2022 s2 final sol - 。 。 。

Programming Methodology II (National University of Singapore)

NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
FINAL ASSESSMENT FOR
Semester 2 AY2021/2022

CS2030S Programming Methodology II

April 2022 Time Allowed 120 minutes

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 30 questions and comprises 18 printed pages, including this page.
2. Write all your answers in the answer sheet provided.
3. The total marks for this assessment is 90. Answer **ALL** questions.
4. This is a **OPEN BOOK** assessment. You are only allowed to refer to hard copy materials.
5. All questions in this assessment paper use Java 11.

Section I: OOP Principles (10 points)

For this section, you will need to refer to the following classes:

```
abstract class Railcar {
  private Railcar nextRailcar;

  public void tow(Railcar nextRailcar) {
    this.nextRailcar = nextRailcar;
  }

  public Railcar getNextRailcar() {
    return this.nextRailcar;
  }

  public int getTrainLength() {
    if (nextRailcar != null) {
      return 1 + nextRailcar.getTrainLength();
    } else {
      return 1;
    }
  }
}

public class Carriage extends Railcar {
  public double weightPerPackage;
  public int numberOfPackages;

  public Carriage(double weightPerPackage, int numberOfPackages) {
    this.weightPerPackage = weightPerPackage;
    this.numberOfPackages = numberOfPackages;
  }
}

public class TrainEngine extends Railcar {

  public double getTotalWeight() {
    double weight = 0.0;
    Carriage currentCarriage = (Carriage) this.getNextRailcar();
    while (currentCarriage != null) {
      weight += currentCarriage.weightPerPackage * currentCarriage.numberOfPackages;
      currentCarriage = (Carriage) currentCarriage.getNextRailcar();
    }
    return weight;
  }
}
```

1. (2 points) Consider the following code excerpt:

```
Railcar railcar1 = new Railcar();
railcar1.tow(railcar1);
railcar1.getTrainLength();
```

Will the above code excerpt compile? If not, why not?

A. Will compile.
B. Will not compile, as incorrect constructor arguments are provided.
C. Will not compile, as the `getTrainLength` method does not terminate.
D. Will not compile, as `Railcar` is abstract.
E. Will not compile, as the `tow` method can't be called on itself.

> **Solution:** The program does not compile. Whether a program will terminate or get into an infinite loop does not stop `javac` from compiling. The `tow` method can be called on any object of type `Railcar` (or its subtypes). The railcar class is abstract and cannot be instanatiated and therefore `D` is the correct option.

2. (1 point) Consider the following code excerpt:

```
TrainEngine engine = new TrainEngine();
Carriage carriage1 = new Carriage(100.5, 1);
Carriage carriage2 = new Carriage(200.0, 1);
Carriage carriage3 = new Carriage(50.0, 2);
Carriage carriage4 = new Carriage(80.0, 1);
carriage2.tow(carriage3);
carriage1.tow(carriage2);
engine.tow(carriage1);
```

Now, consider the following line:

```
int length = engine.getTrainLength();
```

After execution, what is the value of `length`?

> **Solution:** This question involves some simple tracing through the code. The important thing to note is that both `Carriage`'s and `TrainEngine`'s count towards train length. The answer is 4.

3. (1 point) Given the above code excerpt, consider the following line:

```
double weight = engine.getTotalWeight();
```

After execution, what is the value of `weight`?

> **Solution:** This question involves some simple tracing through the code excerpt. The important thing to note is that `carriage4` is not attached to the train. The answer is 400.5.

4. (2 points) Do the classes `TrainEngine` and `Carriage` follow the principle of Tell, Don't Ask? If not, why not?

A. Does follow the principle.

B. Does not follow the principle, as `TrainEngine` gets information directly from `Carriage`.

C. Does not follow the principle, as `Carriage` has no getters or setters.

D. Does not follow the principle, as `TrainEngine` has no getters or setters.

E. Does not follow the principle, as `Railcar` is abstract.

Solution: Here we need to check where `TrainEngine` and `Carriage` interact with each other, namely the `getTotalWeight` method. We can see here that `TrainEngine` is asking for the `weightPerPackage` and `numberOfPackages` fields to calculate the carriage weight, rather than telling the carriage to calculate its own weight. The prescence of setters and getters does not necessarily indicate whether the Tell-Don't-Ask principle is being followed. Railcar being abstract is immaterial to the question. Therefore, the answer is `B`.

5. (2 points) In the `getTotalWeight` method, there are two typecasts. These are examples of:

   A. Type erasure

   B. Type inference

   C. Narrowing type conversion

   D. Widening type conversion

   E. Generic types

Solution: No generic types are used and therefore it does not have anything to do with Type erasure, Type inference, or Generic types. Therefore we must conisder which type of conversion it is, either narrowing or widening. We can see that it is casting from a more general type to a more specific one, therefore this is a narrowing type conversion and this option is `C`. Note that widening type conversion is implicit in Java (i.e. you do not need to cast).

