NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
PRACTICAL ASSESSMENT II FOR
Semester 2 AY2022/2023

CS2030S Programming Methodology II

April 2023 Time Allowed 90 minutes

INSTRUCTIONS TO CANDIDATES

1. This practical assessment consists of **one** question. The total mark is 20: 2 marks for design; 2 for style; 16 for correctness. Style and correctness are given on the condition that reasonable efforts have been made to solve the given tasks.
2. This is an OPEN BOOK assessment. You are only allowed to refer to written/printed notes. No online resources/digital documents are allowed, except those accessible from the PE nodes (peXXX.comp.nus.edu.sg) (e.g., man pages are allowed).
3. You should see the following files/directories in your home directory.
   - `Bank.java`, `Account.java`, and `Customer.java` are for you to edit and solve the given task.
   - `pristine`, that contains a clean copy of the files above for your reference.
   - `Test1.java`, `Test2.java`, and `Test3.java`, for testing your code.
   - `cs2030s/fp`, that contains the cs2030s.fp package, including `Maybe.java`.
   - `checkstyle.sh`, `checkstyle.jar`, and `cs2030s_checks.xml`, for style checks.
   - `StreamAPI.md` and `MapAPI.md`, for references to Java's Stream and Map API.
4. Solve the programming tasks by editing `Bank.java`, `Account.java`, and `Customer.java`. You can leave the files in your home directory and log off after the assessment is over. There is no separate step to submit your code.
5. Only the files directly in your home directory will be graded. Do not put your code under a subdirectory.
6. Write your student number on top of EVERY FILE you edited as part of the `@author` tag. Do not write your name.
7. To compile your code, run `javac -Xlint:unchecked -Xlint:rawtypes *.java`. You can also compile the files individually if you wish.
8. You can run each test individually. For instance, run `java Test1` to execute `Test1`.
9. To check the style of your code, run `./checkstyle.sh`. You can also check the style of individual file with `java -jar checkstyle.jar -c cs2030_checks.xml <FILENAME>`.

IMPORTANT: If the submitted classes `Bank.java`, `Account.java`, and `Customer.java` cannot be compiled, 0 marks will be given for the corresponding task. Make sure your program compiles by running

```
javac -Xlint:unchecked -Xlint:rawtypes *.java
```

before submission.

You have been given three classes `Bank`, `Account`, and `Customer`.

Customers. Each `Customer` has a name and an address. Customers can change their names and their addresses using the `changeName` and `changeAddress` methods respectively.

Account. An `Account` keeps track of an account number, the account balance, one or more account owners, and a boolean flag to indicate if the account has been closed. An account can have money deposited into it with the `deposit` method and withdrawn from it using the `withdraw` method. Money can also be transferred between two bank accounts using the `transferTo` method. An account can have a negative balance. Two `Account`s can be combined with the `combine` method. The combined `Account` has a combined list of owners and an account balance that is the sum of the two accounts combined. The account number of the combined account is the account number of the first account.

Bank. The `Bank` class keeps track of accounts using a `Map` class, with account numbers as keys. The method `allAccounts` returns a string listing all the accounts in the bank; The method `totalMoneyInBank` computes the total amount of money in the bank (excluding closed accounts). You can close an account using the `closeAccount` method. You can `deposit` and `withdraw` money to a specific account. Money can be transferred between two bank accounts using the `transfer` method. Both accounts must not be closed. If one or both of the accounts is/are closed, the method does nothing. The account to transfer money from must have a positive balance. Otherwise, the method does nothing. Finally, a closed account can be combined into a non-closed account with `combineBankAccount`. If the account to be combined from is not closed, or the account to be combined into is closed, the method does nothing. If the methods above are applied to a non-existing account, the methods do nothing.

You should read through the files `Account.java`, `Customer.java`, and `Bank.java` to understand what methods they have, the parameters and return value of the methods, and their behavior in detail.

The accounts are stored using a `Map` (a collection of key-value pairs) as you have seen in Lab 5: `Maybe`. The keys in a map are unique, so you cannot have duplicate keys. To put a key-value pair into a map `m` use `m.put(K key, V value)`, and to remove a key-value pair from a map `m` use `remove(Object key)`. Although `Map` may throw exceptions, you do not have to handle exceptions and may assume that all test cases will not cause exceptions.

Task 1 (4 marks): Rewrite `Account.java` and `Customer.java` classes to be immutable.
All methods that mutate the class should return a new object. You are not allowed to add new methods in `Account.java` and `Customer.java`. As `Account` is dependent on `Customer` we advise that you start by modifying the `Customer` class first, followed by the `Account` class. Make small changes at a time and make sure that your code compiles after every change.

Note that for `Account::transfer`, you are expected to return a `Pair` of accounts `(first, second)` such that `first` corresponds to the current account and `second` corresponds to the `toAccount` after the transfer.

The tests `Test1.java` and `Test2.java` will test the immutability of `Customer.java`, and `Account.java` respectively. Note that the given skeleton does not correctly pass all these tests. As in previous labs and PE1, make sure all OO principles, including LSP, tell-don't-ask, and information hiding, are adhered to, where applicable.

Task 2 (12 marks): Remove all loops, conditional statements, and `null` from `Bank`
The `Bank` class has multiple methods that have loops, conditional statements, and `null`. Remove these from the methods in `Bank.java`, except from `Bank::equals`, whilst retaining all of the functionality of `Bank.java`, You may use Java `Stream` and our `Maybe` to achieve this result.

Note that you are also not allowed to use block statements in your lambdas and you are not allowed to add new methods in `Bank.java`. For instance, the following is not allowed `x.map(x -> { .. })`.

To begin, you must change the return type of `findAccount` into `Maybe<Account>`. Make small changes at a time and make sure that your code compiles after every change.

The last test `Test3.java` will test the functionality of `Bank.java`. It correctly works on the mutable implementation you have been given.

END OF PAPER