COMP 1731 (Winter 2019)
Programming Techniques and Algorithms

Assignment 3

Due date: Saturday, April 6, 2019, by 10:00pm (submit to Moodle)

Submitting Your Work

When you are finished, submit the following three (3) files to Moodle:

- `Hotmail.java`
- `NQueensVerifier.java`
- `TestNQueensVerifier.java`

Commenting Your Code

All your code should be sufficiently well commented. In particular, the `NQueensVerifer` class should use Javadoc-style comments.

Hotmail

You have just been hired by the marketing department of a company that does all of its advertising by bombarding millions of helpless Internet users with unsolicited e-mail. As a proud member of the *Society for the Proliferation of Antagonizing Messages* (S.P.A.M.), you are thrilled with your job. Your first assignment is to help the company compile an even larger list of e-mail addresses. Other programmers have already written code that scans Web pages and extracts strings of characters. You will write a Java method that takes one such string as input and returns `true` if the string is a valid e-mail address and `false` otherwise. The method signature is:

```
public static boolean isValidEmail(String test)
```

Define an *eName* to be a sequence of one or more characters, each of which is either a lowercase letter, an uppercase letter, or a digit ($0 \ldots 9$). A valid e-mail address consists of one or more eNames separated by periods, followed by the `@` character, followed by *two or more* eNames separated by periods. (Note that an e-mail address cannot contain spaces, and for the purpose of this question, any other characters that are usually allowed in e-mail addresses, such as underscores, are excluded.)

Here are some example strings and the value your method should return for each. Note that this may not be a complete list, i.e., even if your program produces the correct response for each of these, there may be other cases you need to think of and test in order to ensure that your program is 100% correct:

- `person@hopper.mta.ca` ⟶ `true`
- `mE.2@yOU.3` ⟶ `true`
- `dil@bert.com.` ⟶ `false` (trailing period not allowed)
- `first.last@net` ⟶ `false` (must have more than one eName after `@`)
- `.me@gmail.gee` ⟶ `false` (leading period not allowed)
- `me..you@jotmail.org` ⟶ `false` (consecutive periods not allowed)
- `i.J.k@W.X.y.z` ⟶ `true`
- `john@DOE@gmail.com` ⟶ `false` (only one `@` character allowed)
- `surfing.safari.info` ⟶ `false` (must have at least one `@` character)
- `bill.@gates.tv` ⟶ `false` (substring before `@` not valid)
- `warren@.buffett.money` ⟶ `false` (substring after `@` not valid)
- `mark zuckerberg@facebook.com` ⟶ `false` (spaces not permitted)
- `bill$$$gates@microsoft.com` ⟶ `false` (symbol `'$'` not permitted)

For this problem, you should use stepwise refinement:

Write other helper methods (also `static`) that are called by `isValidEmail`. For example, you could include a method named `countAtSymbol` that returns the number of time the character `'@'` occurs in a string. Another good idea is a method named `isEName` that takes a `String` argument and returns `true` if the string is a valid eName and `false` otherwise. Then inside `isValidEmail` you could extract *sub*-strings that should be eNames (e.g., between consecutive periods, between the last period and the end of the string, etc., etc.) and pass them to `isEName`. (You can extract sub-strings yourself, or you can use the `substring` method in the `String` class.)

Put all your (`static`) methods in class `Hotmail`. Make sure you include a `main` method that makes plenty of calls to `isValidEmail` to verify its correctness.

The N Queens Problem

In the game of chess, which is played on an 8×8 board, a queen can attack another piece if that piece is in the same row, in the same column, or on the same diagonal as the queen. The well-known *Eight Queens* problem is the following: starting with an empty chess board, place 8 queens so that no two queens can attack each other.

There are various ways to approach the Eight Queens problem. Here is the simplest (although probably not the most efficient): generate every possible placement of 8 queens on a chess board, and for each placement, check whether or not it represents a valid solution. Clearly there are two main challenges here, but we will focus on the second one: *given an arrangement of queens on a chess board, check whether or not it represents a valid solution.* For this question, we will generalize from an 8×8 board to an $N \times N$ board.

`NQueensVerifier` Class

The `NQueensVerifier` class should have the following components:

- a class constant (`static final`) of type `char` named `QUEEN` that is assigned the literal character '`Q`'; you can use any access modifier you want (but think about why you might choose one over the others)
- another class constant of type `char` named `BLANK` that is assigned the literal character '`B`'; again, you can use any access modifier you want (but make it the same as the one you chose for the `QUEEN` constant)
- a (private) instance variable of type `int` called `size`; this represents the size of the board (i.e., it plays the same role as N in the description above)
- a (private) instance variable called `board` that is a 2D `char` array
- a constructor with two parameters, one of type `int` (say, `inSize`) and one of type `char[][]` (say, `inBoard`)
  - if `inSize` is not positive, throw an `IllegalArgumentException` with an appropriate internal message
  - pass `inBoard` to the method `checkBoardBasics` (described below); if this method returns false, throw an `IllegalArgumentException` with an appropriate internal message
  - assign `inSize` to `size` (you'll only reach this point if you didn't already throw an exception)
  - instantiate a 2D `char` array referenced by `board`, and copy, character by character, each element of `inBoard` into `board` (using nested `for` loops)
- a private method named `checkBoardBasics` with two parameters, one of type `int` (say, `inSize`) and one of type `char[][]` (say, `inBoard`); `checkBoardBasics` should check all of the following, returning `false` if any of them fail, and `true` otherwise:
  - `inBoard` is not `null`
  - `inBoard` has `inSize` rows
  - each row of `inBoard` is not `null`
  - each row of `inBoard` has length `inSize`

  - there are no illegal characters; use nested `for` loops to visit every position on the board and check that no position contains a character other than '`Q`' or '`B`' (use your class constants here)

- a private method named `oneQueenPerRow` with no parameters and a `boolean` return type; this method should return `true` if each row of the board contains exactly one queen, and `false` otherwise
- a private method named `oneQueenPerColumn` with no parameters and a `boolean` return type; this method should return `true` if each column of the board contains exactly one queen, and `false` otherwise
- a private method named `noDiagonalAttacks` with no parameters and a `boolean` return type; this method should return `true` if no two queens can attack each other diagonally, and `false` otherwise
- a *public* method named `isValidSolution` with no parameters and a `boolean` return type; this method should call the methods `oneQueenPerRow`, `oneQueenPerColumn`, and `noDiagonalAttacks`, and should return `true` if all three helper methods return `true`; otherwise, it should return `false`

`TestNQueensVerifier` Class

The `TestNQueensVerifier` should contain a `main` method in which you do the following:

- prompt the user for the full path name of an input file containing a board configuration
- use a `Scanner` associated with keyboard input to read in the user's response
- construct a `File` object associated with the disk file specified by the user
- use this `File` object to construct a second `Scanner`, and use this second `Scanner` to read in the contents of the file
- construct an `NQueensVerifier` object, passing to the constructor the appropriate board information obtained from the input file
- call `isValidSolution` on the `NQueensVerifier` object, and, based on the result, print an appropriate message for the user

Final Points

1. The second `Scanner` constructor can potentially throw a checked exception, so use a try/catch structure to handle this.
2. You can assume that any input file is a text file with the following format:
   - the first line contains a positive integer; this is the board size (N)
   - each of the next N lines contains a sequence (string) of N characters, each of which is either '`Q`' (representing a queen) or '`B`' (representing a blank square)