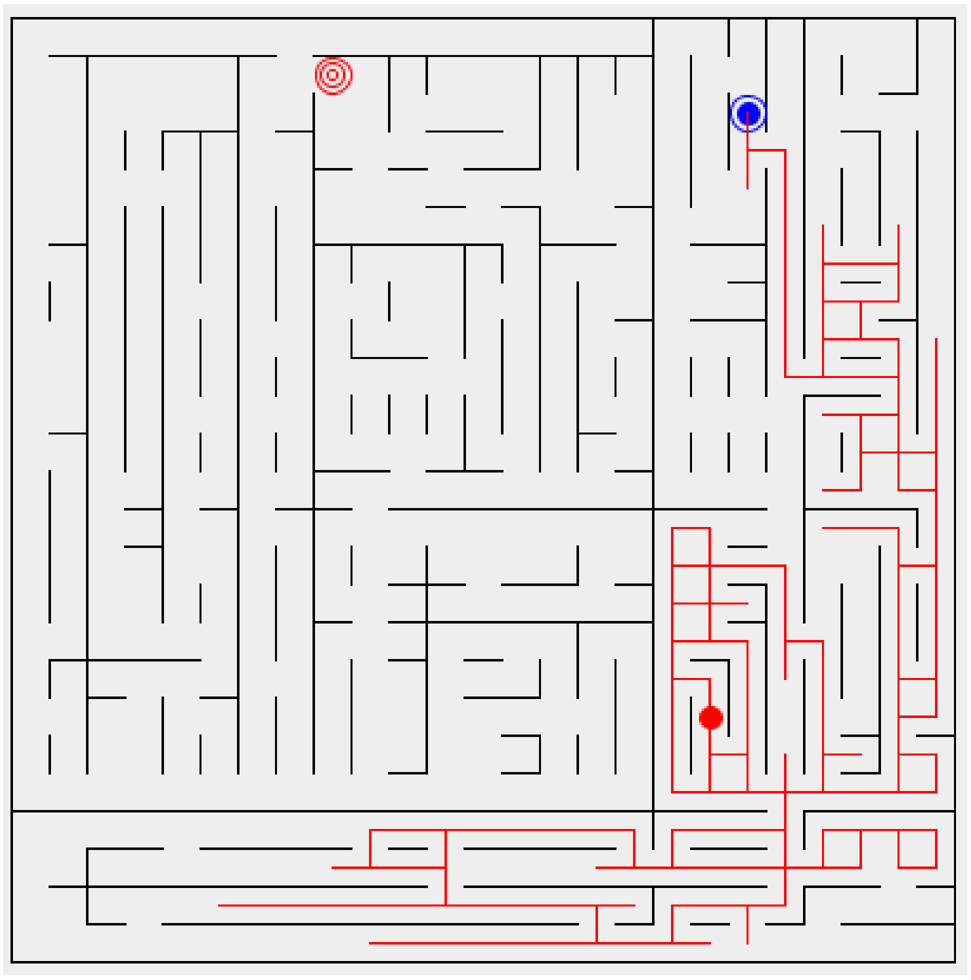
Victoria University of Wellington
School of Engineering and Computer Science

SWEN221: Software Development

Assignment 1

1 Maze Walker

This assignment asks you to write a simple algorithm that is capable of finding the exit in a certain type of maze. Both start and end points are provided as input and the aim is to calculate a path from the former to the latter. The following screen-shot illustrates the objective:



g

The blue dot indicates the start point, the red target icon the end point, and the red lines the current path covered by the search algorithm.

1.1 Maze Software and Documentation

The respective software splits into two components: a *predefined infrastructure part* and the *search componentn*. The predefined part comprises numerous classes responsible for reading, writing and drawing mazes. **Its source code is not provided, as you will not need to modify it**. Documentation (JavaDoc) is provided, however, and linked from the lecture schedule.

