CSC 115: Fundamentals of Programming II
Assignment #3: Recursion

Due date

Wednesday, March 4, 2015 at 9:00 pm via submission to conneX.

How to hand in your work

Submit the requested files through the Assignment #3 link on the CSC 115 conneX site. Please ensure you follow all the required steps for submission (including confirming your submission).

Learning outcomes

When you have completed this assignment, you should have become acquainted with:

- How to create an implementation of a backtracking maze-solver *using recursion*.
- How to *use a linked list* to represent the computed path through the maze.
- How to write a Tester class alongside your implementation.

Path finding

Problem description

A *maze* may be represented as 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 text file containing all details of a maze but without indicating the path through the maze. In the example that follows (corresponding to *maze01.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 details of the maze's squares.

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

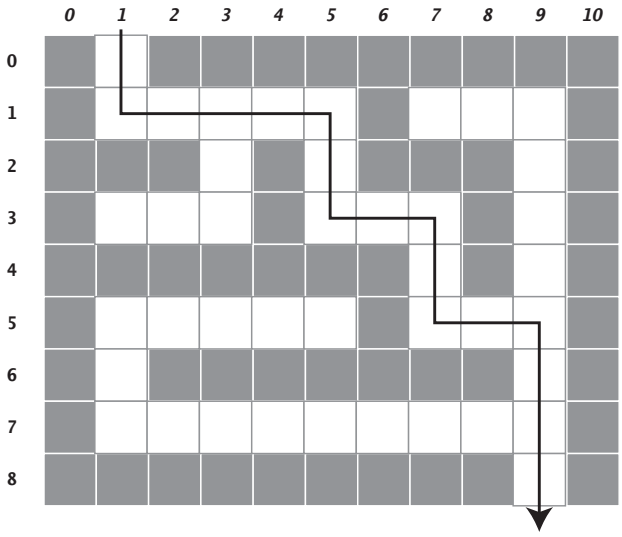
We could also represent this maze Java as a 2D-array of booleans:

	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)
```

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



When representing such a path, each of the squares is stored as a *MazeLocation* instance, and the sequence of squares comprising the path can be stored in a *MazeLocationList* (with is itself made up of *MazeLocationListNodes* each containing a *MazeLocation*). In such a list the head node stores information indicating the starting square, and the tail node stores information indicating the ending square. If the list is null, then this can indicate the absence of path. The code for the three classes mention above is provided to you.

What you are to do

Also provided to you is a Java program called *RunSolver.java*. However, this program only handles the work of reading in a maze file and printing out a path.

You are to write the code for a class named *Maze.java* that will be used to find a path through a maze. The code provided in *RunSolver* will therefore invoke methods on the implementation of Maze that you provide. The class must have the following methods:

- A constructor that accepts five parameters: an array of Strings containing details of each maze square, the row and column of the starting square (ints), and the row and column of the finish square (ints). The constructor will also create any other data structures you will need to complete the assignment. The constructor's signature will be:

```
public Maze(String[] textmaze, int startRow, int startCol,
    int finishRow, int finishCol)
```

- A public method named `solve()` that takes no parameter and returns an instance of *MazeLocationList*. If there is no path through the maze, then

`solve()` returns null. The method `solve()` will call the recursive method. The signature of `solve()` must be:

```
public MazeLocationList solve()
```

- A private method named `findPath()` taking four parameters (starting row and column, finish row and column), returning true if a path is found between the two square and false otherwise. The signature for `findPath()` must be:

```
private boolean findPath(int fromRow, int fromCol,
    int toRow, toCol)
```

- A public method named `toString()` that will print a representation of the data stored in your instance of Maze. You will use this primarily when developing and debugging your solution. Therefore the output produced by the method must be in some format that helps you understand your own code. (No particular output format is required.) The signature for `toString()` must be:

