CSE 8B Spring 2022

Assignment 6

Exception handling and Text I/O

Due Date: Monday, May 16, 11:59 PM

Welcome back! Be sure to start this assignment too as EARLY as possible! You got this!

Learning goals:

- Understanding exception handling in practice
- Using text file I/O to read and write to files

NOTE: This programming assignment must be done individually. Paired programming is NOT allowed for this assignment.

Coding Style (10 points)

For this programming assignment, we will be enforcing the CSE 8B Coding Style Guidelines. These guidelines can also be found on Canvas. Please ensure to have COMPLETE file headers, class headers, and method headers, to use descriptive variable names and proper indentation, and to avoid using magic numbers.

Part 0: Getting started (0 points)

1. Make sure there is no problem with your Java coding environment. If there is any, then review Assignment 1, or come to the office/lab hours before you start Assignment 6.
2. Review lectures 11 and 12.
3. Once again, in this PA, you will start with some code that we have written for you.
4. If you are using UCSD Linux Cloud for the assignment, download the starter code from Piazza → Resources → Homework by opening a web browser on the Remote Desktop. You will be downloading the **Assignment6.zip**. You can open Firefox by going to Applications → Firefox. Then, unzip Assignment6.zip. Here is one way to unzip Assignment6.zip on the *ieng6 Server Remote Desktop*:

a. Create an empty directory named PA6 anywhere in your remote desktop.
b. Right click Assignment6.zip.
c. Select “Open with Archive Manager”
d. Double click the Assignment6 folder in the Assignment6.zip window that pops up
e. Click once somewhere in the window and select all of the files with **<CTRL>+A**
f. Drag the files onto the PA6 directory you created above.

5. If you are working on your local machine, then you can download the starter code from a local web browser on your machine.
   Next, please make the **readOnlyFile** inside Assignment6 to be read-only. Here’s what you should do:
   a. If you’re using Linux (macOS, Ubuntu, etc), go to your PA6 directory. Run the following command.

```
$ chmod 444 readOnlyFile
```

   b. If you’re using Windows, go to the your PA6 folder. Right click the **readOnlyFile**. Click **properties**. Then check the **Read-only** box.

Part 1: Overview

Conway’s Game of Life

In this assignment, you will implement a Game Of Life board game simulator. If you don’t know how the game works yet, don’t worry, we’ll walk you through the rules!

Before you start programming, please take some time to review the starter code and to read the instructions below **CAREFULLY**. Some methods are already implemented for you, but you will still need to supply those methods with a method header for coding style points. You should fully understand the purpose of each variable and the usage for each method before you implement anything.

NOTE 1: you should NOT change any data field or method signature in the starter code. As such, do NOT add any additional parameters to methods, and do NOT import any Java packages. Feel free to add any helper methods if desired.

NOTE 2: You must implement and comment on everything with a "**TODO".** Do NOT forget to adhere to the CSE 8B style guidelines.

NOTE 3: You can assume that all inputs will be valid. For example, all `int`s and `double`s will be non-negative.

Be sure to compile your code often, so that you can catch compile errors early on!

Description

Game of Life is a board game that would be simulated as a 2D grid of 0's and 1's in our program. Each cell in the grid can be alive or dead. The game starts with a grid initialized with some number of dead cells and some number of alive cells. In each step of the game, you need to generate the next generation of cells based on the following rules:

- If a cell is alive, and has fewer than two alive neighbor cells, it dies from underpopulation.
- If a cell is alive, and has two or three alive neighbor cells, it lives on to the next generation.
- If a cell is alive, and has more than three alive neighbors cells, it dies from overpopulation.
- If a cell is dead and has exactly three alive neighbor cells, it comes alive from reproduction.

Important note: Here, the neighbor of a cell includes its adjacent cells as well as diagonal ones, so for each cell, a total of 8 neighbors are there.

