Joanna Klukowska
joannakl@cs.nyu.edu

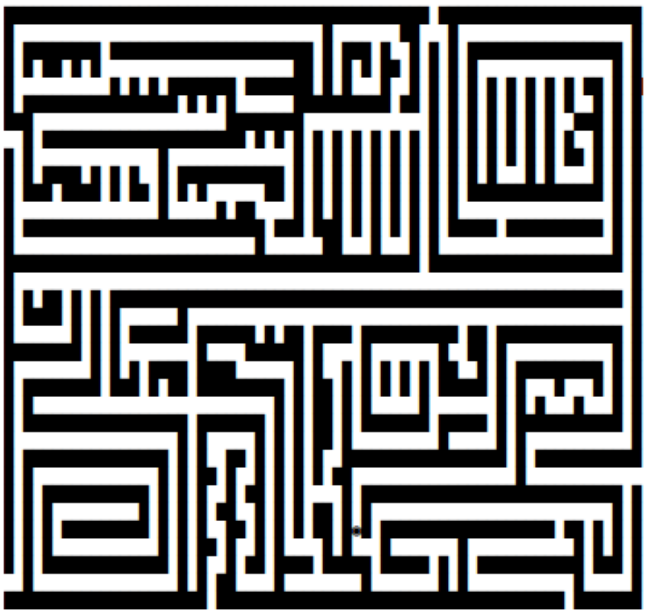
Project 3: Finding Your Way Out Of A Maze

Due date: Nov. 4, 11:55PM EST.

You may discuss any of the assignments with your classmates and tutors (or anyone else) but all work for all assignments must be entirely your own . Any sharing or copying of assignments will be considered cheating (this includes posting of partial or complete solutions on Piazza or any public forum). If you get significant help from anyone, you should acknowledge it in your submission (and your grade will be proportional to the part that you completed on your own). You are responsible for every line in your program: you need to know what it does and why. You should not use any data structures and features of Java that have not been covered in class (or the prerequisite class). If you have doubts whether or not you are allowed to use certain structures, just ask your instructor.

You are going to write a program that uses our new data structures, stack and queue, to explore a maze and, hopefully, find a way out of it. There are many algorithms that use different data structures and different strategies to explore mazes. In this assignment you will try two such approaches. In both of these approaches you will start from some initial position within the maze and evaluate the neighboring locations until either you find a way out, or you discover that there is no way out. The rough outline of both approaches is to examine locations around the current position and decide which need to be examined further and which definitely do not lead to an exit (more details below in the algorithm description).

Objectives



Main goal: be able to get out of a maze.
The other goal of this programming project is for you to master (or at least get practice on) the following tasks:

- using and understanding command line arguments
- implementing reference based stack and queue classes according to the provided interface
- writing classes
- working with existing code
- testing implementation of classes using jUnit

Start early! This project should take a fairly short time to implement, but there you may need some time for debugging. This is most likely first time that you are implementing reference based data structures.

For the purpose of grading, your project should be in the package called `project3`. This means that each of your submitted source code files should start with a line:

```
package project3;
```

Program Usage

Your program is a console based program (no graphical interface).

The program is started with **two command line arguments** (you can either run it in a terminal or in an IDE, like Eclipse, as long as there is a way of specifying the command line arguments for the program).

If the program is run with fewer than two arguments, it is an error (the program prints an error message and terminates).

If the program is run with more than two arguments, the additional arguments are ignored.

The first command line argument is a name of an input file containing a maze to be used by the program. (This project comes with two such files. You can (and should) create your own mazes for testing.) The name of the file can be entered as an absolute or relative path.

Joanna Klukowska
joannakl@cs.nyu.edu

The second command line argument is one of the following keywords: `stack` or `queue`. If any other keyword is used, the program prints a message indicated that this option is not supported. (The meaning of the two keywords is described below.)