The search component consists of the following classes:

```
swen221/assignment1/LeftWalker.java
swen221/assignment1/LeftWalkerTests.java
```

A walker implementation, i.e., a class that contains code that will find a path through a maze, is a subclass of `maze.Walker`. Your job is to complete the `LeftWalker` class to implement the "*left hand*" rule for navigating a maze. A simple implementation for `LeftWalker` is already provided; it simply produces randome moves.

Eclipse. To begin working in Eclipse, create a new project and import the `client.jar` file. You should then add the `maze.jar` file as an "External Archive". Do this by right-clicking on the project and selecting **"Build Path→Add External Archives"**. You will also need to add JUnit as a library. Do this by right-clicking on the project and selecting **"Build Path→Add Libraries→ JUnit→JUnit 4"** from the menu.

1.2 Maze File Format

The maze layout is specified using a comma-separated text-file. Each square of the maze is represented by a single integer number. The first six bits of the number are used to determine which walls of the square are present, and to identify the start and target locations:

Value	Meaning
1	North Wall
2	South Wall
4	East Wall
8	West Wall
16	Start Position
32	Target Position

To compute the value for a square, you simply add up the appropriate numbers from this table. For example, a square having just the north and east walls is: $1 + 4 = 5$. An example maze file is given below:

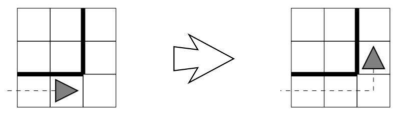
```
% cat maze1.dat
  5,5
  9,3,1,1,5
  8,1,4,44,12
  10,2,2,4,12
  9,19,1,6,12
  10,3,2,3,6
```

The two numbers of the first row give the dimensions of the maze. Each subsequent row of the file corresponds to a row in the maze and, likewise, each column in the file to a column in the maze.

Can you spot an invariant that holds with respect to the maze representation?

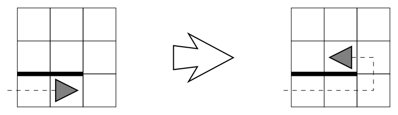
Part 1 - Simple Left Walker (50%)

The aim in this part is to implement a search strategy in the form of a walker implementation that follows the "*left hand rule*" for navigating a maze. Class `LeftWalker` class should make the walker follow the wall on its left and, upon arriving at a turning to the left, take it. The following scenario illustrates the intended behaviour.



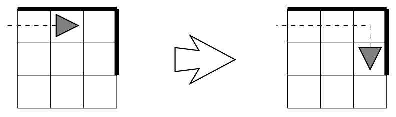
Here, the walker enters the southern-most west square from the west following the wall on its left (i.e. to the north). When it enters the southern-most east square, it finds there is a gap to the north. At this point it turns to the left and moves north following the wall on its left.

As another example of the intended behaviour, consider the following scenario:



Here, the walker enters the southern-most west square from the west following the wall on its left (as before) and turns left to the north when it reaches the gap to its left. At this point, it is in the middle row of the rightmost column facing north. Again, it sees that there is a gap to its left and turns left again to continue west following the wall on its left.

As a final example, consider the following:



Here, the walker enters the northern-most west square from the west following the wall on its left. Eventually, it encounters the wall to its east, and cannot proceed in that direction. Then, since it cannot continue east or turn to the north, it turns clockwise and continues south following the wall on its left.

The `LeftWalker` initially is facing the challenge of identifying a left wall to follow, in other words, it must initially search for a left wall to follow. The simple respective strategy to use is the following: If there are no adjoining walls at all, the walker moves north (one step) and keeps searching for a left wall. Otherwise, the walker turns clockwise until it has a wall to its left and then starts following it.

HINT: The JUnit tests provided in `LeftWalkerTests` provide a range of simple tests for the left walker. Initially, these fail as there is no LeftWalker class.

HINT: You need to determine the direction in which the walker initially faces. This can be done by analysing the JUnit tests provided. In other words, you should observe that the tests provide a specification of how the `LeftWalker` should operate.

