National University of Singapore

SCHOOL OF COMPUTING

CS2030/S — PROGRAMMING METHODOLOGY II
(Semester 1: AY2021/2022)

Due: 11:00 hrs on Thursday, 25 November 2021

INSTRUCTIONS TO CANDIDATES

1. This assessment paper consists of **FIVE(5)** questions and comprises **ELEVEN(11)** 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/S_E** 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.
7. At the end of the assessment, separately upload the video of your screen capture as per the assessment protocol.

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

1. [**8 marks**] Below is the implementation of the immutable list `ImList`, with the `add` and `set` methods:

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

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

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

    ImList<T> add(T elem) {
        ImList<T> newList = new ImList<T>(this.list);
        newList.list.add(elem);
        return newList;
    }

    ImList<T> set(int index, T elem) {
        ImList<T> newList = new ImList<T>(this.list);
        newList.list.set(index, elem);
        return newList;
    }

    // other methods omitted for brevity
}
```

Notice that `add` and `set` (as well as `remove` which is not shown) are implemented in a similarly way by delegating the underlying task to the mutable list structure encapsulated within `ImList`.

By employing the **abstraction principle**,

> Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts
>
> — *Benjamin C. Pierce*

(a) [**4 marks**] Identify "similar functions (that) are carried out by distinct pieces of code", and "combine them into one" by defining an `update` method.

(b) [**4 marks**] Demonstrate how "abstracting out the varying parts" is achieved by rewriting the `add` and `set` methods.

2. [**8 marks**] The following is a design of the `TrafficLight` class for a standard red-green traffic light that toggles between red and green lights.

```
class TrafficLight {
    private final String color;

    TrafficLight(String color) {
        this.color = color;
    }

    TrafficLight toggle() {
        if (this.color.equals("red")) {
            return new TrafficLight("green");
        } else {
            return new TrafficLight("red");
        }
    }

    @Override
    public String toString() {
        return this.color;
    }
}
```

In particular, calling the `toggle()` method repeatedly toggles the lights to switch between red and green. This can be demonstrated by a client `toggling` method:

```
jshell> void toggling(TrafficLight t, int n) { // toggle n times
   ...>     System.out.print(t);
   ...>     for (int i = 1; i < n; i++) {
   ...>         t = t.toggle();
   ...>         System.out.print(" -> " + t);
   ...>     }
   ...>     System.out.println();
   ...> }
|  created method toggling(TrafficLight,int)

jshell> toggling(new TrafficLight("red"), 5);
red -> green -> red -> green -> red
```

Notice that the `TrafficLight` class has the *desirable property* that it can be toggled between red and green only.

(a) [**3 marks**] By employing the **substitutability principle**, redesign the `TrafficLight` class with the inclusion of sub-classes `RedLight` and `GreenLight`, such that `TrafficLight` need only have the *desirable property* that it can be toggled, but not the specific color of the lights.

Note that the `TrafficLight` class should not be dependent on its sub-classes. However, the sub-classes (i.e. siblings) are dependent on each other.

(b) [**1 mark**] Without changing the client `toggling` method, demonstrate how the `toggling` method can be used to produce the same output

`red -> green -> red -> green -> red`.

(c) [**2 marks**] Suppose we now include an amber light, such that toggling the lights starting with red would result in, say

`red -> green -> amber -> red -> green -> amber -> ....`

Redesign the classes with the addition of an `AmberLight` sub-class. However, unlike question 2a, there should be no cyclic dependencies among the classes.

(d) [**2 marks**] Without modifying the four classes in question 2c, as well as the `toggling` method, demonstrate how the lights could be re-configured such that an appropriate call to the `toggling` method to toggle the light seven times would result in the output:

`red -> amber -> green -> amber -> red -> amber -> green`

3. [**10 marks**] In the following questions, you are **not allowed** to use looping constructs `for`, `while`, `do`, or `null` values.

   (a) [**2 marks**] Complete the following method `convert` that takes in an integer array `arr` and a function `fn` and returns a new integer array with **all** the elements mapped over `fn`. The method should comprise of **only one** `return` statement.

   ```
   /* complete the method header */ {

       return /* complete the statement */

   }
   ```

   The following shows a sample run:

   ```
   jshell> int[] arr = new int[]{1, 2, 3}
   arr ==> int[3] { 1, 2, 3 }

