CSc 127B — Introduction to Computer Science II
Fall 2015 (McCann)

http://www.cs.arizona.edu/classes/cs127b/fall15/

Program #6: A-Maizing!

Due Date: October 22nd, 2015, at 9:00 p.m. MST

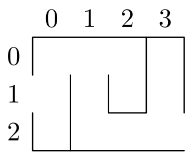
Overview: Using a stack to help solve (that is, to find a path through) a 2D maze is a common programming assignment. Less common is an assignment that asks students to use a stack to construct the maze. We're going to be less common. (As for the title, it's corn field maze season . . . corn . . . maize . . . get it?)

To build a maze using a stack, we will select a location in a matrix, and 'cut out' a random path until we reach a dead-end – a spot with no adjacent corn to cut. As we cut, we'll push onto the stack the locations that comprise the path we've followed. When we reach a dead-end, we'll back up (by popping locations from the stack) until we reach a location from which we can cut in a new direction. We continue this cutting and retreating until we're back at the starting location and there are no more directions to try. When that happens, the maze is almost done. All that's left to do is select the entrance and exit, and display the result.

Here's a fairly detailed pseudocode outline of the algorithm:

```
allocate a 2*r+1 by 2*c+1 2D array representing an r by c 2D corn maze
clear a randomly-selected starting location within the r by c maze
push the starting location onto the stack
while the stack is not empty:
    peek at the top of stack to learn current location
    if there is at least one new direction in which to cut:
        randomly select one of the available directions
        cut from the current location to the next cell in that direction
        push the location of that next cell on the stack
    otherwise:
        discard the location at the top of the stack
    end if
end while
select and place the maze entrance and the maze exit
display maze
```

To keep everyone consistent, we want you to use a $2r + 1$ by $2c + 1$ array of characters to represent an r by c maze. Here are two representations of the same maze; the one on the left is a graphical representation that's 3×4 (r by c), and the other is the character representation that's 7×9 ($2r + 1$ by $2c + 1$):



```
  012345678
0 #########
1 #     # #
2 # # # # #
3   # # # #
4 # # ### #
5 # #
6 #########
```

This representation allows alternate ("every-other") array locations to represent maze locations, with the in-between locations available to represent the walls and corridors between maze locations. Think of the maze as being created in a corn field. Initially, the maze builders start somewhere within the maze, pick a direction, and cut down corn stalks in that direction until they clear the next location in that direction, along with the path between the locations. In the example shown, the random starting location was (0,3). The random path cut a spiral to (1,2), backed up to (0,1), and cut some more to (2,0) before backing up to the starting location.

(Continued ...)

Assignment: Write a complete, well-documented Java program that uses the algorithm described above to create mazes with a number of rows (r, above) and a number of columns (c) entered by the user (have your program read these values, in that order, from the command line).

You need two classes, `Prog6` and `MazeHolder`. As the name suggests, `MazeHolder` represents the maze (using the $2r + 1$ by $2c + 1$ character representation) and provides methods to do the low-level operations such as 'cutting' to the next cell in a given direction, placing the entrance, etc. `Prog6` is in charge of running the maze creation algorithm, and will use a `MazeHolder` object to hold the maze being created. All of the code in `Prog6` thinks that the maze is r by c in size; it doesn't care how `MazeHolder` views the maze.

Cutting the maze doesn't provide the maze entrance and exit; we have to do that separately, after the maze has been created. For this program, we want you to pick a random maze location from the left side of the maze and remove the wall to its left to form the entrance. Do the same on the right side to form the exit.

You must create your own stack class with your own push, pop, and peek methods. Your stack of locations may be represented by any of the classes of Java's Collections Framework (JCF) that seem appropriate for the job (`Vector`, `ArrayList`, etc.), *except* that you can't reuse any existing stack methods (any named push, pop, and/or peek), because we want you to create your own versions of those methods.

Data: Your program is to get the size of the maze (the numbers of rows and columns) from the command line. If the user doesn't give two acceptable integer values, print a suitable error message and halt the program. As the content of the maze will be determined randomly, there is no other data to be gathered from the user.

Output: We'll try something a little different on this assignment. If you are content to earn no higher than 90% on this assignment, you may output just the $2r + 1$ by $2c + 1$ character dump of the internal maze representation. (This is the same output format shown above in the 7x9 representation you see in the figure on the right — you don't even have to display the numbers.) This will be very easy to do, hence the reduced number of possible points.

For a shot at 100%, you will need to augment the basic program using the `StdDraw` class from the `stdlib` graphics package (a.k.a. the `stdlib.jar` file we had you add to DrJava in Section #1) to produce a 'thin-walled' graphic representation, like the 3x4 figure on the left of the above character representation. (It's fairly easy to produce a graphic version in which each character position is a square. That's why we want to see the fancier version (with thin walls) for full credit.)

Turn In: For this assignment, you may place all of your classes in the `Prog6.java` file, or you many arrange them as one class per file; it's your choice. However you arrange the code, be sure to use the 'turnin' utility to electronically submit your `Prog6.java` and **all** other files to the `cs127bsXp06` directory at any time before the stated due date and time. Of course, you can turn them in late if you still have late days to use, or don't mind losing 20% per day if your late days are exhausted.

Hints, Reminders, and Other Requirements:

- Many of you have used the `stdlib` graphics package in CSc 127A. To help you refresh your memories, and to help those who have never used it before, links to the documentation and to some example graphics programs from my spring 127A class are on the class web page, below the link to this handout.
- We encourage you to get the maze algorithm working before you worry about the graphical representation of the maze. The text version of the maze is easy to display, and you can use it to help debug your algorithm before worrying about adding the graphics.
- We anticipate that the 'external' vs. 'internal' representation issues will be a significant source of confusion for this assignment. Remember that your application (which has the maze creation logic that uses the stack) thinks of the maze as having r rows and c columns, while the class maintaining the maze uses a representation that has $2r+1$ rows and $2c+1$ columns. Make sure you are mentally in the appropriate frame of mind when working in a particular area of your program.
- Helpful hint: Due to the coordinate system used by the `stdlib` graphics package, to avoid producing a 'mirror-image' graphical maze, you'll need to think (column,row) instead of (row,column) when drawing the lines that form the maze.