Part 2 — Refined Left Walker (40%)

When you have a simple implementation of the LeftWalker, you will notice that some tests still fail due to the walker getting into an infinite loop following the same left walls without making any progress. To improve the chances that the walker will find a solution, refine its behaviour by using a "memorisation" trick as follows:

1. The `LeftWalker` remembers the squares it has visited together with the direction it exited those squares.
2. When the `LeftWalker` visits a square that it has already visited, it first chooses an exit based on the previous rules. However, if the walker has already taken that exit before, then it chooses another exit which it has not yet taken. This exit is chosen in a clockwise fashion from the initial exit chosen.
3. After that, the `LeftWalker` returns to the usual strategy of following the wall on its left.

Finally, note that if the walker reaches a previously visited square for which it has tried all exits, then it is *stuck* and does not move any further. This means that there are mazes with solutions that the `LeftWalker` still cannot solve.

Submission

Your solution should be submitted electronically via the *online submission system*, linked from the course homepage. The required files are:

`swen221/assignment1/LeftWalker.java`

You must ensure your submission meets the following requirements (which are needed for the automatic marking script):

1. **Your submission is packaged into a jar file, including the source code**. *Note, the jar file does not need to be executable.* See the following Eclipse tutorials for more on this:

   `http://ecs.victoria.ac.nz/Support/TechNoteEclipseTutorials`

2. **The names of all classes, methods and packages remain unchanged**. That is, you may add new classes and/or new methods and you may modify the body of existing methods. However, you may not change the name of any existing class, method or package. *This is to ensure an automatic marking script can test your code.*
3. **All testing mechanisms supplied with the assignment remain unchanged.** Specifically, you must not alter the way in which your code is tested as the marking script relies on it. However, this does not prohibit you from adding new tests. *This is to ensure an automatic marking script can test your code.*
4. **Any debugging code that produces output, or otherwise affects the computation has been removed.** *This ensures the output seen by an automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in your submission being rejected by the submission system and/or zero marks being awarded.

Assessment

This assignment will be marked as a letter grade (A+ ... E), based primarily on the following criteria:

- **Correctness of Part 1 (50%)** — does the submission adhere to specification given for Part 1?
- **Correctness of Part 2 (40%)** — does the submission adhere to specification given for Part 2?
- **Style (10%)** — does the submitted code follow the style guide and have appropriate comments (inc. Javadoc)?

As indicated above, part of the assessment for the coding assignments in SWEN 221 involves a qualitative mark for coding style. Coding style is an important aspect of coding since all code should be expected to be revisited again, i.e., it is important that you are someone else can make sense of the code it in the future and change it without accidentally introducing errors.

The qualitative marks for style are given for the following points:

- **Division of Functionality into Classes**. This refers to how *cohesive* your classes are. That is, whether a given class is responsible for single specific task (has high cohesion), or for many unrelated tasks (has low cohesion). In particular, big classes with lots of weakly related functionality should be avoided.
- **Division of Work into Methods**. This refers to how well a given task is split across methods. That is, whether a given task is broken down into many small reusable methods (good) or implemented as one large monolithic method (bad).
- **Self-Explanatory Naming**. This refers to the choice of names for the classes, fields, methods and variables in your program. First, naming should be consistent and follow the recommended Java Coding Standards (see `http://g.oswego.edu/dl/html/javaCodingStd.html`). Secondly, names of items should be descriptive and reflect their purpose in the code.
- **JavaDoc Comments**. This refers to the use of JavaDoc comments on classes, fields and methods. We expect all `public` and `protected` items to be properly documented. For example, when documenting a method, an appropriate description should be given, including despriptions for (if present) its parameters and return value. Good style dictates that `private` features are documented as well.
- **Code Comments**. This refers to the use of commenting within a given method. Comments should be used to elaborate the purpose of the code, rather than simply repeating what is evident from the code already.
- **Layout**. This refers to the consistent use of indentation and other layout conventions. Code must be properly indented and make consistent use of conventions e.g. for placing curly braces.

In addition to a mark, you should expect some written feedback, highlighting the good and bad points of your solution.