Maze Solver

In this lab, you will use the convenience of the Stack data type to explore and find your way through a maze.

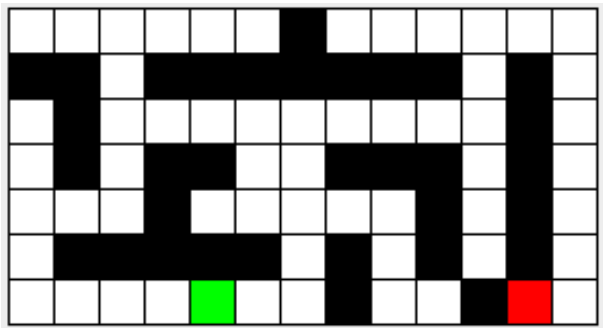
This lab is a little larger than past labs; it has many different pieces that all need to work together. Most pieces are small, but problems can pile up quickly if you're not careful. Test everything as you go! Comprehensive, thoughtful tests may take time to write, but they will undoubtedly save you time in the long run.

Begin by importing the starter code and data files into your project. Don't initially change any of the instance variables or method declarations in the starter code!

Part 1 - Representing the Maze

A basic maze has walls and pathways, and it has a starting point and an exit point (to keep things simple, let's assume it has no more than one of each). Furthermore, one wall is just like another, and any open space (not including start and finish) is also identical. So, we can think of a maze as being made up of individual squares, where each square either empty, a wall, the start, or the exit.

Below is a graphical representation of the maze found in the file "maze-2". The green box represents the start, the red box the exit, and the black squares the walls.



We will represent such a maze with a text file with the following format:
The first line of the file contains two integers; the first indicates the number of rows `R`, the second the number of columns `C`.
The rest of the file will be `R` rows of `C` integers. The value of the integers will be as follows:

- `0` – an empty space
- `1` – a wall
- `2` – the start
- `3` – the exit

In terms of coordinates, the **upper left** corner is position [0,0] and the lower right is [R-1, C-1].
For example, below is the text version of the maze shown previously (start is at [6, 4] and exit at [6, 11]).

```
7 13

0 0 0 0 0 0 1 0 0 0 0 0 0
1 1 0 1 1 1 1 1 1 0 1 0
0 1 0 0 0 0 0 0 0 0 1 0
0 1 0 1 1 0 0 1 1 1 0 1 0
0 0 0 1 0 0 0 0 0 1 0 1 0
0 1 1 1 1 1 0 1 0 1 0 1 0
0 0 0 0 2 0 0 1 0 0 1 3 0
```

The Square Class

Make a class **Square.java** that represents a single square (position) in a maze. For convenience, Square should maintain `public static final` integer variables (class constants) that represent its "types" (space, wall, start, or exit - see the `toString` method for more info). By convention, `final` variables (constants) are in `ALL_CAPS`. A Square should also have an *instance* variable that stores the `type` of `this` Square.

A square should also know where it is positioned in the maze. Give each square private integer variables named `row` and `col` to store its location within the maze.

Finally, a square should maintain its `status`. A square's `status` is used by the GUI / maze solver to display the progress of the solver as it explores the maze. See the `toString` method for more info, also see here if you're completely confused. Implement the following API:

`public Square(int row, int col, int type)`

Constructor to create a new Square object at a specific location and of the given type (wall, empty/open, etc.)

`public String toString()`

Returns a textual representation of the Square.

A Square's `type` is as follows**:**

- `_` Empty space; an open room that's currently unexplored (int value 0)
- `#` Wall (int value 1)
- `S` Start (int value 2)
- `E` Exit (int value 3)

A Square's `status` is as follows:

- `o` On the solver's "work list" (a data structure storing Square objects); in other words, this room is still being explored by the maze solver
- `.` Already explored by the maze solver
- `x` On the final path to the exit (you will use this later)

The `status` symbols are only applied to empty spaces. These values are used by the GUI to display the progression of the solver, while a maze is being solved. Watching the GUI demo run may help.

```
public boolean equals(Object obj)
```

Squares are considered equivalent if they have the same `row` and `column` (`type` isn't necessary for this).

```
public int  getRow()
public int  getCol()
public int  getType()
public char getStatus()
```

Accessor methods to get the values of the various instance variables (which should be `private`).

```
public void  setRow(int row)
public void  setCol(int col)
public void  setType(int type)
public void  setStatus(char status)
```

Mutator methods to change the values of the various instance variables (which should be `private`).

The Maze Class

Now we can set up the maze itself. Create a class **Maze.java** that stores the logical layout of a maze. It should contain a 2D array of Square objects. Initially, this array will be empty - you will use a method to populate it. Maze should also store references to the `start` and `exit` squares of `this` maze. Implement the following API:

```
public Maze()
```

A constructor that takes no arguments. The work of initializing the maze will be done in the `loadMaze` method described below.

```
boolean loadMaze(String fileName)
```

Loads the maze that is referenced by `fileName`. The format of the file was described previously. While logically this method could be private, this method must be public as it is called by the GUI.

If you encounter a problem while reading in the file, you should return false to indicate that it failed. Returning true indicates that you have now loaded the file from disk successfully.

Use `try` / `catch` blocks to catch the exception that is raised if the user specifies an incorrect file. Print out an appropriate error message when this occurs and return false (don't throw the exception, which would then need to be thrown by other calling methods - bad practice, but tolerated in AP CS for the sake of simplicity). Check the "exceptions" powerpoint in the lab folder for more info.

`List<Square> getNeighbors(Square s)`

Returns a List of the squares neighboring the parameter `s`. There will be at most four of these (to the North, East, South, and West) and you should list them in that order.

If the square is on a border, skip over directions that are out of bounds of the maze. Do not add `null` values.

Note: this method does NOT check whether a neighboring square is a wall - it just adds all neighboring squares. Checking for walls will be handled by the solver.

`Square getStart()`
`Square getExit()`

Accessor methods that return the saved start/exit locations for this maze.

`void reset()`

Return the maze back to the initial state after loading. Erase any marking on squares (e.g., visited or on the current worklist, stored in Square's `status` variable), but keep the layout (walls, empty, etc.).

One way you might do this is by also giving Square a `reset` method, then just loop through the squares and ask them to reset themselves.

`public String toString()`

Returns a String representation of this maze in the format given below (hopefully you have a working `Square.toString` method). Example, for the maze shown previously (the "maze-2" file):

```
_ _ _ _ _ _ # _ _ _ _ _ _
# # _ # # # # # # # _ # _
_ # _ _ _ _ _ _ _ _ # _
_ # _ # # _ _ # # # _ # _
_ _ _ # _ _ _ _ _ # _ # _
_ # # # # # _ # _ # _ # _
_ _ _ _ S _ _ # _ _ # E _
```

Before you continue, (in a `main` method) you should test that your Maze class works properly! Among other things, load a maze from one of the supplied files, get the neighbors of some specific square (the start square, for example), and assert that (1) there are the correct number of neighbors, and (2) the neighbors are in the correct locations. You probably should do this for the corners and border cases, at least. You should print out the maze, and to confirm your `getStart` and `getExit` methods return the correct squares.

Part 2 - Using your MyStack class

In a previous lab, you wrote a MyStack class that could store Integer objects. Copy this class into your current project, and make the necessary changes such that MyStack instead stores Square objects. Your MyStack class must also now implement the supplied interface StackADT, a contract that ensures your class is consistent with what the GUI expects.

Before continuing, ensure that your MyStack class is working properly by confirming it matches the behavior of the **java.util.Stack** class using an object of each (e.g. `MyStack mine = new MyStack()` and `Stack<Square> javas = new Stack<>()`). You may be wondering how Java's version is able to specify the type of object being stored (i.e., how you could make your MyStack class work with *any* type of object) - you will learn in due time! For now, your MyStack class will only store Square objects.

Part 3 - Solving the Maze

Now that you have working maze and stack data structures, you can use them to solve mazes! You'll next be writing the **MazeSolver.java** class, which bundles a maze with the functionality of determining if a given maze has a valid solution. That is, whether you can get from the start to the exit (without jumping over walls).

Our maze solving algorithm goes something like this:

> *Begin at the start location, and trace along all possible paths to (eventually) visit every reachable square. If at some point you visit the exit square, it was reachable. If you run out of squares to check, it isn't reachable.*

Boiling this down into pseudo-code, we have the following algorithm:

At the start

1. Create an empty worklist (e.g. a stack, some type of data structure that will remember all the squares to visit) of locations to explore.
2. Add the start location to it.

Each step thereafter

1. Is the worklist empty? If so, the exit is unreachable, as there are no more valid paths to try.
2. Otherwise, grab the "next" location to explore from the worklist.
3. Does the location correspond to the exit square? If so, you're done! The maze is solved.
4. Otherwise, it is a reachable (non-exit) location that we haven't explored yet. Explore it as follows:
    a. Get all the adjacent locations that are inside the maze and aren't walls, and...

b. Add them to the worklist for later exploration, *provided they have not previously been added to the worklist (they're not already explored).*

5. Also, record the fact that you've explored this location so you won't ever have to explore it again. Note that a location is considered "explored" once its neighbors have been put on the worklist. The neighbors themselves are not "explored" until they are removed from the worklist and checked for **their** neighbors.

Note that this pseudo-code is entirely agnostic as to what kind of "worklist" data structure you use (which, in this lab, is a stack). You'll pick one when you create the worklist, but subsequently everything should work abstractly in terms of the worklist's operations.

The MazeSolver abstract class

Create an `abstract` class **MazeSolver.java** that will implement the above algorithm, assuming the use of a "general worklist" (some data structure maintaining the locations in the maze to be explored - in this lab, it will be a stack). The MazeSolver class will not contain the data structure that actually stores the Squares currently being explored in the maze - that will be handled by its sub-classes. Why the mystery as to what type of "worklist" you'll use? You will find out eventually! The MazeSolver class should have a private member of type Maze, and should have the following methods:

`MazeSolver(Maze maze)`

Constructor that takes a Maze as a parameter (the maze to be solved) and stores it in a variable that extending classes can access.

Should perform the two initialization steps of creating an empty worklist (by calling the `makeEmpty` abstract method) and adding the maze's start location to it (using the `add()` abstract method).

`abstract void makeEmpty()`

Create an empty worklist, i.e. make a new worklist object with no elements. Normally initializing the worklist would take place in the constructor, however the GUI uses a call to this method, so it must be kept separate.

`abstract boolean isEmpty()`

Return true if the worklist is empty

`abstract void add(Square s)`

Add the given square to the worklist

`abstract Square next()`

Return the "next" item from the worklist

`boolean isSolved()`

A concrete (non-abstract) method that the application program can use to see if the maze has been solved. That is, has it determined the path to the exit or that there is no path.
This method will return true if either:

1. A path from the start to the exit has been found; OR
2. You determine there is no such path (i.e. if the worklist is empty)

Adding variables to store these values (solved, solvable, etc.) may help.

`void step()`

Perform one iteration of the algorithm shown previously (i.e. steps 1 through 5 in the pseudo-code). Note that this is not an abstract method, that is, you should fully implement this method in the MazeSolver class by calling the abstract methods listed above (which works, as sub-classes *must* override them to be instantiated).

In order to keep track of which squares have previously been added to the worklist, you will "mark" each square (i.e. the square's `status`) that you place in the worklist. Then, before you add a square to the worklist, you should first check that it is not marked (and if it is, refrain from adding it). Recall that the various `status` constants are `static` and accessible from the Square class with e.g. `Square.WORKING`.

Reminder: here is a video showing the maze solver at work that may help you understand the algorithm.

`String getPath()`

Return whether this maze is not yet solved, solved, or unsolvable.

`void solve()`

Repeatedly call `step` while the maze is not solved

The MazeSolverStack class

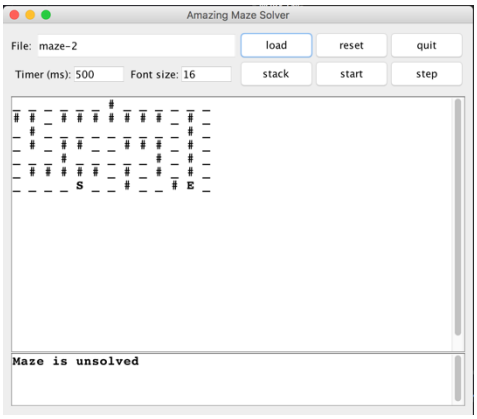
Next, create a concrete class **MazeSolverStack.java** that `extends` the MazeSolver class and implements (overrides) the super-class' abstract methods. MazeSolverStack should contain, as an instance variable, a "worklist" of the appropriate type (i.e. MazeSolverStack should have an instance variable of type MyStack). All you have to do to implement the abstract methods is perform the appropriate operations on the stack. For example, the MazeSolverStack's overridden `add` method would look like this:

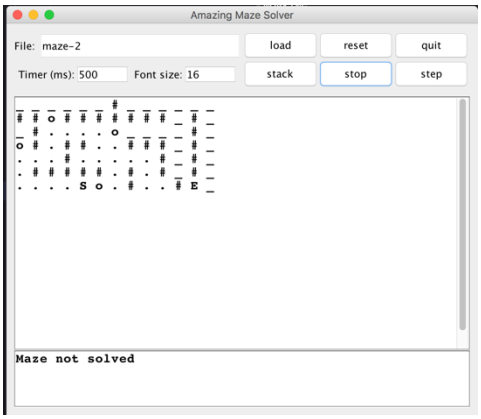
```
@Override
public void add(Square s)
{
    stack.push(s);  //stack is an instance (object) of your MyStack class
}
```

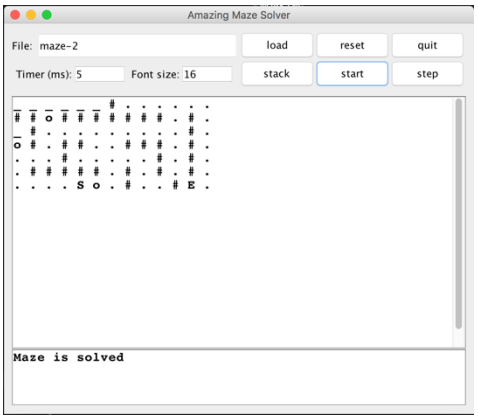
Implementing MazeSolverStack should not take you very long. Don't forget to include a call to `super(maze)` as the first line of your constructor!

Part 4 - Watch it work!

If everything is working you should be able to run the MazeApp program and see a GUI that animates the process of finding a solution to the maze. You should not need to modify anything in this file.







The load and quit buttons operate as you might expect. The reset button will call the Maze's `reset` method and then create a new MazeSolver. The step button performs a single step of the MazeSolver and start will animate things taking one step per timer delay interval. Note that you must to hit enter after changing values in the text boxes (e.g. timer) for the GUI to recognize it.

Maze's `toString` method is used to display the maze in the main window, and the `getPath` method from MazeSolver is used for the bottom window.

(Advanced) Better, faster maze solving

Once you have the maze solving algorithm working, try these challenges:

- Find the *shortest* path (e.g. *maze-2* has two possible solutions that are different lengths)
- Outputting the path itself
    - In other words, output all of the locations, beginning at the start, that lead to the exit square