Peg Solitaire Game

Start Assignment

Due No Due Date **Points** 1 **Submitting** a file upload

This assignment has been designed as a self-assessment for students considering enrollment in CS300. Our expectation is that well-prepared students should be comfortable completing an assignment like this within two weeks (or about 15 hours, since CS300 is a 3 credit class).

Although students will use Java to complete their CS300 assignments, this self-assessment can be completed in whatever language they feel most comfortable working in. Students with sufficient experience in languages like C, C++, Javascript, Python, Matlab, and others can expect to find many familiar data types and control structures in Java. Experience with designing, implementing, testing, and debugging code written in these kinds of languages should transfer nicely toward developing code in Java. You'll note in the required organization below that your solution will be in a more procedure-oriented rather than object-oriented form. This is because the later organizational paradigm will be taught in CS300.

The rest of this document is organized into the following sections:

- **The Rules of Peg Solitaire** – The rules of the game you are implementing for this assignment.
- **Organizational Requirements** – How your solution to this assignment must be organized.
- **Sample Runs** – Demonstrate the output of a working implementation.
- **Code Templates** – Files containing method stubs, ready for you to implement.

Once you complete this project, upload your code here in Canvas.

The Rules of Peg Solitaire

Peg solitaire is a board game (or puzzle) in which a single player takes turns removing pegs from a board until either: they have removed all but the final peg to win the game, or they have lost because there are no legal moves left for them to make despite more than one peg remaining on the board. The number of pegs and board shapes vary, but these pegs are always set into holes arranged in a grid on the board. A legal move involves jumping one peg over a neighboring peg to rest in a hole on the other side and then removing the peg that was jumped over. Diagonal jumps are not allowed. In the following examples, an at sign (@) is used to represent the position of pegs, and a dash (-) is used to represent the position of empty holes.

```
@@-           --@           @-@           @-@           @--
@@@           @@@           -@@           -@@           -@-
@@@ (turn1)   @@@ (turn2)   -@@ (turn3)   @-- (turn4)   @-@
```

In the example above, the first turn involves moving the peg from the upper-left corner of the board, over the peg in the top-middle position, and landing in the hole in the upper-right corner. The peg in the top-middle position is then removed since it was jumped over. Can you follow the subsequent moves starting with the lower-left peg on turn 2, lower-right peg on turn 3, and upper-right peg on turn 4? After this sequence of moves, there are no more legal moves to be made. And since there is more than one peg left on the board, this example demonstrates a lost game.

More information about this game is available on **Peg Solitaire Game on Wikipedia (https://en.wikipedia.org/wiki/Peg_solitaire)** .

Organizational Requirements

Although you are free to implement this assignment in a language of your choice, well-prepared CS300 students are comfortable organizing their code into methods (also called functions, procedures, subroutines, etc). You are allowed to make use of global (or static) constants, but all mutable variables should be defined locally within some method. All communication between methods must be passed through their arguments and return values. To demonstrate your ability to implement this game with a procedure-oriented organization, your code for this assignment must be organized into the following methods.

-
  ```
  void main(String[] args) // drives the entire game application
  ```

  This method is responsible for everything from displaying the opening welcome message to printing out the final thank you. It will clearly be helpful to call several of the following methods from here, and from the methods called from here. See the Sample Runs below for a more complete idea of everything this method is responsible for.

-
  ```
  int readValidInt(Scanner in, String prompt, int min, int max) // returns a valid integer from th
  e user
  ```

  This method is used to read in all inputs from the user. After printing the specified prompt, it will check whether the user's input is in fact an integer within the specified range. If the user's input does not represent an integer or does not fall within the required range, print an error message asking for a value within that range before giving the user another chance to enter valid input. The user should be given as many chances as they need to enter a valid integer within the specified range. See the Sample Runs to see how these error messages should be phrased, and to see how the prompts are repeated when multiple invalid inputs are entered by the user.

-
  ```
  char[][] createBoard(int boardType) // returns an array, initialized according to a specific boa
  rd type
  ```

  This method creates, initializes, and then returns a rectangular two-dimensional array of characters according to the specified boardType. Initial configurations for each of the possible board types are depicted below. Note that pegs are displayed as @s, empty holes are displayed as -s, and extra blank positions that are neither pegs nor holes within each rectangular array are displayed as #s.

