Assignment 1

DUE DATE: THURSDAY MAY 27, 2021 AT 11:59 PM (CDT)

Notes:

- Submit ONE .java file per question. Each .java file must contain ALL of the source code for a question. See the Programming Standards document for how to store several classes in the same file.
- Each filename must have the format `<your last name><your first name>A1Q2.java` (e.g. `SmithJohnA1Q1.java`). Please use your name as shown in UM Learn.
- Your program must compile, run, and end normally (not crash or get stuck in an infinite loop) to receive any marks. See the Assignment Information file for some tips on how to make sure your code runs for the markers.
- Assignments must follow the Programming Standards posted in UM Learn.
- Assignment submissions are only accepted via UM Learn. Submissions by email will not be accepted.
- You may submit your assignment multiple times, but only the most recent version will be marked.
- Assignments become late immediately after the posted due date and time. Late assignments will be accepted up to 49 hours after that time, at a penalty of 2% per hour or portion thereof. After 49 hours, an assignment is worth 0 marks and will no longer be accepted.
- The time of the last submission controls the late penalty for the entire assignment.
- These assignments are your chance to learn the material for the tests. *Code your assignments independently*. We use software to compare all submitted assignments to each other, and pursue academic dishonestly vigorously.

Question 1: A Library [18 marks]

This question can be done in Week 1 of the course.

Libraries contain a catalogue of books, that can be searched by author or title. Books can be loaned and returned. In this question you will use commands given in an input file to build a library, and to search for, and loan books.

Your solution should include three classes:

- An **application (main) class**, that will read input from a file (see below) and output results.
- A **Book class**, that will store information on one book. You should store (1) the book title, (2) author first name/initials, (3) author last name, and (4) a boolean flag that indicates whether the book is out on loan.
- A **Library class**, that contains a list of books (stored in an **array**). This class should have, at minimum, the following methods:

- A **constructor**, which creates an empty Library.
- An **addBook** method, that takes the information for a book and adds a new Book to the library. Books must be stored in order by an author's last name, then by first name, and then by title (i.e. perform an ordered insert, to make sure that the list is always ordered). It is possible for libraries to have multiple copies of the same book, so you may end up with multiple Book objects with the same author and title.
- A **listByAuthor** method, that will return a String containing a list of books written by that author. Use '\n' (newline) in the String so that it can be printed and there will be one book per line, as seen in the sample output below.
- A **listByTitle** method, that will return a String containing a list of books with that title.
- A **loanBook** method, that will take an author (first and last names) and title, and return a boolean indicating whether that Book is available for loan. If the Book is available, this method should update the flag in the Book object, to indicate that the book is now out on loan, and return true. If the book is not found in the Library, or is already out on loan, return false.
- A **returnBook** method, that will take an author and title, and return a boolean indicating whether the book was successfully returned. This method should update the flag in the Book object, to indicate that the book has been returned and is again available for loan.

Use the object-oriented programming techniques that you learned in COMP 1020. All instance variables should be private, and appropriate methods should be written any time access to private data is required.

Input format:

The name of an input file will be entered by the user. That file should be read one line at a time, performing operations and printing output, as you go. As commands are read from file, call the appropriate Library methods. The user (the application class) knows they are searching for/borrowing/returning books but should not need to know any details on how the books are stored.

Valid commands in the input file are:

- ADD – A line beginning with ADD will be followed by the book details (author last name, author first name, title). Commas separate the names and title (author last name, a comma, the author first name, a comma, and then the title). It is possible that there may be additional commas in a line of input, if there are commas in the title of the book. The ADD command adds a book to the Library, but does not produce any output.
- SEARCHA – A line beginning with SEARCHA should result in a list of books with a given author last name. If multiple authors have the same last name, the list should include all books from all of those authors.
- SEARCHT – A line beginning with SEARCHT should result in a list of books with the given title.

- GETBOOK – A line beginning with GETBOOK will attempt to borrow a book from the library. GETBOOK will be followed by the author last name, a comma, the author first name, a comma, and then the title. If a book is loaned, output reflecting that will be displayed.
- RETURNBOOK – A line beginning with RETURNBOOK will return a book to the library. RETURNBOOK will be followed by the author last name, first name and title, in the same format as for GETBOOK.

