CSSE2002 / CSSE7023 **Semester 1, 2016**

Assignment 1

Goal: The goal of this assignment is to gain practical experience with data abstraction, unit testing and using the Java class libraries (the Java 8 SE API).

Due date: The assignment is due at **4:00pm on Friday 15 April.** 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, and the following two assignments in this course, you will develop the component classes of a program for simulating the behaviour of a train management system.[1]

A train management system controls the behaviour of trains on a railway track. A railway track is composed of sections of track that are connected at junctions in the track. Each junction in the track has between one and three connections to sections of track. Each of these connections is referred to as a branch of the junction, and each of those branches has a type: they are either a FACING, NORMAL or REVERSE branch. Each junction can only have one branch of each type. Locations on the track can be defined relative to their offset from a junction along one of its branches.

Task Overview: In this assignment, you will be writing data types for a *Section*, a *Location*, and a *Track*. If you are a CSSE7023 student you will also be required to write a JUnit4 test suite for the *Section* class in the *SectionTest* class. (All students will be expected to write their own tests to help them to debug their implementations, but only CSSE7023 students will be required to submit their tests for the *SectionTest* class.)

[1] The specification of this system has been adapted from the work of Jason Darlow in his Bachelor of Engineering Honours thesis *A Useful Approach in Developing a Track Data Model for Modern High Integrity Railway Control* from the University of Queensland, 2015.

Task Details:

Skeletons for the classes *`Section`*, *`Location`* and *`Track`* classes are provided in the zip file in the `railway` package. These skeletons include javadoc specifications of the constructors and methods that you need to complete. You must complete these class skeletons according to their specifications in the files. You will need also to declare variables, add import clauses, and add comments and javadocs as required.

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

Copy or rename the skeleton files before you start. (Don't forget that the package and class names inside the files must correspond to the file and the directory names … otherwise you'll have difficulty compiling and running it.)

You must implement these classes as if other programmers were, at the same time, implementing the code that instantiates them and calls their methods. 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.
- Don't add any new methods or variables to the classes unless they are private (that would be changing their specification by adding new features accessible from outside the class).
- 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.
- Your source files 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.
- Your code should use private methods and private instance variables and other means to hide implementation details and protect implementation invariants.
- Your methods, fields and local variables (except for-loop variables) should have appropriate Javadoc comments or normal comments.
- Comments should be used to document your code's invariants, and to describe any particularly tricky sections. (You must provide an implementation invariant for each of the *`Section`*, *`Location`* and *`Track`* classes). 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.
- The `checkInvariant()` method of each class (other than the test suite) should check that the implementation invariant you have specified in your comments is satisfied.
- Your code should not be overly complex and hard to understand.

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 *`Section`*, *`Location`* and *`Track`*, 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 `SectionTest.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, too!

If necessary, there may be some small changes to the files that are provided, up to one 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.

Hints:

1. It will be easier to implement the `Section` class first, followed by `Location` and then `Track`. The tests you may wish to write before you start coding these classes.
2. Read the specification comments carefully. They have details that affect how you need to implement and test your solution.

Submission: Submit your files `Section.java`, `Location.java`, `Track.java` (and `SectionTest.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 source (i.e. .java) 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 10, and if you are a CSSE7023 student, your assignment will be given a mark out of 12, according to the following marking criteria. (Overall the assignment is worth 10% for students from both courses.)

Testing (5 marks)

- All of our tests pass — 5 marks
- At least 3/4 of our tests pass — 4 marks
- At least 1/2 of our tests pass — 2 marks
- At least 1/4 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 (5 marks)

- Code that is clearly written and commented, and satisfies all specifications, style rules and rules of data abstraction — 5 marks
- Minor problems, e.g., lack of commenting or private methods — 3-4 marks
- Major problems, e.g., code that does not satisfy the specification or rules of data abstraction, or is too complex, or is too difficult to read or understand — 1-2 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);**
- not using private methods and private instance variables and other means to hide implementation details and protect implementation invariants;
- not having appropriate comments for methods, 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;
- not having meaningful implementation invariants commented and implemented for each of the classes; or
- 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 `SectionTest` to test an implementation of `Section` that contains some errors in an environment in which the other classes `Section.java` depends on exist and are correctly implemented.

Marks for the JUnit4 test suite in `SectionTest.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.