Hug the Angry Jarvis Game Instructions Part 1

Introduction

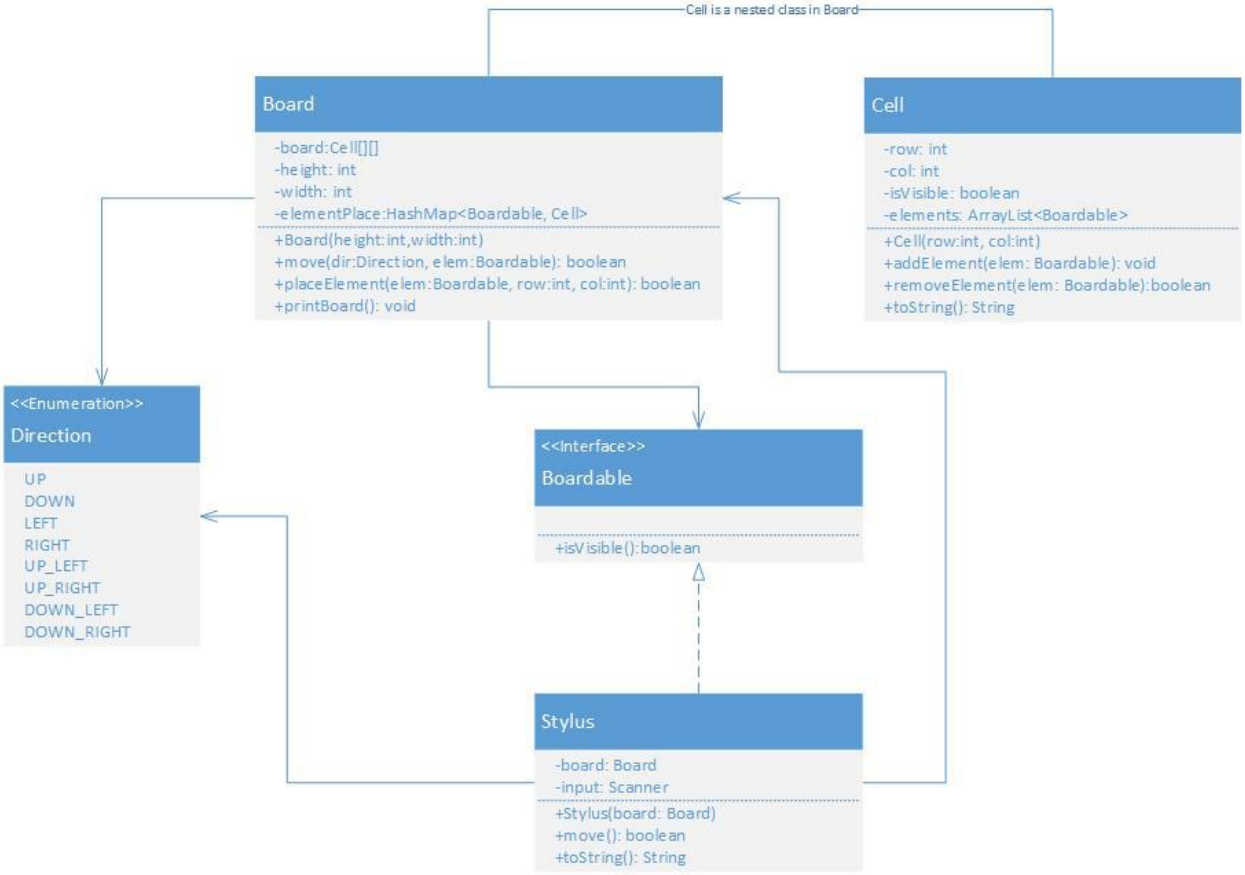
In this assignment, you have the opportunity to design and work with Java objects, interfaces, nested classes, and throwing exceptions. Additionally, you will be using HashMap, ArrayList, Strings, Scanner, and other basic Java constructs.

This assignment is to be done individually.

On our journey toward building the "Hug the Angry Jarvis" game, we have the opportunity to create another fun game; a digital etch a sketch. We can do this by implementing a portion of our ultimate game with a few modifications.

The Assignment

Below you will see the UML design of our "Etch a Sketch" game. Your classes MUST HAVE all of the entities (classes, methods, and fields) and they MUST HAVE the exact same names (case matters as does the order of parameters). You are welcome and possibly should add additional PRIVATE methods to your classes. However, you are not allowed to add any additional public methods.



The Board class will be responsible for displaying our sketches. Board contains a grid of Cells (more on them in a bit). When it is initialized, Board creates the grid (2-d array). Boardable elements can be placed in a particular cell (specified by the row and column values). After an element has been placed, the Board is responsible for tracking its location. This is done by updating the elementPlace map when

the element is initially placed and every time the element is moved. The Broad class also displays the board (i.e., printBoard). In doing so it relies, in part, on the toString of the individual Cells. The Board is (re-)displayed on initial creation and every time a board element is moved. In addition to the methods and return values shown in the UML, the Board constructor should throw an IllegalArgumentException if the passed in width and/or height that is not in the range [1-100]. Similarly, the move method should also throw an IllegalArgumentException if the passed in elem has not been previously placed on the board.

Cell is a nested class in Board. It is used to create the individual cells of the grid. A Cell can contain an element (actually later they can contain multiple elements.) If an element that is visible is added to a Cell, then that Cell becomes visible (and will be visible as long a Board exists.) A Cell's toString that is not visible should return "#", an empty Cell that is visible should return " ", and a Cell that isVisible that contains an element should return the element's toString value.

Stylus is the "pen" we are using to draw on the board. Its constructor takes the Board that it will be drawing on. Its move function invokes the Board's move passing a Direction and the "this" pointer. This is essentially telling the Board to move the Stylus in that direction. The Stylus's move function should take input from the user. The user inputs map to the directions as follows:

- "q" = UP_LEFT
- "w" = UP
- "e" = UP_RIGHT
- "a" = LEFT
- "d" = RIGHT
- "z" = DOWN_LEFT
- "x" = DOWN
- "c" = DOWN_RIGHT

Note that the top of the board (so the most UP) is row 0 in the gird. Any correct input should result in the Stylus moving on the board if possible (it isn't allowed to go outside of the bounds of the grid) and a return value of true (even if the move wasn't possible.) Any input other than the specified above should return false. You will lose points if your system crashes because of wrong user input. The Stylus's toString should return "*".

In addition to the classes specified in UML you need to also turn in two driver classes, TestDriver and DrawingDriver. TestDriver should comprehensively and automatically test all components of the entire system. Make sure to carefully document each test in comments. This should include what is being tested and the expected outcome. DrawingDriver should ask the user the size of the Board they would like to draw on (which it should then create). It should then create a Stylus for that board and repeatedly invoke move until the user enters a value that is not a valid move.

Grading

I'll be watching for style, design, as well as correct output. Your output should be neat and easily understood. At no point and time should your code crash. Make sure your code is well commented and carefully follows the class Style Guide (including a Javadoc-style comment (/** comment */) before each method). Use constants where appropriate, and focus on elegant solutions. If you find that you are

writing very large methods, that likely means they should be broken into multiple smaller ones. If you have lots of repeated code, that indicates that you should write a method to do that work. When you are designing your classes, try to keep them very tightly focused.

Submitting

Submit a *zip* all of your .java files to Canvas.