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

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

Section 8: Stacks

Pair up with anyone who is agreeable to pairing up with you, pick the first driver, and let's get to work!

PART I: Escaping a Maze Using a Stack

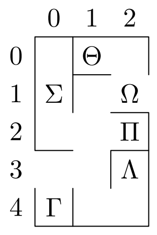
In Program #6, you are implementing in Java a stack-based algorithm that constructs a maze. There's also a stack-based algorithm for finding the exit of a maze from any starting point within it. This section activity asks you to execute that algorithm by hand, and to complete diagrams along the way to show that you are following it correctly.

1. The following algorithm is essentially Trémaux's Algorithm, which was designed in the late 1800s for people walking a hedge maze, not computers solving virtual mazes. Here's the computer-friendly, stack-based version; read it carefully.

   We start by ordering the four major compass directions. For this activity, we'll use this ordering: S, W, N, E. From your starting position within the maze, start by pushing your location on a stack of locations. Next, attempt to move south (the first direction). If you cannot, try the next direction (west). When you find a direction in which you can move, move one step in that direction, and push this new location on the stack.

   Continue until you reach the exit or reach a dead-end – a location in the maze from which the only way out is the way you came in. In this latter case, pop the stack (removing the dead-end), and return to the location currently on the top of the stack. Eventually, you'll return to a location from which you can go in a new direction (e.g., you went west on your first visit, but now you can try north), and you'll move that way. When you reach the exit, the stack contains the route you followed to get there, without all of the dead-ends.

2. Get out a sheet of paper and a pencil/pen. Your task is to use this algorithm to escape from the following maze, entering at (3,0). As you follow the algorithm, you will reach some of the locations marked with Greek letters. When you encounter a letter for the first time, draw the content of your stack, labeling the stack with the letter. When you reach Omega (Ω, the exit) and have drawn Ω's stack, you're done.



3. Question: It's easy to see that the stack's content is the path from the start to the exit. But our stack operations are just push, pop, peek, and isEmpty(). Create an algorithm that uses those operations, and a second stack, to print the path (the list of locations) in order from the start to the exit. Write down this algorithm (which can be in *pseudocode*, English statements structured to look like code), so that your SL can examine it.

✔ **CHECKPOINT 1** Raise your hand. Your SL will come over and check your work.

PART II: StringBuilder Stack

We've been using arrays, and lately members of the Java Collections Framework, to represent our lists of items. A string is also a list of items – characters! If we need a stack of characters, a mutable string is a perfectly reasonable representation.

1. For our representation of a stack of characters, we're going to use a mutable string object. Why use this instead of an ordinary `String` object?
2. We are going to use a `StringBuilder` object instead of a `StringBuffer` object. Use the API to answer this question: Why is `StringBuilder` the better choice for this application?
3. On the class web page is the shell of `StackSec8.java`. Load it into DrJava.
4. At the moment, all it contains is a `main()` method that contains some simple tests for instance methods that do not yet exist. Your job for the rest of this part: Write those methods. Specifically:
   - A no-argument constructor that creates the object representing the stack. You need a suitable instance variable to reference it, of course.
   - A boolean-returning, no-argument `isEmpty()` method.
   - An int-returning, no-argument `getOccupancy()` method. (Remember, occupancy and capacity are different things.)
   - The standard `push()`, `pop()`, and `peek()` methods. You should be able to figure out the arguments, if any, for each. For those that return a value, code them to return an `int`, so that both stack elements and error codes (`PEEK_POP_ERROR` is already defined for you) can be returned as necessary.
5. Compile and run your completed `StackSec8` code, and verify that the results match the expected output given at the top of the file.

✔ **CHECKPOINT 2** Raise your hand. Your SL will come over and check your answers to the questions, in addition to your output.

PART III: Matching Brackets

In mathematical expressions and Java programs, brackets of various forms are used to group elements in ways that clarify intent. The Java compiler needs to make sure that the brackets in your programs are appropriately paired together, and, if brackets appear inside of pairs of brackets, all of the brackets are correctly 'nested'. You get one guess as to the data structure it uses to do this . . .

1. On the class web page is the shell of `Brackets.java`. Load it into DrJava.
2. This program accepts a string of characters on the command line and checks that any pairs of brackets it contains pair up correctly. The program includes some example situations that the program needs to handle, and provides the expected output. It also includes some internal comments to help you along, but that's all the help you get. Write the program! Be sure to test it on the situations given in the top comment block to make sure the program does what it needs to do in those situations.

✔ **CHECKPOINT 3** Raise your hand. Your SL will come over and verify that your program is correctly matching brackets.

(Continued . . .)

PART IV: Clean Up!

1. Log out of your computer.
2. Pick up your papers, writing implements, cell phones, trash, etc.
3. Push in your chair(s).

✓ **CHECKPOINT 4** Raise your hand. Your SL will come over and see if your workspace is so clean that its jokes would be funny only to pre-schoolers.

You're free to go! But, if you have time, we recommend that you use it to work some more on the programming assignment (individually, of course), if you're not already done.