Sample input file:

```
ADD Atwood, Margaret, The Handmaid's Tale
ADD Montgomery, L.M., Anne of Green Gables
ADD Martel, Yann, Life of Pi
ADD Atwood, Margaret, Alias Grace
ADD Ondaatje, Michael, The English Patient
ADD Toews, M., A Complicated Kindness
SEARCHA Atwood
GETBOOK Martel, Yann, Life of Pi
ADD Shields, Carol, The Stone Diaries
ADD Mistry, Rohinton, A Fine Balance
ADD Fitzgerald, ,The Great Gatsby
GETBOOK Shields, Carol, The Stone Diaries
ADD Margaret, Atwood, Alias Grace
SEARCHT Alias Grace
RETURNBOOK Martel, Yann, Life of Pi
SEARCHA Fitzgerald
```

Sample output:

```
Please enter the input file name (.txt files only):
input.txt

Processing file input.txt...

Books by Atwood:
Atwood, Margaret, Alias Grace
Atwood, Margaret, The Handmaid's Tale

Book loaned:
Martel, Yann, Life of Pi

Book loaned:
Shields, Carol, The Stone Diaries

Books named Alias Grace:
Atwood, Margaret, Alias Grace
Margaret, Atwood, Alias Grace

Book returned:
Martel, Yann, Life of Pi
```

```
Books by Fitzgerald:
Fitzgerald, unknown, The Great Gatsby

Program terminated normally.
```

Notes:

- The markers will use additional input to test your program. You should create your own test data to make sure that your program behaves as expected. Your program should not crash.
- If a SEARCHA, SEARCHT, GETBOOK, or RETURNBOOK command has an incorrect number of parameters, your program should output an error message and move to the next command, without modifying the Library.
- If a parameter for the ADD command is blank, your program should list it as “unknown” in the library (see The Great Gatsby in the above output).
- When reading book details from the ADD command, don’t expect a specific format for spaces around commas. You should split the input into parameters based on the location of commas, and then trim off any extra whitespace that may have been located before or after a comma. If you have a string named s, s.trim() will remove any leading or trailing whitespace.
- Your program is not expected to detect errors such as an incorrectly spelled name or title, or mixed up first & last names (see Margaret, Atwood in the above output).

Question 2: A Recursive Sudoku Solver [20 marks code + 4 marks questions]

This question requires material from Week 2.

Sudoku is a puzzle game, played on a grid (https://en.wikipedia.org/wiki/Sudoku). The Sudoku game is played on a 9x9 grid, where each cell contains a number from 1 through 9. To solve a puzzle, you must fill in the empty cells so that the numbers 1 through 9 appear once in every row, column, and 3x3 block.

2		7			3		8	
	5			9				6
								9
	4		2					
		8	5		1	6		
					4		3	
4								
5				7			2	
	3		9			4		5

For a human, solving a Sudoku puzzle requires careful logical reasoning. For a computer, even the most difficult Sudoku puzzles can easily be solved using a recursive backtracking approach.

An algorithm used to solve Sudoku puzzles is:

```
while there are empty cells
        choose an empty cell and place a number in it
        if a conflict occurs (i.e. if there is a duplicate in any row, column, or block)
                then remove the number, backtracking
        if there is no conflict
                then attempt to fill the rest of the grid
        if you are unable to fill the rest of the grid (i.e. a future dead-end is reached)
                then remove the number, backtracking
```

In this question you will write a program that accepts an initial board (from keyboard input) and attempts to solve the game. If a solution exists, your program will output the solution. (If multiple solutions are possible, output any one of the solutions.) If a solution does not exist, your program will inform the user that their initial board can't be solved.

Your solution should include two classes:

- The application (main) class, which will ask the user to enter a board, create a Sudoku object, call a solve() method on that object, and output the result (either the solution, or a message that the provided board is impossible to solve).
- A Sudoku class, which will store the game board, and contain a **recursive** solve() method. You should also write helper methods as needed.
    - The constructor for this class should be passed the string input (see below), and build a board from that input.
    - The solve() method should modify the board, so that it contains a valid solution (every cell filled with a number).
    - Hint: Store the size of the game board as an instance variable in this class. Your program should work for any valid value for n. See the generalization note below.

