The University of Queenland, School of ITEE
CSSE2002/7023 — Semester 1, 2019
Assignment 2 (15%)
Due: 03 May 2019 16:00
Revision: 2.1

Abstract

The goal of this assignment is to implement a set of classes and interfaces[1] to be used in later assignments. You will implement precisely the public and protected items described in the supplied documentation (no extra public/protected members or classes). Private members may be added at your own discretion.

Language requirements: Java version 8, JUnit 4

Preamble

All work on this assignment is to be your own individual work. As detailed in Lecture 1, code supplied by course staff (from this semester) is acceptable, but there are no other exceptions. You are expected to be familiar with "What not to do" from Lecture 1 and `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`. If you have questions about what is acceptable, please ask course staff.

Supplied Material

- This task sheet
- Code specification document (javadoc).[2]
- A subversion repository for submitting your assignment.[3] The repository contains supplied source code for you to build on, **you must use this as the base for your second assignment**. Do not modify the supplied files except to add required methods according to the `Javadoc`.

Javadoc

Code specifications are an important tool for developing code in collaboration with other people. Although assignments in this course are individual, they still aim to prepare you for writing code to a strict specification by providing a specification document (in Java, this is called Javadoc). The Javadoc outlines the required modifications which need to be made to the supplied support code. You will need to implement the specification precisely as it is described in the specification document. Viewing the Javadoc can be done as follows:

1. Open `https://csse2002.uqcloud.net/assignments/a2/` in your web browser, or navigate to the relevant assignments folder under `Assessment` on Blackboard.
2. Download the Javadoc `.zip` file containing html documentation. Unzip the bundle somewhere, and open `doc/index.html` with your web browser.

[1] From now on, classes and interfaces will be shortened to simply "classes"
[2] Detailed in the `Javadoc` section
[3] Detailed in the `Submission` section

Tasks

1. Fully implement each of the classes described in the Javadoc.
2. Write JUnit4 tests for the methods in the following classes:
    - `Network` (in a class called `NetworkTest`)

Marking

The 100 marks available for the assignment will be divided as follows:

Symbol	*Marks*	*Marked*	*Description*
F	55	Electronically	Functionality according to the specification
S	25	Course staff	Code style and design
J	20	Electronically	Whether JUnit tests identify and distinguish between correct and incorrect implementations

The overall assignment mark will be $A_1 = F + S + J$ with the following adjustments:

1. If $F < 5$, then $S = 0$ and code style will not be marked.
2. If $S > F$, then $S = F$.
3. If $J > F$, then $J = F$.

For example: $F = 22, S = 25, J = 17 \Rightarrow A_1 = 22 + 22 + 17$.
The reasoning here is to place emphasis on functional code and to not to give marks to well styled code and well implemented JUnit tests when the code is not functional.

Functionality Marking

The number of functionality marks given will be

$$F = \frac{\text{Tests passed}}{\text{Total number of tests}} \cdot 55$$

Each of your classes will be tested independently of the rest of your submission. Other required classes for the tests will be copied from a working version of the assignment.

Style Marking

Your assignment will be style marked with respect to the course style guide, located under `Learning Resources > Guides`. The marks are broadly divided as follows:

Naming	5
Commenting	6
Structure and Layout	8
Good Object-Oriented Practices	6

Note that style marking does involve some aesthetic judgement (and the marker's aesthetic judgement is final).

JUnit Test Marking

Marks will be awarded for test sets which distinguish between correct and incorrect implementations[4]. A test class which passes every implementation (or fails every implementation) will likely get a low mark. This will be assessed by running your JUnit test classes on a number of correct and incorrect assignment implementations. Marks will be rewarded for tests which pass or fail correctly.

There will be some limitations on your tests:

1. If your tests take more than 20 seconds to run, or
2. If your tests consume more memory than is reasonable or are otherwise malicious

then your tests will be stopped and a mark of zero given. These limits are very generous, (e.g. your tests shouldn't take anywhere near 20 seconds to run).

Electronic Marking

The electronic aspects of the marking will be carried out in a linux environment. The environment will not be running Windows, and neither IntelliJ nor Eclipse (or any other IDE) will be involved. *It is also critical that your code compiles.* If one of your classes does not compile, **you will receive zero** for any electronically derived marks for that class.

Submission

Submission is via your subversion repository. Details for how to submit your assignment is available in the `Version Control Guide`. Your repository url is:
`https://source.eait.uq.edu.au/svn/csse2002-s???????/trunk/ass2`.

Your submission should have the following internal structure:

`src/`	folders (packages) and `.java` files for classes described in the Javadoc
`test/`	folders (packages) and `.java` files for the JUnit test classes

A complete submission would look like:

```
src/exceptions/EmptyRouteException.java
src/exceptions/IncompatibleTypeException.java
src/exceptions/NoNameException.java
src/exceptions/OverCapacityException.java
src/exceptions/TransportException.java
src/exceptions/TransportFormatException.java
src/network/Network.java
src/passengers/ConcessionPassenger.java
src/passengers/Passenger.java
src/routes/BusRoute.java
src/routes/FerryRoute.java
src/routes/Route.java
src/routes/TrainRoute.java
src/stops/Stop.java
src/vehicles/Bus.java
src/vehicles/Ferry.java
src/vehicles/PublicTransport.java
```

[4] And get them the right way around

`src/vehicles/Train.java`
`src/utilities/Writeable.java`
`test/network/NetworkTest.java`

Ensure that your assignments correctly declare the package they are within. For example, `Network.java` should declare `package network`.
You may submit additional text files to your repository for use in your test classes.

Prechecks

Prechecks will be performed on your assignment repository twice before the assignment is due. They will assess whether your folders and files are in the correct structure and whether your public interface aligns with the expected public interface. **Successfully passing a precheck does not guarantee any marks. No functionality or style is assessed.**

`Precheck #1: Approximately 4pm on the 26/04`
`Precheck #2: Approximately 4pm on the 01/05`

Please endeavour to have code written and in your repository before at least one of these prechecks in order to make the most of them. No additional prechecks will be run for people who did not start the assignment in time, or who neglected to commit their code to their repository. Prechecks are valid only for currently released version of the Javadoc, if an update is made it may invalidate the precheck results.

Late submission

Assignments submitted after the due date will recieve a mark of zero unless an extension is granted as outlined in the ECP — see the ECP for details.

Revisions

If it becomes necessary to correct or clarify the task sheet or Javadoc, a new version will be issued and a course announcement will be made on Blackboard. No changes will be made on or after 29/04/2019.

Change Log

26 April 2019

Task Sheet

- Included TransportFormatException in the list of expected files.
- Allowed the use submission of extra text files for testing purposes.

Javadoc

- Network.Network(String) now addresses what to do if there are spaces before or after integer, for negative integers, and for null filenames.
- Network.save() addresses what to do for null filenames.
- Stop.decode(String) addresses what to do for spaces before or after integers, and for empty Stop names in the stopString.

- Route.decode(String, List) addresses what to do for duplicate Stop names in the routeString or in the existingStops list, as well as what to do for spaces before or after integers. It also addresses empty stop names in the routeString, a routeString with no stops, and an empty route name in the routeString.
- PublicTransport.decode(String, List) addresses what to do for duplicate Routes in the existingRoutes list, as well as what to do for spaces before or after integers.

27 April 2019

Javadoc

- Removed contradiction relating to null handling in Network.Network(String) which was introduced in previous update