National University of Singapore

SCHOOL OF COMPUTING

CS2030 — PROGRAMMING METHODOLOGY II
(Semester 1: AY2023/2024)

Nov / Dec 2023 Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment contains **NINE(9)** questions.
2. Answer **ALL** questions. The maximum mark is **40**.
3. This is an **EXAMPLIFY-SECURE OPEN-BOOK** assessment. You may refer to your lecture notes, recitation guides, lab codes, and the Java API.
4. You will be liable for disciplinary action which may result in expulsion from the University if you are found to have contravened any of the clauses below,
   - Violation of the NUS Code of Student Conduct (in particular the part on Academic, Professional and Personal Integrity), NUS IT Acceptable Use Policy or NUS Examination rules.
   - Possession of unauthorized materials/electronic devices.
   - Bringing your mobile phone or any storage/communication device with you to the washroom.
   - Unauthorized communication e.g. with another student.
   - Reproduction of any exam materials outside of the exam venue.
   - Photography or videography within the exam venue.
   - Plagiarism, giving or receiving unauthorised assistance in academic work, or other forms of academic dishonesty.
5. Once you have completed the assessment,
   - Click on the "Exam Controls" button and choose "Submit Exam".
   - Check off "I am ready to exit my exam" and click on "Exit" to upload your answers to the server.
   - You will see a green confirmation window on your screen when the upload is successful. Please keep this window on your screen.
   - If you do not see a green window, please disconnect and reconnect your WIFI and try again.
   - Please be reminded that it is your responsibility to ensure that you have uploaded your answers to the Software.

1. [**4 marks**] You are given the following `Shape` interface and `Circle` class.

```
interface Shape {
    public double getArea();
}

class Circle implements Shape {
    private final double radius;

    Circle(double radius) {
        this.radius = radius;
    }

    Circle(Circle c) {
        this(c.radius);
    }

    public double getArea() {
        return Math.PI * this.radius * this.radius;
    }

    public String toString() {
        return "Circle with radius " + this.radius;
    }
}
```

Without modifying the `Circle` class, write the `FilledCircle` class with the `unfill()` method according to the sample test cases below:

```
jshell> Circle circle = new Circle(1.0)
circle ==> Circle with radius 1.0

jshell> FilledCircle filledCircle = new FilledCircle(circle, Color.BLUE)
filledCircle ==> FilledCircle with radius 1.0 and java.awt.Color[r=0,g=0,b=255]

jshell> circle = filledCircle
circle ==> FilledCircle with radius 1.0 and java.awt.Color[r=0,g=0,b=255]

jshell> filledCircle.unfill()
$.. ==> Circle with radius 1.0
```

Remember to keep to the OOP design discipline instilled throughout the course. Moreover, DO NOT modify any given code unless otherwise specified. This applies to all other questions throughout this assessment.

2. [**6 marks**] *This question is NOT related to the implementation of question 1.*

   You are given the following `Shape` interface and `Square` class.

```
interface Shape {
    public double getArea();
}

class Square implements Shape {
    private final double side;

    Square(double side) {
        this.side = side;
    }

    Square(Square c) {
        this(c.side);
    }

    public double getArea() {
        return this.side * this.side;
    }

    public String toString() {
        return "Square with side " + this.side;
    }
}
```

   Write the class `FilledSquare` by implementing the `unfill` method specified by the interface `Fillable<T>`.

```
interface Fillable<T> {
    public T unfill();
}
```

   In particular, the following is-a (sub-type) relationships should now hold:

   - `Square <: Shape` (`Square` is-a `Shape` as given above)
   - `FilledSquare <: Square`
   - `FilledSquare <: Shape`
   - `FilledSquare <: Fillable<Square>`

   To demonstrate the application of the abstraction principle, your answer should focus on where the color property should be declared and where the `unfill()` method should be implemented so as to support extensions such as `Rectangle` and `FilledRectangle`.

   You may include any other class/interface construct(s); you may ignore the implementation of the `toString` method.

   Hint: `FilledSquare` should *delegate* implementations to other class(es).

3. [**5 marks**] You are given the following `Func` class with the `compose` method.

```
abstract class Func {
    abstract Integer apply(Integer x);

    Func compose(Func other) {
        return new Func() {
            Integer apply(Integer x) {
                return Func.this.apply(other.apply(x));
            }
        };
    }
}
```

Rewrite the `Func` class and include the `decompose` method so that calling `decompose()` will undo the last function composition.

```
jshell> Func f = new Func() {
   ...>    Integer apply(Integer x) {
   ...>       return 2 * x;
   ...>    }
   ..>> }
f ==> 1@604ed9f0

jshell> Func g = new Func() {
   ...>    Integer apply(Integer x) {
   ...>       return 2 + x;
   ...>    }
   ..>> }
g ==> 1@6a4f787b

jshell> f.apply(10)
$.. ==> 20

jshell> f.compose(g).apply(10)
$.. ==> 24

jshell> f.compose(g).decompose().apply(10)
$.. ==> 20

jshell> f.compose(g).decompose().decompose().apply(10)
$.. ==> 20
```

Use `Optional` to handle decomposition beyond the first function, but keep to a disciplined and proper use of `Optional` methods as emphasized throughout the course. Moreover, DO NOT use `null`. This applies to all other questions throughout this assessment.

