The University of Birmingham
School of Computer Science


# *Assignment 02 — Addendum*

## 1 Generating JavaDoc

You may find it helpful to generate the JavaDoc HTML files so that you can easily browse and navigate through the JavaDoc information for the assignment classes as well as the related library classes. Once they are generated you will be able to view the JavaDoc for your project classes, methods and variables in a web browser just like you can for the standard library classes (hover over an identifier and press `<Shift-F2>` on Linux).

To do this, in eclipse, when you have no compilation errors in the project:

1. `Project|Generate Javadoc...`
2. Select the `src/main/java` "types" to generate Javadoc for
3. Select "`Private`", and set the destination to be a directory called "`javadoc`" in the top level of your project directory (beside `bin` and `src`). It is important to use that name instead of the default "`doc`", because your `.gitignore` file causes GIT to never store the `javadoc` directory (as it should not), but it will store a `doc` directory.
4. Click on `Next>`
5. Select all the `Basic Options`, all the `Document these tags` and click the `Select All` button on the right.
6. Click on `Next>`
7. Click on `Finish`

It may ask if you want to update the Javadoc location for your project with the chosen destination folder. If so, click on `Yes`. In such a case, the generation may fail without giving any warning. If you run the Javadoc generation again, it will work the second time.

## 2 Support Classes

### 2.1 Immutable Objects

The concept of *immutable objects* is important in the design of Java data structures. An immutable object is an object of a class that does not allow the value of the object to be changed after it has been constructed. This is accomplished by ensuring:

- that any *instance fields* (sometimes called *"class fields"* or *"class variables"* are declared to be private,
- that there are no mutators (setters) for the instance fields,
- that the code of the class itself never modifies the class fields after construction, and
- that we do not *"leak"* references to modifiable class fields outside the class. For example, if we have a private array of ints as an instance field, and we have an accessor (getter) method that returns this array to an object of another class, then this other object can modify the contents of this array and our supposedly immutable object will find that the array contents that were supposed to be fixed have now been changed without its knowledge.

  To avoid this problem we must ensure that we never return a reference to non-immutable instance field from any method of our immutable class. We can safely return immutable values or references (e.g. String, Integer, etc.), but if we need to return an inherently mutable object, such as an array, or Collection class object, we must not return the direct reference to it but instead, either provide controlled getters for the information in the mutable structure, or we return a copy of the object so that if the copy is changed, the original version is not affected.

  For a collection class object which contains immutable objects, you can call the `toArray()` method to return an array of the contained objects. For an array you can call `Arrays.copyOf(..)` method to get a new copy of the array.

### 2.2 Portal

Our maze has chambers and each chamber has a number of doors. The passageways between chambers connect a particular door in one chamber to a particular door in another. This pairing of door number with the number of the chamber that the door is in is such a basic element in this program that the `Portal` class has been written to capture it.

It is an immutable object, thus it is safe to pass around references to these objects even if they are in other data structures.

Portal has proper `equals()` and `hashCode()` methods. These are necessary to be able to put objects of this type into various collection classes, such as `Set<>`, `Map<>`, etc.

## 2.3 Maze

The `Maze` class is designed to represent the properties of an underground maze as seen by a drone. The drone can only ever see the chamber it is in (i.e. identify the chamber and the doors in that chamber), and it can go through a doorway to find the chamber connected to it. The drone cannot see or query the maze about anything beyond that. Any other information it needs it has to build up by exploring the maze. As the drone wanders through the maze, the maze maintains the information about what chamber it is in so that it can respond appropriately to the drone's queries.

`Maze`'s `traverse(...)` method handles moving to a different chamber by going through a particular door (the parameter) in the current chamber. The `Portal` object through which the drone enters the new chamber is returned.

The drone can always ask what chamber it is currently in and how many doors the current chamber has. The drone cannot just jump, or teleport, to any chamber in the maze, it has to find its ways there.

There are a number of constructors for `Maze`. With one exception they generate a random maze with some number of chambers, some average number of doors per chamber and, optionally, a seed number for the random number generator used so that you can ensure that the same maze is reproduced each time if needed for debugging purposes. The maze will never have connections that loop back from one chamber to the same chamber. If you set the average number of doors too high it will stop adding connections, and hence doors, when every chamber is directly connected to every other chamber.

The exceptional constructor is a copy constructor. This makes a new maze which is a copy of the parameter maze. Since the new maze shares the data structures of the old maze (except for the `currentChamber` instance field), this is a very quick and cheap operation. The purpose of this copy constructor is to make testing easier. You can see it in action in the `DroneTest` class.

There is a simple `toString()` method that returns a printable `String` version of the maze connections, and a `toDotFormat()` method, that generates a version of the same information that can be used to generate a graphical map of the maze. See the JavaDoc of the method for more information and see the `DroneTest` class for using them in log messages.