6. (2 points) The classes above were designed based on the following specification:

   "A train consists of multiple railcars connected in a line. The first railcar should be a train engine, and this can be connected to either a carriage or another train engine. Your program must be able to calculate the train length and the total carrying weight of the train. The total carrying weight of a train is the weight of all packages in all of the connected carriages."

   Do the provided classes `Railcar`, `TrainEngine`, and `Carriage` meet all aspects of the above specification? Why or why not? Choose the most correct answer.

   A. The code meets the specification. This is because the programmer correctly used OOP principles when designing their code.

   B. The code meets the specification. This is because all trains consist of both train engines and carriages, and the train length and carrying weight can be calculated correctly.

   C. The code does not meet the specification. This is because while it is possible create a train with both train engines and carriages, it is not possible to calculate the total carrying weight as this method assumes all remaining railcars are carriages.

   D. The code does not meet the specification. This is because trains can only contain carriages and engines are not allowed to be the front of a train.

   E. The code does not meet the specification. This is because trains can be infinite in length and therefore the program will never terminate, and it is never possible to calculate the train length.

   F. The code does not meet the specification. This is because no train tracks or railway stations have been implemented, as such the trains can not move.

Solution: This question requires a deeper reading of the program and excerpt above. We can note that the method `getTotalWeight` has an implicit assumption that the runtime type of the returned object of `getTotalWeight` will be of type `Carriage`, and therefore it is safe to cast. This is not true in the specification, as a train can have many `TrainEngine`'s (not just the one at the front). Therefore the answer is `C`.

Section II: Types (18 points)

For this section, you will need to refer to the following classes and interfaces:

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

abstract class Product {
  public void repair() {
    System.out.println("Repairing");
  }
}

class Car extends Product {
  @Override
  public void repair() {
    System.out.println("Repairing car");
  }
}

class Computer extends Product {
  @Override
  public void repair() {
    System.out.println("Repairing Computer");
  }
}

class Laptop extends Computer {
  @Override
  public void repair() {
    System.out.println("Repairing Laptop");
  }
}

class Factory<T extends Product> {
  private Producer<? extends T> producer;

  private Factory(Producer<? extends T> producer) {
    this.producer = producer;
  }

  public T produceNew() {
    return producer.produce();
  }

  public void changeProduction(Producer<? extends T> producer) {
    this.producer = producer;
  }

  public static <S extends Product> Factory<S> newFactory(Producer<? extends S> producer) {
    return new Factory<>(producer);
  }
}
```

7. (3 points) Select all valid relationships between the classes below:

   A. `Computer` <: `Product`

   B. `Product` <: `Computer`

   C. `Factory` <: `Product`

   D. `Laptop` <: `Product`

   E. `Car` <: `Car`

   F. `Car` <: `Product`

   G. `Car` <: `Computer`

   H. `Producer` <: `Object`

> **Solution:** This question involves looking through the class and interface structure and determining all of the subtyping relationships between classes/interfaces. This question also tests your understanding of the transitivity and reflexivity of subtype relationships. The answer is therefore `A`, `D`, `E`, `F` and, `H`.

8. (2 points) Consider the following code excerpt:

```
Factory<Product> computerFactory = Factory.newFactory(Computer::new);
Product product = computerFactory.produceNew();
product.repair();
```

   Will this code excerpt compile? If so what will be printed out? If not, why not?

   A. Compiles. Will print "Repairing".

   B. Compiles. Will print "Repairing Laptop".

   C. Compiles. Will print "Repairing Computer".

   D. Does not compile. This is because the factory `computerFactory` can not be instantiated due to a type conflict.

   E. Does not compile. This is because the return type of `computerFactory::produceNew` is not compatible with the variable `product` due to a type conflict.

> **Solution:** This code excerpt will compile. The factory produces new `Computer` objects using `Computer::new`. As a result, on calling the `repair` method, "Repairing Computer" will be printed. Therefore the answer is `C`.

9. (2 points) Consider the following code excerpt:

```
Factory<Product> computerFactory = Factory.newFactory(Laptop::new);
Computer computer = computerFactory.produceNew();
computer.repair();
```

   Will this code excerpt compile? If so what will be printed out? If not, why not?

   A. Compiles. Will print "Repairing".

   B. Compiles. Will print "Repairing Laptop".

   C. Compiles. Will print "Repairing Computer".

D. Does not compile. This is because the factory `computerFactory` can not be instantiated due to a type conflict.

E. Does not compile. This is because the return type of `computerFactory::produceNew` is not compatible with the variable `computer` due to a type conflict.

Solution: This code excerpt will not compile. We can see that the first line will compile as the `Factory:new` uses the upper bounded wildcard, and it is possible to assign the variable to this newly created factory object. However, the second line will not compile as the return type of `computerFactory::produceNew` is `Product` which is the supertype of `Computer`. Therefore the answer is `E`.

10. (1 point) After type erasure, what will be the compile-time type of the field `producer` in the class `Factory<T>`?

A. `Object`

B. `T`

C. `Producer`

D. `Producer<T>`

E. `? extends T`

Solution: After type erasure, all wildcards and generic type parameter will be erased, leaving only `Producer`. Therefore the answer is `C`.

11. (2 points) A general factory which produces a product of type `A` can be repurposed to produce products of type `B`, but only if `B` <: `A`.

Consider the following code excerpt:

```
Factory<Computer> computerFactory = Factory.newFactory(Computer::new);
computerFactory.changeProduction(Laptop::new);
```

What is the compile-time type before type erasure of `computerFactory` after this code excerpt is executed?

A. `Factory<Computer>`

B. `Factory<Laptop>`

C. `Factory<Object>`

D. `Factory<Product>`

E. `Factory<Car>`

Solution: The key for understanding this question is to realise that the Compile-Time type (before type erasure) of `computerFactory` will not change even if we change the "type" the factory is producing because Java is statically typed. Therefore it remains `Factory<Computer>`, and the answer is `A`.

12. (1 point) What is the compile-time return type of `computerFactory.produceNew();` after the above code excerpt is executed?

A. `Object`

B. `Product`
C. `Computer`
D. `Laptop`
E. `Factory`

> **Solution:** Similarly, the Compile-Time type of the return type of `computerFactory.produceNew();` is unchanged. Therefore the answer is `C`.

13. (1 point) What is the run-time return type of `computerFactory.produceNew();` after the above code excerpt is executed?

A. `Object`
B. `Product`
C. `Computer`
D. `Laptop`
E. `Factory`

> **Solution:** However, the Run-Time type can change, and in this case it changes to `Laptop`. Therefore the answer is `D`.

14. (3 points) Consider the following method signature:

```
public <T extends Product> T fun(Factory<? extends T> factory)
```

Now, consider the following code excerpt:

```
Factory<Computer> factories;
Product p = fun(factories);
```

What type is `T` inferred as?

A. `Product`
B. `Computer`
C. `Laptop`
D. `Object`
E. No type is inferred

Solution: To answer this question we must look at the constraints on the type parameter `T`. First, we know that `T` is a bounded type parameter and therefore we know that `T` <: `Product`. Secondly, we know that the return type of the method `fun` is of type `T` and it is constrained by the type of variable `p`, therefore we know that `T` <: `Product`. Finally, we know that the object passed to the argument of the method `fun` also constrains `T`. Specifically, we know that `Factory<Computer>` <: `Factory<? extends T>`, which means that `T` >: `Computer`. We now know that `Computer` <: `T` <: `Product`. We take the most specific type which is `Computer`. Therefore the answer is `B`.

15. (3 points) Consider the following method signature:

```
public <T extends Product> T fun(Factory<? extends T> factory)
```

Now, consider the following code excerpt:

```
Factory<Product> factories;
Laptop l = fun(factories);
```

What type is `T` inferred as?

A. `Product`
B. `Computer`
C. `Laptop`
D. `Object`
E. No type is inferred

Solution: We follow the same procedure as the previous question. First, we know that `T` is a bounded type parameter and therefore we know that `T` <: `Product`. Secondly, we know that the return type of the method `fun` is of type `T` and it is constrained by the type of variable `p`, therefore we know that `T` <: `Laptop`. Finally, we know that the object passed to the argument of the method `fun` also constrains `T`. Specifically, we know that `Factory<Product>` <: `Factory<? extends T>`, which means that `T` >: `Product`. We now know that `T` >: `Product` and `T` <: `Laptop`. There is no type parameter that can satisfy these constraints. Therefore the answer is `E`.

Section III: Functional Interfaces (8 points)

Recall our `Transformer` and `Combiner` interfaces from the labs:

```
@FunctionalInterface
public interface Transformer<U, T> {
  T transform(U u);
}

@FunctionalInterface
public interface Combiner<S, T, R> {
  R combine(S s, T t);
}
```

You will need to use these for this section.

16. (1 point) What is the main defining of property of Functional Interfaces.

    A. Functional Interfaces are more functional.
    B. Functional Interfaces must have the @FunctionalInterface annotation.
    C. Functional Interfaces are interfaces with default implementations.
    D. Functional Interfaces have a single abstract method.
    E. Functional Interfaces can only be implemented using a lambda function.

> **Solution:** The main defining property of Functional Interfaces is that they have a single abstract method. Therefore the answer is `D`.

17. (4 points) Consider the following code excerpt:

```
Combiner<Integer, Integer, Boolean> isDivisible =
                                  (x, y) -> x % y == 0 ? true : false;
```

Write the equivalent anonymous class of the above lambda function. You may omit the declaration of the variable `isDivisible` in your answer.

> **Solution:**
>
> ```
> Combiner<Integer, Integer, Boolean> isDivisible = new Combiner<>(){
>     @Override
>     public Boolean combine(Integer x, Integer y) {
>         return x % y == 0 ? true : false;
>     }
> };
> ```

18. (1 point) If we were to implement a curried version of the above lambda function what would the compile-time type of `isDivisible` be?

    A. `Combiner<Integer, Integer, Boolean>`
    B. `Combiner<Integer, Combiner<Integer, Boolean>>`
    C. `Transformer<Integer, Combiner<Integer, Boolean>>`
    D. `Transformer<Integer, Integer, Boolean>`

E. `Transformer<Integer, Transformer<Integer, Boolean>>`

Solution: Our `Combiner<Integer, Integer, Boolean>` will be come a chain of two unary functions (two `Transformer`s). Therefore the answer is `E`.

19. (2 points) Consider the following code excerpt:

```
double m1 = 0.3;
double m2 = 0.7;

Combiner<Double, Double, Double> weightedSum = (x, y) -> (m1 * x + m2 * y);

double firstSum = weightedSum.combine(1.0, 2.0);

m1 = 0.4;
m2 = 0.6;

double secondSum = weightedSum.combine(1.0, 2.0);
```

Will the code excerpt compile? If it compiles, what will the value of `secondSum` be? If it does not compile, why not?

A. It will compile. `secondSum` will have the value 1.7.

B. It will compile. `secondSum` will have the value 1.6.

C. It will not compile. This is because `m1` and `m2` are out of scope of the lambda function.

D. It will not compile. This is because the lambda function is not properly declared (syntax error).

E. It will not compile. This is because `m1` and `m2` must be effectively final.

Solution: The variables `m1` and `m2` are captured by `weightedSum`. Therefore they can not be changed and must be effectively final. In this code excerpt, `m1` and `m2` are changed and as such it will not compile. Therefore the answer is `E`.

Section IV: Streams (8 points)

For this question you will have to evaluate a stream pipeline and describe what elements are present in the stream. For your reference we have given you some helpful stream methods:

- `Stream<T> filter(Predicate<? super T> predicate)` Returns a stream consisting of the elements of this stream that match the given `predicate`.
- `<R> Stream<R> map(Function<? super T, ? extends R> mapper)` Returns a stream consisting of the results of applying the given function to the elements of this stream.
- `T reduce(T identity, BinaryOperator<T> accumulator)` Performs a reduction on the elements of the stream, using the accumulation function, and returns the reduced value.
- `Stream<T> limit(long maxSize)` Returns a stream consisting of the elements of this stream, truncated to be no longer than `maxSize` in length.
- `static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)` Returns an infinite sequential ordered `Stream` produced by iterative application of a function `f` to an initial element seed, producing a Stream consisting of seed, f(seed), f(f(seed)), etc. (Note `UnaryOperator<T>` is equivalent to `Function<T, T>`)

20. (3 points) Consider the following stream pipeline.

```
Stream.iterate(1, i -> i + 1)
      .map(x -> x * 10)
      .filter(x -> x > 50);
```

What will be the elements in the resulting stream above when it is eventually evaluated? If this Stream is infinite, write the first 5 elements, followed by "...". If it is a finite stream, write out all the elements of the stream.

Solution: We will break this down by method call:
`Stream.iterate(1, i -> i + 1)`
Will give the following stream:
1, 2, 3, 4, 5, ...
`Stream.iterate(1, i -> i + 1).map(x -> x * 10)`
Will give the following stream:
10, 20, 30, 40, 50, ...
Therefore the final answer is:
`Stream.iterate(1, i -> i + 1).map(x -> x * 10).filter(x -> x > 50)`
60, 70, 80, 90, 100, ...

21. (5 points) Consider the following stream pipeline.

