

- All classes shown in the UML should be declared as public classes.
- The `MazeApplication` is a JavaFX application that allows users to specify a text

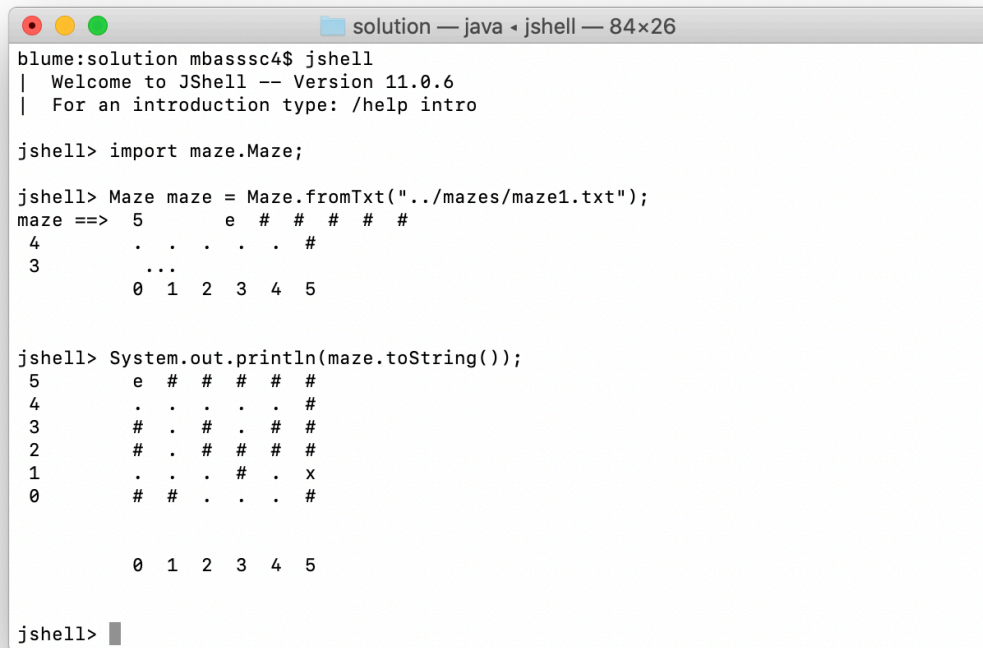
file containing a maze representation. The graphical representation of the maze is displayed to the user. The user will also be able to: (i) step through the solving process with the visualisation updating to the current state, (ii) save the current route-solving state to a file, and (iii) load route-solving state from a file. For example, a simple implementation of this might look as follows:



- The `MazeDriver` class is an optional Java application used during the development process. It may be used to print `Maze` and `RouteFinder` state to the console/terminal. **This class will only be marked if there is no `MazeApplication` class, or if the `MazeApplication` class provided does not compile and run successfully.**
- A `Maze` is an object that represents the maze to be solved. Each `Maze` contains a two-dimensional `ArrayList` of `Tile` objects. Each `Maze` contains exactly one entrance, one exit, and all rows of tiles must be the same length. If any of these constraints are not met, an appropriate subclass of `InvalidMazeException` should be thrown. It should not be possible to use the setter methods to set the entrance or exit to a tile that is not present in the maze. Calls to methods that may result in an `InvalidMazeException` must require the caller to explicitly handle the

exception.

- The `fromTxt` method in the `Maze` class allows a `Maze` object to be created by reading in a `txt` file (the full file path is passed as a `String` parameter), and the `toString` method should return a string that visualises the entire maze. The format of this string **is not specified** but should accurately present the complete maze, for example:



```
blume:solution mbasssc4$ jshell
| Welcome to JShell -- Version 11.0.6
| For an introduction type: /help intro

jshell> import maze.Maze;

jshell> Maze maze = Maze.fromTxt("../mazes/maze1.txt");
maze ==> 5      e # # # # #
4      . . . . . #
3      . . . . . #
0 1 2 3 4 5

jshell> System.out.println(maze.toString());
5      e # # # # #
4      . . . . . #
3      # . # . # #
2      # . # # # #
1      . . . # . x
0      # # . . . #

0 1 2 3 4 5

jshell>
```

- A `Tile` is an object that represents one space within a maze. A tile may be of `Type` `CORRIDOR` (represented in text files as a `.`), `ENTRANCE` (`e`), `EXIT` (`x`), or `WALL` (`#`). The `fromChar` method creates a new `Tile` from its text representation (supplied as a `char`). Conversely, the `toString` method returns the the string representation used in text files (i.e. `"e"` for tiles of `Type` `ENTRANCE`). A `Tile` of `Type` `WALL` cannot be navigated through; all other `Tile` types can be.
- Each position within the `Maze` can be represented as a `Coordinate` whose `toString` method returns `"(x, y)"` where `x` represents the column number (indexed from 0, where 0 is the left of the maze) and `y` represents the row number (indexed from 0, where 0 is the bottom of the maze). The `Maze` class provides methods to get the `Coordinate` of a specified `Tile` (`getTileLocation`), and to get the `Tile` at a specified `Coordinate` (`getTileAtLocation`).
- From each `Tile` it may be possible to navigate in one of four directions (diagonal movement is not possible). These directions are represented by an enum

`Direction` with the values `NORTH` (up), `SOUTH` (down), `EAST` (right), and `WEST` (left). The `Maze` class provides a `getAdjacentTile` method that returns the tile next to a specified `Tile` in a given `Direction`.

