Victoria University of Wellington
School of Engineering and Computer Science

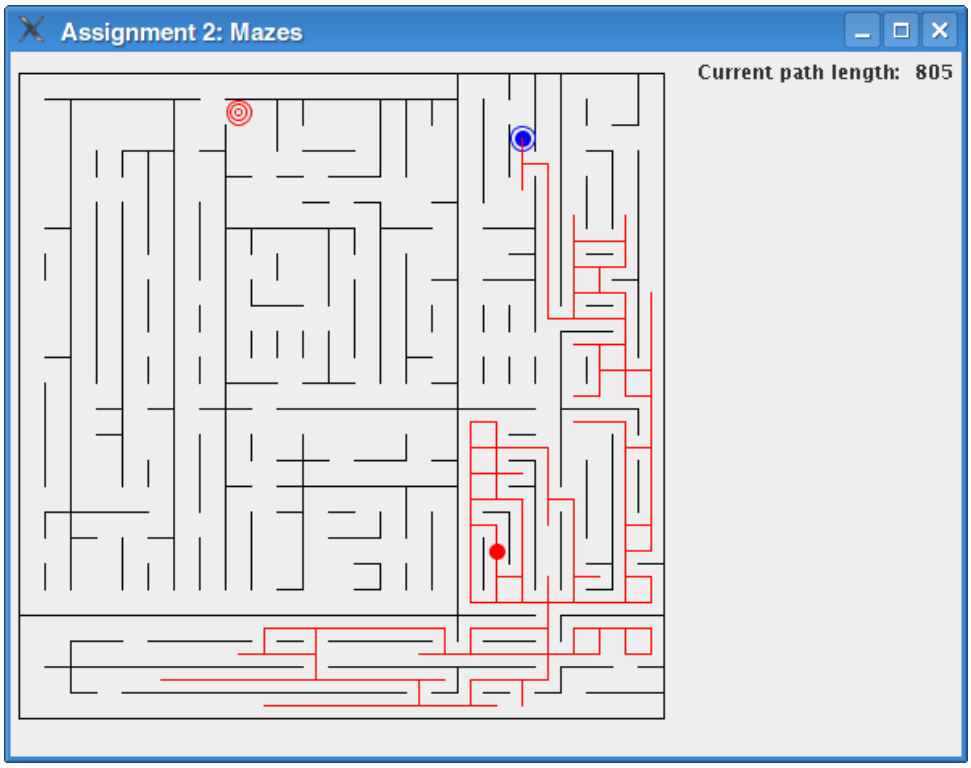
SWEN221: Software Development

Assignment 2

Due: Monday 23rd March @ Mid-night

1 The Amazing Maze

This assignment concerns a simple program known as the "Amazing Maze". As the name suggests, the program is a simple maze game, where the aim is to find a path through a maze from a given starting point to a given ending point. The following screen-shot illustrates the game being played:



g

The game should provide two modes of operations: *user-controlled* and *computer-controlled*. In the user-controlled mode, the user moves around the maze using the cursor keys, whilst the program records the number of steps taken. In the computer-controlled mode, a computer program navigates the maze, attempting to find a given end point.

1.1 Maze Software and Documentation

The maze software splits into two components: the *library code*, and the *client code*. The library code consists of numerous classes responsible for reading, writing and drawing mazes. **The source code for the maze library is not provided, as you will not need to modify it**. Documentation (JavaDoc) for the maze library is provided and linked from the lecture schedule.

The client code consists of the following classes:

```
assignment2/RandomWalker.java
assignment2/Main.java
assignment2/LeftWalker.java
assignment2/KeyWalker.java
assignment2/LeftWalkerTests.java
```

The class `Main.java` passes command-line arguments, loads the maze and creates the appropriate "walker". A walker is a subclass of `maze.Walker`, and is responsible for navigating a path through the maze according to some pre-determined strategy. The `RandomWalker` provides an illustration of a walker which, in this case, simply moves around the maze at random. The classes `KeyWalker` and `LeftWalker` are to be completed as part of this assignment.

1.2 Maze File Format

The maze library supports the loading and saving of mazes. The `Main` class in the client code allows you to load a predefined maze like so:

```
% java assignment2.Main -file maze1.dat
```

The format for the maze is a fairly simple 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.

NOTE: If you omit an input file when running `assignment2.Main`, the client will automatically construct a randomly generated maze.

Part 1 - Key Walker

The aim in this part is to implement the `KeyWalker` class. This is a simple extension of `maze.Walker` that implements the `java.awt.event.KeyListener` interface. When the key walker is fully implemented, the user should be able to control the walker in the maze using the arrow keys.