   jshell> convert(arr, x -> x + 1)
   $.. ==> int[3] { 2, 3, 4 }

   jshell> arr
   arr ==> int[3] { 1, 2, 3 }

   jshell> arr = convert(arr, x -> x + 1)
   arr ==> int[3] { 2, 3, 4 }

   jshell> arr = convert(arr, x -> x % 2)
   arr ==> int[3] { 0, 1, 0 }
   ```

   (b) [**1 mark**] Notice that the method in question 3a allows the client to pass in a functionality that specifies how an element is converted to another; the method is still responsible for accessing all elements.

   Complete the following method `convert` that takes in an integer array `arr`, and a function that specifies how the array is accessed, and how the elements are to be mapped. The method should comprise of **only one** `return` statement.

   ```
   /* complete the method header  */ {

       return /* complete the statement */

   }
   ```

As an example, suppose that `f` is a function that picks alternate elements starting from the front, and multiplies these elements by 2.

```
jshell> int[] arr = new int[]{1, 2, 3}
arr ==> int[3] { 1, 2, 3 }

jshell> convert(arr, f)
$.. ==> int[3] { 2, 6 }

jshell> arr
arr ==> int[3] { 1, 2, 3 }
```

(c) [**1 mark**] Write a suitable function for `f` such that `convert(arr, f)` would give the output shown in question 3b.

(d) [**3 marks**] A one-dimensional Conway's Game of Life is a population of cells represented in an array where each element represents a cell organism. Each cell may be alive or dead.

Let's assume a population of nine cells with only one living organism (marked 1; 0 being dead) in the initial state, or generation 1:

0	0	0	0	1	0	0	0	0

For each generation, a cell is alive or dead depending on its own previous state and the previous states of the two neighbour cells. We adopt the following rule:

For each cell in the population,

- If a cell is alive in one generation, it will be dead in the next.
- If a cell is dead in one generation, but has one and only one live neighbour cell, it will be alive in the next generation.

Applying the above rules to the initial generation above, we obtain the next generation 2:

0	0	0	1	0	1	0	0	0

Applying the rules again to obtain generation 3:

0	0	1	0	0	0	1	0	0

Write a `void` method `conway` that takes in an integer array `arr`, the rules of the game `r`, and the number of generations `n` of the game. The method plays the game of life for `n` generations. After each generation, the method displays the population as a `String` where an array element of 0 is represented as a blank space, and 1 is represented as an asterisk `*`.

Using the three generations described above, the output will be

```
   *
  * *
 *   *
```

The `conway` method should comprise of **only one** statement. You may make use of helper methods to simply your implementation, on the condition that all these methods should also comprise of **only one** `return` statement.

(e) [**2 marks**] Write the rule `r` such that the following can be generated.

```
jshell> arr = new int[]{0, 0, 0, 1, 0, 0, 0}
arr ==> int[7] { 0, 0, 0, 1, 0, 0, 0 }

jshell> conway(arr, r, 4)
   *
  * *
 *   *
* * * *
```

(f) [**1 mark**] Give the output of the following:

```
int[] arr = new int[63]
arr[31] = 1;
conway(arr, r, 32)
```

4. [**6 marks**] Study the following `MyStream` class.

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

abstract class MyStream<T> {
    static <T> MyStream<T> generate(Supplier<T> seed) {
        return new MyStream<T>() {
            public T get() {
                return seed.get();
            }
        };
    }

    abstract T get();
}
```

annoymous inner class

As an example, to obtain an element from the stream

```
jshell> MyStream.generate(() -> 1).get()
$.. ==> 1
```

You are advised to model how the above works before proceeding to the following questions.