Solving the Maze - An Algorithm

You should be able to solve the above maze on paper easily. But how do you tell a computer to find a way out? The computer cannot just look at the whole maze and figure out where the nearest exit is. Think of yourself being stuck in a maze without having a global view. You only have a local view of what is immediately next to you and before you take a step it might be wise to decide if there ever might be a reason to come back to your current location. The algorithm below describes this type of search for a way out of a maze.

The program should keep a set of locations that need to be examined (places that we might need to get back to in order to test another alternative, for example when we have two choices and we can only follow one of them right away). The exact representation of this set does not matter from the point of view of the algorithm. At the very beginning the set consists of a single location that is the initial position. We explore the maze by repeating the following steps:

- if the set is empty, there is no way out of the maze - algorithm ends;
- otherwise we take the next element from the set
  - if the location that we just picked has been visited before, no need to look at it again, we skip the rest of this step
  - if the location is an exit, we found the solution - algorithm ends
  - otherwise, we examine all of the neighbors of that location (the order does not matter and for a more interesting simulation, the order of "examining the neighbors" should be randomized), for each location that is not a wall, we add it to the set of locations we are maintaining and then mark the current location as visited

This algorithm is implemented for you. Your task is to provide two different implementations for the "set of locations" that is used in the algorithm. For this purpose you need to implement a stack of locations and a queue of locations (details below).

Implementation

The project comes with several classes that provide a program that implements a maze simulation. Study the source code to understand what it is doing.

The classes given to you are:

`Labyrinth` - the class that represents a 2D rectangular maze

`Simulation`- the actual program that simulates an exploration of a maze

`PossibleLocations` - the interface that provides requirements for the two classes that you need to implement

`Location` - the class that represents a position/location of a single square in the maze

`SquareType` - the enumerated type describing different types of squares in the maze

Your task is to provide two different implementations of the following interface included in the project:

```
1 /**
2  * Represents a set of locations used for the
3  * search for an exit out of a maze.
4  */
5 public interface PossibleLocations {
6         /**
7          * Add a Location object to the set.
8          * @param s object to be added to this PossibleLocations
9          */
```

Joanna Klukowska
joannakl@cs.nyu.edu

```
10          void add ( Location s );
11
12          /**
13           * Remove the next object from the set. The specific
14           * item returned is determined by the underlying structure
15           * of set representation.
16           * @return the next object, or null if set is empty
17           */
18          Location remove ();
19
20          /**
21           * Determines if set is empty or not.
22           * @return true, if set is empty, false, otherwise.
23           */
24          boolean isEmpty();
25  }
26
```

You may implement additional classes if you wish. You are not allowed to modify any of the classes provided to you as part of this project.

`PossibleLocationsStack` class

This class should implement the `PossibleLocations` interface. It should be a reference based stack (the implementation should use a linked list like storage for its nodes). You have to implement all of this class yourself. You cannot use any of the Java provided classes that implement a stack or a list.

Note that in order to implement the interface the typical `push` method needs to be called `add` and the typical `pop` method needs to be called `remove`.

You my implement additional methods if you wish.

`PossibleLocationsQueue` class

This class should implement the `PossibleLocations` interface. It should be an array based queue (the implementation should use an array, not an ArrayList object, as storage). You have to implement all of this class yourself. You cannot use any of the Java provided classes that implement a stack or a list.

This class should be able to handle any number of elements to be stored in the queue. This means that you will need to *grow* the array when its size reaches the capacity of the storage array.

This class should provide two constructors:

- a default constructor
- a one parameter constructor that takes a single integer as its parameter - this value indicates the initial capacity of the array

Note that in order to implement the interface the typical `enqueue` method needs to be called `add` and the typical `dequeue` method needs to be called `remove`.

You my implement additional methods if you wish.

Unit Tests

Testing of the code that you implement is as crucial as writing it in the first place. For the two classes that you implement, you need to develop a set of unit tests that will test the class' correctness independent of the program itself.

