

—Assignment #5—

Make sure to review the submission requirements on pg. 4 of the Lab #1 document.

Follow the Style Guide as given in the Resources section of our `conneX` course page.

Learning Outcomes

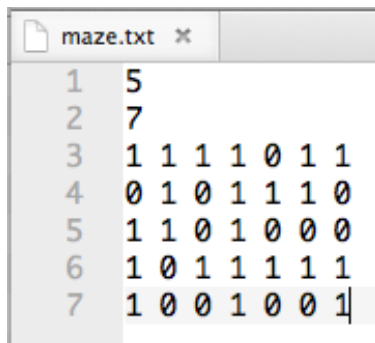
When you have completed this assignment, you should understand:

- How to use procedural decomposition to break down a problem into methods.
 - How to use the `File` class to read from a file.
 - How to use `Scanner` to process file input.
 - How to use 2D arrays to store data.
 - How to print a grid map using 2D arrays.
 - How to develop your own strategy to test code for correctness.
-

You will write a program that lets a user play a game navigating a maze. Download the template file `Maze.java` from Assignment #5 on `conneX`. Write the missing code for the methods in this template, and as usual, compile and test your code often as you write it up.

Make a file called `maze.txt` that will hold the data for your own maze. The first number in the file should be the number of rows, and the second number should be the number of columns. The number of rows should be between 5 and 10 inclusive, and so should the number of columns. Let `r` represent the number of rows, and `c` represent the number of columns. The remaining `r*c` numbers should all either be 0 or 1, separated by spaces. The 0 or 1 values describe the maze, and the user will be able to exist in corridor positions labelled with a 1. A value of 0 indicates a wall. You should space the 0 or 1 values in a rectangular format matching the number of rows and columns, so that it is easy to see the paths a user can take.

For example



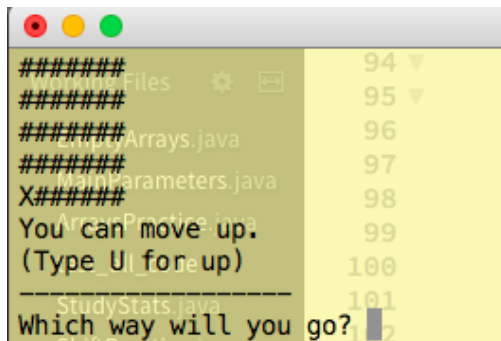
```
maze.txt x
1 5
2 7
3 1 1 1 1 0 1 1
4 0 1 0 1 1 1 0
5 1 1 0 1 0 0 0
6 1 0 1 1 1 1 1
7 1 0 0 1 0 0 1
```

You should design the maze to start in the bottom left corner and finish in the top-right corner. The user will only be able to move left, right, down, or up according to your maze, so make your paths possible for the user to finish.

The starting output for your game should prompt the user:

```
-----  
Welcome to the maze!  
Your current position is marked with an X.  
To leave, find your way to the top-right corner.  
Press Enter to start!
```

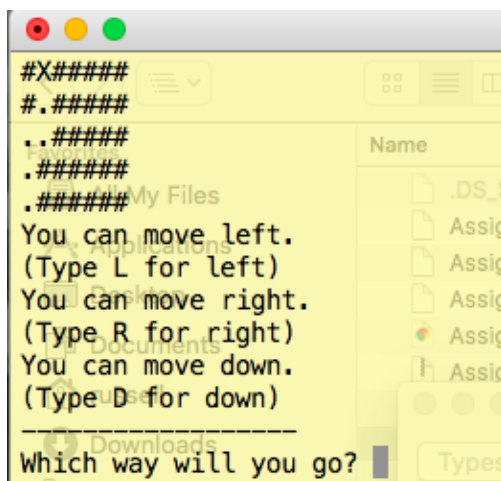
There is a method you can use called `clearScreen` to clear the screen of your console to give a nice presentation to the user playing your game. Make sure to clear the old screen output each time the user must decide the next move they will make. For example, a screen shot of your program on the first user's move should follow the same format below.



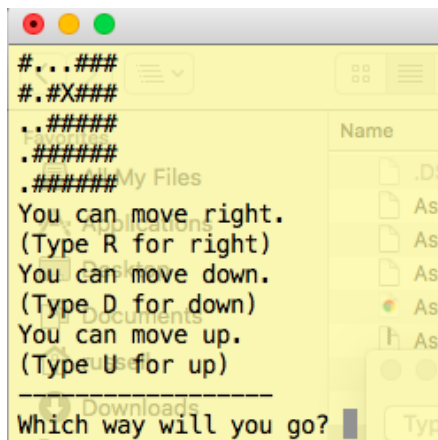
A screenshot of a Java IDE window. The console output shows a maze game interface. It starts with a 6x6 grid of '#' characters. The second row, first column contains an 'X'. Below the grid, it says 'You can move up. (Type U for up)'. After a separator line, it asks 'Which way will you go?'. The background shows a file explorer with various Java files like 'MainParameters.java' and 'StudyStats.java'.

There should be a 2D array to hold the map data from your `maze.txt` file, and another 2D array to keep track of the route where the user has been. Make sure to always print the route information, using `#` for unknown map information, and a period for each position the user has been. The user's current location should have an uppercase `X` printed at that position.

Then the choices for which direction the user can move should be printed below the route information. And finally, the user should be prompted for their choice of which direction to move. Some screen shots are available below.

