` is incomplete.

```
class IntMonad {
  private int v;

  private IntMonad(int v) {
    this.v = v;
  }

  static IntMonad of(int v) {
    return new IntMonad(v);
  }

  IntMonad flatMap(Function<Integer, IntMonad> map) {
    ..
  }
}
```

Now, consider the following three versions of `flatMap`.

```
// (a)
return IntMonad.of(this.v);

// (b)
return map.apply(this.v + 2);

// (c)
return IntMonad.of(Math.max(this.v, map.apply(this.v).v));
```

Let's represents of the three laws of Monad with letter `L`, `R`, and `A`:

- `L`: Left Identity
- `R`: Right Identity
- `A`: Associative

Which of the above implementation of `flatMap` would cause `IntMonad` to violate the Laws of Monad?

Fill in the blank with the letter (or letters) representing the laws that a given `flatMap` implementation violates. Fill in the blank with the string `none` if no law is violated. For instance, if a given implementation violates the Right Identity and the Associative law, fill in the blank with the string `RA` or `AR`.

Note that this question will be graded by a bot. So, filling in with any other text, such as "R, A", "Right Identity and Associative", "none, because ..", will lead to the answer being marked as wrong even if the intention of the answer is correct.

Solution: For this question, we want to assess if students have developed the right intuition about the operations and the rules of monads.

You get 1 mark for correctly including a law, and 1 mark for correctly excluding a law.

For this question, we want to assess if students have developed the right intuition about the operations and the rules of monads.

You get 1 mark for correctly including a law, and 1 mark for correctly excluding a law.

(a) L If all `flatMap` does is `IntMap.apply(this.v)`, it is returning the object with the original `v` and ignore the `map` function completely.

Left identity says that `IntMonad.of(i).flatMap(f)` must be the same as `f.apply(i)`. So, as long as `f` is not `x->x`, the left identity law is violated.

The right identity law says that `monad.flatMap(x -> IntMonad.of(x))` must equals to `monad`. Since this version of `flatMap` does nothing, the right identity law holds.

The associative law says that:

`monad.flatMap(g).flatMap(f)`

must be the same as

`monad.flatMap(x -> g.apply(x).flatMap(f))`.

Since `flatMap` does nothing, the left-hand side is the same as `monad`, and the right-hand side is also the same as `monad`. Regardless of what `f` and `g` are (since they are ignored). So, only left identity law is violated.

(b) LR

Here `flatMap` does not faithfully apply the function to `v`, but it applies it to `2+v` instead.

Left identity says that `IntMonad.of(i).flatMap(f)` must be the same as `f.apply(i)`. So, the left identify law is violated since the left-hand side applies `f` to `2+i`, the right-hand side applies `f` to `i`.

The right identity law says that `monad.flatMap(x -> IntMonad.of(x))` must equals to `monad`. The left-hand side would lead to a monad containing `2+v`, which is different from the right-hand side. The right identity law is violated.

The associative law says that

`monad.flatMap(g).flatMap(f)`

must be the same as

`monad.flatMap(x -> g.apply(x).flatMap(f))`.

Suppose `monad` contains a number `v` to begin with, then `monad.flatMap(g).flatMap(f)` would lead to (intuitively) `f(2+g(2+v))`, while the right-hand side would also lead to `f(2+g(2+v))`. The associate law holds.

(c) LA

This version of `flatMap`, intuitively, returns `max(v, f(v))`. We know that if `flatMap` returns `f(v)` then the law holds. Thus, we only need to pay attention to the cases where `v > f(v)`.

Left identity says that `IntMonad.of(i).flatMap(f)` must be the same as `f.apply(i)`. So, the left identify law is violated since left-hand side returns `i` (when i > f(i)) and the right-hand side returns `f(i)`.

The right identity law says that `monad.flatMap(x -> IntMonad.of(x))` must equals to `monad`. Here, `f(x)` is (intuitively) just `x`, since `x -> IntMonad.of(x)` does not change the value. So the right identity law holds.

The associative law says that

`monad.flatMap(g).flatMap(f)`

must be the same as

`monad.flatMap(x -> g.apply(x).flatMap(f))`.

