National University of Singapore

SCHOOL OF COMPUTING

CS2030 — PROGRAMMING METHODOLOGY II
(Semester 2: AY2020/2021)

Due: 19:00 hrs on Wednesday, 28 April 2021

INSTRUCTIONS TO CANDIDATES

1. This assessment paper consists of **SEVEN(7)** 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 and lab codes.
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. At the conclusion of the assessment, zip all your files (except the video) into a single zip file, without password. Name your zip file: `ABC.zip` where `ABC` is your student number starting with `A0`. Upload your file to the folder `Enn submission folder` in LumiNUS Files, where `Enn` is your Exam Group. The submission folder opens 15 minutes before the end of the assessment. You may then upload as many times as you wish, but the latest file will be graded. In addition, submit your individual program files to the CodeCrunch task titled `Final Assessment Submission`.
6. Separately upload the video of your screen capture as per the assessment protocol.
7. **Announcements during the assessment will be made via the following link:** `https://www.comp.nus.edu.sg/~cs2030/exam`

Question	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Total
Marks	5	10	4	5	4	8	4	40

1. Study the following program.

```
class A {
    private int x;

    A(int x) {
        this.x = x;
    }

    // skipping many other methods here...

    @Override
    public String toString() {
        return "[" + this.x + "]";
    }
}
```

(a) **[2 marks; Q1a.jsh]** Given the following `client` method in `jshell`,

```
jshell> void client(A a1) {
   ...>     A a2 = a1.set(20);
   ...>     System.out.println("a2: " + a2);
   ...>     System.out.println("a1: " + a1);
   ...>     A a3 = a2.set(30);
   ...>     System.out.println("a3: " + a3);
   ...>     System.out.println("a2: " + a2);
   ...>     System.out.println("a1: " + a1);
   ...> }
|  modified method client(A)
```

write the `set` method (and only this method) that behaves as follows:

```
jshell> client(new A(10))
a2: [20]
a1: [20]
a3: [30]
a2: [30]
a1: [30]
```

(b) **[3 marks; Q1b.jsh]** Let's test out an immutable version of the method `set`. Without changing the implementations of class `A` or method `client`, demonstrate how you can achieve this using `jshell`. The `client` method should now produce the following output (*you will need to call the* `client` *method appropriately*).

```
a2: [20]
a1: [10]
a3: [30]
a2: [20]
a1: [10]
```

2. [**10 marks; Q2.java**] Suppose you are a project manager overseeing a team of three programmers building part of a banking system. One of your programmers is writing the `Account` class and handles the `credit` and `debit` methods. A sample run is given below. You may assume that positive integer amounts of type `int` are passed as arguments to constructors and methods.

```
jshell> new Account(10)
$.. ==> Amount: $10

jshell> new Account(10).credit(1)
$.. ==> Amount: $11

jshell> new Account(10).credit(1).debit(2)
$.. ==> Amount: $9
```

Another programmer handles the `static` methods for crediting, such as `deposit`, `interest`, etc. while the last programmer handles the `static` methods for debiting, such as `withdraw`, `loan`, etc.

To maintain the integrity of the system, crediting methods can only increase the amount of an account, while debiting methods can only decrease the amount.

```
jshell> Creditor.deposit(new Account(20), 10)
$.. ==> Amount: $30

jshell> Debitor.withdraw(new Account(20), 10)
$.. ==> Amount: $10
```

Design the system and include the implementations to meet the requirements of the above sample runs. For simplicity, you do not need to perform validation checks such as debiting more than the amount in the account, crediting a negative amount, etc.

3. [**4 marks; Q3.jsh**] Study the following `jshell` program fragment.

```
jshell> int sum = 0
sum ==> 0

jshell> int doSomething(int x) {
   ...>     return sum + x * x;
   ...> }
|  created method doSomething(int)

jshell> for (int i = 0; i < 10; i++) {
   ...>     sum = doSomething(i);
   ...> }

jshell> sum
sum ==> 285
```

Notice that the above uses *side-effects* in terms of updating the variables `sum` and `i`. Rewrite the code fragment above to remove the side-effects.

Hint: Other than making the loop functionality side-effectless, you should also retain the use of the `doSomething` method, but remove its side-effect.

4. [**5 marks; Q4.jsh**] Study the following generic method `processList` which takes in a list of elements of `Comparable` type `T`. The list could possibly be `null` or comprises `null` elements.

```
<T extends Comparable<T>> T processList(List<T> list) {
    T max = null;
    try {
        max = list.get(0);
        for (T item : list) {
            if (item.compareTo(max) > 0) {
                max = item;
            }
        }
    } catch (NullPointerException | ArrayIndexOutOfBoundsException ex) {
        return max;
    }
    return max;
}
```

Even though Java exceptions are used in handling `null` values, the main logic of the method relies on the occurrence of the `null` value within the list, and hence should not be considered as exceptional or rarely occurring.

By removing Java exceptions, deduce the behaviour of the `processList` method and rewrite it by completing the `return` statement below:

```
<T extends Comparable<T>> T processList(List<T> list) {
    return ...
}
```

You are not allowed to create other methods.