Part 2: readBoard (20 points)

This method will read the board's initial state from a file and parse the board to a 2D integer array, which is used to initialize the `private static int[][] board` variable. The method takes an input argument `fileToRead` which is the name of the file containing the board whose initial state we want to read. Apply what you've learned in lectures to read the input file (Hint: use the `Scanner` class).

The file contains the following information of the board:

1. The first line of the file will always be a single number, which represents the size of the board. We assume that the number of rows and columns is the same as specified by the size, i.e., the board is a square.
2. The rest of the file is the state of the board - containings 0's (for dead cells) and 1's (for alive cells)
3. Between every row and every column, there will be a single space.
4. No blank lines in the input file. No extra spaces between rows and columns. No extra spaces before and after each line.

Read the input file `fileToRead` and parse it to a 2D integer array which would be used to initialize the `private static int[][] board` variable. While reading the input file, you would also need to handle `FileNotFoundException` in the case when the file does not exist. Print a message while handling the exception using the following method and throw the exception.

```
exception.getMessage()
```

You can assume that as long as there is no exception, the file is valid for you to initialize the 2D array. Remember to close the `Scanner` object that you created to read the input file after the reading is finished.

Hint: try -with-resources statements might/might not be helpful, depending on whether you want to close resources explicitly.

Part 3: generateNextState (20 points)

In this method, you would be implementing the simulation of generating the next state of the board as per Game Of Life rules. To do so, first, make a copy of the 2D integer board array into a new 2D integer array. Once the copy of the board is created, you can use this to store the next state of the board.
To determine the next state, iterate over the board array using nested for-loops and compute the state of each cell by looking at its neighbors and counting the number of alive neighbor

cells. Based on this count, the cell is determined to be dead or alive in the next state using the rules of the game, which are reiterated below for reference:

- If a cell is alive, and has fewer than two alive neighbor cells, it dies from underpopulation.
- If a cell is alive, and has two or three alive neighbor cells, it lives on to the next generation.
- If a cell is alive, and has more than three alive neighbors cells, it dies from overpopulation.
- If a cell is dead and has exactly three alive neighbor cells, it comes alive from reproduction.

Important note: Here, the neighbor of a cell includes its adjacent cells as well as diagonal ones, so for each cell, a total of 8 neighbors are there.

Once the next state of the board is determined and stored in the copy created, copy back the state of the board to the original 2D board array (i.e., `private static int[][] board` variable) to update the state of the game.

Part 4: writeBoard (20 points)

This method will write the board given by the `private static int[][] board` variable to a file specified by the `fileToWrite` argument. Apply what you've learned in lectures to write to the output file (Hint: use the `PrintWriter` class).
The output file should follow the following format:

1. You will not output the size of the board. Your first line should be the board itself.
2. Between every row and every column, there will be a single space.
3. No blank lines in the output file. No extra spaces between rows and columns. No extra spaces before and after each line.
4. You will lose points if your output file's format does not meet the requirements.

You would also need to handle `FileNotFoundException`. For `PrintWriter`, a new file will be created if the file specified by `fileToWrite` does not exist. However, you still need to handle

the `FileNotFoundException` to avoid other cases. For example, throw this exception if the output file is a read-only file. Print a message while handling the exception using the following method and then throw the exception.

```
exception.getMessage()
```

You can assume that as long as there is no exception, the board state is valid for you to write to the file. Close the `PrintWriter` object that you created for writing to the output file after the writing is finished.

Part 5: Compile, Run and UnitTest Your Code (10 points)

Just like in previous assignments, **in this part of the assignment, you need to implement your own test cases in the method called `unitTests` in the `Assignment6.java` class.**

In the starter code, this time, no test cases have been implemented as a sample to slightly make the process of writing unit tests more thoughtful. Recall, the general approach is to come up with different inputs and manually give the expected output, then call the method with that input and compare the result with expected output.

To compare arrays by the equality of contents, you must use **`Arrays.equals()`**. You can use the `getBoard()` getter method and the `setBoard()` setter methods to access and modify the value of the `private static int[][] board` member variable inside `unitTests()` method if needed. Remember to reset the `int[][] board` member variable whenever required between multiple unit tests.

