Project 3: Card Hand simulator project **Assigned: Thursday February 20, 2020**	**CS 212: Spring 2020**

Due part A (see in the tables below what is included in part A): Thursday March 12, 2020
You cannot turn part A late. I will be giving the answer to part A on Thursday March 12, 2020
Due (whole project): Thursday March 19, class time
Objectives: Coding a class, using classes.
Assignment: You need to write two classes:
FirstLastBridgeHand and FirstLastSimulateHand (for me, it would be HerveFranceschiBridgeHand and HerveFranceschiSimulateHand)

NOTE: Although this project is applicable to the game of bridge, you do not need to know how to play bridge to do this project.

The FirstLastBridgeHand class encapsulates a hand of cards (13 different cards): it includes the following:

- An int constant storing 13, the number of cards per hand.
- An int constant storing 52, the number of cards in the deck.
- A char array constant storing characters representing the face of the cards, from A (Ace, K (King), to 2. Note: 10 is T.
- An array of ints, representing the cards in this hand. Each int represents a card as follows:
    - 0-12: spades, from the Ace of spades (0) to the 2 of spades (12)
    - 13-25: hearts, from the Ace of hearts (13) to the 2 of hearts (25)
    - 26-38: diamonds, from the Ace of diamonds (26) to the 2 of diamonds (38)
    - 39-51: clubs, from the Ace of clubs (39) to the 2 of clubs (51)
- A default constructor: it generates randomly the ints in the array; no duplicates are allowed, all the cards must be different (see isDuplicate method below).
- A **private** isDuplicate method: it accepts an int parameter and checks if that int is already in the array instance variable. This method can be useful for the default constructor.
- A public sort method: it sorts the array of ints so that they are ordered in ascending order (thus, spades will be first in ascending order, then hearts in ascending order, …). The constructor should call that method. This will be useful when you code the toString method.
- 4 public methods named spades, hearts, diamonds, and clubs. They each return an array of ints representing the spades, hearts, diamonds, or clubs, respectively, in this hand. Important note: each array must have minimal size. For example, if there are 4 ints in an array, it should have size 4.
- **Testing**: At that point, you should be able to test your code in the GUI class that is provided to you. Just replace the 4 hard coded arrays with spades, hearts, diamonds, and clubs arrays generated by your class. For this to work, you need to download card gifs from https://www.waste.org/~oxymoron/cards/; these card gifs are made available

under the GNU General Public License. The card directory (which contains the card gif files after you unzipped the file) should be in the same directory as your java files.

- A public method named value: it accepts an int parameter representing a card, i.e. an int between 0 and 51. It returns 4 if the corresponding card is an Ace, 3 if it is a king, 2 if it is a queen, 1 if it is a jack, and 0 otherwise.
- A public method named highCardPoints: it returns the total number of high card points in this hand (ace = 4, king = 3, queen = 2, jack = 1).
- A public method named shape: it returns an array of 4 elements representing the number of spades, hearts, diamonds, and clubs in this hand, respectively. For example, it could return an array with values 3, 4, 3, 3 or 3, 3, 5, 2, …
- A public method named sumOfLengthsOfTwoLongestSuits: it returns the sum of the number of cards in the two longest suits (for example, if there are 5 spades, 2 hearts, 4 diamonds, and 2 clubs, it returns 9 (5 + 4)). It could be useful to call the shape method inside this method.
- A public method named ruleOf20: it returns the total number of high card points in this hand plus the total number of cards in the two longest suits.
- A public method named hasRegularOpening: it returns true if the ruleOf20 method returns 20 or more, false otherwise.
- A public method named typeOfHand: it returns either BALANCED or UNBALANCED. A balanced hand is defined as a hand having no void (no card in a suit), no singleton (one card in a suit), and at most one doubleton (two cards in a suit).
- A public method named weak3: it returns true if this hand has exactly 7 cards of a suit, has between 6 and 10 high card points (both) included, and has at least 2 "honors" in the 7-card suit (honors are the Ace, King, Queen, Jack, and 10).
- A public method named weak4: it returns true if this hand has exactly 8 cards of a suit, has between 6 and 10 high card points (both) included, and has at least 2 "honors" in the 7-card suit (honors are the Ace, King, Queen, Jack, and 10).
- A public method named has1NoTrump: it returns true if this hand is balanced and has between 15 and 17 high card points (both) included.
- A public method named has2NoTrump: it returns true if this hand is balanced and has either 20 or 21 high card points.
- A public method named weHaveGame: it takes an int parameter and returns true if the total of high card points in this hand added to the int parameter is 25 or more, false otherwise.
- A public method named weHaveSlam: it takes a BridgeHand parameter and returns true if the total of high card points in both hands is 33 or more, false otherwise.
- A public method named weHaveAFit: it takes a BridgeHand and a char parameter (representing a suit) whose value is either S, H, D, or C and returns true if the total number of cards in the suit in both hands is greater than or equal to 8, false otherwise.
- A public method named romanKeyCard: it takes a char parameter (representing a suit) whose value is S, H, D, or C; it returns the number of key cards in this hand. A key card is either an ace (any ace) or the king of trump (the char parameter is the trump suit).