5. [**4 marks; Q5.jsh**] You are given the `Pair` class and `foo` method as shown below.

```
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;
    }

    @Override
    public String toString() {
        return "(" + this.t + ", " + this.u + ")";
    }
}

Stream<Pair<Integer,String>> foo(List<Pair<Integer,String>> list) {
    return list.stream().
        filter(x -> x.first() % 2 == 0);
}
```

For the following `jshell` program fragment

```
Scanner sc = new Scanner(System.in)

List<Pair<Integer,String>> list = IntStream.range(1,10).
    mapToObj(x -> new Pair<Integer,String>(x, sc.next())).
    collect(Collectors.toList())

foo(list).forEach(System.out::println)
```

The input
`the quick brown fox jumps over the lazy dog`
will produce the output

```
jshell> foo(list).forEach(System.out::println)
(2, quick)
(4, fox)
(6, over)
(8, lazy)
```

By employing the concept of *delayed data*, rewrite the `foo` method to meet the following specifications:

- `foo` should take as the first argument a `List` of `Pair`s of any type
- `foo` should take as the second argument a test of the first element of the `Pair`
- `foo` should only assign input to valid elements that pass the test
- You should not include other methods

Here is a sample `jshell` script that you can use a guide:

```
Scanner sc = new Scanner(System.in)

jshell> List<Pair<Integer,Supplier<String>>> list =
   ...>    IntStream.range(1,10).
   ...>    mapToObj(x ->
   ...>        new Pair<Integer,Supplier<String>>(x, () -> sc.next())).
   ...>    collect(Collectors.toList())
list ==> [(1, $Lambda$34/5406421720@5a8e6209), ... da$34/5406421720@5a8e6209)]

jshell> foo(list, x -> x % 2 == 0).forEach(System.out::println)
(2, the)
(4, quick)
(6, brown)
(8, fox)
```

6. Given a series of elements of the same type `T`, the *second-best* element is the one which is *next-after-the-best* when elements are ranked in some specified ordering. A skeleton program for the `SecondBest` class is given below:

```
class SecondBest<T> {
    private final T first;
    private final T second;
    private final Comparator<? super T> cmp;

    SecondBest(T x, T y, Comparator<? super T> cmp) {
        this.cmp = cmp;
        if (cmp.compare(x,y) < 0) {
            this.first = y;
            this.second = x;
        } else {
            this.first = x;
            this.second = y;
        }
    }

    @Override
    public String toString() {
        return "" + this.second;
    }
}
```

For each of the parts below, write the corresponding method following the sample run provided.

(a) **[2 marks; Q6a.jsh]** Complete the `add` method that adds an element to the `SecondBest`. You may just write the two `return` statements in your answer but make sure you order them correctly.

```
    SecondBest<T> add(T t) {
        if (this.cmp.compare(second, t) < 0) {
            return ...
        } else {
            return ...
        }
    }
```

```
jshell> new SecondBest<Integer>(1, 2, (x, y) -> x - y)
$.. ==> 1

jshell> new SecondBest<Integer>(1, 2, (x, y) -> x - y).
   ...> add(3)
$.. ==> 2
```

```
jshell> new SecondBest<Integer>(1, 2, (x, y) -> x - y).
   ...> add(3).
   ...> add(4).
   ...> add(2)
$.. ==> 3
```

(b) **[2 marks; Q6b.jsh]** Write the `map` method that maps the elements of the `SecondBest` and re-orders it based on the new `Comparator` passed in. **Use only one `return` statement in the method body.**

```
jshell> new SecondBest<Integer>(1, 2, (x, y) -> x - y).
   ...> map(x -> x % 2, (x,y) -> x - y)
$.. ==> 0

jshell> new SecondBest<String>("hi", "five", (x, y) -> x.compareTo(y)).
   ...> map(x -> x.length(), (x,y) -> x - y)
$.. ==> 2
```

(c) **[2 marks; Q6c.jsh]** Write the `flatMap` method that maps the two elements of a `SecondBest` to another `SecondBest` as specified by mapping. **Use only one `return` statement in the method body.**

```
jshell> new SecondBest<String>("hi", "five", (x, y) -> x.compareTo(y)).
   ...> flatMap((x,y) -> new SecondBest<Integer>(
   ...>     x.length(), y.length(), (p,q) -> p - q))
$.. ==> 2

jshell> new SecondBest<String>("hi", "five", (x, y) -> x.compareTo(y)).
   ...> flatMap((x,y) -> new SecondBest<Integer>(
   ...>     x.length(), 5 - y.length(), (p,q) -> p - q))
$.. ==> 1
```

(d) **[2 marks; Q6d.jsh]** Write the method `flip` that flips the ordering of the `SecondBest`. **Use only one `return` statement in the method body.**

```
jshell> new SecondBest<Integer>(1, 2, (x, y) -> x - y).flip()
$.. ==> 2

jshell> new SecondBest<Integer>(1, 2, (x, y) -> x - y).flip().
   ...> add(3)
$.. ==> 2

jshell> new SecondBest<Integer>(1, 2, (x, y) -> x - y).flip().
   ...> add(3).
   ...> add(4)
$.. ==> 2
```

7. [**4 marks; Q7.jsh**] Suppose you are given the following methods:

   - `int a()` that returns a value of type `int`; calling `a()` at different times will give different values
   - `int b()` that returns a value of type `int`; calling `b()` at different times will give different values
   - `void add(int x, int y)` that outputs the result of `x + y`
   - `void sub(int x, int y)` that outputs the result of `x - y`
   - `void mul(int x, int y)` that outputs the result of `x * y`

   For some unknown reasons, all the above methods take some time to complete, and the time taken for each method to complete varies throughout the day.

   By making use of asynchronous computation techniques, output the sum, difference and product of `a()` and `b()` via the `add`, `sub` and `mul` methods as efficiently as possible. As an example if `a()` and `b()` returns 4 and 2 respectively, then `add`, `sub` and `mul` should return 6, 2 and 8 correspondingly. Also note that as the asynchronous computation proceeds, we should still be allowed to perform other independent tasks.

— END OF PAPER —