```
public String toString()
```

Please note that you are not permitted to change *RunSolver.java*.

Some of the details of path finding through a maze will be described in a forthcoming lecture. Those wishing a head start may read through section 6.1 in the textbook ("Java: Walls & Mirrors" by Prichard and Carrano).

If you make use of code from the textbook or lecture or from the web as part of your solution, then you must provide a comment in your Java files near your relevant code indicating the original source of the idea (e.g., text page number, or URL, etc.)

The five pairs of test files provided for this assignment are to be used for final testing of the solver. For example, `maze01.txt` contains the maze diagrammed earlier in this assignment description, and the correct result (i.e., a path through the maze) is stored in `path01.txt` (i.e., it contains one line per square in the path). The program will be run as follows:

```
$ java RunSolver maze01.txt
```

If you want to store the results into a file for further examination or comparison with the expected results, then use I/O redirection. Below the output would be redirected to a file named results.txt (i.e., the file will contain the output and that output will not appear on the console window):

```
$ java RunSolver maze01.txt > results.txt
```

What you are to write

1. Please keep all code within a single directory. We will take advantage of what is called the *default package* in Java, i.e., all Java files in the same directory are considered to be in the same package. This will permit you to use package-private access for the assignment without a complex directory structure.

2. Write *Maze.java* as specified earlier in this assignment description.

3. Use the *MazeLocation*, *MazeLocationList*, and *MazeLocationListNode* files. An implementation of a reference-based linked list is also provided for you (*MazeLocationListRefBased*).

4. As you complete the work for step 2, you must write tests in *MazeTester.java*. These tests will be in order of increasing difficulty and this order should reflect your implementation strategy. Ideally you will write a test in *MazeTester.java* before adding relevant implementation code in *Maze.java*. (An appendix at the end of this assignment description gives an example a test suggesting how you could write such tests.) **You must have at least six tests, and <u>none</u> may be equivalent to the five sets of test files.**

5. In all of your coding work, please follow the coding conventions posted at conneX (i.e., a document in the "Lectures & stuff" section).

Files to submit (three in total):

1. *Maze.java*
2. *MazeTester.java*
3. *A file named test_output.txt (which also includes a description of what each test is checking followed by the test input and output)*

Grading scheme

- "A" grade: An exceptional submission demonstrating creativity and initiative going above and beyond the assignment requirements. The program runs without any problems and implements the required classes and methods in those classes. Public methods are tested with output showing tests and their results. Any extra work appears in class files having the name *<file>Extra.java*, and identified within the file (i.e., Class comment) are details of your extension to the assignment demonstrating creativity and initiative.

- "B" grade: A submission completing the requirements of the assignment. The program runs without any problems and implements the required classes and methods in those classes. Public methods are tested with output showing tests and their results.

- "C" grade: A submission completing most of the requirements of the assignment. The program runs with some problems but does implement the required classes and methods, and yet might not have the expected output. Methods are tested with output showing tests and their results.

- "D" grade: A serious attempt at completing requirements for the assignment. The program runs with major problems, or does not contain implementations of all required classes and methods.

- "F" grade: Either no submission given, the submission does not compile, or submission represents very little work.

Appendix

Example of a test case suitable for *MazeTester.java*

```
boolean passed = true;
String testcase[] = {"* ****",
                     "*    *",
                     "**** *"};
Maze maze = new Maze(testcase, 0, 1, 2, 4);
MazeLocation expected[] = new MazeLocation[6];
expected[0] = new MazeLocation(0, 1);
expected[1] = new MazeLocation(1, 1);
expected[2] = new MazeLocation(1, 2);
expected[3] = new MazeLocation(1, 3);
expected[4] = new MazeLocation(1, 4);
expected[5] = new MazeLocation(2, 4);
MazeLocationList result = maze.solve();
if (result == null) {
    passed = false;
} else if (result.isEmpty() == true) {
    passed = false;
} else {
    for (int i = 0; i < result.getLength(); i++) {
        MazeLocation loc = result.retrieve(i);
        if (loc.equals(expected[i]) == false) {
            passed = false;
            break;
        }
    }
}
if (passed) {
    System.out.println("Test (horizonal path): passed");
} else {
    System.out.println("Test (horizonal path): FAILED");
}
```