

CSSE2002 / CSSE7023

Semester 1, 2016

Assignment 2

Goal: The goal of this assignment is to gain practical experience with procedural abstraction – how complex functionality can be broken up between different methods and different classes.

Due date: The assignment is due at **4:00pm on Wednesday 11th May**. Late assignments will lose 20% of the total mark immediately, and a further 20% of the total mark for each day late.

Only extensions on Medical Grounds or Exceptional Circumstances will be considered, and in those cases students need to submit an application for extension of progressive assessment form (<http://www.uq.edu.au/myadvisor/forms/exams/progressive-assessment-extension.pdf>) no later than 48 hours prior to the submission deadline. The application and supporting documentation (e.g. medical certificate) must be submitted to the ITEE Coursework Studies office (78-425) or by email to enquiries@itee.uq.edu.au. If submitted electronically, you must retain the original documentation for a minimum period of six months to provide as verification should you be requested to do so.

School Policy on Student Misconduct: You are required to read and understand the School Statement on Misconduct, available on the School's website at:

<http://ppl.app.uq.edu.au/content/3.60.04-student-integrity-and-misconduct>

This is an individual assignment. If you are found guilty of misconduct (plagiarism or collusion) then penalties will be applied.

If you are under pressure to meet the assignment deadline, contact the course coordinator **as soon as possible**. Do not discuss or post solutions to the assignment on public forums, like piazza.

Problem Overview: In this assignment, you will continue to implement the component classes of a program for simulating the behaviour of a train management system.

Task Overview:

In brief, you will write a method for reading in the sections of a track from a file, and you will write a method for allocating routes to trains (so that they don't collide). If you are a CSSE7023 student you will also be required to write a JUnit4 test suite for testing the track-reading method.

More specifically, you must code method `read` from the `TrackReader` class and method `allocate` from the `Allocator` class that are available in the zip file that accompanies this handout, according to their specifications in those files.

If you are a CSSE7023 student, you will also need to complete a systematic and understandable JUnit4 test suite for the `read` method in the skeleton of the `TrackReaderTest` class from the `railway.test` package. You may write your unit tests assuming that the classes that `TrackReader` depends on (e.g. the `Junction`, `Section`, `Track` classes and any of the Java 8 SE API classes) are implemented and functioning correctly. That is, you don't need to create test stubs for these classes.

As in Assignment 1, you must complete these methods and classes as if other programmers were, at the same time, implementing classes that use it. Hence:

- Don't change the class names, specifications, or alter the method names, parameter types, return types, exceptions thrown or the packages to which the files belong.
- You are encouraged to use Java 8 SE API classes, but no third party libraries should be used. (It is not necessary, and makes marking hard.)

- Don't write any code that is operating-system specific (e.g. by hard-coding in newline characters etc.), since we will batch test your code on a Unix machine.
- Any new methods or fields that you add to `TrackReader` or `Allocator` must be private (i.e. don't change the specification of these classes.)
- Your source file should be written using ASCII characters only.

Implement the classes as if other programmers are going to be using and maintaining them. Hence:

- Your code should follow accepted Java naming conventions, be consistently indented, readable, and use embedded whitespace consistently. Line length should not be over 80 characters. (Hint: if you are using Eclipse you might want to consider getting it to automatically format your code.)
- Any additional methods that you write, and fields that you introduce should be private to hide implementation details and protect invariants.
- Private methods that you write must be commented using preconditions and postconditions (require and ensures clauses). Informal description is OK.
- Fields and local variables (except for-loop variables) should have appropriate comments. Comments should also be used to describe any particularly tricky sections of code. However, you should also strive to make your code understandable without reference to comments; e.g. by choosing sensible method and variable names, and by coding in a straightforward way.
- Any exceptions that are created and thrown should have appropriate messages to help the user understand *why* the exception was thrown. This is particularly important for the `read` method in `TrackReader`, since if there is an error with the file format, then the user will want to know what is wrong with it when a `FormatException` is thrown. Each `FormatException` thrown should have a meaningful message that accurately describes the problem with the input file format, including the line of the file where the problem was detected. (You can create a new `FormatException` with a message using the constructor that takes a string parameter.)
- The methods that you have to write must be decomposed into a clear and not overly complicated solution, using private methods to prevent any individual method from doing too much.

I recommend that you attempt to write loop invariants for all non-trivial while-loops in your code, but this is not compulsory.

The Zip file for the assignment also includes some other code that you will need to compile your classes as well as some junit4 test classes to help you get started with testing your code.