```
Stream.iterate(1, i -> i + 1)
      .map(n -> Stream.iterate(1, i -> i + 1).limit(n).reduce(0, (x, y) -> x + y))
      .limit(6);
```

What will be the elements in the resulting stream above when it is eventually evaluated? If this Stream is infinite, write the first 5 elements, followed by "...". If it is a finite stream, write out all the elements of the stream.

Solution: We will break this down by method call, and we will start with the lambda passed to `map`:
`n -> Stream.iterate(1, i -> i + 1).limit(n)`
For a given n, we can see that we will get a stream containing the numbers from 1 to n.

For example, for $n = 1$, we will get a stream containing 1, and for $n = 4$ we will get a stream containing 1, 2, 3, and 4. Now we add back in the reduce method:

```
n -> Stream.iterate(1, i -> i + 1).limit(n).reduce(0, (x, y) -> x + y)
```

This will return the sum of all the elements in the stream. For example, for $n = 1$, we will get the sum 1, and for $n = 4$ we will get the sum 10.

We can now go back into the original map method:

```
Stream.iterate(1, i -> i + 1)
.map(n -> Stream.iterate(1, i -> i + 1).limit(n).reduce(0, (x, y) -> x + y))
```

We can see that this will create an infinite stream with the following elements:

1, 3, 6, 10, 15, ...

We now put it all together with the final limit method:

```
Stream.iterate(1, i -> i + 1)
.map(n -> Stream.iterate(1, i -> i + 1).limit(n).reduce(0, (x, y) -> x + y))
.limit(n)
```

This gives us the following finite stream:

1, 3, 6, 10, 15, 21

Section V: CompletableFuture (6 points)

For the next three questions you will need to use the following methods:

```
public Integer zero() {
    slowTask(); // Takes 1 second
    return 0;
}

public Integer addOne(Integer i) {
    slowTask(); // Takes 1 second
    return i + 1;
}

public Integer timesTwo(Integer i) {
    slowTask(); // Takes 1 second
    return i * 2;
}

public Integer sum(Integer i, Integer j) {
    slowTask(); // Takes 1 second
    return i + j;
}
```

22. (3 points) Consider the following code excerpt:

```
int i = timesTwo(addOne(zero()));
```

Which of the following single line `CompletableFutures` pipelines will compile and give the same result as the above code excerpt.

A.
```
int i = CompletableFuture.supplyAsync(() -> zero())
                         .thenAcceptAsync(x -> addOne(x))
                         .thenAcceptAsync(x -> timesTwo(x)).join();
```

B.
```
int i = CompletableFuture.supplyAsync(() -> zero())
                         .thenApply(x -> addOne(x))
                         .thenApply(x -> timesTwo(x)).join();
```

C.
```
int i = CompletableFuture.supplyAsync(() -> zero())
                         .thenApplyAsync(x -> addOne(x))
                         .thenApplyAsync(x -> timesTwo(x)).join();
```

D.
```
int i = CompletableFuture.supplyAsync(() -> timesTwo(addOne(zero()))).join();
```

E.
```
int i = CompletableFuture.supplyAsync(() -> zero())
                         .thenApply(x -> timesTwo(x))
                         .thenApply(x -> addOne(x)).join();
```

Solution: B, C, D

23. (3 points) Consider the following three code excerpts:

I.
```
CompletableFuture<Integer> cf1 = CompletableFuture.supplyAsync(() -> zero());
CompletableFuture<Integer> cf2 = cf1.thenApplyAsync(x -> addOne(x));
CompletableFuture<Integer> cf3 = cf1.thenApplyAsync(x -> addOne(x));
cf2.thenCombineAsync(cf3, (x, y) -> sum(x, y)).join();
```

II.
```
CompletableFuture<Integer> cf1 = CompletableFuture.supplyAsync(() -> zero());
CompletableFuture<Integer> cf2 = cf1.thenApply(x -> addOne(x));
CompletableFuture<Integer> cf3 = cf1.thenApply(x -> addOne(x));
cf2.thenCombineAsync(cf3, (x, y) -> sum(x, y)).join();
```

III.
```
CompletableFuture<Integer> cf1 = CompletableFuture.supplyAsync(() -> zero());
CompletableFuture<Integer> cf2 = cf1.thenApplyAsync(x -> addOne(x));
CompletableFuture<Integer> cf3 = cf1.thenApply(x -> addOne(x));
cf2.thenCombine(cf3, (x, y) -> sum(x, y)).join();
```

Compare the time taken to run the code excerpts above, focusing on the delay caused by `slowTask()` and ignore any other threading or computational overhead. We assume that an infinite number of worker threads are available, and the system can run at maximum concurrency.

Which of the following statement is correct?

A. I and II take the same amount of time; III is one second faster than I and II.

B. II and III take the same amount of time; I is one second faster than II and III.

C. I and III take the same amount of time; II is one second slower than I and III.

D. I is one second faster than III; III is one second faster than II.

E. I, II, and III take the same amount of time.

F. None of the above.

Solution: B

Section VI: Monad (12 points)

Consider the following class:

```
class FlatMapCount<T>{
    private T value;
    private int count;

    public FlatMapCount(T value, int count) {
      this.value = value;
      this.count = count;
    }

    public static <S> FlatMapCount<S> of(S value) {
      return new FlatMapCount<>(value, 0);
    }

    public FlatMapCount<T> flatMap(Transformer<T, FlatMapCount<T>> map) {
      FlatMapCount<T> tempFMC = map.transform(this.value);
      return new FlatMapCount<>(tempFMC.value, tempFMC.count + this.count + 1);
    }

    public String toString() {
      return "[" + this.value + ", " + this.count + "]";
    }
}
```

Test if `FlatMapCount` is a Monad by checking if it obeys each of the Monad laws.

Use the following functions:

```
Transformer<Integer, FlatMapCount<Integer>> mapper1 = x -> new FlatMapCount<> (x + 1, 1);
Transformer<Integer, FlatMapCount<Integer>> mapper2 = x -> new FlatMapCount<> (2 * x, 1);
```

24. (4 points) Does `FlatMapCount` obey the left identity law?

If it does not, demonstrate it by filling in the following blanks. Use the string representation of `FlatMapCount<T>` as returned by its `toString` method to describe the return values.

If it does obey the law write in "NA" in all the blanks below.

`FlatMapCount.of(2).flatMap(_____);` returns _____

`_____.transform(2);` returns _____

Solution: No, it does not.

```
FlatMapCount.of(2).flatMap(mapper1);
// returns [3, 2] (does not pass)
mapper1.transform(2);
// returns [3, 1]  (does not pass)
```

25. (4 points) Does `FlatMapCount` obey the right identity law?

If it does not, demonstrate it by filling in the following blanks. Use the string representation of `FlatMapCount<T>` as returned by its `toString` method to describe the return values.

If it does obey the law write in "NA" in all the blanks below.

`FlatMapCount<Integer> monad = FlatMapCount.of(2).flatMap(mapper1);`

`monad.flatMap(_____);` returns _____

`monad;` returns _____

Solution: No, it does not.

```
FlatMapCount.of(2).flatMap(mapper1).flatMap(x -> FlatMapCount.of(x));
// returns [3, 3]  (does not pass)
FlatMapCount.of(2).flatMap(mapper1);
// returns [3, 2]  (does not pass)
```

26. (4 points) Does `FlatMapCount` obey the associative law?

If it does not, demonstrate it by filling in the following blanks. Use the string representation of `FlatMapCount<T>` as returned by its `toString` method to describe the return values.

If it does obey the law write in "NA" in all the blanks below.

`FlatMapCount.of(2).flatMap(_____).flatMap(_____);` returns _____

`FlatMapCount.of(2).flatMap(_____);` returns _____

Solution: Yes, it does.

```
FlatMapCount.of(2).flatMap(mapper1).flatMap(mapper2);
// returns [6, 4] (passes)
FlatMapCount.of(2).flatMap(x -> mapper1.transform(x).flatMap(mapper2));
// returns [6, 4] (passes)
```

Section VII: Stack and Heap Diagram (10 points)

Consider the following `Lazy` class. This class is similar to the one you implemented except that that it does not use the `Maybe` class.

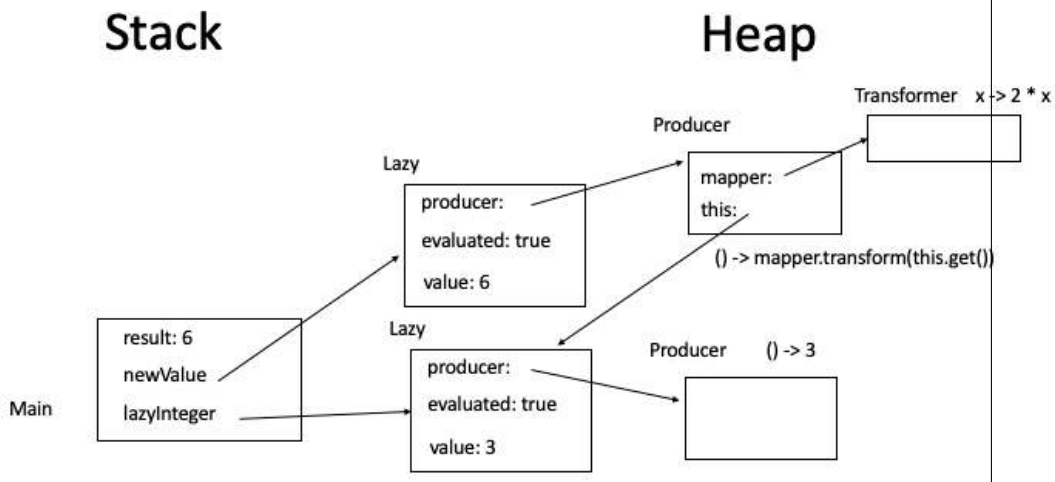
```
public class Lazy<T> {
  private Producer<T> producer;
  private T value;
  private boolean evaluated;

