National University of Singapore

SCHOOL OF COMPUTING

CS2030 — PROGRAMMING METHODOLOGY II
(Semester 2: AY2021/2022)

Apr / May 2022 Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper consists of **FIVE(5)** questions and comprises **TEN(10)** printed pages, including this page.
2. Answer **ALL** questions. The maximum mark is **40**.
3. This is an **OPEN BOOK** assessment. You may refer to your lecture notes, recitation guides, lab codes, and the Java API.
4. By taking this assessment, you are agreeing to abide by the following Honor Code:
   i. You will not discuss with, or receive help from, anyone.
   ii. You will not search for solutions or help, whether online or offline.
   iii. You will not share your answers with, or give help to, anyone.
   iv. You will act with integrity at all times.

   Breaching the Honor Code will result in severe penalties!
5. Write your answers within the individual program files and submit to the CodeCrunch course **CS2030_E1** with the task titled **Final Assessment Submission**.

   `https://codecrunch.comp.nus.edu.sg/`
6. You are advised to submit all your files after attempting every question. No extra time will be given to submit your work at the end of the assessment. You may upload as many times as you wish, but only the latest submission will be graded.

Question	Q1	Q2	Q3	Q4	Q5	Total
Marks	8	8	8	10	6	40

1. [**8 marks**] A university comprises two types of students: undergrads (`UG`) and postgrads (`PG`). Every student has a unique string identifier (`id`). A student can read modules and be credited marks at the end of the semester. These marks are managed by the Admin office (`Admin`) which has access to the list of students and their CAP (`getCAP`). CAP computation is the same for any type of student.

   A student can also apply for study loans during their candidature. The Bursar's office (`Bursar`) has access to the list of students and their loan amount (`getLoan`). Loan amounts are computed differently for different types of students.

   Notice that student CAP and loans are handled by different offices. Due to prevailing privacy data protection policies, no one party shall have access to data that does not come under the purview of the office.

   We would like to design the high level structure of the classes and test the program with method stubs (or methods with dummy implementations). Define the classes `PG`, `UG`, `Admin`, `Bursar` and their dependencies; you may include other classes or interfaces. Include the following methods where appropriate:

   - `double getCAP()` which returns `5.0` for all students;
   - `double getLoan()` which returns `99.99` for undergrads, and `111.11` for postgrads;
   - `String getId()` which returns the student identifier.

   In each of the `Admin` and `Bursar` classes, include separate `process` methods that takes in a list of students in order to "process" (for simplicity, to print out) the CAP and loan respectively for the students.

   A sample run is shown below:

```
jshell> List<Student> students = List.<Student>of(
   ...> new UG("U123"), new PG("P456"))
students ==> [UG@39aeed2f, PG@724af044]

jshell> new Admin().process(students)
U123 : 5.0
P456 : 5.0

jshell> new Bursar().process(students)
U123 : 99.99
P456 : 111.11
```

   ANSWER:

2. [**8 marks**] The value of π can be obtained either by summing the terms of an iterative series, or by some approximation method. In the questions below, write your solutions in a declarative style using Java `Stream`.

   (a) An approximation for π is given by summing the terms of the series:

   $$\pi \approx \frac{4}{1} - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \ldots$$

   i. [**2 marks**] Complete the given jshell method `pow` that takes as arguments `double a` and `long b`, and returns a^b as a `double` value. Do not use `Math.pow` or any math function from `java.lang.Math`.

   ANSWER:

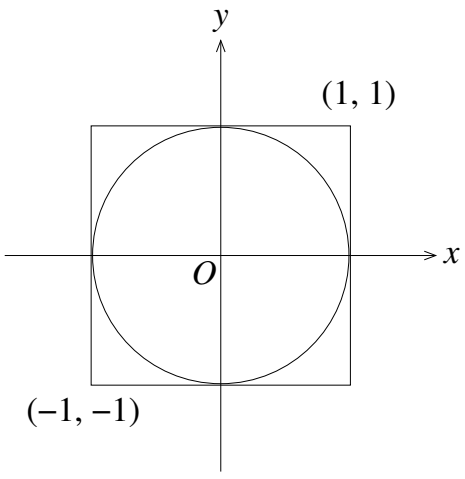
   ```
   double pow(double a, long b) {
      return Stream.<Double>generate(() -> a)...
   ```

   ii. [**3 marks**] Complete the given jshell method `seriesPi` that takes as argument `long n`, and returns an approximation for π by summing the first n terms of the series.

