# National University of Singapore

## SCHOOL OF COMPUTING

**CS2030 — PROGRAMMING METHODOLOGY II**
(Semester 2: AY2022/2023)

Apr / May 2023 Time Allowed: 2 Hours

---

## INSTRUCTIONS TO CANDIDATES

1. This assessment paper consists of **FIVE(5)** questions and comprises **NINE(9)** 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_EX** with the task titled **CS2030 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** | 4 | 10 | 12 | 10 | 4 | 40 |

1. [**4 marks**] A route is a pathway between an origin and a destination. Each route is associated with a distance (in km) and is undirectional, i.e. a route from `A` to `B` and a route from `B` to `A` are different.

   Complete the given class `Route` with the following specifications

```
class Route {
    private final String origin;
    private final String destination;
    private final String distance;
    ...
```

   (a) [**2 marks**] Write two constructors for the `Route` class:
   - A three argument constructor that takes in the origin, destination and the distance;
   - A two argument constructor that takes in the origin and destination. In this case, the distance should be set to an empty `String`.

   (b) [**1 mark**] Write a method `getDistance` that takes no arguments and returns the distance of a route.

   (c) [**1 mark**] Write a method `toString` that outputs the route according to the sample run below.

```
jshell> new Route("Clementi", "BuonaVista", "12")
$.. ==> Clementi --> BuonaVista

jshell> new Route("Clementi", "BuonaVista", "12").getDistance()
$.. ==> "12"

jshell> new Route("Clementi", "BuonaVista")
$.. ==> Clementi --> BuonaVista

jshell> new Route("Clementi", "BuonaVista").getDistance()
$.. ==> ""
```

2. [**10 marks**] Write a `City` class to model a city comprising of a list of routes. We can obtain the distance of a specific route from the city, and also update a route in the city by either adding it as a new route, or editing an existing route. Complete the given `City` class following the specifications below:

```
import java.util.Optional;
class City {
    ...
```

   (a) [**2 marks**] Include a property of type `ImList` to represent the routes of the city. Write a constructor with no arguments to initialize a `City` object with no routes. You may add other `private` constructors as you deem necessary later.

   (b) [**3 marks**] Write an `updateRoute` method that takes in a `Route` and updates the list of routes in the city. Note that the locations are **case insensitive**, e.g. the origin `Clementi` is the same as the origin `clementi`. You may also assume that routes added have positive distances.

   (c) [**3 marks**] Write a `getDistance` method that takes in a `Route` and returns the distance of the route in the city wrapped within `Optional`. For a route that cannot be found in the city, return an `Optional.empty()` instead.

   (d) [**2 marks**] To adhere to proper design abstraction, include additional method(s) in the `Route` class.

```
jshell> City city = new City()
city ==> City@d2cc05a

jshell> city.updateRoute(new Route("Clementi", "BuonaVista", "12"))
$.. ==> City@548a9f61

jshell> city
city ==> City@d2cc05a

jshell> city = city.updateRoute(new Route("Clementi", "BuonaVista", "12"))
city ==> City@2a2d45ba

jshell> city.getDistance(new Route("clementi", "buonavista"))
$.. ==> Optional[12]

jshell> city.getDistance(new Route("clementi", "jurongwest"))
$.. ==> Optional.empty

jshell> city = city.updateRoute(new Route("Clementi", "JurongWest", "24"))
city ==> City@28feb3fa

jshell> city = city.updateRoute(new Route("clementi", "buonavista", "11"))
city ==> City@51565ec2

jshell> city.getDistance(new Route("clementi", "buonavista", "12"))
$.. ==> Optional[11]
```

For the rest of the questions, you will need to study the following `CityGuide` class. A copy of the class is given to you.

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

class CityGuide {
    private static final int PARAM_POSITION_START_LOCATION = 0;
    private static final int PARAM_POSITION_END_LOCATION = 1;
    private static final int PARAM_POSITION_DISTANCE = 2;
    private static final String MESG_INVALID_FORMAT = "invalid command format: %s";
    private static final String MESG_NO_ROUTE = "No route exists from %s to %s!";
    private static final String MESG_DISTANCE = "Distance from %s to %s is %s";
    private static final String MESG_UPDATED =
        "Route from %s to %s with distance %skm updated";

    public static void main(String[] args) {
        City city = new City();
        Scanner scanner = new Scanner(System.in);
        while (scanner.hasNextLine()) {
            String userInput = scanner.nextLine();
            String command = parseCommand(userInput);
            String feedback = "";
            if (command == "UPDATE_ROUTE") {
                List<String> parameters = splitParameters(userInput);
                Pair<String,City> pair = updateRoute(parameters, city);
                feedback = pair.first();
                city = pair.second();
            } else if (command == "GET_DISTANCE") {
                List<String> parameters = splitParameters(userInput);
                feedback = getDistance(parameters, city);
            } else {
                feedback = String.format(MESG_INVALID_FORMAT, userInput);
            }
            System.out.println(feedback);
        }
    }

    static String parseCommand(String userInput) {
        String commandString = getFirstWord(userInput);
        if (commandString.equalsIgnoreCase("updateroute")) {
            return "UPDATE_ROUTE";
        } else if (commandString.equalsIgnoreCase("getdistance")) {
            return "GET_DISTANCE";
        } else {
            return "INVALID";
        }
    }
```

