National University of Singapore

SCHOOL OF COMPUTING

CS2030 — PROGRAMMING METHODOLOGY II
(Semester 1: AY2018/2019)

Time Allowed: 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper consists of **FIVE(5)** questions and comprises **SIXTEEN(16)** printed pages, including this page.
2. Answer **ALL** questions in the spaces provided. You may use pen or pencil.
3. This is an **OPEN BOOK** assessment. The maximum mark is **80**.
4. Calculators are allowed, but not electronic dictionaries, notebooks, tablets, or other computing devices.
5. Do not look at the questions until you are told to do so.
6. Please write your **Student number** below. Do not write your name.

This portion is for examiner's use only.

Question	Marks	Remarks
Q1	/16	
Q2	/16	
Q3	/16	
Q4	/16	
Q5	/16	
Total	/80	

1. [**16 marks**] Consider the following Java implementation for a banking application that facilitates the transfer of money between two accounts.

```
import java.util.Scanner;

class BankApp {
    static void transfer(Account source, Account target, double amount) {
        source.withdraw(amount);
        target.deposit(amount);
    }

    static double getAmount() {
        System.out.print("Enter amount to transfer: ");
        return new Scanner(System.in).nextDouble();
    }

    public static void main(String[] args) {
        Account s = getAccount();
        Account t = getAccount();
        double amt = getAmount();
        transfer(s, t, amt);
        System.out.println("Transfer successful");
    }

    // getAccount and other methods omitted
}

class Account {
    private double balance;

    Account(double balance) {
        this.balance = balance;
    }

    void withdraw(double amount) {
        this.balance = this.balance - amount;
    }

    void deposit(double amount) {
        this.balance = this.balance + amount;
    }

    // other methods omitted
}
```

You may assume that the functionality for getting accounts to initiate the transfer has been handled correctly. **This question focuses only on the transfer of money between two valid accounts**.

By employing good OOP design principles, rewrite the `BankApp` and `Account` classes to include the following:

- Check that the transfer amount is greater than zero.
- Check that the transfer amount is within the balance of the withdrawal account.
- Ease of inclusion of different types of accounts, each with a specific withdrawal limit. Include a `SavingsAccount` with a withdrawal limit of $1000.
- Terminate the transfer immediately for any violations above.
- Ensure that a deposit does not follow a failed withdrawal.

ANSWER:

— **BLANK PAGE** —

2. [**16 marks**] The following program shows a typical setup for a system that comprises a **console** that handles the input/output, and the (business-)**logic** part of the system.

```
import java.util.Scanner;

class Console {
    private String id;
    private Logic logic;

    Console(String id, Logic logic) {
        this.id = id;
        this.logic = logic;
    }

    void start() {
        Scanner sc = new Scanner(System.in);
        String command;
        do {
            System.out.print("Enter a command: ");
            command = sc.next();
            logic.invoke(command, this);
        } while (!command.equals("exit"));
    }

    void feedback(String mesg) {
        System.out.println(id + ": " + mesg);
    }
}

class Logic {
    void invoke(String command, Console console) {
        // do something based on the command
        console.feedback(command + " executed");
    }
}

class Main {
    public static void main(String[] args) {
        Logic logic = new Logic();
        Console console = new Console("main", logic);
        console.start();
    }
}
```

In particular, when a command is entered via the console, the logic component invokes the command and initiates a feedback call to the console. In the above design, the `Console` class is dependent on the `Logic` class (via the private instance field), while the `Logic` class depends on the `Console` class (via the parameter in method `invoke`).

This establishes a cyclic-dependency, which makes isolation and testing of individual components difficult. A sample run of the above program is given in the following.

```
Enter a command: load
main: load executed
Enter a command: store
main: store executed
Enter a command: exit
main: exit executed
```

Redesign the system to remove the cyclic dependency while maintaining the feedback call. Moreover, the system should facilitate the inclusion of other secondary consoles that receives the same feedback as the primary console. A sample run is given below where input is provided via the `primary` console, and feedback is provided to both `primary` and `secondary` consoles.

```
Enter a command: load
primary: load executed
secondary: load executed
Enter a command: store
primary: store executed
secondary: store executed
Enter a command: exit
primary: exit executed
secondary: exit executed
```

ANSWER:

— **BLANK PAGE** —

— **BLANK PAGE** —

3. [**16 marks**] Write a static method `findMinMax` with the signature

```
static Optional<MinMax> findMinMax(Stream<Integer> instream)
```

that takes a `Stream` of `Integer` values and finds both the maximum and minimum values via the `MinMax` class given below.

```
class MinMax {
    final int min, max;

    public MinMax(int min, int max) {
        this.min = min;
        this.max = max;
    }

    @Override
    public String toString() {
        return min + ", " + max;
    }
}
```

Take note of the following:

- An `Optional<MinMax>` empty instance is returned if the input stream is empty
- The steam pipeline should work if parallelized
- You are **not allowed** to use any Java collections

Using the following program fragment as an example

```
System.out.print("From range: ");
int from = (new Scanner(System.in)).nextInt();
System.out.print("To range: ");
int to = (new Scanner(System.in)).nextInt();

Stream<Integer> instream = IntStream
    .rangeClosed(from, to)
    .mapToObj(x -> Integer.valueOf(x));

System.out.println(findMinMax(instream));
```

the sample runs are:

- ```
  From range: -123
  To range: 456
  Optional[-123, 456]
  ```
- ```
  From range: -123
  To range: -456
  Optional.empty
  ```