1. Cross

```
    ###@@@###
    ###@@@###
    @@@@@@@@@
    @@@@-@@@@
    @@@@@@@@@
    ###@@@###
    ###@@@###
```

2. Circle

```
    #-@@-#
    -@@@@-
    @@@@@@
    @@@@@@
    -@@@@-
    #-@@-#
```

3. Triangle

```
    ###-@-###
    ##-@@@-##
    #-@@-@@-#
    -@@@@@@@-
```

4. Simple T

```
    -----
    -@@@-
    --@--
    --@--
    -----
```

- `void displayBoard(char[][] board) // prints out the contents of a board to for the player to see`

  This method prints out the contents of the specified board using @s to represent pegs, -s to represent empty hole, and #s to represent empty positions that are neither pegs nor holes. In addition to this, the columns and rows of this board should be labelled with numbers starting at one and increasing from left to right (for column labels) and from top to bottom (for row labels). See the Sample Runs for examples of how these labels appear next to boards with different dimensions.

- `int[] readValidMove(Scanner in, char[][] board) // reads a single peg jump move in from the user`

  This method is used to read in and validate each part of a user's move choice: the row and column that they wish to move a peg from, and the direction that they would like to move/jump that peg in. When the player's row, column, and direction selection does not represent a valid move, your program should report that their choice does not constitute a legal move before giving them another chance to enter a different move. They should be given as many chances as necessary to enter a legal move. The array of three integers that this method returns will contain: the user's choice of column as the first integer, their choice of row as the second integer, and their choice of direction as the third.

-
  ```
  boolean isValidMove(char[][] board, int row, int column, int direction) // checks move validity
  ```

  This method checks whether a specific move (column + row + direction) is valid within a specific board configuration. In order for a move to be a valid: 1)there must be a peg at position row, column within the board, 2)there must be another peg neighboring that first one in the specified direction, and 3)there must be an empty hole on the other side of that neighboring peg (further in the specified direction). This method only returns true when all three of these conditions are met. If any of the three positions being checked happen to fall beyond the bounds of your board array, then this method should return false. Note that the row and column parameters here begin with one, and may need to be adjusted if your programming language utilizes arrays with zero-based indexing.

-
  ```
  char[][] performMove(char[][] board, int row, int column, int direction) // applies move to a bo
  ard
  ```

  The parameters of this method are the same as those of the isValidMove() method. However this method changes the board state according to this move parameter (column + row + direction), instead of validating whether the move is valid. If the move specification that is passed into this method does not represent a legal move, then do not modify the board.

-
  ```
  int countPegsRemaining(char[][] board) // returns the number of pegs left on a board
  ```

  This method counts up the number of pegs left within a particular board configuration and returns that number.

-
  ```
  int countMovesAvailable(char[][] board) // returns the number of possible moves available on a b
  oard
  ```

  This method counts up the number of legal moves that are available to be performed in a given board configuration. HINT: Would it be possible to call the isValidMove() method for every direction and from every position within your board? Counting up the number of these calls that return true should yield the total number of moves available within a specific board.

Note that these method headers correspond to their Java implementations, and may differ from another language in some ways. The Scanner parameters in the methods above are used to read input from the user through standard-in. Although a parameter like this may not be necessary for some languages, seeing this parameter still provides a helpful hint about which methods may require extra information from the user. The other types are likely to look more familiar: int is an integer numeric type, char[][] is the type for a random access two-dimensional array of characters, and void means that no value is returned from a method. Another thing worth noting is that the size of an array is accessible through an array reference in Java. If this is not the case in your language, additional parameters may be necessary to communicate this information.

Sample Runs

The user's input in each of the following sample runs is displayed as `bold orange text`.