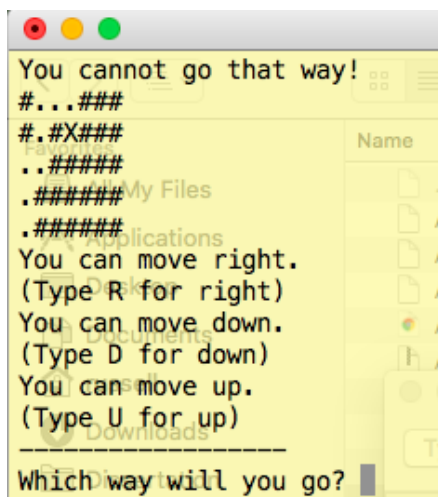


A screenshot of a Java IDE window, similar to the previous one. The console output shows the maze game after a move. The grid now shows the path taken: the first row is all '#', the second row is '#.', the third row is '..#.', the fourth row is '.#.', and the fifth row is '.#.'. The 'X' is still in the second row, first column. Below the grid, it lists movement options: 'You can move left. (Type L for left)', 'You can move right. (Type R for right)', and 'You can move down. (Type D for down)'. After a separator line, it asks 'Which way will you go?'. The background shows a file explorer with various files.



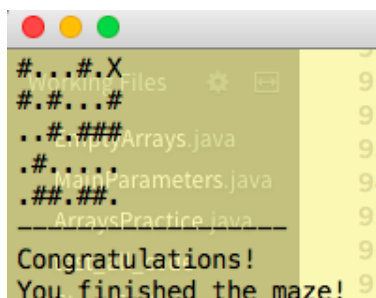
```
#...##  
#.#X##  
..####  
.#####  
.#####  
You can move right.  
(Type R for right)  
You can move down.  
(Type D for down)  
You can move up.  
(Type U for up)  
-----  
Which way will you go? █
```

The user should be able to move in any direction from those available according to your map data. The user should not be able to move in a direction that takes them to a position with value 0 in your map. If they try to enter an invalid direction, simply clear the screen and print the same options to prompt them again, but print a message at the top of the screen to tell them they cannot go that direction.



```
You cannot go that way!  
#...##  
#.#X##  
..####  
.#####  
.#####  
You can move right.  
(Type R for right)  
You can move down.  
(Type D for down)  
You can move up.  
(Type U for up)  
-----  
Which way will you go? █
```

Once the user navigates the maze successfully to the top-right corner, make sure to clear the screen and print the final route information, and congratulate the user on finding the exit of the maze.



```
#...#.X  
#.#...#  
..#...#  
#...#  
#...#  
#...#  
-----  
Congratulations!  
You finished the maze!
```

Make sure to write the template methods shown, and do not try to do everything in the `main` method of your program. The object types you will need are available with the import declarations

needed at the top of the file, and do not import any others. Keep track of the user's current position in the maze by using an instance of the `Point` class. Remember, you can modify the field variables (`x` and `y` for an instance of `Point`) of an object inside a method if you pass the object as a parameter.

Use four boolean variables to help you keep track of whether the user can move left, right, down, or up. The user should not be able to move in a direction that takes them to a position outside the legal indices of the 2D map array. Pass the boolean variables as arguments to the parameters of both `printDirections` and `updatePosition`. The boolean return type for `updatePosition` should be used to decide whether to print that the user cannot move in their chosen direction.

You should check that you can print your maze map in the same orientation as seen in your `maze.txt` file. The indices of a 2D array do not automatically match up with the coordinate system of the 2D plane. I recommend you let the first index keep track of the y coordinate height which would correspond with a "row" index. Let the second index keep track of the x coordinate width which would correspond with a "column" index. You have to fill the rows of your 2D array in reverse order from the rows in the file and you need to print the rows of the maze in reverse order if you want to have it match the orientation in the file. Then a move "up" in your maze would correspond with an index increment of `y`, and a move "down" would correspond with an index decrement of `y`. If you do not fill and print your maze this way, your `y` variable would need to be updated in the reverse way. Whatever you decide, you must be consistent!

If you decide to write a method to test parts of your code, you could leave that extra code in your submission, but make sure to get rid of any calls to it before you submit. If you have an adequate testing method, then it can earn you up to 5 buffer marks (doesn't contribute to a mark above 30).

Please submit both your `Maze.java` program and your `maze.txt` file.

Grading

Marks will be allocated out of 30 points:

- (0 points) You do not submit any program file.
 - (5 points) Your program does not compile. The marker cannot fix the compiler error easily, or if they can, the output is severely mismatched with the assignment requirements.
 - (10 points) Your program does not compile, but the marker can fix the error easily.
 - (15 points) Your program compiles, but only outputs a few of the assignment requirements.
 - (20 points) Your program compiles, outputs most of the assignment requirements, and is missing documentation comments or has formatting issues.
 - (25 points) Your program compiles, outputs everything required, but is missing documentation comments or has formatting issues.
 - (30 points) Your program compiles, outputs everything required, and has proper documentation and formatting.
-

If you enjoy making this game, on your own you could make creative additions, such as secret items for the user to find at certain locations, or portals that transport them from a dead end to another location in the maze. You could even make a narrative to go along with the user's traversal of the maze, printing parts of a story as they find their way through to the end.