Do not modify any of the files in package `railway` other than `TrackReader` and `Allocator`, since we will test your code using our original versions of these other files. Do not add any new files that your code for these classes depends upon, since you won't submit them and we won't be testing your code using them.

The JUnit4 test classes as provided in the package `railway.test` are *not intended to be an exhaustive test for your code*. Part of your task will be to expand on these tests to ensure that your code behaves as required by the javadoc comments. (Only if you are a CSSE7023 student will you be required to submit your test file `TrackReaderTest.java`.) We will test your code using our own extensive suite of JUnit test cases. (Once again, this is intended to mirror what happens in real life. You write your code according to the "spec", and test it, and then hand it over to other people ... who test and / or use it in ways that you may not have thought of.)

If you think there are things that are unclear about the problem, ask on the piazza forum, ask a tutor, or email the course coordinator to clarify the requirements. Real software projects have requirements that aren't entirely clear!

If necessary, there may be some small changes to the files that are provided, up to 1 week before the deadline, in order to make the requirements clearer, or to tweak test cases. These updates will be clearly announced on the Announcements page of Blackboard, and during the lectures.

More about the `allocate` method from the `Allocator` class:

In this section we explain some of the terminology used in the specification of the `allocate` method.

- A *route* describes a path through a track. A route is a list of zero or more (non-null) *segments*.
- A segment, implemented as the class `Segment` from the `railway` package, is a part of route. A segment is located on one section of track, and it has a direction of travel along that section, and a first and last location that are on that section.

The direction of travel is specified with respect to the end-points of the section: one of the section end-points is designated as the departing end-point and the other as the approaching end-point. The direction of travel is away from the departing end-point towards the approaching end-point.

Let `startOffset` and `endOffset` be the offset of the segment's first location and last location (respectively) with respect to the departing end-point of the segment. We must have that `startOffset < endOffset`. For every offset `x` satisfying `startOffset <= x <= endOffset`, the segment of track contains every location at distance `x` from the segment's departing end-point.

The location at distance `startOffset` from the departing end-point is referred to as the first location on the segment, the location at distance `startOffset+1` from the departing end-point is the second location on the segment, etc. The length of the segment is `endOffset - startOffset`, and it contains `(1 + endOffset - startOffset)` locations. Since `startOffset < endOffset`, this length must be greater than or equal to one.

For example, consider a segment that lies on a section of length 9 with end-points `(j0, FACING)` and `(j1, FACING)`. If `(j1, FACING)` is the designated departing end-point of the segment, the first location is 3 meters from `(j1, FACING)` and the last location is 7 meters from `(j1, FACING)`, then the segment denotes a part of a route that starts at 3 meters from `(j1, FACING)`, and travels towards end-point `(j0, FACING)` until it reaches the location that is 7 meters from `(j1, FACING)`. It contains 5 locations, and is 4 meters long.

- A segment is *part of track* if it is on a section that lies on that track. A route is part of a track if all of its segments are part of the track.
- A route `r` is *valid* if and only if the following conditions are all satisfied:
 - The segments are connected, and they only connect at junctions. That is, for all indices `i` of route `r` satisfying `0 <= i < r.size()-1`, the last location of `r.get(i)` equals the first location of `r.get(i+1)`, and that location is at a junction.

- The direction of travel through junctions is possible. This means that if a train approaches a junction on a FACING branch, then it may only depart on the NORMAL or REVERSE branch of the junction. If it approaches the junction on one of the other two branches, then it may only depart the junction on the FACING branch. I.e. for all indices i of route r satisfying $0 \leq i < r.size() - 1$, if $(r.get(i)).getApproachingBranch()$ is FACING, then $(r.get(i+1)).getDepartingBranch()$ may be either NORMAL or REVERSE. If $(r.get(i)).getApproachingBranch()$ is not FACING, then $(r.get(i+1)).getDepartingBranch()$ must be FACING.
- A route $r1$ *intersects* another route $r2$, if there is a location that is contained in both $r1$ and $r2$. A route contains a location, if that location is contained in one or more of its segments.
- A segment p is a *prefix* of a segment s if both segments p and s have the same section, departing end-point and start offset (i.e. first location), and the length of p is less than or equal to the length of s . (As a special case, a segment is a prefix of itself.)
- A route p is a prefix of a route r if $p.size() \leq r.size()$ and for all indices i satisfying $0 \leq i < p.size() - 1$, we have that $p.get(i)$ equals $r.get(i)$ and if $p.size() \neq 0$, then the last segment in p , $p.get(p.size() - 1)$, is a prefix of $r.get(p.size() - 1)$. (As a special case, a route is a prefix of itself, and the empty route – the route with no segments – is a prefix of any route.)
- The length of a route r is the sum of the lengths of its constituent segments, i.e. the sum of $(r.get(i)).getLength()$ for each index i in the domain of the list.

