CSSE2002 / CSSE7023 **Semester 1, 2017**

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 **noon on Friday 5th May**. Late assignments will lose 20% of the maximum mark immediately, and a further 20% of the maximum 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 (level 4 of GPSouth) or by email to studentenquiries@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. Assignment extensions longer than 5 calendar days will not be possible due to the incremental nature of the assignments.

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. You should 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 allocating events to venues in a municipality, taking into consideration the traffic that they generate.

Task Overview:

In brief, you will write a method for reading in the venues in a municipality from a file, and you will write a method for finding a safe allocation of events to venues (one which does not cause the traffic on any corridor to exceed its maximum capacity). If you are a CSSE7023 student you will also be required to write a JUnit4 test suite for testing the allocation method.

More specifically, you need to code method `read` from the `VenueReader` class and method `allocations` from the `Allocator` class (a private helper method that is used to implement the public `Allocator.allocate` method) 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 `Allocator.allocate` method in the skeleton of the `AllocatorTest` class from the `planner.test` package. You may write your unit tests assuming that the classes that `Allocator` depends on (e.g. the `Event`, `Venue`, `Traffic` 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. You may also assume the existence of a correctly implemented `VenueReader` class – since that might make it easier to define lists of venues to test with.

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 `VenueReader` 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 `VenueReader`, 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 `planner` other than `VenueReader` 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 `planner.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 `AllocatorTest.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.

Given

- a list of events, called `events` such that `events` is not `null`, it does not contain any `null` events, and it does not contain duplicates
- a list of the venues available in a municipality, called `venues`, such that `venues` is not `null`, it does not contain any `null` venues, and it does not contain duplicate venues

we say that `allocation`, a mapping from events to venues (i.e. an object of type `Map<Event,Venue>`), is a *safe allocation* of `events` to `venues` if and only if

(i) `allocation` is not null

(ii) `allocation` contains a mapping from each event in `events` to a venue from `venues` that can host that event. It does not contain any other mappings (i.e. the set of keys in `allocation` is equal to the set of events from `events`). Note that it is not necessary for each venue in `venues` to have an event allocated to it: some venues may not be used in the allocation.

(iii) The same venue is not allocated to more than one event in `allocation` (i.e. you can't have more than one event allocated to the same venue.)

(iv) The traffic caused by the allocation (i.e. the sum of the traffic generated by hosting each of the events in `events` at their respective venues in `allocation`) is *safe*, where we say that traffic is *safe*, if the traffic on each corridor is less than or equal to the capacity of that corridor. (Note: there is now an `isSafe` method in the `Traffic` class that can be used to check if a traffic object is safe.)

When its precondition is satisfied, the method

```
Allocator.allocate(List<Event> events, List<Venue> venues)
```

is defined to return a safe allocation of `events` to `venues`, if there is at least one possible safe allocation, or `null` otherwise. It is implemented by using the private method

`Allocator.allocations(List<Event> events, List<Venue> venues)`

that is defined to return the set of all possible safe allocations of `events` to `venues`. This method is currently unimplemented. Your job is to write a recursive implementation of the private method `Allocator.allocations`.

Example of the `Allocator.allocations` method:

Given the following list of `events`:

```
e0 (10)
e1 (7)
e2 (5)
```

And the following list of `venues`:

```
v0 (10)
Corridor l0 to l1 (15): 10
Corridor l1 to l2 (15): 10

v1 (4)
Corridor l4 to l5 (20): 4

v2 (10)
Corridor l1 to l2 (15): 10

v3 (9)
Corridor l3 to l4 (20): 9
```

There are two possible safe allocations of `events` to `venues`:

```
eo is allocated to v0
e1 is allocated to v3
e2 is allocated to v2
```

and

```
eo is allocated to v2
e1 is allocated to v3
e2 is allocated to v0
```

The method `Allocations.allocations` should return the set containing both of those allocations. The method `Allocations.allocate` may return either one.

For each of those possible allocations it is trivial to see that conditions (i), (ii) and (iii) are satisfied. Let's have a closer look at (iv) for the first allocation. For the first allocation we have that the traffic caused by the allocation is the sum of the traffic caused by each of the allocations, i.e. it is the sum of the traffic caused by hosting `eo` at `v0`:

```
Corridor l0 to l1 (15): 10
Corridor l1 to l2 (15): 10
```

and the traffic caused by hosting `e1` and `v3`:

```
Corridor l3 to l4 (20): 7
```

and the traffic caused by hosting `e2` at `v2`:

```
Corridor l1 to l2 (15): 5
```

The sum of those traffic objects (i.e. the traffic caused by the allocation) is:

```
Corridor l0 to l1 (15): 10
Corridor l1 to l2 (15): 15
Corridor l3 to l4 (20): 7
```

and we can check that it is safe because the traffic on each of the corridors does not exceed the maximum capacity for that corridor.

Any allocations other that the two listed above are not safe. For example the allocation:

```
eo is allocated to v0
e1 is allocated to v2
e2 is allocated to v3
```

is not safe because the traffic caused by that allocation:

```
Corridor l0 to l1 (15): 10
Corridor l1 to l2 (15): 17
Corridor l3 to l4 (20): 5
```

is not safe because the traffic on `Corridor l1 to l2` exceeds the capacity of that corridor.

Hints for the recursive implementation of the `Allocator.allocations` method (in case you get stuck):

In the case that `events` is an empty list, there is exactly one safe allocation of `events` to `venues`: the empty map (i.e. the map with no entries).

In the case that `events` is not an empty list, the problem needs to be solved using recursion. The first element in `events`, let's call it `first`, may be able to be allocated to any of the venues in `venues` that can host it. For each venue from `venues`, let's call it `venue`, that may be able to host `first`, we can recursively calculate all of the possible safe allocations of the "rest of the events" (i.e. the `events` without `first`) to the "rest of the venues" (the `venues` without `venue`). We can then use those possible allocations (of the "rest") to calculate the possible safe allocations of `events` to `venues` in which `first` is allocated to `venue`.

Efficiency concerns for the allocate method:

The `Allocator.allocations` method, and hence the `Allocations.allocate` method will be very inefficient. Hence it is not prudent to test the allocate method with large inputs (i.e. any more than about 5 events and 10 venues). If you are a CSSE7023 student, you are asked to limit your tests to these small cases. None of your tests should be expected to run for more than 5 seconds – you should include a timeout, i.e.

```
@Test(timeout = 5000)
public void test() {
    // …
}
```

to make sure of that. Note that if the test times out, then it will fail.

Hints:

1. It may be easier to implement the `VenueReader.read` method first since you can use it to read in lists of venues to test the `allocate` method from the `Allocator` class.
2. Since the `Allocator.allocate` method may be nondeterministic (i.e. it may return any possible safe allocation, if there is more than one), when you are writing your tests for the method in the `AllocatorTest` class, it may be easier to write a helper method to check that the returned allocation is safe, rather than trying to enumerate all the possible allocations that might be returned by a test, and then checking that the returned result equals one of those values.

Submission: Submit your files **`VenueReader.java`**, **`Allocator.java`** (and **`AllocatorTest.java`** and any of your venue files that are used for testing in `AllocatorTest.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 `AllocatorTest` to test an implementation of `allocate` that contains some errors in an environment in which the other classes `Allocator.java` depends on exist and are correctly implemented.

Marks for the JUnit4 test suite in `AllocatorTest.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.