You are encouraged to create as many test cases as you think to be necessary to cover all the edge cases. The `unitTests` method should `return true` only when all the test cases are passed. Otherwise, it should `return false`. **To get full credit for this section,** you need to create **at least two test cases** for each of the methods - `readBoard()`, `generateNextState()` and `writeBoard()`. In total, you will have **at least six test cases**.

If a test is not passing, try temporarily printing the result of your method(s) and comparing them to the expected output.

You can compile and run your unit tests from `main()` using the following commands: (Make sure you are in the correct directory, else navigate to the starter code using `cd`).

```
> javac Assignment6.java
> java Assignment6
```

Part 6: Implement main() method (20 points)

In this part, you will be completing `Assignment6.java`'s main method to simulate the game play.

Once your unit tests pass and you are confident that the above methods are working, it's time for you to put everything together.

Similarly to the previous assignments, all of the command line functionality will be implemented in the `main` method. We have defined several `String` constants for you (at the top of `Assignment6.java`) to use for printing and for throwing exceptions. Assume that every `String` constant that is printed needs to be with the cursor moved to the next line. (Think about print vs println).

Do not change any code above the comment "TODO". Below that comment, your task is to implement a text file I/O to read from an input file and compute the game state after generating the next state for a predefined number of times, and writing the final state to an output file.
For this, you will be using the methods you have implemented earlier, i.e., `readBoard()`, `generateNextState()`, and `writeBoard()`.

First, your program should print the following prompt:

```
Which file do you want to read?
```

and wait for the user to enter the input file name. Read the file and parse the board from the file using the `readBoard()` method. You need to handle the exception from `readBoard()`, print **FILE_NOT_READ_EXCEPTION** and stop the program if there's an exception.

Next, prompt the user for the number of times the game state has to be computed using the following prompt:

```
How many states do you want the game to go through?
```

and wait for the user to enter the number ‘n’ as an integer.
Then compute the final state of the game starting from the initial state and generating the next state of the board for ‘n’ number of times using the `generateNextState()` method.
Finally, prompt the user to input the name of the file the final state of the board should be written to:

```
Which file do you want to write?
```

and wait for the user to enter the output file name. Then write your board to the file. You need to handle the exception from `writeBoard()`, print **FILE_NOT_WRITTEN_EXCEPTION** and stop the program if there’s an exception.

Finally, close any I/O if there’s still some open ones.

We have provided some input and output files so that you can test your program for different cases. `input1` and `input2` are valid input files. The `readOnlyFile` is for you to test if your `PrintWriter` exception handling is correct.

Example: you should be able to reproduce a similar output with your program:

```
Which file do you want to read?
input1
How many states do you want the game to go through?
3
Which file do you want to write to?
readOnlyFile
readOnlyFile (Permission denied)
File is not written because of exception
[cs8bsp22ta1@ieng6-253]:PA6:51$ java Assignment6
All unit tests passed.

Which file do you want to read?
input1
How many states do you want the game to go through?
4
Which file do you want to write to?
output1
[cs8bsp22ta1@ieng6-253]:PA6:51$ java Assignment6
All unit tests passed.

Which file do you want to read?
randomFile
randomFile (No such file or directory)
File is not read because of exception
```

Part 7: Optional Challenge (0 points)

As an optional, ungraded extension of this assignment, you can extend the Game of Life simulation to make it slightly more interesting. This section is not mandatory for points and will not be graded. If you wish to skip this, please proceed to Submission.

We could extend the board to wrap around itself, just to make this more fun. For all the cells at the edges and corners of the board, while looking at their neighbors, instead of considering only the neighboring indices that are positive and within the range of indices in the board, we could wrap the indices using a little math. Whenever the index of the neighbor goes out of range, we can add the board size to its index value with a % (boardSize) to wrap the board.

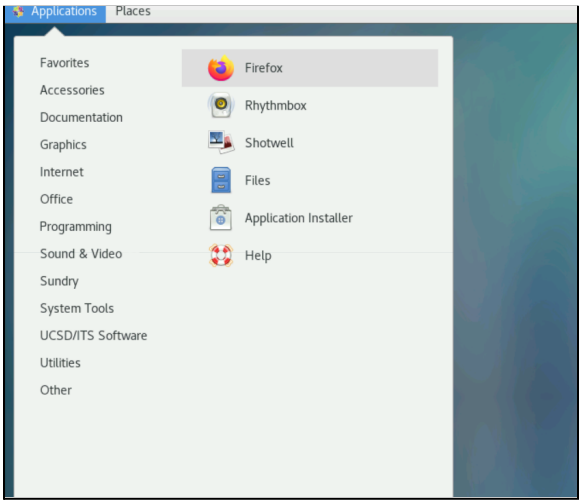
For example, while calculating the neighbors of (0,0) - instead of considering only (0,1), (1,0), and (1,1) we could also consider all the other five neighbors wrapping them around the board as follows:
(-1,0) could now be (-1 + boardSize, 0)
(0,-1) could now be (0, -1 + boardSize) and so on.
Similarly, for the cells at the right edges and corners, you could wrap them around using the % operator. When the index is greater than the board size, you can wrap it using `(index + boardSize) % boardSize.` This formula could be generalized for both cases as

`(index + boardSize) % boardSize`. You are free to implement any other ideas for extending the game and making it more interesting. This is just one of them! Remember, this section will not be graded but feel free to discuss your approaches with us if required.

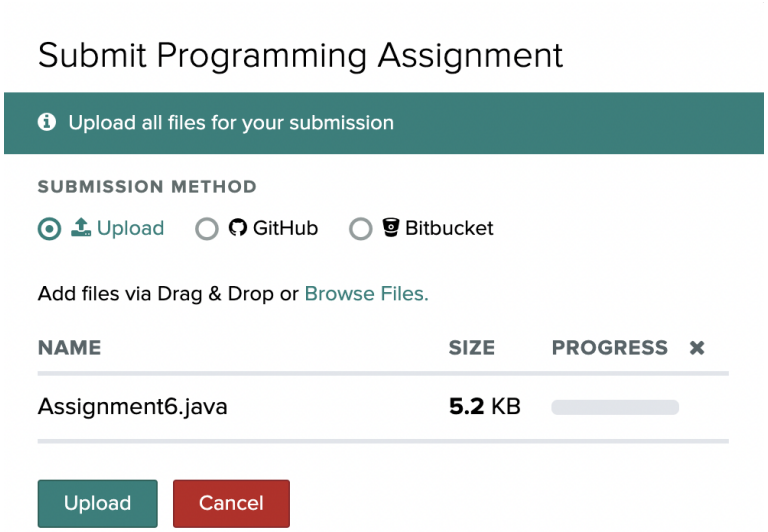
Submission

You're almost there! Please follow the instructions below carefully and use the **exact submission format**. Because we will use scripts to grade, **you may receive a zero** if you do not follow the same submission format.

1. Open a web browser on the Remote Desktop. You can open Firefox by going to Applications → Firefox.



2. Open Gradescope in Firefox and login. Then, select this course → PA6.
3. Click the DRAG & DROP section and directly select the required file - **Assignment6.java**. Drag & drop is fine. Please make sure you don't submit a zip, just the file in one Gradescope submission.
4. You can resubmit unlimited times before the due date. Your score will depend on your final (most recent) submission, even if your former submissions have higher scores.
5. Your submission should look like the below screenshot. If you have any questions, feel free to post on Piazza!


Submit Programming Assignment
Upload all files for your submission
SUBMISSION METHOD
Upload
GitHub
Bitbucket
Add files via Drag & Drop or Browse Files.
NAME
SIZE
PROGRESS
Assignment6.java
5.2 KB
Upload
Cancel