IS1220 - Object Oriented Software Design

Tutorial 05

Paolo Ballarini

General instructions:

- Ask if you have any problem.
- Import the project that has been defined for your tutorial `fr.ecp.is1220.pc05.zip`.
- Carefully document your code (JavaDoc), i.e., explain the general idea of your algorithm, explain the relevant steps in the code implementing the algorithm, document assumptions (if any), corner cases, and error conditions.

Main Concepts:

- Java colletions
- Abstract classes
- JUnit testing

Exercise 1. A class hierarchy for Complex numbers

Bearing in mind that a complex number can be expressed as `a + bi`, where `a` and `b` are real numbers and `i` is the imaginary unit you are required to develop (using the Eclipse IDE) a class hierarchy for representing complex numbers. In particular

E1 Q1) Develop a class called `Complex` that extends class `java.lang.Number`. In so doing make sure that `Complex` implements all required methods (**Hint**: use the java.lang API to find documentation about `Number` and `Object`).

E1 Q2) Equip the `Complex` class with a `toString` to display `Complex` objects and `equals` for comparing a `Complex` objects.

E1 Q3) In addition, implement the following interface for the `Complex` class:

```
public interface BasicOps {

        // Add to this number and return it as the result
        public Number addNum(Number a);

        // Subtract from this number and return it as the result
        public Number subNum(Number b);

        // Multiply to this number and return it as the result
        public Number multNum(Number a);

        // Divide this number by and return it as the result
        public Number divNum(Number b);

    }
```

E1 Q4) Implement a `main` method and test all features (methods) of the class in it.

Similarly implement a class `Fraction`. This class should represent fraction numbers (e.g. 1/3) as two integers. Likewise Complex, Fraction should be derived from Number and implement the interface BasicOps. Implement a main method and test

all features (methods) of the class in it.

E1 Q5) Using both classes (`Complex` and `Fraction`), create a class Test with only a `main` method. In this method test the interaction between the two classes using the operations of the interface BasicOps, as the example below:

```
BasicOps a = new Complex(2, 3);
Number b = new Fraction(1, 3);
Number c = a.multNum(b);
System.out.println("Result = " + c);
```

Perform the tests with a reasonably large number of combinations of Complex and Fraction numbers and check that the results are correct!

E1 Q6) You used two Java mechanisms: subclassing and interfaces. When did you use each? Should Fraction be implemented as a subclass of Complex? Why?

E1 Q7) Using JUnit define appropriate test units for testing the implementation of the four basic operations for both classes Complex and Fraction. Run the tests and check the results.

Exercise 2. Code Design - the `MetroUserTerminal`

You are required to implement a Java program that allows the user to operate on maps of metro network of a given city, typically the user should be capable of searching which metro stations are in the network and searching for a path in the network connecting a source station to a destination station. For this you have to design an abstract data type (ADT) for representing graph-like structures which you will use for storing a dedicated representation of a metro network. Your program should be capable of loading a city metro network map by parsing it from a formatted textual file. You do not have to worry about the parsing of metromap files, for this you are given a Java implementation of a class called `MetroMapParser` (file `MetroMapParser.java` available on Claroline). However your client program (which you will implement in a class called `MetroUserTerminal`) will have to use the API of `MetroMapParser`. To test your implementation you are also given a formatted textual file called `bostonmetromap.txt` (available on Claroline) which contains a textual description of the *graph* consisting of the metro station of the Boston underground.

Table 1 describes the format of the textual file `bostonmetromap.txt` containing a textual description of a metropolitan network. Notice that for solving this exercise you do not need to really worry about the format of files like `bostonmetromap.txt` because in fact the parsing is already supported by `MetroMapParser` class which you are given (however it might be useful to be aware of the format of the textual file if you wish to figure out how the code in `MetroMapParser.java` works).

