CSCI 1101 – Winter 2017
Laboratory No. 2

This lab is a continuation of the concepts of object-oriented programming.

Note:

1. *All submissions must be made on Brightspace (dal.ca/brightspace).*
2. ***Submission deadline is 11.55 p.m. (5 minutes to midnight) on Saturday, January 28th, 2017.***
3. *Put the java source code files and the text outputs for each exercise in a folder. Zip the folder into one file and submit the zip file.*

***4. Marking Scheme**:*

Each exercise carries 10 points. Your final score will be scaled down to a value out of 10.
Working code, Outputs included, Efficient, Comments included: 10/10
No comments: subtract one point
Unnecessarily long code and inefficient program, improper use of variables: subtract one point
No outputs: subtract two points
Code not working: subtract up to six points depending upon the extent to which the program is incorrect.

***5. Error checking**: Unless otherwise specified, you may assume that the user enters the correct data types and the correct number of input entries, that is, you need not check for errors on input.*

***6. Testing your code and generating the outputs:** If the test data is included in the question, use that to test your classes. In addition, test it with two more input sets. Otherwise, create your own test data and run your program for at least 3 input sets such that they cover the range of results expected.*

Exercise 1:
In this exercise, you are to write a simple object-oriented program to simulate the movement of a bug along a horizontal line on the floor of a room. The horizontal line starts at the 0^{th} position and ends at the 50^{th} position. In other words, the length of the horizontal line is 51 steps - there is a wall at the left and right ends of the line.

The bug starts at an initial position between 0 and 50 (both inclusive) and an initial direction, left or right. It can move one step at a time, either to the left or to the right (depending upon its direction). If it is at the left wall, it changes direction and moves one step. Similarly, if it is at the right wall, it changes direction and moves one step.

You need to write a class called Bug that models the bug's position and direction, and has the method move() in addition to other methods. The tester program simulates a bug at an initial random position between 0 and 50, makes it move a random number of steps and displays its initial and final position. The tester program has been given! All you need to write is the code for the Bug class. Complete the code, compile and run it, and you should see results similar to the one given at the end of the program below.

```
public class Bug
{

      //TODO: add the attributes
      //an integer variable for position
```

```
//another integer variable for direction (0 means left
//and a 1 means right)

//TODO: add a constructor that sets the position and direction

//TODO: add the appropriate get and set methods

//TODO: add the move method
public void move()
{
//Here the algorithm:
/*if the direction is left, decrement the position by 1;
if the direction is right, increment the position by 1;
However, before you do the above, if the position is 0 and the direction
is left, change the direction to right and increment the position by 1;
Similarly, if the position is 50 and the direction is right, change the
direction to left and decrement the position by 1
*/

}

public String toString()
{
      //TODO: return a String that contains the Position and //the
Direction of the bug
}


//BUG TESTER MAIN METHOD
public static void main(String[] args)
  {
      //randomly set the initial position(between 0 and 50) and direction
//(between 0 and 1)
      int initialPos = (int)(Math.random()*51);
      int initialDir = (int)(Math.random()*2);
      Bug bugsy = new Bug(initialPos, initialDir);
      System.out.println(bugsy);

      //display the line with the bug
      for(int i=0; i<=50;i++)
      {
         if (i==bugsy.getPosition())
            System.out.print("X");
         else
            System.out.print("-");
      }
      System.out.println();

      //make the bug move a random number of times
      int moves = (int)(Math.random()*51);
      System.out.println("Moves: " + moves);
      for(int i=1;i<=moves;i++)
         bugsy.move();
      System.out.println(bugsy);

      //display the line with the bug
      for(int i=0; i<=50;i++)
      {
         if (i==bugsy.getPosition())
            System.out.print("X");
         else
```

```
                System.out.print("-");
            }

            System.out.println();


        }



    }
```

Sample outputs:

```
----jGRASP exec: java -ea Bug
Position: 25    Direction: Left
-------------------------X-------------------------
Moves: 20
Position: 5     Direction: Left
-----X-----------------------------------------

----jGRASP: operation complete.


----jGRASP exec: java -ea Bug
Position: 15    Direction: Left
---------------X---------------------------------
Moves: 38
Position: 23    Direction: Right
-----------------------X-------------------------

----jGRASP: operation complete.

----jGRASP exec: java -ea Bug
Position: 30    Direction: Right
------------------------------X-------------------
Moves: 49
Position: 21    Direction: Left
---------------------X----------------------------

----jGRASP: operation complete.
```

Exercise 2: Implement the Point class (code for this up to the isHigher method was discussed in the lecture). The class has the following instance variables:

- x coordinate (an int)
- y coordinate (an int)

and the following methods:

- Constructor that sets the x and y coordinates
- Get and set methods
- Method equals if this point is equal to another point object (if the x and y coordinates are the same).
- Method isHigher if this point is higher than another point object (Note that we assume that the top left corner is the coordinate 0,0).

- Method findDistance that calculates the distance between this point and another point. (Formula for the distance between two points (x1,y1) and (x2,y2) is square root of $((x2-x1)^2 + (y2-y1)^2)$

Test the class. In the demo program, construct four points p1, p2, p3 and p4 using user-defined values. Determine

- Which point is the highest. (If more than one point is at the same height, you may report any one point).
- Whether the distance p1→ p2 is more than the distance p3→ p4, or p3→ p4 is more than p1→p2, or whether they are the same.