(a) [**3 marks**] Write a method `forEach` that takes in a `Consumer` followed by an integer `n`. The method performs an action as specified by the consumer for each of the `n` elements of the stream.

```
jshell> Random r = new Random(123)
r ==> java.util.Random@735b5592

jshell> MyStream.generate(() -> r.nextInt(10)).
   ...>     forEach(x -> System.out.print(x + " "), 5)
2 0 6 9 5
```

(b) [**3 marks**] Write a method `map` that takes in a `Function` and applies the given function to the elements of the stream.

```
jshell> Random r = new Random(123)
r ==> java.util.Random@735b5592

jshell> MyStream.generate(() -> r.nextInt(10)).
   ...>     map(x -> x % 2).
   ...>     forEach(x -> System.out.print(x + " ", 5)
0 0 0 1 1
```

5. [**8 marks**] An arithmetic expression is an expression made of operators and operands, e.g. 3 + 4. In particular, a postfix expression is one in which the operands are placed before the operator, e.g. 3 4 +. A postfix expression allows the evaluation of complex expressions without the need for brackets, e.g. 1 2 + 3 4 + ∗ is equivalent to $(1 + 2) * (3 + 4)$.

   The typical way to evaluate a postfix expression is using a stack via the push and pop actions. For example, using the postfix expression above,

Operator/Operand	Stack Action	Stack [Top..Bottom]
1	push 1	[1]
2	push 2	[2 1]
+	pop 2; pop 1; add $1 + 2$; push 3	[3]
3	push 3	[3 3]
4	push 4	[4 3 3]
+	pop 4; pop 3; add $3 + 4$; push 7	[7 3]
*	pop 7; pop 3; multiply $3 * 7$; push 21	[21]

   Notice that the final value in the stack is the evaluated answer.

   As this question involves the use of the `Optional` class, you are reminded not to use the `get()`, `isEmpty()` or `isPresent` methods in your answers.

   You are given the `Pair` and `Stack` classes.

```
class Pair<T,U> {
    private final T t;
    private final U u;

    Pair(T t, U u) {
        this.t = t;
        this.u = u;
    }

    T first() {
        return this.t;
    }

    U second() {
        return this.u;
    }
}
```

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

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

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

    private Stack(List<T> oldList) {
        this.list = new ArrayList<T>(oldList);
    }

    Stack<T> push(T elem) {
        Stack<T> newStack = new Stack<T>(this.list);
        newStack.list.add(0, elem);
        return newStack;
    }

    boolean isEmpty() {
        return this.list.isEmpty();
    }

    public String toString() {
        return "Top -> " + this.list;
    }
}
```

(a) [**4 marks**] Complete the `pop` method of the `Stack` class. The method returns the top element, as well as the resulting stack encapsulated within a `Pair` object.

```
Pair<Optional<T>,Stack<T>> pop() {
    ...
}
```

The following is a sample run:

```
jshell> Stack<Integer> stack = new Stack<Integer>()
stack ==> Top -> []

jshell> stack.push(1)
$.. ==> Top -> [1]

jshell> stack.push(1).push(2)
$.. ==> Top -> [2, 1]
```

```
jshell> stack.push(1).push(2).pop()
$.. ==> Pair@28feb3fa

jshell> stack.push(1).push(2).pop().first()
$.. ==> Optional[2]

jshell> stack.push(1).push(2).pop().second().pop().first()
$.. ==> Optional[1]

jshell> stack.push(1).push(2).pop().second().pop().second().pop().first()
$.. ==> Optional.empty
```

(b) [**4 marks**] Complete the method `postfix` that takes in a postfix expression as a `String` and performs postfix expression evaluation using an integer stack.

```
int evaluate(String expr) {
    Scanner sc = new Scanner(expr);
    ...
    while (sc.hasNext()) {
        String term = sc.next();
        ...
        if (term.equals("+") || term.equals("*")) {
            ...
        } else {
            Integer value = Integer.parseInt(term);
            ...
        }
    }
    ...
    return...;
}
```

Here, we restrict our expressions to integer expressions with only two operators add `+` and multiply `*`. In addition, we will assume that each operator takes some time to complete. As an example, suppose the time taken for each operation is represented by the sum of the left and right operand values, i.e. for the postfix expression `1 2 + 3 4 + *`,

- for `1 2 +`, the add operation will take 3 units of time
- for `3 4 +`, the add operation will take 7 units of time
- for `3 7 *`, the multiply operation will take 10 units of time; 3 and 7 being evaluated from the two preceding expressions

Although it seems that we need a total of 20 units of time to complete the evaluation, notice that the first two sub-expressions can be evaluated asynchronously. That is to say, both evaluations would have been completed within 7 units of time. And since multiplication proceeds right after, the total time taken for evaluation is 17 units of time.

— END OF PAPER —