To get started, we suggest you follow these steps:

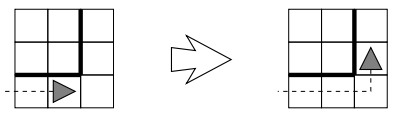
1. Create a Java project in eclipse.
2. Add the JUnit library.
3. Add the file `maze.jar` as an *external archive*. Do this by right-clicking on the project, selecting "BuildPath→Add External Archives...".
4. Import the files from `client.jar` into the project.
5. At this stage, you should be able to run the class `assignment2.Main`. By default, this will create a random maze and use the random walker to try and find a solution (although the random walker isn't very good, and probably won't actually find a solution!).
6. Now, look at the RandomWalker implementation as well as the `javadoc` that accompanies the maze library (linked from the SWEN221 homepage); in particular the class `maze.Walker`. You'll also need to find out how `java.awt.event.KeyListener` works.
7. Finally, implement the class `assignment2.KeyWalker` which extends `Walker` and implements `KeyListener`. You'll also need to remove the comments which create the `KeyWalker` in the `main()` method for class `assignment2.Main`.

Part 2 - Left Walker

The aim in this part is to implement a computer-controlled walker that follows some simple heuristics to find the exit.

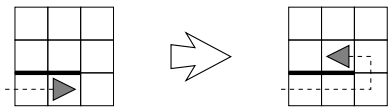
a) A simple LeftWalker

The first implementation of the `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 illustrates:



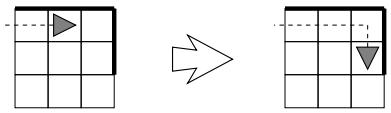
Here, the walker enters the southern-most west square from the west following the wall on its left (i.e. to the north). When it enters the southern-most east square, it finds there is now a gap to the north. At this point it turns to the left and moves north following the wall on its left.

As another example, consider the following:



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 (i.e. to the north). 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 in a clockwise-direction and continues south following the wall on its left.

One issue the `LeftWalker` is faced with initially is to identify a left wall to follow. So when placed into a maze, the walker searches for a left wall and once it has identified one, follows it. Searching for the left wall is performed as follows: If there are no adjoining walls (to the north, east, south, or west), 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.

b) A refined LeftWalker

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. At this point, 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. Unfortunately, this means that there are mazes with solutions that the `LeftWalker` still cannot solve.

Submission

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

```
assignment2/RandomWalker.java
assignment2/Main.java
assignment2/LeftWalker.java
assignment2/KeyWalker.java
assignment2/LeftWalkerTests.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**. See the export-to-jar tutorial linked from the course homepage for more on how to do this. *Note, the jar file does not need to be executable.*

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 the automatic marking script can test your code.*

3. **All JUnit test files supplied for the assignment remain unchanged.** Specifically, you cannot alter the way in which your code is tested as the marking script relies on this. This does not prohibit you from adding new tests, as you can still create additional JUnit test files. *This is to ensure the automatic marking script can test your code.*

4. **You have removed any debugging code that produces output, or otherwise affects the computation.** *This ensures the output seen by the automatic marking script does not include spurious information.*

Note: Failure to meet these requirements could result in you getting zero marks for the assignment.

Assessment

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

- **Correctness of Part 1 (30%)** — does submission adhere to specification given for Part 1.
- **Correctness of Part 2 (60%)** — does 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 SWEN221 involves a qualitative mark for style, given by a tutor. Whilst this is worth only a small percentage of your final grade, it is worth considering that good programmers have good style.

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

- **Division of Concepts into Classes**. This refers to how *coherent* your classes are. That is, whether a given class is responsible for single specific task (coherent), or for many unrelated tasks (incoherent). In particular, big classes with lots of 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 methods (good) or implemented as one large method (bad). The approach of dividing a task into multiple small methods is commonly referred to as *divide-and-conquer*.
- **Use of Naming**. This refers to the choice of names for the classes, fields, methods and variables in your program. Firstly, 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 program.
- **JavaDoc Comments**. This refers to the use of JavaDoc comments on classes, fields and methods. We certainly expect all `public` and `protected` items to be properly documented. For example, when documenting a method, an appropriate description should be given, as well as for its parameters and return value. Good style also dictates that `private` items are documented as well.
- **Other Comments**. This refers to the use of commenting within a given method. Generally speaking, comments should be used to explain what is happening, rather than simply repeating what is evident from the source code.

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