A sample screen dialog is given below:

```
Enter the x and y coordinates of point1: 8 9
Enter the x and y coordinates of point2: 4 3
Enter the x and y coordinates of point3: 2 1
Enter the x and y coordinates of point4: 5 6
[2,1] is the highest point
The distance between [8,9] and [4,3] is 7.211102550927978
The distance between [2,1] and [5,6] is 5.830951894845301
[8,9]-->[4,3] is longer than [2,1]-->[5,6]
```

Exercise 3: Implement a Stock class that has the following attributes:
symbol (String), price (double), number of shares (int)
and the following methods:
Constructor (that sets the symbol, price and number of shares to user defined values).
Get and set methods.
toString method returns the symbol, price and number of shares
method `public int compareTo(Stock s)` compares the share price of this stock with another stock and returns:
-1 if the value of this stock is lower than the other stock.
1 if the value of this stock is higher than the other stock.
0 if both the values are the same.
(Value of the stock = price * number of shares)

For example, let's say IBM has a price of $105.23 and you have bought 45 shares and MOS has a price of $89.88 and you have bought 60 shares. Then the value of IBM in your portfolio is 105.23*45= $4735.35 and the value of MOS in your portfolio is 89.88*60 = $5392.80. Comparing this stock(IBM) with another stock(MOS) will return a -1 since the value of MOS is greater in your portfolio.

Test the Stock class with a client program that asks the user to enter the symbols, share prices, and number of shares for two stocks, prints their values, determines which stock is higher than the other and by how much and prints the total value. You must use the methods in the Stock class wherever appropriate.

A portion of the client program is given below for your programming convenience.

```
import java.util.Scanner;
public class StockDemo
{
	public static void main(String[] args)
	{
		Scanner keyboard = new Scanner(System.in);
		String sym1, sym2;
		double prc1, prc2;
```

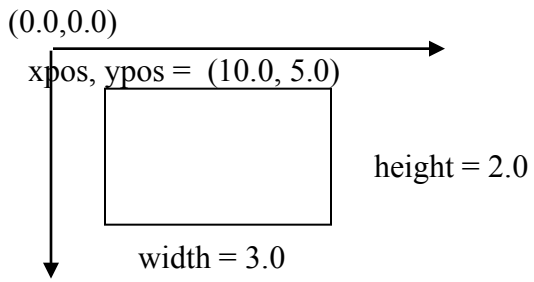
```
            int sh1, sh2;
            //get the values for two stocks
            System.out.print("Enter the symbols for the two stocks: ");
            sym1 = keyboard.next();
            sym2 = keyboard.next();
            System.out.print("Enter their prices: ");
            prc1 = keyboard.nextDouble();
            prc2 = keyboard.nextDouble();
            System.out.print("Enter the number of shares for the two stocks: ");
            sh1 = keyboard.nextInt();
            sh2 = keyboard.nextInt();

            //create the first Stock
            Stock s1 = new Stock(sym1,prc1,sh1);


            //create the second Stock
            Stock s2 = new Stock(sym2,prc2,sh2);

            // continue the rest of the code here
      }
}
```

The following is a sample screen dialog:

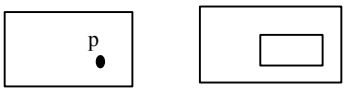
```
Enter the symbols for the two stocks: IBM MOS
Enter their prices: 105.23 89.88
Enter the number of shares for the two stocks: 45 60

I have the following stocks:
Stock: IBM
Price: 105.23
Shares: 45

Stock: MOS
Price: 89.88
Shares: 60

The value of MOS is higher than IBM
The total value of my portfolio is: $ 10128.15
```

Exercise 3: Define the Rectangle2D class that contains:

- Double data fields named xpos, ypos (that specify the top left corner of the rectangle), width and height. (assume that the rectangle sides are parallel to the x and y axes. See example below).
-



- A no-arg constructor that creates a default rectangle with (0.0, 0.0) for (xpos, ypos) and 0.0 for both width and height.
- A constructor that creates a rectangle with the specified xpos, ypos, width, and height.
- Get and set methods for all the instance variables.
- A method getArea() that returns the area of the rectangle.
- A method getPerimeter() that returns the perimeter of the rectangle.
- A method contains(x, y) that returns true if the specified point (x, y) is inside this rectangle. See Figure 1(a).
- A method contains(Rectangle2D r) that returns true if the specified rectangle is inside this rectangle. See Figure 1(b). Note that the condition for contains is that all four corners of the second rectangle must be contained in the first. This means, that if the two rectangles are exactly identical, we still say that the second rectangle is contained within the other.



(a) **(b)** **Figure 1**

Write a test program that creates a Rectangle2D object and tests various methods. For example, as a first test case, create a Rectangle2D object r1 with parameters 2, 2, 5.5 and 4.9, which represent xpos, ypos, width and height, respectively, displays its area and perimeter, and displays the result of r1.contains(3, 3) and r1.contains(new Rectangle2D(4, 5, 10.5, 3.2)).

The sample screen dialog for the above input is given below:

```
Enter the xpos, ypos, width and height of the rectangle: 2 2 5.5 4.9
The perimeter of the rectangle is 20.8
The area of the rectangle is 26.950000000000003
Rectangle [[2.0,2.0],width=5.5,height=4.9] contains point [3,3]
[[2.0,2.0],width=5.5,height=4.9] does not contain Rectangle
[[4.0,5.0],width=10.5,height=3.2]
```

Try it for two other cases.

What to submit:
ONE Zip file containing the following:

1. Bug.java (the tester main method is already included in Bug.java)
2. Point.java
3. PointDemo.java
4. Stock.java
5. StockDemo.java
6. Rectangle2D.java
7. Rectangle2DDemo.java
8. A text file containing sample outputs for all the programs.