Lab 9

For parts 1, 2 and 3 you will need to use the BlackJack.java and Card.java files from the class website (under the “Lab assignments” section). BlackJack is a card game, where you try and reach the number 21. If you go over 21 you lose and otherwise the winner is the closest person to 21. If the card is an ace, it can be counted as either 1 or 11 (whichever is most beneficial). Face cards (Jack, Queen and King) are worth 10. All other cards are worth the value written on the card (A “2 of hearts” is worth 2). Only a basic version of BlackJack is provided (without split, double down, etc.).

**Part 1.** Familiarize yourself with the code (you can run the code in it’s current form, but some features are not implemented yet). There is a method hit() that is supposed to add a card to your current hand. However, this method currently does nothing. Use the variable “hand” and “numberOfCards” to keep track of how many cards you have in your partially filled array.

**Part 2.** There is a getCards() method that should return an array of exactly the cards you are using in your partially filled array. Currently this method returns an array of length zero and does not show what cards you have. By correctly fixing the getCards() method, the println() statements should display your hand.

**Part 3.** There is currently a valueOfHand() method that returns an (often) incorrect result. Fix the valueOfHand() to always return the correct value. To do this you should write a method that counts the number of aces in your hand. The number of aces are important since aces can be counted as either 11 or 1. For example if your hand is two aces, you should have the value of 12 (11 + 1) and not 22 (11 + 11) since 22 is a worse result than 12.

A “magic square” is a square where all rows, columns and diagonals add up to the same value. (See: [http://en.wikipedia.org/wiki/Magic_square](http://en.wikipedia.org/wiki/Magic_square)). The method we will use to create a magic square is under the section called “Method for constructing a magic square of odd order” (An example is provided in this section). A formula for this method is provided below (Note: this only works for odd sized grids):

Step 1: Insert the value 1 anywhere in the grid.

Step 2:

(a)If the square up and to the right is already filled with a number, then put the next integer (counting up) in the square directly below the current position.

(b) Otherwise, go one square up and to the right and put in the next integer (counting up). If you go off the grid, wrap around to the bottom and/or left.

Step 3: Go to step 2 until the whole grid is filled.
Part 4. Write a method to check and see if the square to the up and right is currently occupied (as described in Step 2). This should return true if the square is already filled in with a number, and otherwise return false if the square is blank. (Note: you may need to provide arguments to this method, depending on how you wrote your code.)

Part 5. Write code to actually fill in the grid (Step 2, 3 and 4) and then display the grid. You may start your initial value 1 (Step 1) anywhere you like. The size of your grid can either be defined in your code, or you can use Scanner to ask the user for the size they want.

Part 6 (extra credit). Rewrite the portion of your magic square code where you filled in the array (Step 2, 3 and 4) using only two lines of code. One line is a (for) loop statement and the other is an assignment statement. You may not use methods in either line of code. You may not simply condense your code to fit on one line (you can only use 2 semicolons for the for loop and one semicolon for the assignment statement). Your array must follow the formula explained before part 4 (numbers must increase going up and to the right).

(Hint: one solution requires using operators you may not have used much and operators you normally don't like. It might also help to actually write out a few of these magic squares (on paper) and see the pattern that emerges when following the algorithm described before part 4.)