Winning Simple T Run

```
WELCOME TO CS300 PEG SOLITAIRE!
===============================

Board Style Menu
  1) Cross
  2) Circle
  3) Triangle
  4) Simple T
Choose a board style: 4

  12345
1 -----
2 -@@@-
3 --@--
4 --@--
5 -----
Choose the COLUMN of a peg you'd like to move: 3
Choose the ROW of a peg you'd like to move: 2
Choose a DIRECTION to move that peg 1) UP, 2) DOWN, 3) LEFT, or 4) RIGHT: 4

  12345
1 -----
2 -@--@
3 --@--
4 --@--
5 -----
Choose the COLUMN of a peg you'd like to move: 3
Choose the ROW of a peg you'd like to move: 4
Choose a DIRECTION to move that peg 1) UP, 2) DOWN, 3) LEFT, or 4) RIGHT: 1

  12345
1 -----
2 -@@-@
3 -----
4 -----
5 -----
Choose the COLUMN of a peg you'd like to move: 2
Choose the ROW of a peg you'd like to move: 2
Choose a DIRECTION to move that peg 1) UP, 2) DOWN, 3) LEFT, or 4) RIGHT: 4

  12345
1 -----
2 ---@@
3 -----
4 -----
5 -----
Choose the COLUMN of a peg you'd like to move: 5
Choose the ROW of a peg you'd like to move: 2
Choose a DIRECTION to move that peg 1) UP, 2) DOWN, 3) LEFT, or 4) RIGHT: 3

  12345
1 -----
2 --@--
3 -----
4 -----
5 -----
Congrats, you won!
```

```
==========================================
THANK YOU FOR PLAYING CS300 PEG SOLITAIRE!
```

Losing Simple T Run

```
WELCOME TO CS300 PEG SOLITAIRE!
===============================

Board Style Menu
  1) Cross
  2) Circle
  3) Triangle
  4) Simple T
Choose a board style: 4

  12345
1 -----
2 -@@@-
3 --@--
4 --@--
5 -----
Choose the COLUMN of a peg you'd like to move: 3
Choose the ROW of a peg you'd like to move: 3
Choose a DIRECTION to move that peg 1) UP, 2) DOWN, 3) LEFT, or 4) RIGHT: 1

  12345
1 --@--
2 -@-@-
3 -----
4 --@--
5 -----
It looks like there are no more legal moves.  Please try again.

==========================================
THANK YOU FOR PLAYING CS300 PEG SOLITAIRE!
```

Bad Input Triangle (Partial) Run

```
WELCOME TO CS300 PEG SOLITAIRE!
===============================

Board Style Menu
  1) Cross
  2) Circle
  3) Triangle
  4) Simple T
Choose a board style: three
Please enter your choice as an integer between 1 and 4: 3!!!
Please enter your choice as an integer between 1 and 4: 3

  123456789
1 ###-@-###
2 ##-@@@-##
3 #-@@-@@-#
4 -@@@@@@@-
Choose the COLUMN of a peg you'd like to move: -2 4 5
Please enter your choice as an integer between 1 and 9: five
Please enter your choice as an integer between 1 and 9: 4
Choose the ROW of a peg you'd like to move: 10
Please enter your choice as an integer between 1 and 4: 2
Choose a DIRECTION to move that peg 1) UP, 2) DOWN, 3) LEFT, or 4) RIGHT: 3
Moving a peg from row 2 and column 4 LEFT is not currently a legal move.

Choose the COLUMN of a peg you'd like to move: 5
Choose the ROW of a peg you'd like to move: -1
Please enter your choice as an integer between 1 and 4: 1
Choose a DIRECTION to move that peg 1) UP, 2) DOWN, 3) LEFT, or 4) RIGHT: 0
```

```
Please enter your choice as an integer between 1 and 4: down
Please enter your choice as an integer between 1 and 4: 2

  123456789
1 ###---###
2 ##-@-@-##
3 #-@@@@@-#
4 -@@@@@@@-
```

(note that the above sample run is not complete)

Code Templates

Files containing stubs for the required methods are available in the following languages:

- Java: **PegSolitaireGame.java (https://canvas.wisc.edu/courses/327719/files/28717674?wrap=1)** **(https://canvas.wisc.edu/courses/327719/files/28717674/download?download_frd=1)**
- Python: **peg_solitaire_game.py (https://canvas.wisc.edu/courses/327719/files/28717675?wrap=1)** **(https://canvas.wisc.edu/courses/327719/files/28717675/download?download_frd=1)** , **peg_solitaire_game.txt (https://canvas.wisc.edu/courses/327719/files/28717676?wrap=1)** **(https://canvas.wisc.edu/courses/327719/files/28717676/download?download_frd=1)** (python template)

Submission

Once you complete this project, upload your code here in Canvas.