  private Lazy(Producer<T> s) {
    this.producer = s;
    this.evaluated = false;
    this.value = null;
  }

  public static <T> Lazy<T> of(Producer<T> producer) {
    return new Lazy<>(producer);
  }

  public <R> Lazy<R> map(Transformer<? super T, ? extends R> mapper) {
    return new Lazy<R>(() -> mapper.transform(this.get()));
  }

  public T get() {
    if (!this.evaluated) {
      this.value = this.producer.produce();
      this.evaluated = true;
    }
    return this.value;
  }
}
```

27. (10 points) Now, consider the the following code excerpt.

```
Lazy<Integer> lazyInteger = Lazy.of(() -> 3);
Lazy<Integer> newValue = lazyInteger.map(x -> 2 * x);
int result = newValue.get();
// Line A
```

On the provided sheet, draw the stack and the heap when the program gets to line A, include all objects that are created.

You may assume that the code excerpt is being run in the `main` method. You may ignore any variables used in `main` but not shown in the excerpt above.



Solution:

Section VIII: InfiniteList (18 points)

For the next 3 questions you will need to refer to the following partial `InfiniteList` implementation which is similar to the one you implemented in labs. You can assume that the `Lazy` class used in Section VII is given.

```
public class InfiniteList<T> {
  private final Lazy<T> head;
  private final Lazy<InfiniteList<T>> tail;

  public static <T> InfiniteList<T> iterate(T seed, Transformer<T, T> next) {
    return new InfiniteList<T>(Lazy.of(() -> seed),
        Lazy.of(() -> InfiniteList.iterate(next.transform(seed), next)));
  }

  private InfiniteList(Lazy<T> head, Lazy<InfiniteList<T>> tail) {
    this.head = head;
    this.tail = tail;
  }

  public T head() {
    return this.head.get();
  }

  public InfiniteList<T> tail() {
    return this.tail.get();
  }

  public <R> InfiniteList<R> map(Transformer<T, R> t) {
    return new InfiniteList<R>(head.map(t), tail.map(x -> x.map(t)));
  }

  public InfiniteList<T> limit(long n) {
    if (n <= 0) {
      return sentinel();
    }
    return new InfiniteList<T>(head, tail.map(x -> x.limit(n - 1)));
  }

  private InfiniteList<T> sentinel() {
    // return a new sentinel.
    // This method implementation is hidden for ease of reading.
  }

  public List<T> toList() {
    // a terminal operator the elements of the InfiniteList as a List.
    // This method implementation is hidden for ease of reading.
  }

  private static class Sentinel extends InfiniteList<Object> {
    // The rest of the implementation is hidden for ease of reading.
  }
}
```

This is a simplified version of `InfiniteList` that does not provide a `filter` method, and therefore it does not need to use the `Maybe` class. Furthermore, we have simplified the type of the method parameters by not applying PECS.

In this question, you will need to implement three methods for `InfiniteList`.

Solution: This section aims to assess students on a variety of concepts. The most important ones are: (i) lambda expression, (ii) lazy evaluation, (iii) infinite list, (iv) type matching.

In general, an answer is classified into three categories: wrong (0 marks); partially correct (half of the maximum marks); almost correct/correct (full marks).

Any answer that eagerly evaluates the list receives 0 marks. These include answers that call `tail()` or `head()`.

Any answer with egregious type mismatch receives 0 marks. These include many answers that pass in a producer to the constructors of the infinite list.

Students are expected to be able to determine the type of an expression. Common mistakes include trying to call methods of `Lazy` (e.g., `map`) on an `InfiniteList` or a `T`; or call methods of `InfiniteList` (e.g., `append`) on a `Lazy`.

Another common mistake is to redundantly calling the factory method or constructor of `Lazy`. The `head` and `tail` of the `InfiniteList` are already `Lazy` objects, so that is no need the "re-wrap" the value of type `T` or the `InfiniteList` in a `Lazy` instance. For instance, `Lazy.of(() -> head())` is equivalent to `head`.

28. (4 points) First, implement the `append` method to concatenate a given infinite list to the current list.

Consider the following examples of use:

```
InfiniteList<Double> l1 = InfiniteList.iterate(1.0, x -> x + 1).limit(2);
l1.toList(); // returns [1.0, 2.0]

InfiniteList<Double> l2 = InfiniteList.iterate(0.1, x -> x + 0.1);
l2.limit(4).toList(); // returns [0.1, 0.2, 0.3, 0.4]

InfiniteList<Double> l3 = l1.append(l2);
l3.limit(6).toList(); // returns [1.0, 2.0, 0.1, 0.2, 0.3, 0.4]
```

Reminder: `l3` should be lazily evaluated so that values shown should be produced only on demand.

Note that it is possible for `l1` to be a list with an infinite number of elements (not truncated with `limit(2)`) and therefore we may not see any elements from `l2`.

Implement the `append` method below. You may assume the `Sentinel::append` method is correctly implemented.

```
public InfiniteList<T> append(InfiniteList<T> list) {
    return new InfiniteList<T>(???, ???);
}
```

Solution:

```
public InfiniteList<T> append(InfiniteList<T> list) {
    return new InfiniteList<T>(
        head,
        tail.map(t -> t.append(list))
    );
}
```

Some students included checks for `this` being a sentinel. This check is not necessary, as dynamic binding would cause the `Sentinel::append` to be called and handle this special case for us.

29. (6 points) Now, implement the `zipWith` method to combine two lists, element-wise, with a lambda expression.

    Consider the following examples of use:

```
InfiniteList<Integer> l1 = InfiniteList.iterate(1, x -> x + 1);
l1.limit(4).toList(); // returns [1, 2, 3, 4]

InfiniteList<Integer> l2 = InfiniteList.iterate(0, x -> x + 2);
l2.limit(4).toList(); // returns [0, 2, 4, 6]

InfiniteList<Integer> l3 = l1.zipWith(l2, x -> y -> x + y);
l3.limit(4).toList(); // returns [1, 4, 7, 10]
```

    Remember: `l3` should be lazily evaluated so that values shown should be produced only on demand.

    The output of `zipWith` will have the length equivalent to the shorter of the two lists provided.

    Implement the `zipWith` method below. You may assume the `Sentinel::zipWith` method is correctly implemented.

```
public <U,R> InfiniteList<R> zipWith(InfiniteList<U> list,
      Transformer<T, Transformer<U, R>> transform) {
    return new InfiniteList<R>(???, ???);
}
```

    Solution:

```
public <U, R> InfiniteList<R> zipWith(InfiniteList<U> list,
    Transformer<T, Transformer<U, R>> transform) {
  return new InfiniteList<R>(
      head.map(transform).map(h -> h.transform(list.head())),
      tail.map(t -> t.zipWith(list.tail(), transform))
    );
}
```

    This question additionally assess if students know how to invoke a curried function. As such, students who treated the curried lambda as a lambda expression with two parameters (e.g., `BiFunction` or `Combiner`) would get 0 marks.

    An alternative to the first argument is

    - `head.map(x -> transform.transform(x).transform(list.head()))`

    An alternative to the second argument is

    - `tail.map(x -> list.tail.map(l -> x.zipWith(l, transform)).get())`

      Note that there is an implicit assumption that the given `list` is not a sentinel. Otherwise, the answer cannot be neatly fit into the given code template.

30. (8 points) Now, we will finally make our `InfiniteList` class into a monad by implementing the `flatMap` method.

    Consider the following examples of use:

```
InfiniteList<Integer> l1 = InfiniteList.iterate(1, x -> x + 1);
l1.limit(4).toList(); // returns [1, 2, 3, 4]
```

```
Transformer<Integer, InfiniteList<Integer>> t;
t = x -> InfiniteList.iterate(100 * x, y -> y + x).limit(2);
InfiniteList<Integer> l2 = l1.flatMap(t);

l2.limit(4).toList(); // returns [100, 101, 200, 202]
```

Reminder: `l2` should be lazily evaluated so that values shown should be produced only on demand.

It is possible for the lambda expression passed into `flatMap` to produce an infinite list. In which case, we will never see an element from `l2` produced by the second element of `l1`.

Implement the `flatMap` method below. You may assume the `Sentinel::flatMap` method is correctly implemented.

```
public <R> InfiniteList<R> flatMap(Transformer<T, InfiniteList<R>> transformer) {
    Lazy<InfiniteList<R>> list = this.head.map(transformer);
    return new InfiniteList<R>(???, ???);
}
```

Solution:

```
public <R> InfiniteList<R>
    flatMap(Transformer<T, InfiniteList<R>> transformer) {
  Lazy<InfiniteList<R>> list = this.head.map(transformer);
  return new InfiniteList<>(
      list.map(l -> l.head()),
      list.map(l -> l.tail().append(tail().flatMap(transformer)))
  );
}
```

Note that there is an implicit assumption that `list` is not a sentinel. Otherwise, the answer cannot be neatly fit into the given code template.

END OF PAPER