Assignment 4

Due: Sunday, March 20th , 2019 before 11:55 pm

Objectives

- Implement a reference-based stack as a generic Stack class
- Create a stack of `Integer`s using your generic Stack class
- Write an implementation of `MazeSolver` which will use your generic Stack class as an element in the solution.

Introduction

We will continue to explore different data structures in this course – and in particular concept of abstract data types – by working with a Stack. In addition there will be some problem solving you will complete that makes use of your work implementing the stack.

More specifically:

1. You will create the class named `StackRefBased` which will implement the interface named Stack (e.g., `push`, `pop`, `isEmpty`, etc.) using the same kind of nodes we have seen in linked lists. As part of this step you will test your implementation by working with a stack of `Integer` class instances.
2. Using your `StackRefBased` and several other class provided to you, write `MazeSolver` which finds a path from the *start* to the *finish* of a two-dimensional maze.

Quick Start

a) Download into some directory the files named `a4tester.java`, `Maze.java`, `MazeLocation.java, Stack.java`, `StackEmptyException.java`, and `StackNode.java`.

b) Into that directory create a subdirectory named `tests` and download into this directory all of the text files in the assignment's `tests` directory.

c) (If you know how to use ZIP files, you can download `a4_files.zip` into a directory, and after unzipping it, all of the files described in a) and b) will be ready for you.)

d) Compile and run the test program `a4tester.java`. As in the previous two assignments, use this tester program as you complete the requirements of the assignment.

e) The “Grading” section at the end of this document describes how marking will be based, amongst other criteria, upon which tests pass or fail.

Part 1: StackRefBased.java

In lectures we have looked at the Stack abstract data type. We have also seen how the Java “Generics” mechanism permits us to write the implementation of some abstract data type once, and then re-use that ADT for different kinds of data items (i.e., stack of `Integer`, stack of `String`, stack of `Movie`, stack of `Song`, etc.).

In the first part of this assignment you will complete an implementation of `StackRefBased.java`. The methods are currently empty of all save the absolute minimum amount of code for compilation. The behavior expect of each method is described in the Java interface file named Stack.java; this description is in the form of comments.

You will notice that `StackRefBased.java` is a generic class. Specifically it is meant to be a stack of items of some type `T` (where `T` must be a `class`). More precisely, `StackRefBased.java` will be implemented using instances of `StackNode.java`, which is also a generic class. That is, your ref-based structure will consist of instances of `StackNode`, while within each `StackNode` will be an instance of the real type you wish to push onto the stack.

For example, here are some lines of Java very similar to the first bit of `testBasicStack()` in `a4tester.java`:

```
Stack<Integer> s;
s = new StackRefBased<Integer>();
displayResults(s.isEmpty());
displayResults(s.size() == 0);
s.push(22);
displayResults(!s.isEmpty());
displayResults(s.size() == 1);
```

The variable `s` is a reference variable, and in the second line we assign to it an instance of `StackRefBased`. However, notice the syntax that involves angle brackets(“<”, “>”): we indicate this new stack instance is to be used for `Integer` values. A few more lines further along we `push` a value onto the stack `s`. (We also take advantage here of *autoboxing*. This helps us as `push` is expecting an `Integer` as a parameter, but we have given it an `int`; Java automatically wraps (or “autoboxes”) our `int` value into a new `Integer` object.)

Once you have correctly completed the implementation of `StackRefBased.java` – or, more precisely, once all of `a4tester.java` tests in the method `testBasicStack()` have passed – then you will have a stack that can be used for any class, including one for `MazeLocation` as will be needed in the next part of the assignment.

Part 2: MazeSolver.java

Problem Description

A *maze* may be represented textually by a two-dimensional array of characters; the ("*") character represents a wall, and a space (" ") represents an open square. One open square on the top wall represents the start square, and one open square on the bottom wall represents the finish square. Finding a path through the maze means finding a path of open squares (where each square is denoted by its *row* and *column*) that connect the start and finish squares. (The path usually includes the start and finish squares, too). The only moves allowed on a path from any given square are up, down, left and right (respectively *row-1*, *row+1*, *column-1*, *column+1*).

A *maze file* is a plaintext file containing all details of a maze but without indicating the path through the maze. In the example that follows (corresponding to *maze06.txt*) the first three lines indicate that the maze described in the file is 9 rows tall and 11 columns wide; the start square is at row 0, column 1; and the end square is at row 8, column 9. After these first three lines are the walls and spaces of the maze.

```
9 11
0 1
8 9
* *********
*     *   *
*** * *** *
*   *   * *
******* * *
*     *   *
* ******* *
*         *
********* *
```

We could also represent the walls and spaces of this maze in the form of a two-dimensional boolean array:

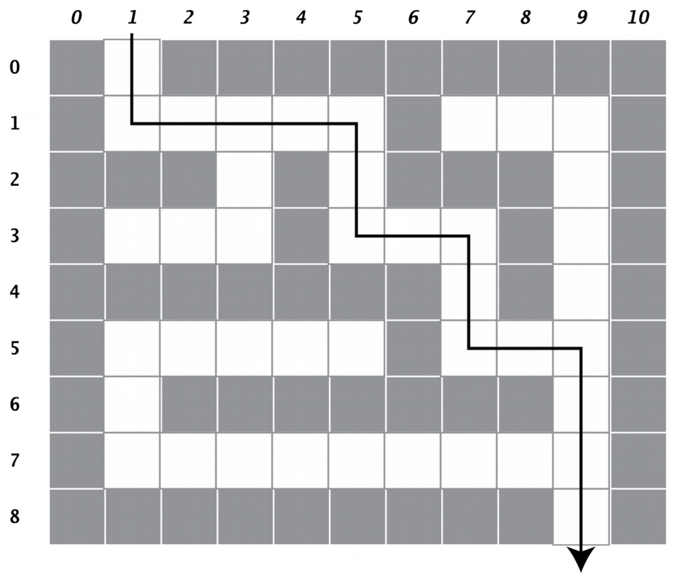
	0	1	2	3	4	5	6	7	8	9	10
0	true	false	true	true	true	true	true	true	true	true	true
1	true	false	false	false	false	false	true	false	false	false	true
2	true	true	true	false	true	false	true	true	true	false	true
3	true	false	false	false	true	false	false	false	true	false	true
4	true	true	true	true	true	true	true	false	true	false	true
5	true	false	false	false	false	false	true	false	false	false	true
6	true	false	true	true	true	true	true	true	true	false	true
7	true	false	false	false	false	false	false	false	false	false	true
8	true	true	true	true	true	true	true	true	true	false	true

The mazes we will consider in this assignment have a single path from start to finish. A path is represented by this sequence of row and column coordinates, such as the following:

```
(0,1) (1,1) (1,2) (1,3) (1,4) (1,5) (2,5) (3,5) (3,6) (3,7) (4,7) (5,7) (5,8)
(5,9) (6,9) (7,9) (8,9)
```

(The line break in the path – i.e., into two lines – is done here for aesthetic reasons. Ultimately the path expressed in the way would be a single string with no line breaks.)

This path is represented by the line shown in the following diagram:



The algorithm for finding a path through the maze from *start* location to *finish* location can be expressed using the following pseudocode:

```
Keep track of locations visited so far with a 2D-boolean array
Push the start location onto a new stack
While (the stack is not empty && stack top is not the finish location)
    Use the location at the top of the stack as the current location
    Mark the current location as visited
    If there is an unvisited location next to the current location
        Push the unvisited location onto stack, and continue the loop
    Else
        Pop the stack (i.e., remove current location as it is top of stack
If stack is empty
   There is no path from start to finish
Else
    Stack contains path from finish to end
```

(Note: This is an example of a *backtracking algorithm*. ***Your solution to Part 2 must not use recursion to implement backtracking.***)

What you are to do

Several files are already provided to you:

- `Maze.java` accepts the pathname to a maze file in its constructor, and creates a new `Maze` object by reading the data from the text of the maze file. Please read this class, especially the comments, to understand what its methods do as you will need to use most – if not all – of these methods in your submission. You are not permitted to change this `Maze.java` without express written permission from Jason or Celina.
- `MazeLocation.java` encapsulates the row and column of a maze location. (Note, however, that when using the `isWall()` method in `Maze`, you need only provide the row and column and not a `MazeLocation` instance.)
- A version of `MazeSolver.java` which returns an empty string when `findPath()` is called.

Your task in Part 2 is to complete `findPath()`. The method accepts an instance of `Maze` as its only argument, and returns a string corresponding to the path from the start location to end location of the given maze. So, for example, if the filepath for the example shown in this description is given as the argument to the `Maze` constructor (e.g., "`tests/maze06.txt`"), then the string to be returned by `findPath()` for this instance of maze is the string shown at the top of page 4 of this description. (This particular test case is used in `a4tester.java`).

Submission

Submit the following files to conneX, and please ensure you submit these files and not simply save a draft.

- `StackRefBased.java`
- `MazeSolver.java`

And remember it is OK to *talk* about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution. (We will use software-similarity analysis tools on assignment submissions.)

Grading

If you submit something that does not compile, you will receive a grade of 0 for the assignment. It is your responsibility to make sure you submit the correct files.

NOTE: Test cases are numbered from 0. The first test is test 0, the last test is test 37.

Requirement	**Marks**
You submit something that compiles	1
Your code passes the first set of test cases (0 to 21 inclusive)	5
Your code passes the second set of test cases (22 to 26 inclusive)	1
Your code passes the third set of test cases (27 to 37 inclusive)	6
Your solution in `MazeSolver.java` follows the posted coding conventions on commenting.	1

Total 14

In order to obtain a passing grade on the assignment, you must satisfy at least the first three requirements by passing all of their test cases.