CSCI 1933 Project 4

Minefield

Due Date: December 11th

1 Instructions

Please read and understand these expectations thoroughly. Failure to follow these instructions could negatively impact your grade. Rules detailed in the course syllabus also apply but will not necessarily be repeated here.

- **Due:** The project is due on **December 11th** by **11:55 PM**.
- **Identification:** Place you and your partner's x500 in a comment near the top of all files you submit. Failure to do so may result in a penalty.
- **Partners:** You may work alone or with *one* partner. Failure to tell us who is your partner is indistinguishable from cheating and you will both receive a zero. Ensure all code shared with your partner is private.
- **Code:** You must use the *EXACT* class and method signatures we ask for. This is because we may use a program to evaluate your code. Code that doesn't compile will receive a significant penalty. Code should be compatible with Java 17, which is installed on the CSE Labs computers. Credit ALL outside references used in completing this project both in the README and within the code that utilizes the referenced material.
- **Questions:** Questions related to the project can be discussed on Discord in abstract. This relates to programming in Java, understanding the writeup, and topics covered in lecture and labs. **Do not post any code or solutions on the forum**. Do not e-mail the TAs your questions when they can be asked on Discord.
- **Grading:** Grading will be done by the TAs, so please address grading problems to them *privately* through the ticket system on Discord or a private Piazza post.
- **README:** Make sure to include a README.txt in your submission that contains the following information:
  - Group member's names and x500s
  - Contributions of each partner (if applicable)
  - Any assumptions
  - Additional features that your project had (if applicable)
  - Any known bugs or defects in the program
  - Credit ALL outside references used in completing this project both in the README and within the code that utilizes the referenced material.
  - Academic Integrity statement

IMPORTANT: You are NOT permitted to use ANY built-in libraries, classes, etc... besides java.awt.Color, java.util.Random, and java.util.Scanner. Double check that you have NO import statements in your code, except for those explicitly permitted.

Code Style

Part of your grade will be decided based on the "code style" demonstrated by your programming. In general, all projects will involve a style component. This should not be intimidating, but it is fundamentally important. The following items represent "good" coding style:

- Use effective comments to document what important variables, functions, and sections of the code are for. In general, the TA should be able to understand your logic through the comments left in the code.

  Try to leave comments as you program, rather than adding them all in at the end. Comments should not feel like arbitrary busy work - they should be written assuming the reader is fluent in Java, yet has no idea how your program works or why you chose certain solutions.
- Use effective and standard indentation.
- Use descriptive names for variables. Use standard Java style for your names: `ClassName`, `functionName`, `variableName` for structures in your code, and `ClassName.java` for the file names.

Try to avoid the following stylistic problems:

- Missing or highly redundant, useless comments.
  `int a = 5; //Set a to be 5` is not helpful.
- Disorganized and messy files. Poor indentation of braces (`{` and `}`).
- Incoherent variable names. Names such as `m` and `numberOfIndicesToCount` are not useful. The former is too short to be descriptive, while the latter is much too descriptive and redundant.
- Slow functions. While some algorithms are more efficient than others, functions that are aggressively inefficient could be penalized even if they are otherwise correct. In general, functions ought to terminate in under 5 seconds for any reasonable input.

The programming exercises detailed in the following pages will both be evaluated for code style. This will not be strict – for example, one bad indent or one subjective variable name are hardly a problem. However, if your code seems careless or confusing, or if no significant effort was made to document the code, then points will be deducted.

If you are confused about the style guide, please talk with a TA.

Project Structure

Your project submission must adhere to the following rules. Failure to do so will impact your grade.

1. Your submission should be one ZIP file named

   `<partner1 x500>_<partner2 x500>_Project4.zip`

2. The ZIP file should contain a single directory (folder) named

   `<partner1 x500>_<partner2 x500>_Project4`

3. This directory should contain **only** these 2 files:
   - Minefield.java
   - main.java
   - README.txt

For example, the following would be a valid project structure:

- shino012_hoang159_Project4.zip
  - shino012_hoang159_Project4
    - Minefield.java
    - main.java
    - README.txt

If you are working alone, just include your single x500 in the naming of the ZIP file and directory. If you have any questions about this structure, ask a TA.

2 Introduction

Welcome to the fourth project in this course, for this project, you will utilize stacks and queues to create our version of *Minesweeper*. If you are unfamiliar with the game, it is recommended that you take a small amount of time to familiarize yourself with it. Here is the official *Minesweeper* website: Click me! *NOTE* this website should be used as a **BRIEF** overview of minesweeper. We will be implementing a slightly different version and so some rules may vary.

In Minefield, users are attempting to reveal every square of a given field while avoiding randomly placed mines. Each square has a value (1-8), that tells the user how many mines surround it in a 3x3 tile. A flag allows users to skip over squares they believe to be mines. Hit a mine, and the game is over. If you reveal all squares and place flags on all the mines, you win. For our version, you will be playing on a field, which is a two-dimensional array that will contain all the relevant information for the game.

There are two main features that require special attention to implement in Minefield: revealZeroes() and revealStartingArea(). These methods allow the user to reveal all surrounding zeros on the field and the first mine(s) found given two starting coordinates, respectively. In *Minefield*, when the user chooses their starting coordinates, the game should reveal a large enough area to give enough information to start solving the field. Additionally, if a user chooses a square with a '0' value in it, meaning there are no mines, the field should also reveal all surrounding zeroes. **These two problems are suitable for utilizing stack and queue data structures and are the focus of this project.**

2.1 Background and Getting Started

More detail about each method will be found in their respective sections. You will also be given a main class that has a game loop, similar to the one you made for the previous project. Do not worry if you have not played *Minesweeper* before. It is recommended to play a few games of it to familiarize yourself with it but the description below explains the premise of the game.

Minesweeper relies on 3x3 tiles. In each of the cells within this 3x3 tile, a value is given based on the number of mines surrounding it.

Below is an example:

1.
0 0 0 0 0
0 1 1 1 0
0 1 * 1 0
0 1 1 1 0
0 0 0 0 0

2.
0 0 1 1 1
0 1 2 * 1
0 1 * 2 1
0 1 1 1 0
0 0 0 0 0

(1) Take note of how when no mines are present in a 3x3 area, a '0' is placed as the value.

(2) Here, and additional mine is placed, notice the difference in how now some cells have a '2'.

2.2 Files Given

Along with the project write-up, you will be given the following files:

- **Minefield.java** The java class for your minefield implementation.
- **Main.java** the java class used to run *Minefield*

- **Cell.java** A helper class for Minefield
- **NGen.java** - Both the queue and stack structure will utilize this generic node class
- **Q1.java** - An interface for a generic queue
- **Q1Gen.java** - An implementation of a generic queue
- **StackGen.java** - An interface for a generic stack
- **Stack1Gen.java** - An implementation of a basic stack

You will not change any of the files given except for the class called `Minefield`. All other files are provided with everything you should need.

2.3 Stack and Queue data structures

Both the stack and queue data structures provided have basic functionality. Take a look at both classes to ensure you know how to instantiate and use them properly. They should be similar to the examples you have seen in lecture.

3 Cell

You do not need to change the Cell class, but this section describes how a cell works. The cell class is how we represent each of the squares in the field. The Cell class has the following attributes:

`boolean revealed` – True if this cell has been revealed to the user, false otherwise.

`String status` – Contains information regarding the Cell's status.

- `"-"` = Blank (defualt value)
- `"F"` = Flagged by user
- `"M"` = Mine
- `"0...9"` = Number of mines in surrounding 3x3 tile.

Each of these attributes have setter and getter functions respectively.

4 Main

To play Minefield, the game will require a *Main* class. The exact implementation is up to you, but the *Main* class should have a game loop and the ability to take in user input.

Furthermore, there is an order in which this game should flow. Specifically, three functions need to be called in the following order: The order in which the game should begin is as follows:

1. When a game is started, the main class should instantiate a new **Minefield** based on user input, **as well as whether the game should be played in debug mode**.

2. After this, the game now has an empty field with which it should place mines on. This is done through the **createMines()** method.
3. Our field now has mines placed on it but does not have any useful information for the user. To help the user, we can change the statuses of each cell to reflect how many mines surround it. This is done through the **evaluateField()** method.
4. We are almost ready to begin playing, except our field does not have any of its cells revealed to the user yet. To do this, we must call the **revealStartingArea()** method.

Finally, the user should be able to do basic things such as taking a guess at a cell. This means the user can either attempt to reveal a cell they believe is safe or place a flag where they believe there is a mine.

4.1 Additional Requirements

- The dimensions and the number of mines are based on three levels that the user can choose from:
  - **Easy**: Rows: 5 Columns: 5 Mines: 5 Flags: 5
  - **Medium**: Rows: 9 Columns: 9 Mines: 12 Flags: 12
  - **Hard**: Rows: 20 Columns: 20 Mines: 40 Flags: 40

- Debug mode can either be on or off, if on, the entire board should be displayed with all cells revealed each turn.

5 Minefield

The `Minefield` class is where the majority of the work will take place. **The methods outlined in the class will all need to be implemented, but you can add any additional methods as you see fit**.

Each of the methods have short descriptions of what they should return or what their purpose is. Further information is given on specific methods in the following sections. All of these methods will need to be implemented, but the following are descriptions about the more challenging methods.

5.1 guess

When a user guesses a coordinate, the guess() method should do the following. It first should see if the guess is in-bounds—this may also be done in the Main class. Next, it should see if the user wishes to place a flag and, if so, whether or not there are enough flags remaining to place the flag. If the user did not place a flag, then the method should check to see if the user has hit a cell with a '0' status. If so, call the revealZeroes() method.

Finally, if the user hits a mine, end the game. Make sure to also set the revealed status of the cell

that the user guesses at the end of the method.

5.2 createMines

This method will place all of our mines on our field. Until all mines has been placed, the method should randomly generate coordinates to place a mine. Before placing a mine, the method should check to see if this coordinate has not been revealed and is not already a mine, and **is not equal to the starting coordinates**. Otherwise, a new coordinate pair should be generated.

5.3 revealZeroes

This method is used to reveal all surrounding zeroes when a user clicks on a square containing a status of "0". At the bottom of the description you will see the desired output, notice how guessing one zero reveals all zeroes nearby. Here we are using an algorithm using a stack to accomplish this. Since a stack has a first-in-last-out structure, this will explore as far in one direction as possible for a '0' status. The pseudo-code is as follows:

- Initialize a stack with the start index {x, y}. This should be the x and y values passed into the method.
- Loop until the stack is empty:
  - Get the top element off the stack.
  - Set the corresponding cell's `revealed` attribute as `true`.
    - Push all valid neighbor's coordinates to the stack. A valid neighbor is one who is in-bounds, has not previously been revealed, and whose status is "0".
  - Neighbors here are the cells located to the left, right, above, and down from the current cell.

```
Enter a coordinate and if you wish to place a flag (Remaining: 5): [x] [y] [f (-1, else)]
0 3 1
   0  1  2  3  4
0  1  1  0  0  0
1  M  2  1  1  0
2  2  3  M  2  1
3  2  M  3  2  M
4  2  M  2  1  1

   0  1  2  3  4
0  -  1  0  0  0
1  M  2  -  -  0
2  -  3  -  -  -
3  -  -  -  -  -
4  -  -  -  -  -
```

NOTE: This pseudo-code should appear familiar. This is very similar to a Depth-First Search algorithm, this also shows why a stack is best used here.

5.4 revealStartingArea

This method will help us reveal enough information for the user to get started at the beginning of each game. The image at the bottom shows the desired output. The goal of the algorithm is to loop until a mine is found. This will reveal enough of the field to help the user. We are instead using a queue here. The pseudo-code is as follows:

- Initialize a queue with the start index {x, y}. This should be the same x and y that were passed into the method.
- Loop until the queue is empty:
  - Dequeue the front cell of the queue.
  - Set the corresponding cell's `revealed` attribute as `true`.
  - If the current cell is the finish point, meaning that the current cell is a mine, then break from the loop. The algorithm is complete.
  - Enqueue all reachable neighbors that are in-bounds and have not already been visited, regardless of their statuses.

```
   0  1  2  3  4
0  -  1  -  -  -
1  M  2  -  -  -
2  -  3  -  -  -
3  -  -  -  -  -
4  -  -  -  -  -
```

NOTE: This pseudo-code should appear familiar. This is very similar to a Breadth-First Search algorithm, this also shows why a queue may be best here.

6 ANSI Color Codes

You have gotten used to printing out information and reading text from the terminal. Until now, this has most likely always been in black and white. When playing Minefield, it would be useful to the player if each number was a different color. We fortunately have a simple way to accomplish this: ANSI Color Codes. These may look familiar to the previous project's trick for displaying ASCII characters. The basic colors have been provided to you, but feel free to add more.

6.1 Implementation of ANSI Color Codes

In order to turn the following print statement:

```
- System.out.println("Change my color!");
```

To print out in red, we add the color code for red "u001b[31m".

```
- static final String ANSI_RED = "u001b[31m";
```

```
- static final String ANSI_GREY_BG = "u001b[0m";
```

```
- System.out.println(ANSI_RED+"Change my color!"+ANSI_GREY_BG);
```

The purpose of ending with ANSIGREYBG, is to act as a reset color. This way any text that is printed after will be reset back to a default color.

Feel free to make any number a specific color, just make sure that each number is unique.

Below is an example of the terminal utilizing these ANSI color codes. **This also shows an example of what the final game should look like, with the first field being what debug mode prints.**