- A public method named has2Clubs: it returns true if the number of high cards points in this hand is 22 or more, false otherwise.
- A public method named hasMichaels: it takes a char parameter (representing a suit) whose value is either S, H, D, or C. It returns true if this hand contains two suits, different from the char parameter, of at least 5 cards each, false otherwise.
- The toString method; it returns a String representation of this hand on 4 lines; a possible output is (each suit in descending order on one line starting with spades, then hearts, ..):

```
S: A K 4
H: 6
D: K J 7 5 3
C: J 10 7 4
```

The FirstLastHandSimulator class (where the main method is located) simulates the following (based on 100,000 randomly generated hands):

- It calculates the average number of high card points in a hand.
- It calculates the percentage of time that a hand has a regular opening (i.e. the hasRegularOpening method returns true).
- It calculates the percentage of time that a hand has a 1 No Trump opening (i.e. the has1NoTrump method returns true).
- It calculates the percentage of time that a hand is balanced (i.e. the typeOfHand method returns BALANCED).
- It calculates the percentage of time that a hand has a weak 3 opening (i.e. the weak3 method returns true).
- It calculates the percentage of time that a hand has a weak 4 opening (i.e. the weak4 method returns true).
- It calculates the average sum of lengths of the 2 longest suits in a hand.
- It outputs a String representation of the last hand generated.

Note: You need to use DecimalFormat, not NumberFormat, for maximum precision (because for Flannery, NumberFormat's precision is likely to be insufficient).

Submission Details: At the top of your program, make sure that you include your name. Test your program. For part A, submit your source code (2 .java files) only on Moodle. For the whole project, submit your source code on Moodle and github. For both part A and the whole project, submit a printout of the 2 classes in class.

Grading (100 points):
Variable names and constants: 3 points
Style: 2 points
Comments: 2 points
Testing questions: 2 points
Correctness: 91 points (see below)

Class	Topic	Points
BridgeHand (part A)	Constants	2
BridgeHand (part A)	Instance variable	2
BridgeHand (part A)	Default constructor	4
BridgeHand	sort method	4
BridgeHand (part A)	isDuplicate method	4
BridgeHand	spades, hearts, diamonds and clubs methods	8
BridgeHand (part A)	highCardPoints method	3
BridgeHand	shape method	3
BridgeHand	sumOfLengthsOfTwoLongestSuits method	3
BridgeHand	ruleOf20Points method	3
BridgeHand	hasRegularOpening method	3
BridgeHand	typeOfHand method	3
BridgeHand	weak3 method	3
BridgeHand	weak4 method	3
BridgeHand	oneNoTrump method	3
BridgeHand	twoNoTrump method	3
BridgeHand	weHaveGame	3
BridgeHand	weHaveSlam	3
BridgeHand	weHaveAFitIn	3
BridgeHand	romanKeyCard	3
BridgeHand	has2Clubs	3
BridgeHand	hasMichaels	3
BridgeHand	toString	3
HandSimulator (part A)	Average number of points	2
HandSimulator	% regular opening	2
HandSimulator	% 1 No Trump opening	2
HandSimulator	% balanced	2
HandSimulator	% weak 3 opening	2
HandSimulator	% weak 4 opening	2
HandSimulator	Average sum of lengths of 2 longest suits	2
HandSimulator	toString called	2

Testing	Topic	Points
highCardPoints method (part A)	On average, what do you expect this method to return?	1
highCardPoints method (part A)	What does your simulation show it returns?	1
sumOfLengthsOfTwoLongestSuits method	On average, what do you expect this method to return? (roughly)	1
sumOfLengthsOfTwoLongestSuits method	What does your simulation show it returns?	1

Extra credits:

1 - Inside the BridgeHand class, add a method to generate a second hand (if you want, you can add another array instance variable), without duplicate, and 100% different from the first hand (no duplicate between the 2 hands). Calculate the percentage of times that both hands have a combined total high card points of 25 points (4 points).

2 - Calculate the percentage of times that both hands (see extra credits # 1) have a combined total high card points of 25 points and do not have a fit in either spades or hearts (see weHaveAFitIn method) and at least one of the two hands is balanced (2 points).

Your program should include appropriate comments (including at the top, your name, date created and date last updated), variables should have proper names, and style should be good (spacing, indentation, ..).