4. [**4 marks**] Java `Stream` has a `toList()` method that accumulates a stream of values into a Java `List` as shown below.

```
jshell> Stream<Integer> stream = Stream.of(1, 2, 3)
stream ==> java.util.stream.ReferencePipeline$Head@64b8f8f4

jshell> List<Integer> list = Stream.of(1, 2, 3).toList()
list ==> [1, 2, 3]
```

However, the list is part of `ImmutableCollections`. Hence we cannot modify the list, e.g. elements cannot be added.

```
jshell> list.add(1)
|  Exception java.lang.UnsupportedOperationException
|        at ImmutableCollections.uoe (ImmutableCollections.java:142)
|        at ImmutableCollections$AbstractImmutableCollection.add...
|        at (#13:1)
```

Write a single `Stream` pipeline to allow elements of the stream to be accumulated into an `ImList`. You may start your answer with

```
stream...
```

and assume that `stream` has just been declared as a fresh stream of elements.

Note that if `stream` is empty, then your answer should result in an empty `ImList`.

If you use the `reduce` method, DO NOT use the three-argument version. This applies to all other questions throughout this assessment.

5. [**4 marks**] Write a generic method `issorted` with the appropriate method header that takes in a `Stream<T>` and returns `true` if the stream elements are sorted in non-decreasing order via the *natural order* of `T`, or `false` otherwise.

```
jshell> issorted(List.<Integer>of(1,2,3).stream())
$.. ==> true

jshell> issorted(List.<Integer>of(1,3,2).stream())
$.. ==> false

jshell> issorted(Stream.<Integer>of(1,1,1))
$.. ==> true

jshell> issorted(Stream.<Integer>of(1).stream())
$.. ==> true

jshell> issorted(Stream.<Integer>of())
$.. ==> false

jshell> class A { }
|  created class A

jshell> class A { }
|  created class A

jshell> issorted(Stream.<A>of())
|  Error:
|  method issorted in class  cannot be applied to given types;
|    required: java.util.stream.Stream<T>
|    found: java.util.stream.Stream<A>
|    reason: inference variable T has incompatible bounds
|      equality constraints: A
|      lower bounds: java.lang.Comparable<T>
|  issorted(Stream.<A>of())
|  ^------^
```

Your method implementation should comprise of one single `return` statement.

```
... issorted(Stream<T> stream) {
   return stream...
}
```

Hint: you may find the `Pair` class useful.

6. [**4 marks**] A classic looping exercise in introductory programming methodology is the number guessing game. The following is a typical implementation for playing the game.

```
void printHint(int guess, int secret) {
    if (guess < secret) {
        System.out.println(guess + " is too low");
    }
    if (guess > secret) {
        System.out.println(guess + " is too high");
    }
}

void playGame(int secret) {
    Scanner sc = new Scanner(System.in);
    int guess = sc.nextInt();
    while (guess != secret) {
        printHint(guess, secret);
        guess = sc.nextInt();
    }
}
```

Notice the presence of the input side-effect within `playGame`. Rewrite the `playGame` method so that the side-effect is isolated from `playGame`.

Here is an example of how `playGame` can be called. User input is underlined.

```
jshell> Scanner sc = new Scanner(System.in)
sc ==> java.util.Scanner...

jshell> playGame(5, () -> sc.nextInt())
50
50 is too high
25
25 is too high
12
12 is too high
6
6 is too high
3
3 is too low
4
4 is too low
5
```

Replace the implementation of `playGame` by a single `Stream` pipeline.

Hint: use an appropriate `Stream` method that takes in a `Predicate` to terminate the stream when the correct guess is made.

7. [**4 marks**] To test the correctness of the `playGame` method in question 6, you are now given a list of integers containing the correct guess:

```
jshell> List<Integer> list = List.of(1, 2, 3, 4, 6, 7, 8, 9, 5, 10)
list ==> [1, 2, 3, 4, 6, 7, 8, 9, 5, 10] // assume 5 is correct guess
```

By writing a `void test(List<Integer> list, int secret)` method that takes in a list of integers and the secret correct guess, show how the test can be constructed such that an appropriate call to `playGame` from within the `test` method gives the following output:

```
jshell> test(list, 5)
1 is too low
2 is too low
3 is too low
4 is too low
6 is too high
7 is too high
8 is too high
9 is too high
```