Suppose `f(g(v)) > v > g(v)`. Then the left-hand side would yield `f(v)`, while the right hand side, `f(g(v))`. So the associative law fails.

QUESTION 5

Question 5: Asynchronous Programming (8 points)

Consider the program below. The method `doSomething()` may run for an undeterministic amount of time.

```
import java.util.concurrent.CompletableFuture;
class CF {
  static void doSomething() { .. }

  static CompletableFuture<Void> printAsync(int i) {
    return CompletableFuture.runAsync(() -> {
        doSomething();
        System.out.print(i);
    });
  }

  public static void main(String[] args) {
    printAsync(1).join();
    CompletableFuture.allOf(printAsync(2), printAsync(3))
      .thenRun(() -> printAsync(4));
    doSomething();
  }
}
```

What are the possible outputs printed by the program if `main` runs to completion normally?

Fill in the blank with the string **yes** if a given output is possible. Fill in the blank with the string **no** if `main` will never print the given output.

Note that this question will be graded by a bot. So, filling in with any other text, such as "NO!", "yes, because ..", "never!", etc, will lead to the answer being marked as wrong even if the intention of the answer is correct.

(a) `1`

(b) `2`

(c) `3`

(d) `4`

(e) `12`

(f) `14`

(g) `23`

(h) `24`

(i) `124`

(j) `134`

(k) `243`

(l) `234`

(m) `213`

(n) `1324`

(o) `4321`

Solution: `1` will always be printed first; Further, `4`, if printed, must appeared after `2` and `3` (in any order).

Question 5: Asynchronous Programming (8 points)

Consider the program below. The method `doSomething()` may run for an undeterministic amount of time.

```
import java.util.concurrent.CompletableFuture;
class CF {
  static void doSomething() { .. }

  static CompletableFuture<Void> printAsync(int i) {
    return CompletableFuture.runAsync(() -> {
        doSomething();
        System.out.print(i);
    });
  }

  public static void main(String[] args) {
    printAsync(1);
    CompletableFuture.anyOf(printAsync(2), printAsync(3))
      .thenRun(() -> printAsync(4))
      .join();
    doSomething();
  }
}
```

What are the possible outputs printed by the program if `main` runs to completion normally?

Fill in the blank with the string `yes` if a given output is possible. Fill in the blank with the string `no` if `main` will never print the given output.

Note that this question will be graded by a bot. So, filling in with any other text, such as "NO!", "yes, because ..", "never!", etc, will lead to the answer being marked as wrong even if the intention of the answer is correct.

(a) `1`

(b) `2`

(c) `3`

(d) `4`

(e) `12`

(f) `14`

(g) `23`

(h) `24`

(i) `124`

(j) `134`

(k) `243`

(l) `234`

(m) `213`

(n) `1324`

(o) `4321`

Solution: `4` will always be printed and must appear after at least one of `2` and `3`.

Question 5: Asynchronous Programming (8 points)

Consider the program below. The method `doSomething()` may run for an undeterministic amount of time.

```
import java.util.concurrent.CompletableFuture;
class CF {
  static void doSomething() { .. }

  static CompletableFuture<Void> printAsync(int i) {
    return CompletableFuture.runAsync(() -> {
        doSomething();
        System.out.print(i);
    });
  }

  public static void main(String[] args) {
    CompletableFuture.anyOf(
         printAsync(1)
           .thenRun(() -> printAsync(2)),
         printAsync(3))
      .thenRun(() -> printAsync(4));
    doSomething();
  }
}
```

What are the possible outputs printed by the program if `main` runs to completion normally?

Fill in the blank with the string `yes` if a given output is possible. Fill in the blank with the string `no` if `main` will never print the given output.

Note that this question will be graded by a bot. So, filling in with any other text, such as "NO!", "yes, because ..", "never!", etc, will lead to the answer being marked as wrong even if the intention of the answer is correct.

(a) `1`

(b) `2`

(c) `3`

(d) `4`

(e) `12`

(f) `14`

(g) `23`

(h) `24`

(i) `124`

(j) `134`

(k) `243`

(l) `234`

(m) `213`

(n) `1324`

(o) `4321`

Solution: `4` if printed, must be either `12` or `3` are printed. `2`, if printed, must be after `1` is printed.

The End