Maze

By the time you are done with this activity, you should be able to:

- write more complicated recursive methods.
- make sophisticated use of arrays, including using three-dimensional arrays and using an extra int to represent a partially-full array.

Playing the Game

Included in the Maze module is a file Maze.jar. This is a compiled version of the working game. To play it, use a terminal to navigate to the directory containing the file and type this on the command line:

```
java -jar Maze.jar
```

A maze, being a puzzle, is essentially a one-player game. Our program both generates and solves the maze, so it becomes a zero-player game: there's nothing for the user to do but watch. That said, watch it run a few times to get a sense of what the program accomplishes.

Overview

You have been given an incomplete skeleton file Maze.java. Your job is to complete the program so that it behaves *exactly* like Maze.jar. As a tiny bit of creative freedom, you are welcome to change the colors.

Take notes as you work. What bugs and conceptual difficulties did you encounter? How did you overcome them? What did you learn?

You have also been given a set of tests in MazeTest.java.

Start by running all of the tests. Most of them do not pass because they depend on methods you haven't written yet! (A few of them pass because a method stub returns, e.g., false, which happens to be the correct answer for some tests.)

Now go through the tests *in the order in which they appear in MazeTest.java*, writing the necessary code in Maze.java to pass them. Pass one test before moving on to the next one.

You should not modify any of the given tests. If it is helpful for debugging, you may write additional tests.

Tests involving `contains`

There are two tricky parts about the `contains` method. The first is that it is only supposed to pay attention to the first `n` pairs in `list`. The second is that you will eventually want to compare `pair` (a pair of numbers) to `list[i]` (also a pair of numbers) for some `i`. You can't use `==` because this will only tell you if `pair` and `list[i]` are references to the same array; you want to know if these two arrays contain the same numbers.

Some questions to ask as you write this or any other method:

- According to the Javadoc comment, what is the method supposed to accomplish?
- What arguments, if any, does the method take? What do they mean?
- What, if anything, is the method supposed to return? What does the returned value mean?
- Is the method supposed to have any side effects, such as printing or modifying existing data structures?
- Would calling existing methods make this one easier to write?

Tests involving `remove`

The `remove` method takes the same arguments as `contains`, but it modifies `list` rather than returning a value. The remaining elements of `list` are *not* kept in order, If, for example, `pair` appears at index 3 and `n` is 10, element 3 of `list` is replaced with element 9.

Drawing pictures of the data structures before and after a call to `remove` can help clarify your understanding of what is changing.

In the test, note the calls to the handy `deepToString` method in the `Arrays` class. (`Arrays` is in the `java.util` package, but that is imported at the top of MazeTest.java, so we don't have to use the long name `java.util.Arrays`.)

Tests involving `addToFront`

The `addToFront` method is sort of the opposite of `remove`. It creates a new array that is slightly longer than the input array `list`.

Tests involving `chooseRandomlyFrom`

The `chooseRandomlyFrom` method should return one of the elements of `list`. Be sure to return one of the original elements (rather than making a copy using `new`).

Tests involving `addPassage`

The `addPassage` method is not terribly complicated, but it does depend on understanding the argument `passages`. See the explanation in the Javadoc comment for `addPassage`.

The constants `NORTH`, `EAST`, `SOUTH`, and `WEST` are defined at the top of Maze.java. These are like variables, but they are not defined inside any particular method. They can be seen from any method in the Maze class, or even from another class if specified as `Maze.NORTH`, etc. A variable with such a vast scope would be a terrible idea, but these are not really variables: because they are declared `final`, they can never be modified. (The compiler will stop us if we try.)

Tests involving `expandLocation`

The `expandLocation` method takes a whopping five arguments, three of which are arrays. If this is intimidating, take a deep breath, read the Javadoc comment, and notice that you've seen these (or things like them) before in this program:

`passages` is just like the array of the same name passed to `addPassage`.

`unexplored` and `n` look a lot like `list` and `n` from `contains`.

`here` is another pair.

`direction` is just an int, specifically one of the constants we dealt with in `addPassage`.

The first few lines, which have been given to you, take advantage of the constant array `OFFSETS` to avoid a four-way if-else statement. Do you understand how this works?

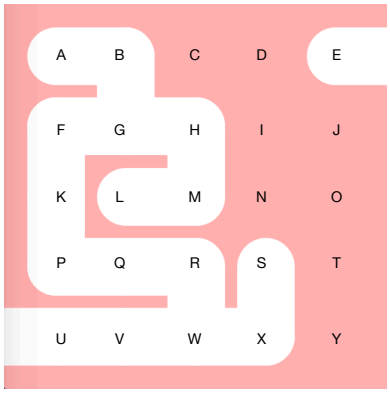
Don't hesitate to call methods that you've already written!

Tests involving `expandMaze`

Six arguments! Is there no end to this madness?

To be frank, the data structures here are getting a bit hairy. Object-oriented programming, which will be introduced shortly after this project, provides powerful tools for dealing with this sort of complexity.

Read the explanations of the arguments `done`, `frontier`, and `explored` in the Javadoc comment. It will be helpful to look at the partially-completed maze below and work out which locations belong in `done`, which in `frontier`, and which in `explored`. (Each location is in exactly one of these lists.)



Since this method is such a doozy, you've been given most of it. You just have to fill in the arguments for the calls to other methods.

Tests involving `solve`

You may, at some point, have a partially-correct version that passes `testSolveEasy` but not `testSolveHard`.

The `solve` method is recursive. It answers the question, "What path is there, if any, from `start` to `goal`?" The simple base case occurs when `start` and `goal` have the same coordinates. In the recursive case, the easier question is, "What path is there, if any, from my neighbor to `goal`?" This question is easier because all passages lead away from the beginning of the maze at 0, 0.

This method is recursive but it also uses a loop because there are four different directions to check for neighbors. The constant array `OFFSETS` comes in handy again here.

You have been given most of this method.

Once you're passing all of the tests, the program should run perfectly. Be sure to test the whole program, though; there's a remote possibility that you still have a bug. In other words, passing the tests is necessary but not sufficient evidence of a correct program.

To celebrate your success, you may enjoy playing with the value of `width` at the beginning of `main` or the argument to `StdDraw.pause` near the end of `main`.

This work was supported by the Google Education and Relations Fund's CS Engagement Small Grant Program grant #TFR15-00411.

The principal investigator, Peter Drake, wishes to thank the following for their useful comments: the members of the CS-POGIL project, specifically Clif Kussmaul and Helen Hu; Maggie Dreyer; and various anonymous reviewers.