Remember: As described in the Programming Standards, no Collections can be use in this course. Your game board must be built using arrays. Consider whether you want to use a 1D or 2D array to store the board, and whether you want to store ints, chars, or something else.

You should verify the input provided by the user. The given board must have a size of n^2 by n^2, where $n > 1$ (see the “generalization” note below). For example, when n=3, the board is 9 by 9, and contains nine 3x3 blocks. If the user-provided board doesn’t contain a valid number of entries, or contains a number that is invalid for the board size, ignore the user input and start with an empty 9 x 9 board.

4x4 and 9x9 boards should be solved quickly, a 16x16 board with only a few blanks should also be solved quickly, a 16x16 board that is half empty may take on the order of 1 hour to solve, and a 25x25 board with more than a few blanks will take a very long time to solve, depending on your system. Start with small boards, and once you think you have the solve method working, try a test with a larger board.

Your solver code is not required to exactly follow the algorithm on page 5. You might find a better way!

Input format:

The game board must be provided by a user, through System.in. The board should be read as a single string, with spaces separating cells, and '-' used to represent an empty cell. The markers will use this format to test your code.

Here are two 9x9 puzzles that you can use for testing:

Puzzle 1:

2 - 7 - - 3 - 8 - - 5 - - 9 - - - 6 - - - - - - - - 9 - 4 - 2 - - - - - - - 8 5 - 1 6 - - - - - - - 4 - 3 - 4 - - - - - - - - 5 - - -
7 - - 2 - - 3 - 9 - - 4 - 5

Puzzle 2:

- - - - - - 5 4 - - - 6 - - - - - 8 4 2 - 7 - - - - - - - 3 6 7 - - - 2 - - - 1 - 8 - - - 9 - - - 4 2 1 - - - - - - - 3 - 6 7 5 - -
- - - 9 - - - 9 2 - - - - - -

A generalization:

The Sudoku game consists of nine 3x3 blocks, in an overall 9x9 grid. However, you could also have similar puzzles with four 2x2 blocks in an overall 4x4 grid, or twenty-five 5x5 block in an overall 25x25 grid, or thirty-six 6x6 blocks in an overall 36x36 grid, etc. Valid possibilities have a size of n^2 by n^2, where $n > 1$, and are filled with the numbers 1 to n^2.

Your solver program should be flexible and accept any valid-sized grid. This flexibility will make testing much easier – start testing your program with a 4x4 grid. For example, a solved 4x4 grid is:

4	3	1	2
1	2	4	3
2	4	3	1
3	1	2	4

Sample execution 1 (input is valid):

```
Please enter a game board, row by row, with - to represent empty cells and
spaces between each cell:
4 - 1 - - 2 - - 2 - - 1 - 1 - -

The original board is:
4 - 1 -
- 2 - -
2 - - 1
- 1 - -

The solution is:
4 3 1 2
1 2 4 3
2 4 3 1
3 1 2 4
Program terminated normally.
```

Sample execution 2 (invalid input):

```
Please enter a game board, row by row, with - to represent empty cells and
spaces between each cell:

3 4 - 8 6


The original board is:

- - - - - - - - -

- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -

The solution is:
2 4 6 1 3 5 8 7 9
1 3 5 8 7 9 2 4 6
8 7 9 2 4 6 1 3 5
5 6 3 4 1 2 7 9 8
4 1 2 7 9 8 5 6 3
7 9 8 5 6 3 4 1 2
6 5 4 3 2 1 9 8 7
3 2 1 9 8 7 6 5 4
9 8 7 6 5 4 3 2 1
Program terminated normally.
```

At the bottom of your code file, in a comment, answer the following questions:

1) Why did you choose to store the grid the way that you did? (1D vs 2D array, chars vs ints)

2) As you were developing your program, were there any approaches that you considered that you rejected as inefficient (e.g. would make many unnecessary recursive calls)? If yes, explain. If no, explain how you know your initial approach is most efficient.

[Programming Standards are worth 8 marks]