Submission: Submit your files **TrackReader.java**, **Allocator.java** (and **TrackReaderTest.java** and any of your track files that are used for testing in **TrackReaderTest.java** if you are a CSSE7023 student) electronically using Blackboard according to the exact instructions on the Blackboard website:

<https://learn.uq.edu.au/>

You can submit your assignment multiple times before the assignment deadline but only the last submission will be saved by the system and marked. Only submit the files listed above.

You are responsible for ensuring that you have submitted the files that you intended to submit in the way that we have requested them. You will be marked on the files that you submitted and not on those that you intended to submit. Only files that are submitted according to the instructions on Blackboard will be marked.

Evaluation: If you are a CSSE2002 student, your assignment will be given a mark out of 15, and if you are a CSSE7023 student, your assignment will be given a mark out of 17, according to the following marking criteria. (Overall the assignment is worth 15% for students from both courses.)

Testing methods read and allocate (8 marks)

- | | |
|---|---------|
| • All of our tests pass | 8 marks |
| • At least 85% of our tests pass | 7 marks |
| • At least 75% of our tests pass | 6 marks |
| • At least 65% of our tests pass | 5 mark |
| • At least 50% of our tests pass | 4 marks |
| • At least 35% of our tests pass | 3 mark |
| • At least 25% of our tests pass | 2 mark |
| • At least 15% of our tests pass | 1 mark |
| • Work with little or no academic merit | 0 marks |

Note: code submitted with compilation errors will result in zero marks in this section. A Java 8 compiler will be used to test code. Each of your classes will be tested in isolation with our own valid implementations of the others.

Code quality (7 marks)

- | | |
|---|-----------|
| • Code that is clearly written and commented, and satisfies the specifications and requirements | 7 marks |
| • Minor problems, e.g., lack of commenting or private methods | 4-6 marks |
| • Major problems, e.g., code that does not satisfy the specification or requirements, or is too complex, or is too difficult to read or understand. | 1-3 marks |
| • Work with little or no academic merit | 0 marks |

Note: you will lose marks for code quality for:

- breaking java naming conventions or not choosing sensible names for variables;
- inconsistent indentation and / or embedded white-space or laying your code out in a way that makes it hard to read;
- having lines which are excessively long (lines over 80 characters long are not supported by some printers, and are problematic on small screens);**
- exposing implementation details by introducing methods or fields that are not private
- not commenting any private methods that you introduce using contracts (pre and postconditions specified using `@require` and `@ensure` clauses).
- not having appropriate comments for fields and local variables (except for-loop variables), or tricky sections of code;
- not setting an appropriate message for exceptions that are created and thrown
- monolithic methods: if methods get long, you must find a way to break them into smaller, more understandable methods using procedural abstraction. (HINT: very important!!)
- incomplete, incorrect or overly complex code, or code that is hard to understand.

** To make sure that your lines are not over 80 characters, you should indent using spaces and not tabs, since tabs may be interpreted as different numbers of characters depending on your code browser. You can set the Eclipse formatter to do this for you.

JUnit4 test – CSSE7023 ONLY (2 marks)

We will try to use your test suite `TrackReaderTest` to test an implementation of `read` that contains some errors in an environment in which the other classes `TrackReader.java` depends on exist and are correctly implemented.

Marks for the JUnit4 test suite in `TrackReaderTest.java` will be allocated as follows:

- Clear and systematic tests that can easily be used to detect most of the (valid) errors in a sample implementation and does not erroneously find (invalid) errors in that implementation.
2 marks
- Some problems, e.g., Can only be used easily to detect some of the (valid) errors in a sample implementation, or falsely detects some (invalid) errors in that implementation, or is somewhat hard to read and understand.
1 marks
- Work with little or no academic merit or major problems, e.g., cannot be used easily to detect (valid) errors in a sample implementation, or falsely detects many (invalid) errors in that implementation, or is too difficult to read or understand.
0 marks

Note: code submitted with compilation errors will result in zero marks in this section. A Java 8 compiler will be used to test code.