- The `RouteFinder` class provides the core logic for solving a given `Maze`. The `RouteFinder` uses a `java.util.Stack` to maintain state as it steps through the maze from the entrance to the exit (possibly/probably via a few dead ends). Only once the stack contains the complete path from entrance to exit (inclusive), will the `isFinished` method return `true`. Once a maze has been solved, the `Stack` representing the route should not contain multiple handles to any one `Tile` (each `Tile` in the maze will appear at most once).
- The `step` method within `RouteFinder` is responsible for updating the `Stack` that holds route-finding state. A call to the `step` method should make exactly one move through the maze -- i.e. either adding or removing one element to/from the `Stack`. It should not be recursive. Once a maze has been solved, further calls to `step` should have no effect on any of the state of the `RouteFinder`. The `step` method returns a `boolean` indicating if the maze is complete. If the `step` method gets stuck in a circular route or otherwise finds that no route to the exit is possible then it should throw a `NoRouteFoundException`. Calls to methods that may result in an `NoRouteFoundException` must require the caller to explicitly handle the exception.
- The `getRoute` method within `RouteFinder` should return a `List` of `Tiles` representing the current (complete or incomplete) route, from start to end (i.e. the first value in the list should be the entrance, and the last should be the element on the top of the route stack).
- The `load` and `save` methods in `RouteFinder` allow `RouteFinder` objects to be serialised out to a file. The `toString` method should return a string that visualises the entire maze and route-solving state. For example, the following screenshot shows the result of calling printing the return value from `toString` for a partially solved maze.



The screenshot shows a terminal window titled "solution — -bash — 85x11". It displays a maze visualization with rows indexed 0 to 5 and columns indexed 0 to 5. The maze is represented by a grid of characters: '#' for walls, '.' for open paths, '-' for a specific path segment, and 'x' for the exit. The entrance is at row 0, column 0. The exit 'x' is at row 1, column 5. The maze structure is as follows:

	0	1	2	3	4	5
5	*	#	#	#	#	#
4	*	*	-	-	-	#
3	#	*	#	-	#	#
2	#	.	#	#	#	#
1	.	.	.	#	.	x
0	#	#	.	.	.	#

In addition to the above functionality, this coursework will require you to:

- Organise your code into packages (see Chapter 19 of book). Package `maze` contains the core code to provide maze representations. Package `maze.routing` contains the maze solving implementation. Package `maze.visualisation` contains JavaFX components to support visualisation of mazes and the stages of route-solving.
- Use Javadoc to document your code (see Chapter 11 of book). Public and protected classes, fields, constructors, and methods should be documented fully.

## 2. How to plan the work.

Do not leave it until the last moment! ☺

The coursework is designed to assess material from Weeks 5-9 (with some bonus material in Week 10 that can enhance the UI further). However, as with Coursework I, there is plenty that you can do on Coursework II before the end of teaching in Week 9.

- **Week 6:** you should be able to implement all classes in the core `maze` package, including reading maze data from a text file.
- **Week 7:** you should be able to implement a basic UI using JavaFX (including classes in `maze.visualisation`).
- **Week 8-9:** you should be able to implement classes in the `maze.routing` package.
- **Week 10:** you should have additional knowledge to refine your JavaFX UI and the `maze.visualisation` package.

Note that unlike Coursework I, we have not provided any of the code for Coursework II. The only Java files in your initial repository are `MazeDriver.java` and `MazeApplication.java`, both are essentially empty files, but are provided to ensure that you're putting code in the correct place within the directory structure.

## 3. Marking

As in Coursework I, your final grade will be a combination of results from automated tests and a set of TA marks.

### Automated testing (40%)

Automated test results will account for 40% of the marks (40 marks). Some of these tests have already been provided to you in the `src\tests` directory, others will be new but will match the provided specification.

The mapping from automated test results to a final grade will be described in full in a separate document.

To ensure that the tests are successful, **you must follow the provided UML diagram and specification.**

## **Everything else (60%)**

This accounts for 60% of the marks (60 marks) distributed as follows:

- **Javadoc:** Is the code fully documented using Javadoc? **[5 marks]**
- **User Interface:** Are users able to fully engage with the application using only the user interface? (i.e. no console IO) Does the user interface demonstrate a variety of JavaFX components? Are component choices appropriate? Does the application work well for mazes of different sizes / configurations? Is the user interface attractive and easy to use? **[10 marks]**
- **Understanding of underlying concepts:** Is the student able to describe aspects of their implementation? Are they able to communicate design choices and rationale? Does the student have a good understanding of alternate design approaches (where appropriate)? Does the student have a good understanding of the topics demonstrated in this coursework task (e.g. advanced OO, file IO, error handling, Java FX, packages, Javadoc)? **[45 marks]**

## **Plagiarism and collusion**

A reminder that your work will be checked for similarity with other sources (including other submissions from this year and prior years). This is an individual coursework exercise and you must not work collaboratively on code with others. You may not include code from any other source. If you are found to have submitted code that you did not write yourself or to have knowingly shared code with others then penalties will be imposed and disciplinary action may be taken. Consult department and university guides for more information about academic malpractice and its penalties.

## **Marking Disputes / Queries**

We appreciate that some students may have questions about their marks. Any questions about your marks or feedback **must be made within one week of the grades and feedback being released.** Queries received after this time will not be accommodated.