   ANSWER:

   ```
   double seriesPi(long n) {
       return Stream.<Integer>iterate(1, x -> x + 1)...
   ```

   (b) [**3 marks**] A circle is inscribed in a square as shown in the following diagram. Let C be the area of the circle, S be the area of the square, and R be the ratio $\frac{C}{S}$. The value of π can be expressed in terms of R as $\pi = 4R$.



Complete the given jshell method `approxPi` that takes as argument `long n`, and returns an approximation for π using the following steps:

Step 1. Generate n uniformly-distributed random 2D points inside the square.

Step 2. Of these random points, determine how many are inside the circle. Let this number be m.

Step 3. The ratio R is approximated as $\frac{m}{n}$, and π approximated as $4R$.

You may assume that the usual `Point` and `Circle` classes are given with the following methods:

- A `Point` constructor that takes in `double x` and `double y`;
- An instance method `distanceTo(Point other)` in class `Point` that returns the distance between the two points as a `double` value;
- A `Circle` constructor that takes in a `Point centre` and `double radius`;
- An instance method `contains(Point point)` in class `Circle` that returns `true` if `point` is inside the circle, or `false` otherwise.

ANSWER:

```
jshell> Random rand = new Random()
rand ==> java.util.Random@482f8f11

jshell> Circle unitCircle = new Circle(new Point(0, 0), 1.0)
unitCircle ==> Circle centered at (0.0, 0.0) with radius 1.0

jshell> double getRand() { // returns a value between -1.0 and 1.0
   ...>     return rand.nextDouble() * 2.0 - 1.0;
   ...> }
|  created method getRand()

jshell> double approxPi(long n) {
    return Stream...
```

3. [**8 marks**] A simplified version of the `Maybe` context to handle invalid or missing values is given below:

```
import java.util.function.Function;

class Maybe<T> {
    private final T thing;

    private Maybe (T thing) {
        this.thing = thing;
    }

    static <T> Maybe<T> of(T thing) {
        return new Maybe<T>(thing);
    }

    static <T> Maybe<T> empty() {
        return new Maybe<T>(null);
    }

    <R> Maybe<R> map(Function<? super T, ? extends R> mapper) {
        if (thing == null) {
            return Maybe.<R>empty();
        } else {
            return Maybe.<R>of(mapper.apply(this.thing));
        }
    }

    @Override
    public String toString() {
        return (thing == null) ? "Maybe.empty" : "Maybe[" + thing + "]";
    }
}
```

Redesign the `Maybe` context while adhering to the following restrictions:

- `null` **cannot** be used;
- additional instance properties **cannot** be included.

You may assume that `null` will not be passed as an argument to the `of` method. A sample run of *Maybe* is shown below:

```
jshell> Maybe.<Integer>of(1).map(x -> x + 1)
$.. ==> Maybe[2]

jshell> Maybe.<Integer>empty().map(x -> x + 1)
$.. ==> Maybe.empty
```

Hint: First, make `Maybe` an abstract class. Then think of how to avoid cyclic dependencies.

4. [**10 marks**] Imperative programming is based upon defining explicit branching and looping control flow to manage state changes when running a program. We have seen how looping control flow can be handled in a `Stream` context so that details of looping are encapsulated from the client. We will now construct a `IfElse` context to handle branching control flow.

   A typical `if..else` statement comprises a conditional expression that evaluates to a `boolean` value, a block of statements to execute if the condition is `true`, and possibly another block of statements to execute if the condition is `false`.

   In the following questions, we shall use the classic leap year problem to motivate the construction of the `IfElse` context.

   (a) [**4 marks**] When your lecturer was a kid, he was told that "a leap year occurs once every four years." In terms of imperative programming, we can express this as a `boolean` method `isLeap` as shown below:

   ```
   boolean isLeap(int year) {
       if (year % 4 == 0) {
           return true;
       } else {
           return false;
       }
   }
   ```

   Calling `isLeap(2022)` will return `false`; `isLeap(2024)` will return `true`.

   Begin by defining the `IfElse` class with a static method `of` that takes as arguments the three parts of the `if..else` construct as shown below. The `IfElse` context can then be evaluated via the `apply` method.