8. [**4 marks**] There are many situations where a class property can be either one of two types, but not both. For example,

```
jshell> class Cat {
   ...>    public String toString() {
   ...>       return "Cat";
   ...>    }
   ...> }
|  created class Cat

jshell> class Dog {
   ...>    public String toString() {
   ...>       return "Dog";
   ...>    }
   ...> }
|  created class Dog

jshell> Either.<Cat,Dog>ofA(new Cat())
$.. ==> A:Cat

jshell> Either.<Cat,Dog>ofB(new Dog())
$.. ==> B:Dog
```

You are given the following `Either` class:

```
class Either<A,B> {
    private final Optional<A> a;
    private final Optional<B> b;

    Either(Optional<A> a, Optional<B> b) {
        this.a = a;
        this.b = b;
    }

    static <A,B> Either<A,B> ofA(A a) {
        return new Either<A,B>(Optional.of(a), Optional.empty());
    }

    static <A,B> Either<A,B> ofB(B b) {
        return new Either<A,B>(Optional.empty(), Optional.of(b));
    }

    public String toString() {
        return this.applyAorB(x -> "A:" + x, y -> "B:" + y);
    }
}
```

Notice that the `toString()` method makes use of the method `applyAorB`.

Complete the `Either` class by defining the following methods in order:

- `applyA`
- `applyB`
- `applyAorB`

A sample run of the methods is given below.

```
jshell> Either.<Integer,String>ofA(1).applyA(x -> x + 1)
$.. ==> 2

jshell> Either.<Integer,String>ofB("1").applyB(x -> x + "1")
$.. ==> "11"

jshell> Either.<Integer,String>ofA(1).applyB(x -> x + 1)
|  Exception java.util.NoSuchElementException: No value present
|        at Optional.orElseThrow (Optional.java:382)
|        at Either.applyB (#3:23)
|        at (#10:1)

jshell> Either.<Integer,String>ofA(1).applyAorB(x -> x + 1, y -> y + "1")
$.. ==> 2
```

You need only write the three methods in your answer.

9. [**5 marks**] You are given a list of tasks where each task is specified by the `doWork` method that is wrapped within a `UnaryOperator<Integer>`.

   Note that `UnaryOperator<T>` is `Function<T,T>`.

   Specifically, `doWork(int t, int x)` takes in two integers `t` and `x`, performs computation for `t` seconds, and returns back the result of `t + x`.

```
jshell> ImList<UnaryOperator<Integer>> list =
   ...> new ImList<UnaryOperator<Integer>>().
   ...>     add(x -> doWork(3, x)).
   ...>     add(x -> doWork(6, x)).
   ...>     add(x -> doWork(2, x)).
   ...>     add(x -> doWork(4, x))
list ==> [$Lambda$... ... $Lambda$...]
```

   The list of dependent tasks can be performed asynchronously in a separate thread, while freeing the main thread to do other work as shown below.

```
jshell> CompletableFuture<Integer> cf = CompletableFuture.completedFuture(0)
cf ==> java.util.concurrent.CompletableFuture@4dfa3a9d[Completed normally]

jshell> for (UnaryOperator<Integer> task : list) {
   ...>     cf = cf.thenApplyAsync(task);
   ...> }

jshell> 1 + 1
$.. ==> 2

jshell> cf.join() // after waiting 15 seconds since for loop was initiated
$.. ==> 15
```

   While attempting to rewrite the above as a `Stream`, complications start to arise:

```
list.stream() // assume that ImList has the stream() method
    .map(??)
    .reduce(CompletableFuture.<Integer>completedFuture(0),
        (cf1,cf2) -> cf1.then??)
```

   Notice the two-argument `reduce(T, BinaryOperator<T>)`, where `BinaryOperator<T>` is `BiFunction<T,T,T>`, requires that the `map` operation above returns a `Stream<CompletableFuture<Integer>>`. How should we map `UnaryOperator<Integer>` to `CompletableFuture<Integer>`?

Now consider creating a `Task` class to encapsulate a `UnaryOperator<Integer>` or `CompletableFuture<Integer>` depending on whether the `Task` object is created during `map` or `reduce`. Eventually, we want to make the following stream pipeline work.

```
jshell> Task task = list.stream().
   ...> map(x -> new Task(x)).
   ...> reduce(new Task(0), (x,y) -> x.thenApply(y));
task ==> Task@5649fd9b

jshell> 1 + 1
$.. ==> 2

jshell> task.get() // after waiting 15 seconds since task was created
$.. ==> 15
```

The `Task` class has been given to you which makes use of the `Either` class in question 8.

```
class Task {
    private final Either<UnaryOperator<Integer>,
        CompletableFuture<Integer>> UOpOrCF;

    Task(UnaryOperator<Integer> f) {
        this.UOpOrCF = Either.ofA(f);
    }

    Task(Integer t) {
        this.UOpOrCF = Either.ofB(CompletableFuture.completedFuture(t));
    }

    Task(CompletableFuture<Integer> cf) {
        this.UOpOrCF = Either.ofB(cf);
    }
}
```

Include additional methods into the `Task` class for the above stream pipeline to work. You need only write these methods in your answer.

— END OF PAPER —