The `MetroUserTerminal` that you are required to realise is a *command-line* type of application, i.e. it is a program capable of executing a number of commands inputed by the user. The `MetroUserTerminal` should feature (at least) the following list of commands:

- `boston`: to load/parse the Boston metro network description contained in file `bostonmetromap.txt`
- `list`: to list the metro stations of the loaded metro map
- `connect` *station*1 *station*2: to display a path of the loaded metromap connection *station*1 to *station*2
- `succ` *station* : to display the list of station which are direct successors of *station*
- `help`: to display this list of commands (the commands featured by the `MetroUserTerminal`)

- `stop`: to halt the `MetroUserTerminal` program

On claroline you find also an image file `BostonMetroMap.png` with a schematic representation of boston metro map.

Below you find screenshots illustrating the output of the `MetroUserTerminal` should look like in response to different user commands.

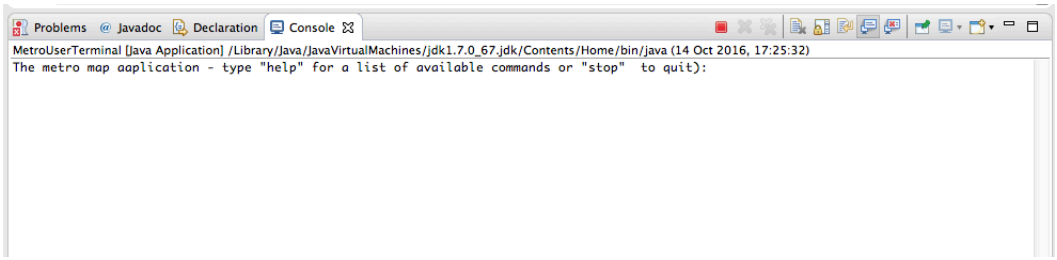

Figure 1: Initial state of the MetroUserTerminal when we launch it: we can type the `help` command to get a list of supported commands through which the user can interact with the terminal.

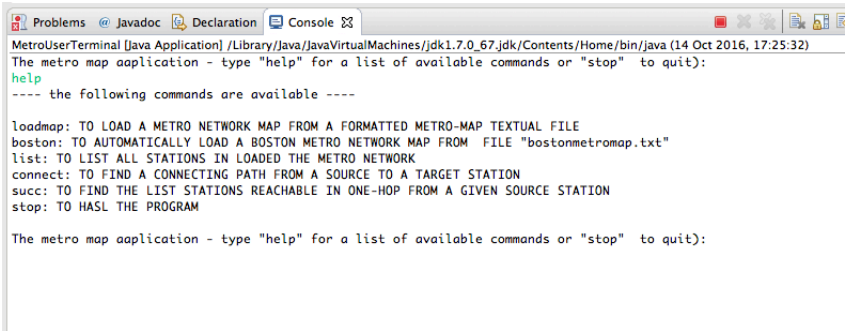

Figure 2: Output of the MetroUserTerminal resulting from processing of the `help` command.

Format of text files representing a metro network The `MetroMapParser` class defined in `MetroMapParser.java` provides one with functionality for parsing formatted textual files containing a representation of the metropolitan network of a city. Such kind of files contains a line for each station in the metro network. Each line is formatted indicated in Table 1:

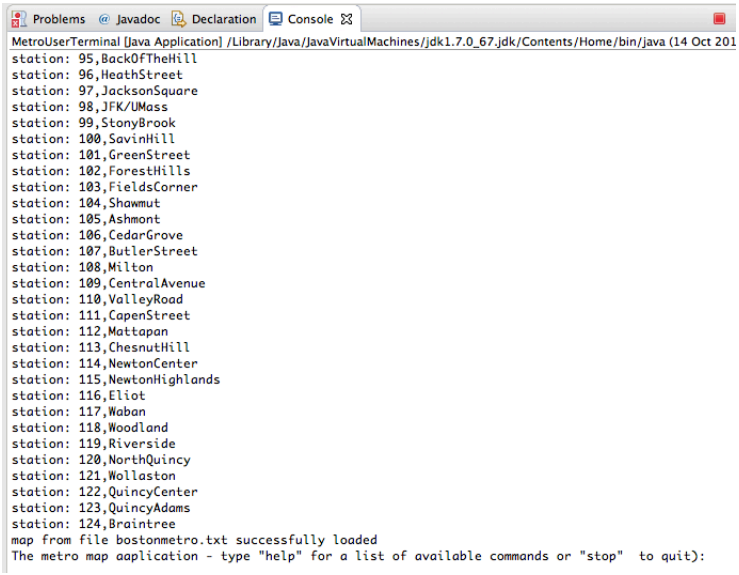

Figure 3: Output of the MetroUserTerminal resulting from processing of the `boston` command (for loading the map of boston metro network stored in formatted textual file `bostonmetromap.txt`).

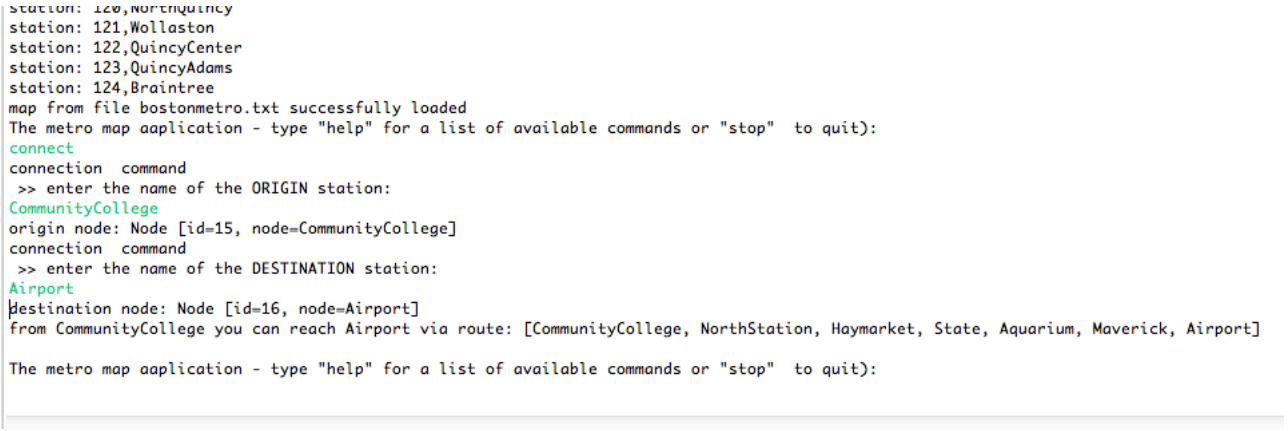

Figure 4: Output of the MetroUserTerminal resulting from processing of the `connect` command for finding a path from the CommunnityCollege station to the Airport station of the Boston metro network (you can verify the path on the image file `BostonMetroMap.png`).

Station num ID	**Station name**	**Station list of connected stations**
1	OakGrove	Orange 0 2
2	Malden	Orange 1 5
20	NorthStation	Green 19 22 Orange 15 22
...	...	...

Table 1: Formatted textual representation of a metropolitan network. Each line of the textual file corresponds to one station and describes the stations directly connected to it. In particular: the first field of a line (i.e. the first column of the Table) is the numeric ID of the station; second field (i.e. second column) is the station name; third field (i.e. third column) is the list of directly connected station, described as: name of the Line the station belong to, numeric ID of directly connected station on one direction followed by the numeric ID of the directly connected station on the other direction. Notice the ID equal to 0 indicates that the station described by this line is a terminal station on the corresponding line. Thus, for example, station OakGrove is a terminal station on the Orange line, whereas station NorthStation is a cross station between the Green and Orange lines. In particular NorthStation is connected with stations 19 and 22 on the Green line and with stations 15 and 22 on the Orange line.