```
    static List<String> splitParameters(String commandParametersString) {
        return List.of(removeFirstWord(commandParametersString)
            .trim().split("\\s+"));
    }

    static String getDistance(List<String> parameters, City city) {
        String newStartLocation = parameters.get(PARAM_POSITION_START_LOCATION);
        String newEndLocation = parameters.get(PARAM_POSITION_END_LOCATION);

        Route other = new Route(newStartLocation, newEndLocation);
        Optional<String> os = city.getDistance(other);

        if (os.isEmpty()) {
            return String.format(MESG_NO_ROUTE, newStartLocation,
                newEndLocation);
        } else {
            return String.format(MESG_DISTANCE, newStartLocation, newEndLocation,
                os.get());
        }
    }

    static Pair<String, City> updateRoute(List<String> parameters, City city) {
        String newStartLocation = parameters.get(PARAM_POSITION_START_LOCATION);
        String newEndLocation = parameters.get(PARAM_POSITION_END_LOCATION);
        String distance = parameters.get(PARAM_POSITION_DISTANCE);

        Route other = new Route(newStartLocation, newEndLocation, distance);

        return Pair.<String,City>of(String.format(MESG_UPDATED, newStartLocation,
            newEndLocation, distance), city.updateRoute(other));
    }

    static String getFirstWord(String userInput) {
        return userInput.trim().split("\\s+")[0];
    }

    static String removeFirstWord(String userInput) {
        return userInput.replace(getFirstWord(userInput), "").trim();
    }
}
```

Suppose you are given the following input to the program comprising of commands to update a route, or get the distance of a route.

```
updateroute Clementi BuonaVista 12
getdistance Clementi BuonaVista
getdistance clementi buonavista
getdistance Clementi JurongWest
updateroute Clementi JurongWest 24
getdistance Clementi JurongWest
updateroute Clementi JurongWest 48
getdistance Clementi JurongWest
exit 1 2 3
```

The output of the program run is:

```
Route from Clementi to BuonaVista with distance 12km updated
Distance from Clementi to BuonaVista is 12
Distance from clementi to buonavista is 12
No route exists from Clementi to JurongWest!
Route from Clementi to JurongWest with distance 24km updated
Distance from Clementi to JurongWest is 24
Route from Clementi to JurongWest with distance 48km updated
Distance from Clementi to JurongWest is 48
invalid command format: exit 1 2 3
```

Here are some observations:

- There are two valid commands: `updateroute` and `getdistance`. Each of these commands is followed by a number of parameters; you may assume that the parameters for a corresponding command is always valid.
- Within the `main` method, input is read one line at a time, with `parseCommand` used to extract the specific command, and `splitParameters` used to form a list of parameters for that command.
- Based on the specific command, the parameters are passed to either `updateRoute` or `getDistance` methods. `getFirstWord` and `removeFirstWord` are helper methods to assist with command parsing.

You are **strongly advised** to understand how `CityGuide` works before proceeding.

3. [**12 marks**] The `parseCommand` method returns the "command" as a `String`, and the `main` method then decides which method to invoke on the city object. In this respect, the `CityGuide` class is parsing the command as well as invoking the command, which violates the *single responsibility principle*. Here, we shall first focus on implementing the different commands.

   (a) [**4 marks**] A `Command` can operate on different types of objects (we call them receivers of the command). As an example, `City` is a receiver, and `UpdateRoute` would be a command that operates on a `City` object.

   Write an abstract generic class `Command` that takes in a generic type `T` to facilitate the implementation of different commands on receivers of type `T`.

   Each implementation of a `Command` will invoke its command on the receiver via the `execute` method, which takes in a receiver object of type `T` as argument, and returns another object of type `T` after executing the command.

   (b) [**4 marks**] Write the `UpdateRoute` class as an implementation of a `Command`. The command is created with the list of parameters from `CityGuide` to facilitate updating the route in the city (refer to the `updateRoute` method). Perform the output as part of executing the command (we shall deal with this *side-effect* later).

   (c) [**4 marks**] Write the `GetDistance` class as another implementation of a `Command`. The command is created with the list of parameters from `CityGuide` to facilitate getting the distance of a route in the city (refer to the `getDistance` method). Perform the output as part of executing the command (we shall deal with this *side-effect* later).

   You may also notice the use of `Optional`'s `isEmpty()` and `get()` methods in `getDistance`. Re-implement so as to avoid the `isEmpty()`, `isPresent()` or any form of methods that exposes the object within the `Optional`. In other words, perform a proper *cross-barrier manipulation*.