Joanna Klukowska
joannakl@cs.nyu.edu

The objective of unit tests is to test individual methods of the class in such a way that the results are as independent of the correctness of other methods as possible (although this is not always doable). The unit tests should be based on the specification of the class, not on specific implementation.

Here are couple of examples of unit tests for either of the classes that implement the `PossibleLocations` interface:

- create an object of the class and assign it to a reference variable; test if the object is empty using the `isEmpty` method (it should be empty at this point) - problems here would indicate errors either in the `isEmpty` method or in the constructor
- create an object of the class and assign it to a reference variable; add an element to the stack; test if the object is empty using the `isEmpty` method (it should NOT be empty at this point) - problems here would indicate errors either in the `isEmpty` method or in the `add` method
- ...

The tests should be short and test one thing at a time.

You need to write two test classes:

- `PossibleLocationsStackTest` - test class for the class `PossibleLocationsStack`
- `PossibleLocationsQueueTest` - test class for the class `PossibleLocationsQueue`

(Note: you will be working on developing jUnit tests in the next recitation activity.)

Programming Rules

You should follow the rules outlined in the document *Code conventions* posted on the course website at https://joannakl.github.io/cs102_f17/notes/CodeConventions.pdf.

You may use any exception-related classes.

You have to implement your own reference based stack and your own array based queue. You are not allowed to use the classes provided in Java API that implement these data types.

Working on This Assignment

You should start right away!

The two classes are independent of one another. You can implement one class at a time.

You should write the unit tests for the two classes and you should fix all mistakes that you discover. The unit tests should be based on the specification, not your own implementation. A few days before the due date, the autograder will be available with a set of limited tests (many of the grader tests will be hidden). You need to convince yourself with your own unit tests, that the two classes are correct.

You should make sure that your program's results are consistent with what is described in this specification by running the program on carefully designed test inputs and examining the outputs produced to make sure they are correct. The goal in doing this is to try to find the mistakes you have most likely made in your code.

You should backup your code after each time you spend some time working on it. Save it to a flash drive, email it to yourself, upload it to your Google drive, do anything that gives you a second (or maybe third copy). Computers tend to break just a few days or even a few hours before the due dates - make sure that you have working code if that happens. (A second copy of the files on the same computer is a good idea to keep multiple versions, but it is NOT a good backup since you do not have access to it if there are problems with your computer.)

Grading

If your program does not compile or if it crashes (almost) every time it is run, you will get a zero on the assignment.

Joanna Klukowska
joannakl@cs.nyu.edu

If the program does not adhere to the specification, the grade will be low and will depend on how easy it is to figure out what the program is doing.

50 points class correctness: correct behavior of methods of the two classes

20 points design and efficiency of the implementation of the required two classes and any additional classes

10 points design and implementation of the two test classes

20 points proper documentation, program style and format of submission

How and What to Submit

For the purpose of grading, your project should be in the package called `project3`. This means that each of your submitted source code files should start with a line:
```
package project3;
```

Your should submit your source code files (the ones with .java extensions only) in a single **zip** file to Gradescope. The files that should be submitted are `Color.java`, `PossibleLocationsStack.java`, `PossibleLocationsQueue.java`, `PossibleLocationsStac` `.java`, `PossibleLocationsQueueTest.java`. If you implement any other classes, they should be included as well. You do not need to include the classes that were provided with this assignment.

You can produce a zip file directly from Eclipse:

- right click on the name of the package (inside the `src` folder) and select Export...
- under General pick Archive File and click Next
- in the window that opens select appropriate files and settings:
  - in the right pane pick ONLY the files that are actually part of the project, but make sure that you select all files that are needed
  - in the left pane, make sure that no other directories are selected
  - click Browse and navigate to a location that you can easily find on your system (Desktop or folder with the course materials or ...)
  - in Options select ”Save in zip format”, ”Compress the contents of the file” and ”Create only selected directories”
- click Finish