ANSWER:

```
import java.util.stream.Stream;
import java.util.Optional;

    ....

    static Optional<MinMax> findMinMax(Stream<Integer> instream) {
```

4. **[16 marks]** The following depicts a classic tail-recursive implementation for finding the sum of values of n (given by $\sum_{i=0}^{n} i$) for $n \geq 0$.

```
static long sum(long n, long result) {
    if (n == 0) {
        return result;
    } else {
        return sum(n - 1, n + result);
    }
}
```

In particular, the implementation above is considered **tail-recursive** because the recursive function is at the tail end of the method, i.e. no computation is done after the recursive call returns. Using an example, `sum(100, 0)` gives `5050`. However, this recursive implementation causes a `java.lang.StackOverflowError` error for large values such as `sum(100000, 0)`.

Although the tail-recursive implementation can be simply re-written in an iterative form using loops, we desire to capture the original intent of the tail-recursive implementation using delayed evaluation via the `Supplier` functional interface.

We represent each recursive computation as a `Compute<T>` object. A `Compute<T>` object can be either:

- a recursive case, represented by a `Recursive<T>` object, that can be recursed, or
- a base case, represented by a `Base<T>` object, that can be evaluated to a value of type `T`.

As such, we can rewrite the above `sum` method as

```
static Compute<Long> sum(long n, long s) {
    if (n == 0) {
        return new Base<>(() -> s);
    } else {
        return new Recursive<>(() -> sum(n - 1, n + s));
    }
}
```

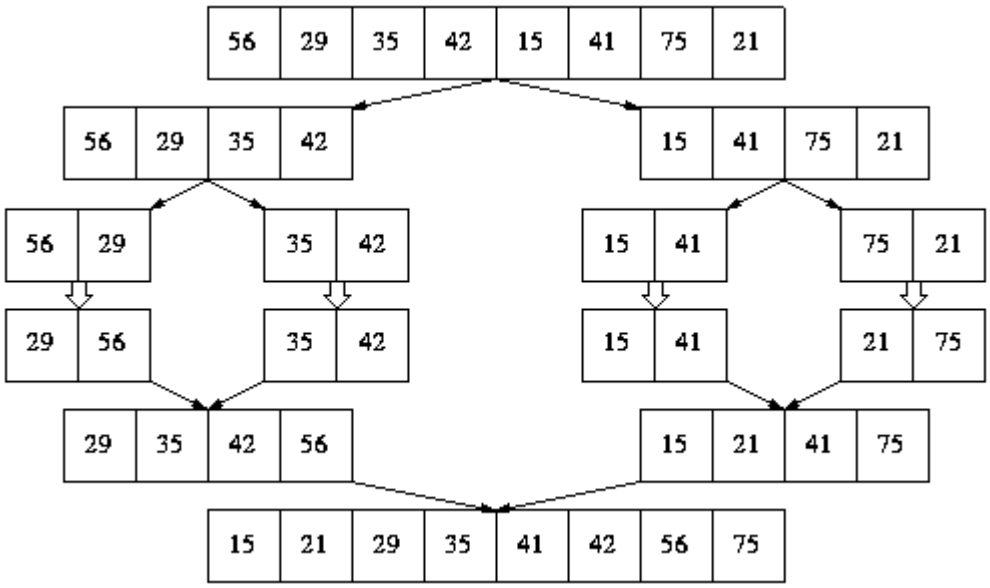
Complete the program by writing the `Compute`, `Base` and `Recursive` classes. By making use of a suitable client class `Main`, show how the "tail-recursive" implementation is invoked.

ANSWER:

— **BLANK PAGE** —

— **BLANK PAGE** —

5. **[16 marks]** Merge sort is a divide-and-conquer sorting technique that divides a list of elements into two halves, and applies the method recursively to the sub-lists. Traditionally, this sub-division is applied until a sub-list of one element, and merging of the sub-lists then takes place. However, the sub-list may be deemed small enough such that applying a more conventional sorting technique will result in the list being sorted more quickly. The figure below shows this case where sub-lists of two elements are immediately sorted, so that merging can then take place.



In the context of concurrent programming, implement the above sorting technique on a `List` of elements of a generic type `T`. Note the following:

- Set up a `MergeSortTask` as a `RecursiveAction` task from Java's fork/join framework. For example, merge-sorting a `List` of type `Integer` would be invoked as

  ```
  MergeSortTask<Integer> task = new MergeSortTask<Integer>(integerList);
  task.compute();
  ```

- Use `List`'s `subList()` method to divide the list.

  ```
  List<E> subList(int fromIndex, int toIndex)
  ```

  ```
  Returns a view of the portion of this list between the specified fromIndex,
  inclusive, and toIndex, exclusive.
  ```

- Use `Collections.sort()` to sort a sufficiently small list. You may decide on a suitable threshold on the length of the list.

  ```
  public static <T extends Comparable<? super T>> void sort(List<T> list)
  ```

  ```
  Sorts the specified list into ascending order, according to the natural
  ordering of its elements.  All elements in the list must implement the
  Comparable interface.
  ```

- Abstract the merging of sub-lists as a separate `merge` method

  ```
  void merge(List<T> list, List<T> leftHalf, List<T> rightHalf)
  ```

ANSWER:

— BLANK PAGE —

— END OF PAPER —