   ```
   jshell> IfElse<Integer, Boolean> div4 = IfElse.
      ...>     of(x -> x % 4 == 0, x -> true, x -> false)
   div4 ==> IfElse@482f8f11

   jshell> div4.apply(2022)
   $.. ==> false
   ```

   (b) [**2 marks**] During his teens around the late 1990s, your lecturer noticed that the past centuries, i.e. 1900, 1800, ... were not leap years, despite that they are divisible by four. As such, we construct the following:

   ```
   jshell> IfElse<Integer, Boolean> not100 = IfElse.
      ...>     of(x -> x % 100 != 0, x -> true, x -> false)
   not100 ==> IfElse@724af044

   jshell> not100.apply(1900)
   $.. ==> false
   ```

Now, we need to combine the two `IfElse` contexts defined above. Include the `mapIf` method so as to adhere to the following sample run:

```
jshell> IfElse<Integer, Boolean> div4Not100 = div4.mapIf(not100)
div4Not100 ==> IfElse@55f3ddb1

jshell> div4Not100.apply(2022)
$.. ==> false

jshell> div4Not100.apply(2024)
$.. ==> true

jshell> div4Not100.apply(1900)
$.. ==> false
```

(c) [**1 marks**] At the recent turn of the century, he realized that year 2000 is a leap year! Indeed years, 1600, 1200, 800, 400 are leap years.

```
jshell> IfElse<Integer, Boolean> div400 = IfElse.
   ...>     of(x -> x % 400 == 0, x -> true, x -> false)
div400 ==> IfElse@4c70fda8

jshell> div400.apply(2000)
$.. ==> true
```

To combine this new context, you wil need to define the method `mapElse`.

```
jshell> IfElse<Integer, Boolean> leap = div400.mapElse(div4Not100)
leap ==> IfElse@46d56d67

jshell> leap.apply(2022)
$.. ==> false

jshell> leap.apply(2024)
$.. ==> true

jshell> leap.apply(1900)
$.. ==> false

jshell> leap.apply(2000)
$.. ==> true
```

(d) [**3 marks**] Define the methods `map` and `flatMap` that takes an appropriate `Function` and performs the following:

```
jshell> Function<Boolean, String> f = x -> x ? "Leap" : "Not leap"
f ==> $Lambda$26/0x00000001000af440@61832929

jshell> leap.map(f).apply(2022)
$.. ==> "Not leap"

jshell> leap.map(f).apply(2024)
$.. ==> "Leap"

jshell> leap.map(f).apply(2000)
$.. ==> "Leap"

jshell> leap.map(f).apply(1900)
$.. ==> "Not leap"

jshell> Function<Integer, IfElse<Integer, Integer>> g = x -> IfElse.
   ...>     of(y -> x % 2 == 0, y -> y + 2, y -> y * 2)
g ==> $Lambda$34/0x00000001000c0c40@3c0ecd4b

jshell> IfElse.<String, Integer>of(x -> x.compareTo("cs2030") > 0,
   ...>     x -> x.length(), x -> 0).flatMap(g).apply("CS2030S")
$.. ==> 2
```

ANSWER:

5. [**6 marks**] Below is a simplified implementation of the `ImList` class adopted in class with the addition of two `println` statements in the `add` and `set` methods.

```
import java.util.List;
import java.util.ArrayList;

class ImList<T> {
    private final List<T> list;

    ImList() {
        this.list = new ArrayList<T>();
    }

    private ImList(List<? extends T> list) {
        this.list = new ArrayList<T>(list);
    }

    ImList<T> add(T t) {
        System.out.println("Adding: " + t);
        ImList<T> newList = new ImList<T>(this.list);
        newList.list.add(t);
        return newList;
    }

    ImList<T> set(int index, T t) {
        System.out.println("Setting: " + index + ", " + t);
        ImList<T> newList = new ImList<T>(this.list);
        newList.list.set(index, t);
        return newList;
    }

    T get(int index) {
        return this.list.get(index);
    }

    @Override
    public String toString() {
        return "ImList";
    }
}
```

The following is a sample run in jshell. Notice that the list is updated *eagerly* via the `add` and `set` methods.

```
jshell> new ImList<Integer>().add(1).add(2).set(1, 3).add(4)
Adding: 1
Adding: 2
Setting: 1, 3
Adding: 4
$4 ==> ImList

jshell> new ImList<Integer>().add(1).add(2).set(1, 3).add(4).get(1)
Adding: 1
Adding: 2
Setting: 1, 3
Adding: 4
$5 ==> 3
```

Modify the implementation of `ImList` such that the list is now updated *lazily* only upon execution of the `get` method. Ensure that the `println` statements are to remain within their own respective methods.

```
jshell> new ImList<Integer>().add(1).add(2).set(1, 3).add(4)
$5 ==> ImList

jshell> new ImList<Integer>().add(1).add(2).set(1, 3).add(4).get(1)
Adding: 4
Setting: 1, 3
Adding: 2
Adding: 1
$6 ==> 3
```

ANSWER: