ASSIGNMENT 4

COMP-202, Fall 2014, All Sections

Due: November 9^{th}, 2014 (23:59)

Please read the entire pdf before starting. You must do this assignment individually and, unless otherwise specified, you must follow all the general instructions and regulations for assignments. Graders have the discretion to deduct up to 20% of the value of this assignment for deviations from the general instructions and regulations. These regulations are posted on the course website. Be sure to read them before starting.

Question 1:	20 points
Question 2:	30 points
Question 3:	45 points
Question 4:	5 points
	100 points total

It is very important that you follow the directions as closely as possible. The directions, while perhaps tedious, are designed to make it as easy as possible for the TAs to mark the assignments by letting them run your assignment through automated tests. While these tests will not determine your entire grade, it will speed up the process significantly, which will allow the TAs to provide better feedback and not waste time on administrative details. Plus, if the TA is in a good mood while he or she is grading, then that increases the chance of them giving out partial marks. Marks can be removed if comments are missing, if the code is not well structured, or if your solution does not follow the assignment specifications.

Part 1 (0 points): Warm-up

Exercise 1: Write a class describing a Cat object. A cat has the following `attributes`: a name (String), a breed (String), an age (int) and a mood (String). The cat `constructor` takes as input a String and sets that value to be the breed. The mood of a cat can be one of the following: `sleepy, hungry, angry, happy, crazy`. Your **Cat** object contains a method called `talk()`. This method takes no input and returns nothing. Depending on the mood of the cat, it prints something different. If the cat's mood is `sleepy`, it prints *meow*. If the mood is `hungry`, it prints *RAWR!*. If the cat is `angry`, it prints *hssssss*. If the cat is `happy` it prints *purrrr*. If the cat is `crazy`, it prints a random String of between 10 and 25 characters (letters).

The cat `attributes` are all **private**. Each one has a corresponding **public** method called `getAttributeName()` (ie: `getName(), getMood()`, etc.) which returns the value of the `attribute`. All but the breed also have a **public** method called `setAttributeName()` which takes as input a value of the type of the attribute and sets the attribute to that value. Be sure that only valid mood sets are permitted. (ie, a cat's mood can only be one of five things). There is no setBreed() method because the breed of a cat is set at birth and cannot change.

Test your class in another file which contains only a main method. Test all methods to make sure they work as expected.

Assignment

Part 2:

Question 1: Merge (20 points)

For this problem, you will write the method **merge** of merge sort,

```
function mergeSort(a)
    if length(a) < 2 then
        return a
    end if
    middle = length(a)/2
    for all x in a up to middle do
        add x to left
    end for
    for all x in a after middle do
        add x to right
    end for
    left ← mergeSort(left)
    right ← mergeSort(right)
    result ← merge(left, right)
    return result
end function
```

Where, `a`, `left`, `right`, and `result` are arrays of type `String`. This method should take as input two sorted arrays of type `String` and return a sorted array containing all the elements from the two input arrays. In order to do this, you will use the compareTo() method[1]. Your method should run in linear time. That is, the number of comparisons[2] you make should be **linearly** dependent of the size of the two input arrays.

Please implement the merge method in the provided **MergeSort.java** file.

Question 2: Intro to Objects: Robot City (30 points)

For this question you have to get a robot (represented by a coloured triangle) from a position a to another position b using simple movement operations. During its movement, the robot should relocate some of the objects in the world around it. You are provided with an object-oriented model of the world using cities, robots and flashing lights. Instead of creating your own classes you will be using software that has already been developed by Byron Becker at the University of Waterloo.

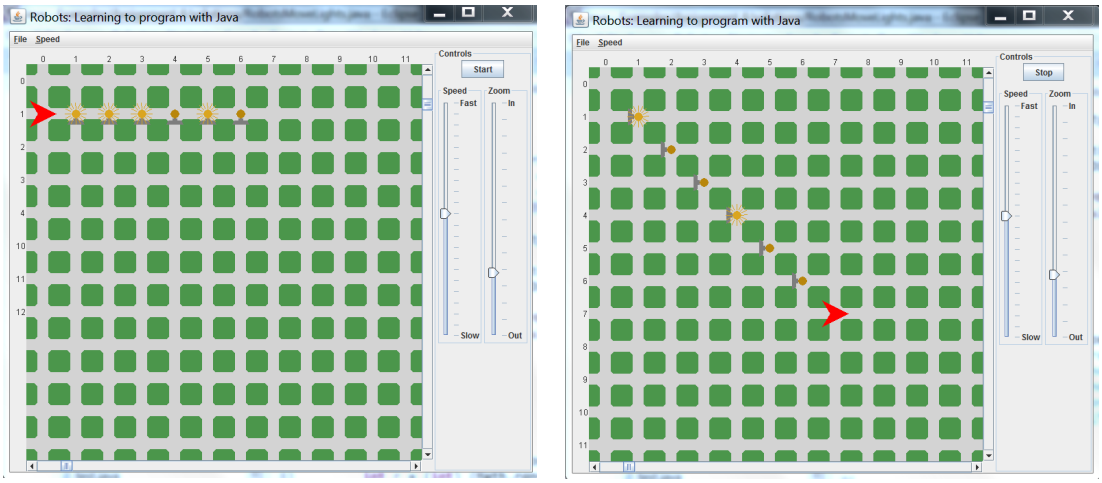
Before you begin, please review the application programming interface (API) for the classes provided in the `becker.robots` package. That is, you need to read the documentation in order to figure out which methods are available to you. To complete the assignment, you will need the following files.

- `a4-becker.jar` This is an archive file which contains all of the additional Java classes that you will need for the assignment.
- `a4-becker-docs.zip` This is an archive file which contains documentation about the classes included in the provided jar file.

All the above files can also be found at: `http://www.learningwithrobots.com/software/downloads.html` Additional information on this package can be found at: `http://www.learningwithrobots.com/doc/becker/robots/package-summary.html`. In order to get the provided file `RobotsMoveLights.java` to compile, and to use the robot objects, you need to tell your computer where to find the `a4-becker.jar` file.

[1] See: `http://www.tutorialspoint.com/java/java_string_compareto.htm` for details

[2] The number of times you call the compareTo() method



(a) Starting Positions

(b) Ending Positions

Figure 1: Result of the `moveToDiagonal()` method

- Dr Java Users: Go into Edit and then click on Preferences. Then click on Resource Locations. Under 'Extra Classpath' click on Add. You then have to find the location of the `a4-becker.jar` file in your computer, select it and the click on 'select'. If you did not move the file, it should be in your 'Downloads' folder. You then click on Apply, and then Okay. You may have to quit DrJava and reopen it for the changes to take effect.
- Eclipse Users: To do this in Eclipse, right click on your project folder (the first folder in the Package Explorer pane on the left hand side of the screen). Go down to 'Build Path' and then click on 'Add External Archives...'. You then have to find the location of the `a4-becker.jar` file, select it, and click 'open'. If you did not move the file, it should be in your 'Downloads' folder.

Now that you have set up the `a4-becker.jar` file, you should be able to compile and execute RobotsMoveLights.java. When you executing, it should display an image similar to Figure 1(a). Your task is to give the robot (the coloured triangle) a series of commands such that at the end your image will look like Figure 1(b).

You should not have to modify the main method in this file - just the `moveToDiagonal()` method. This method takes as input a reference to a `Robot` object. This robot is at an arbitrary intersection in the city and is facing East. In this world, **streets** run from West to East and **avenues** run from North to South. The further South a street is, the higher its street number will be. Similarly, the further East an avenue is, the higher its avenue number will be. This is further described by Figure 2.

Initially, an arbitrary number of flashers are placed at consecutive intersections along the street on which the robot is stationed. These flashers start one block East from the robot. The purpose of the method `moveToDiagonal()` is to have the robot pick up **all the flashers** and then position them such that they form a diagonal line from the North-West to South-East - starting from the original location of the West-most flasher. The final orientation of the flashers does not matter.

Some methods that might be useful to you are: `robot.move()`, `robot.turnLeft()`, `robot.pickThing()`, `robot.putThing()`, `robot.getDirection()`. You will likely need to use others as well. This list should just help get you started.

Your method must work regardless of the number of flashers (including zero flashers) or the specific starting location of the robot. You **may** assume that the robot starts adjacent to and facing the flashers, that they are on the same Street and that there are no gaps between flashers. Note that you may write small 'helper' methods if you wish. For example, there is no `robot.turnRight()` method. You may want to write your own method that does this. This is completely optional.

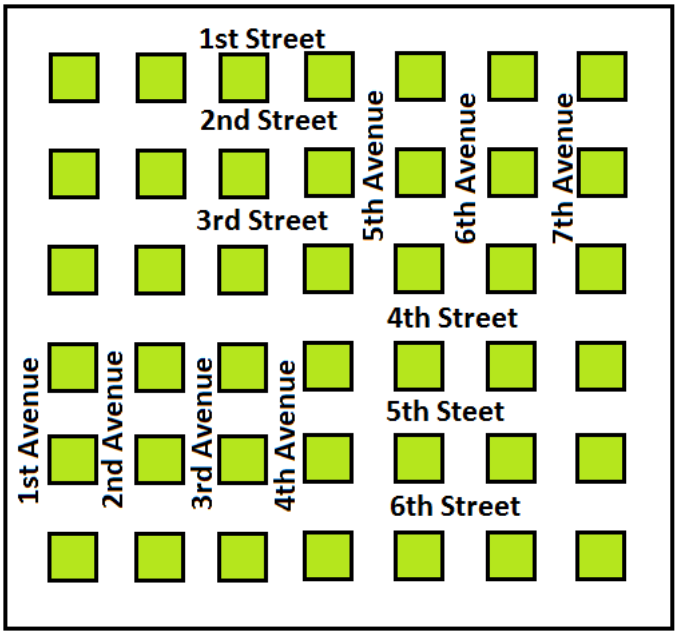
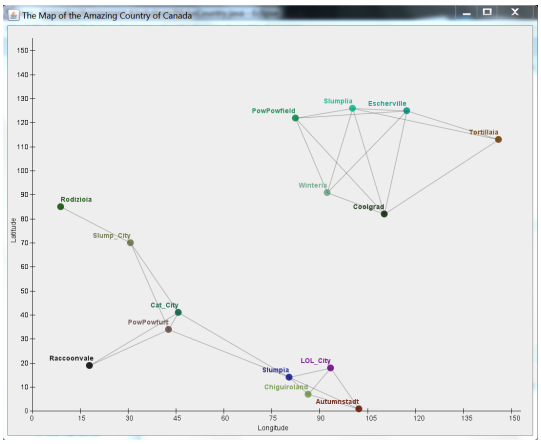


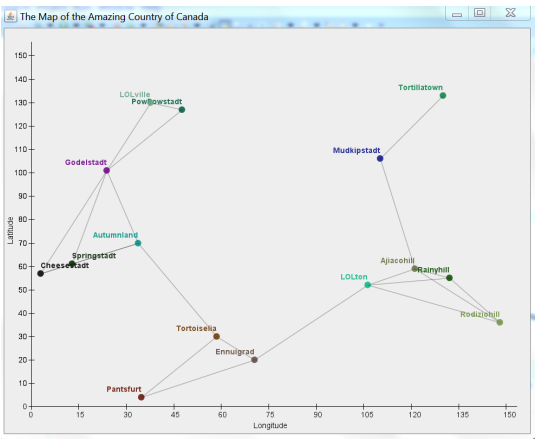
Figure 2: A Rough Sketch of the Robot's World

Question 3: Making your own Objects: Cities and Roads (45 points)

In this problem, you will implement an abstraction of a `Country`. The Country class contains an array of `City` objects. A City is defined by a two dimensional (x, y) position[3], a name, and its neighbouring cities. Once the country is populated with cities, your code will determine whether or not the all the cities in the country are connected[4]. For more information what it means for a *graph* to be connected, see the web links provided at the end of this question.[5] Using your random country, you will be able to create a map of it using the provided `CountryMap.java`. Figure 3 depicts completed countries. You



(a) A non connected sample country



(b) A connected sample country

Figure 3: Examples of connected and non-connected countries

[3] `Vector2D` class

[4] From any city is it possible to reach any other city by traveling along roads

[5] `http://mathworld.wolfram.com/ConnectedGraph.html`, `http://mathworld.wolfram.com/DisconnectedGraph.html`

have been given all the files you need to solve this problem. In order to create your random country, you will have to populate the provided files with class variables and methods. An overview is shown in Figure 4. The following expands further on the content of each class and their functionalities.

1. **Vector2D.java**: This file describes an abstraction of a two dimensional vector, a `Vector2D` object. This class has two *private* attributes of type `double`, *e.g.* an x position and a y position. These values are set by the constructor. This file also contains a `distance` method that takes as input another `Vector2D` object and returns the distance between the argument vector and *this* vector reference, recall $d = \sqrt{(v_x - w_x)^2 + (v_y - w_y)^2}$, where v and w are vectors and d is the euclidean distance that separates them.

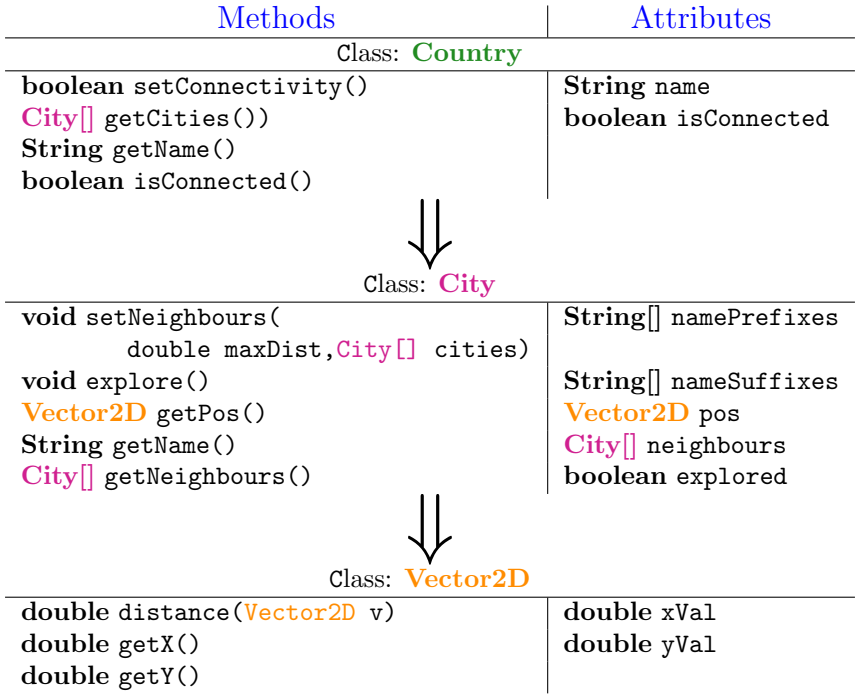


Figure 4: An overview of the Country, City and Vector2D classes

2. **City.java**: This class is an abstraction of a city. The file you have been given contains the attributes, as well as some empty methods. You have to complete the file as follows:

   - The constructor: The `City` constructor takes no input and initializes the `City` class by giving it a random name and a position. In order to create the random name, take an element from a random index of the `namePrefixes` array and an element from a random index of the `nameSuffixes` array. Concatenating the two together will give the String representing the city's name. We invite you (if you wish) to add more prefixes and suffixes to the arrays. The constructor should also initialize the city's position, a `Vector2D` object, by choosing two random integers between 0 and 150 and setting those values to be the `x` and `y` vector coordinates respectively using the `Vector2D`'s constructor.

   - The `City[] neighbours` attribute: Each city has an array of neighbouring cities. Two cities are deemed to be neighbours if the distance between them is less than some threshold. For the image in Figure 3, a maximum distance of 50 was used. You can play with this value. The maximum number of neighbours is, of course, the number of cities in the country minus one[6].

   - The `setNeighbours(double maxDist, City[] cities)` method populates the `neighbours` attribute. It takes as input an array of all cities in the country and determines which to add

[6] A city cannot be its own neighbour

as a neighbour of the current city. A neighbour is defined as any `City` whose distance to the current `City` is less than `maxDist`. In order to determine the size of the array, infer from the following: there are n `City` instances in a `Country` object, and a city cannot be its own neighbour. This method does not return anything.

- The `explore()` method performs a breadth-first exploration of the `Country`. For more information, see `http://en.wikipedia.org/wiki/Graph_traversal`. The pseudo-code is as follows:

```
function EXPLORE( )
    explored ← true
    for all City v in neighbours do
        if v was not explored then
            v.explore()
        end if
    end for
end function
```

  This algorithm starts at any `City`, and then explores all of the city's neighbours. We 'explore' a `City` by setting its class variable, `explored`, to be true. If any of the neighbours are not already explored, then we explore their neighbours, and so on. When this algorithm terminates, the `explored` parameter will be true for every `City` for which there exists a path between itself and the starting `City`.

3. The **Country.java** file. This file defines a class `Country`. A country has the following *private* attributes: a name, an array of `City` objects, and a boolean variable that notes whether or not the country is connected. You will also need:

   - A constructor: This takes as input the name of the country an integer `n`, representing the number of `Cities` that should exist in the `Country`, as well as an integer `maxDist` representing the maximum distance two neighbouring `Cities` can be from each other. The constructor sets the name and then initializes the `n` `City` objects, sets their neighbours, and populates the `City` array.

   - A method called `setConnectivity()`. This method should first call the `explore` method from the first `City` in the `Country` (`cities[0].explore()`). It should then iterate through all cities in the `Country` and verify the values of their boolean `explored` attributes to determine whether or not every city can be reached from every other city. It should return `true` if the cities are connected and `false` otherwise.

4. The **MakeCountry.java** file. Here is where your `main` method is. This method creates a `Country`, determines whether or not its graph is *connected* and then creates the diagram of the `Country`. You should also have a print statement that states whether or not the `Country` is connected.

Additional details:

- For more information on graph connectivity, please see:
  `http://mathworld.wolfram.com/ConnectedGraph.html`
  `http://mathworld.wolfram.com/DisconnectedGraph.html`

Question 4: Simple programming question (5 points)

The decimal number, $585 = 1001001001_2$ (binary), is palindromic in both bases. Find the sum of all numbers, less than two million + 1, which are palindromic in base 10 and base 2. (Please note that the palindromic number, in either base, may not include leading zeros.)[7]

[7] Your answer should be in a .txt file. You can add explanations and logic if you like, but do not include your code.

What To Submit

You have to submit one zip file with all your files in it to MyCourses under Assignment 3. If you do not know how to zip files, please ask any search engine or friends. Google might be your best friend with this, and a lot of different little problems as well.

`Merge.java`	The complete java code for question 2.
`RobotsMoveLights.java`	The complete code for question 3.
`Country.java`, `City.java`, `Vector2D.java`, `MakeCountry.java`	
`question4.txt`	Your answer for question 4.
The merge code for question 1	

Marking Scheme

Up to 20% of points can be removed from each question for bad indentation of your code as well as omitted comments, missing files, or just generally not following instruction. Marks will be removed as well if the class names are not respected.

Question 1

The algorithm uses n steps	10	points
Array is sorted	10	points
	20	**points**

Question 2

The method works regardless of the robot's initial position	5	points
The robot picks up the flashers	7	points
The method works regardless of the number of flashers	8	points
The robot deposits the flashers in the right place	10	points
	30	**points**

Question 3

The distance is checked using the city's `Vector2D` instance	5	points
The attributes are all private	10	points
`MakeCity` creates a random city	10	points
`City` stores its neighbours cities	10	points
The country is properly explored and checked for connectivity	10	points
	45	**points**

Question 4

right answer	5	points
	5	**points**