American University of Beirut
Department of Computer Science
CMPS 200 – Introduction to Programming
Assignment 7: Deadline Tuesday April 19

Exercise 1. Card Data Type

A card from a standard 52-card deck of playing cards has two elements:

- a rank ("2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A"); and
- a suit ("Clubs", "Diamonds", "Hearts", "Spades")

Write a class `Card` that implements the methods whose signatures are shown below. The class should have 2 private data members: `rank` and `suit` (both of type `String`).

```
public Card (String r, String s)    // constructor

public String rank()

public String suit()

public boolean isOfSuit(String s) // checks if card is of given suit

public boolean stronger(Card c)   // true if card is stronger than c

public String toString()         // returns a printed representation
                                  // in the form "8S", "10D", "KC",...
```

A card is stronger than another if its rank is higher. In case of equal rank, the suit determines the relative strength: Spades beat Hearts which beat Diamonds which beat Clubs.

Your code should be in a file `Card.java`. The file should also include a `main()` method to test the methods of the `Card` class. Use the `main()` below and augment it with a few additional tests.

```
public static void main(String[] args) {

    Card c1 = new Card("10", "Hearts");

    Card c2 = new Card("Q", "Spades");

    System.out.println(c1);

    System.out.println(c2);

    System.out.println(c1.isOfSuit("Hearts"));       // should print true

    System.out.println(c2.isOfSuit("Hearts"));       // should print false

    System.out.ptintln(c1.stronger(c2));             // should print false

}
```

Exercise 2. CardDeck

In this program, you are to write a class `CardDeck` to represent a standard 52-card deck with operations to shuffle and draw cards from the deck. The class will use the Card class from the above problem. Here is a

possible API with the private fields to store the information about the deck:

```
public class CardDeck {

	private Card[] cards; // array that holds the 52 cards

	private int top;	// keeps track of the current top of deck,
				// initially zero, incremented as cards are drawn


	public CardDeck() {...}	// constructor, creates and fills cards array

	public boolean isEmpty() {...} // is the deck empty or not?

	public int cardsLeft() {...}	// how many cards are left in deck

	public void shuffle() {...}	// shuffles the deck (see algorithm below)

	public Card draw() {...}	// returns the card on top of deck

}
```

CardDeck constructor:

The array will be filled to contain all the cards. One way to do so can be to create an array storing all possible ranks and one storing all possible suits. Then nested for loops can be used to get all possible cards combinations and store them in the array.

Shuffle:

Shuffling an array of N objects (N cards in this case) may be accomplished by looping through the array (using an index i), picking a random object located between i and the end of the array, and swapping it with the object in location i.

Draw

It returns the card on top of the deck and then updates the top.

Test your implementation of CardDeck by writing a program that keeps drawing from a shuffled deck until an Ace is drawn. The program should report how many cards were drawn.

Exercise 3. BankAccount

Consider the Java class BankAccount.java. The BankAccount class contains as instance variables the following fields:

- **String id**: the name of the owner of the bank account
- **double balance**: the amount of money in the bank account
- **int transactions**: the number of transactions that have been performed by this user
- An **array list** of **transactions**

You are required to implement the following methods:

- ***BankAccount(String id)***: Constructor. Constructs a BankAccount object with the given id, with 0 balance and transactions.
- ***double getBalance()***: returns the value of the balance field.
- ***String getId()***: returns the value of the id field.
- ***int getTransactions():*** returns the value of the transactions field. It will display the list of transactions.
- ***void deposit(double amount):*** Adds the amount to the balance if it is between 0-500. This counts as 1 transaction. And the transaction should be added to the list as: "Deposit of $*amount*"
- ***void withdraw(double amount):*** Subtracts the amount from the balance if the user has enough money. If not, it should print as such. This counts as 1 transaction. It should be added to the list of transaction as: "Withdraw of $amount".
- ***String toString():*** returns the bank account as a String in the following format:

  <id>, $<balance>

  For example, if the account has an id "Mary" and a balance of "517.5" it should be returned as ***Mary, $517.50.*** The balance should always have 2 digits after the decimal.
- ***boolean transactionFee(double amount)***: accepts a fee amount as a parameter and applies that fee to the user's past transactions. The fee is applied once for the first transaction, twice for the second transaction, three times for the third, and so on. If the user's balance is large enough to afford all of the fees with greater than $0.00 remaining, the method returns true. If the balance cannot afford all of the fees or has no money left, the balance is left as 0 and the method returns false.
- ***void transfer(BankAccount acc, double amount):*** The method accepts two parameters: a second BankAccount to accept the money, and a double for the amount of money to transfer. There is a $5.00 fee for transferring money that will be deducted from the sender's account. The method deducts from "this" current object the given amount plus the $5 fee, and the other BankAccount's balance is increased by the given amount. A transfer also counts as a transaction on both accounts. If after deducting the $5.00, 'this' account doesn't have enough to complete the transaction, as much money as possible is transferred to acc. If the account has less than $5.00, the no transfer should occur and neither accounts change.

Exercise 4. Points and Lines

PART A: Creating a Point class.

In this first part of this problem, it is required to create a new class called Point. A point is typically described by its geometric coordinates. In this problem, a two-dimensional cartesian coordinate system is considered. Consequently, a point, say P, in such a system is defined by its abscissa x and ordinate y. Following the definition of a generic point's coordinates, write accessor and mutator

methods, namely, getX(), getY(), setX(), setY(), getXY(), setXY(). Note that the getXY() function will return both the abscissa and ordinate of a point. Also, the setXY() takes two parameters as inputs being the new abscissa and new ordinate of the point. This being done, write a function called distance() that takes a point object, say secondPoint, as a parameter and computes the cartesian distance between the current point and secondPoint. Finally, a toString() method will print out the x and y coordinates of a point in an appropriate string.

PART B: Creating a Line class.

Here, it is required to use the Point class in order to create a new Line class. Recall, a Line, say l, is described by the two points, say p1 and p2that it goes through. It also has an equation of the form $y = sx + i$ where s is the slope and i is the intercept. In addition, in this problem, a line will also have one additional property being the range $[xl; xu]$ over which it can be plotted where xl and xu are the lower and upper bounds delimiting this range. On the next page is an incomplete skeleton of the class Line which you are required to complete.

Description is below:

1. The constructor function takes two input Point parameters p1 and p2 and constructs a new Line object l. Throughout the construction, given the two points, the constructor will resort to a Helper Function called computeEquation()whose description is below.
2. The helper function computeEquation() takes the newly constructed Line object l as a parameter and computes its equation which it returns as a String eq to be stored in the variable e of the Line object l. Throughout the computation of the equation, the computeEquation() function is given the authority to mutate the respective slope and intercept variables s and i of the object l and store in them their corresponding values which it computes using the coordinates of the points p1 and p2 above. Here, note that given two points $P1(x1, y1)$ and $P2(x2, y2)$, the slope and intercept of the line passing through them are:

   $\boldsymbol{s = (y2 - y1) / (x2 - x1)}$;

   $\boldsymbol{i = y1 - s \cdot x1}$

Following the computation of s and i above, special care is due when creating the String variable eq in a professional manner. That is, if $s = 0$ and $i \neq 0$ the String variable eq should contain 'y = i' (where i is replaced by its value). Alternatively, if $s \neq 0$ and $i = 0$ then the String variable eq should only contain 'y = sx' (where s is replaced by its value). In addition, if $s = 1$, then eq should contain 'y = x + i' (where i is replaced by its value).

3. The toString() method which return a meaningful string containing all the information about the newly created line. A sample output of the toString() method once called using an appropriate print() command from the client program is given below as well.

PART C: Driver / Client program.

In this part it is required to create a JAVA client program called LineClient.java to test the functionality of the above classes.

Sample output for the toString() function of the Line class:

Point 1: (1, 2)

Point 2: (3, 4)

Equation: y = x + 1