The only complicated part of this class is in generating a random maze. It is not necessary to understand this to do the assignment, but you might find it worthwhile to study for its examples of using `Set<>` and `Map<>`. Basically it works in two halves. The first half generates a random tree to connect all the chambers together, by starting with chamber 0 in the tree and, until all the chambers are added to the tree, choosing a random chamber that has not yet been added and connecting it to a random chamber in the tree. In each case the next available door number is used for the connection at each end. The second half just adds random chamber connections from a set of possible connections (excluding connections between the same chambers or connections that already have been made.

## 2.4 DroneInterface and DroneTest

These two classes require little discussion. *DroneInterface* is there to ensure that the `Drone` class has the correct method declarations.

`DroneTest` is a standard JUnit test class. You can change the baseMaze constructor to try different mazes with different sizes.

## 2.5 Drone

This is where you have to write your code. You only need to fill in your student id number and your name in the `getStudentID()` and `getStudentName()` and otherwise write the code for 3 methods:

1. `searchStep()`: This executes a single step in searching through the maze and corresponds to the drone choosing a door to go through, traversing through the door and ending up in the connected chamber. The Portal by which the drone enters the connected chamber should be returned. If the drone has completed the search and is back in chamber 0, then any further calls to this method should return null. See section 3 for a discussion on the search algorithm.

2. `getVisitOrder()`: This simply returns an array of Portal objects recording the path that the drone has taken through the maze. Note that both the Portal taken to exit a chamber as well as the Portal taken to enter the next one should be recorded. Thus the first Portal in the array should be the Portal through which the drone left chamber 0 at the start of its search, while the last element in the array should be Portal by which the drone has entered the chamber that it is currently in.

3. `findPathBack()`: This returns an array of Portal objects that show the route that the drone should take to get back to chamber 0 with no diversions on the way. Here the list of Portals should contain only those ones to exit the sequence of chambers that the drone will pass through on its way back to chamber 0. Thus the first element will be the Portal to exit the current chamber, the second element be the one to exit the chamber that you are in after exiting the first chamber, etc. The final Portal should be the one to exit the chamber that returns the drone to chamber 0. If your current chamber is already chamber 0, then the `findPathBack()` method should return an empty list (NOT null!). Note that there may be many different paths back. You are NOT required to find the shortest one and the tests used for

marking will only check that your path does actually work to get the drone back to chamber 0 and that it does not have any loops or dead-end backtracking.

The last page of the main assignment_02 handout has the log output of an execution of a solution to this problem that shows the current chamber, the visit order list and the path back after every search step. If you work through that you will see precisely what is required.

# 3 Search Algorithm

The simplest suitable search algorithm to implement, and the one you are advised to use, is called *Depth First Search*. Here the idea is, at every step, to select, if possible, a doorway out of the current chamber that you have not used before and go through that doorway. If there is no such unused door, you have to backtrack out of the current chamber. To figure out how to do that, when you find an unused door to go through to enter the next chamber, you push the Portal that you enter that next chamber through onto the stack. When you have exhausted all the doors in that chamber, you pop off the Portal from the stack and follow that door to backtrack. You are finished the search when you try to pop a backtrack Portal off the stack but the stack is empty. At this point you should be back in chamber 0.

You should use:

a. the `visitStack` instance field for the backtrack stack,
b. the `visited` instance field to keep track of which Portals you have passed through (either entering or exiting a chamber)
c. the `visitQueue` instance field to keep track of the sequence of Portals you have passed through.

# 4 Tips and Strategy

The tests are designed to support incremental development of your solution. Target a particular test to get working at each stage and only go on to target a different test when the current test is fully working. I suggest you tackle the tests in the following order and try to resist the temptation to add code to handle a later test before you have completed the current test you are working on, because this ends up making your code messier and harder to change as you are developing:

1. `testDroneConstruction()`: should already work!
2. `testCheckStudentIdentification()`: a particularly easy one!
3. `testDroneOneStep()`: Just checks that if you take one step you move through one connection and you correctly report the Portal you connect to.
4. `testDroneSearch()`: This is the full search. It is a little involved but should not be too difficult
5. `testDroneFinalVisitOrder()`: This should be very easy once you have the previous tests passing
6. `testDroneAllVisitOrders()`: Again very easy
7. `testDroneSimpleBackPath()`: Easy again, as you do not, for this test, have to have removed loops or dead-ends that you need to backtrack out of to get home
8. `testDroneBackPathNoLoops()`: More complicated for extra marks. You do not need to find the shortest path home, but you do need to find a path home that does not enter any chamber a second time.

Note that if you have `getVisitOrder()` working, then it is pretty easy to get a basic version of `findPathBack()` working, where a basic version does not remove the loops, or dead-end backtracking. This basic version would pass `testDroneSimpleBackPath()` but not `testDroneBackPathNoLoops()`.