```
jshell> City city = new City().
   ...> updateRoute(new Route("Clementi", "BuonaVista", "12"))
city ==> City@457e2f02

jshell> new UpdateRoute(List.of("Clementi", "JurongWest", "24")).execute(city)
Route from Clementi to JurongWest with distance 24km updated
$.. ==> City@39aeed2f

jshell> new GetDistance(List.of("Clementi", "BuonaVista")).execute(city)
Distance from Clementi to BuonaVista is 12
$.. ==> City@457e2f02

jshell> new GetDistance(List.of("Clementi", "JurongWest")).execute(city)
No route exists from Clementi to JurongWest!
$.. ==> City@457e2f02
```

4. [**10 marks**] With `CityGuide` focused on parsing the command, and each `Command` object focused on operating on the `City`, we now need an invoker to invoke the commands. The invoker plays two roles:

   - it collects the commands;
   - it processes all collected commands on the receiver object.

   The invoker invokes the commands *lazily*. That is to say, the invoker collects all commands first, and then executes all of them one by one, only when initiated by the client, in our case `CityGuide`.

   (a) [**4 marks**] Write an `Invoker` class with generic type `T` to facilitate invoking commands of type `T` receivers.

   TO DO !!!!

   - Include an appropriately type-parameterized `ImList` as a property of the class, and constructor(s) as you deem necessary.
   - Write an `addCommand` method that takes in a `Command` and adds to the list of commands. A new `Invoker<T>` is returned.
   - Write an `executeCommand` method that takes no arguments and returns an `Optional<Command<T>>` that encapsulates all collected commands. In the case of no commands, the method should return `Optional.empty()`.
     You should write a single return statement using Java streams; there is a `stream()` method in `ImList`.
     *Hint: include another method in the* `Command` *abstract class.*

   (b) [**3 marks**] Rewrite the `parseCommand` method in the `CityGuide` class to return a command instead. For the case of an invalid command, return an inner class (lambda or anonymous inner class) implementation of a command that outputs

   ```
   invalid command format: <userInput>
   ```

   (c) [**3 marks**] Rewrite the `main` method in the `CityGuide` class so that it now reads each line of user input, parses the command and adds the command to the invoker. After all commands have been added, the method proceeds to output the commands invoked upon a city with no routes. The output should be the same as that of the original `CityGuide` implementation.

   ```
   Route from Clementi to BuonaVista with distance 12km updated
   Distance from Clementi to BuonaVista is 12
   Distance from clementi to buonavista is 12
   No route exists from Clementi to JurongWest!
   Route from Clementi to JurongWest with distance 24km updated
   Distance from Clementi to JurongWest is 24
   Route from Clementi to JurongWest with distance 48km updated
   Distance from Clementi to JurongWest is 48
   invalid command format: exit 1 2 3
   ```

5. [**4 marks**] So far, an output occurs *during* the invocation of each command as a *side-effect*. Isolate these side effects by using the `Log` class that you have developed in one of your lab execises. The `Log` class has the following method implementations:

   - `static <T> Log<T> of(T t, String log)`
   - `static <T> Log<T> of(T t)`
   - `<R> Log<R> flatMap(Function<? super T, ? extends Log<? extends R>> mapper)`
   - `<R> Log<R> map(Function<? super T, ? extends R> mapper)`
   - `String getLog() // included to return the log`
   - `public boolean equals(Object obj)`
   - `public String toString()`

   (a) [**2 marks**] Rewrite the `Command` class to make use of `Log<T>` instead.
   *Although this entails that the individual implementations of* `Command` *need to be modified accordingly, there is no need to write this down in your answer since it merely involves wrapping the output in the* `Log`*.*

   (b) [**2 marks**] Rewrite the `main` method of the `CityGuide` class to work with the new implementation of the `Command` class so that the log of the accumulated `Log` object so as to give the same output.

```
Route from Clementi to BuonaVista with distance 12km updated
Distance from Clementi to BuonaVista is 12
Distance from clementi to buonavista is 12
No route exists from Clementi to JurongWest!
Route from Clementi to JurongWest with distance 24km updated
Distance from Clementi to JurongWest is 24
Route from Clementi to JurongWest with distance 48km updated
Distance from Clementi to JurongWest is 48
invalid command format: exit 1 2 3
```

**— END OF PAPER —**