CS2030 Programming Methodology II

Semester 4 2018/2019

javac; java; codecrunch

Prerequisites

Assume that students are already familiar with:

- Basic programming

Learning Objectives

After completing this lab, students should:

- be familiar with compiling and running Java programs from the command line
- be familiar with the concept of standard input and standard output; how to redirect the content of a file to a standard input; and how to print to standard output
- be more comfortable with basic Java syntax and semantics, specifically with:
    - adding methods into existing classes
    - invoking methods from the class
    - declaring and using arrays, primitive types, and objects
    - using `if-else` and `for-loop` statements
    - printing to standard output
    - `this` keyword
- be more comfortable reading Java API documentation and find out what are the methods available, and what are the arguments and returned types
- see an example of how class `Scanner` is used
- appreciate how encapsulation of class `Point` and `Circle` allows one to reason about higher-level tasks without worrying about lower level representation
- be familiar with using CodeCrunch

Goals

We have an implementation of `Point` in *Lecture 01* and an implementation of Circle in *Lecture 02*. The goal of this lab is to extend the capability of `Point` class to create a midpoint, compute angle between points, and to move point; use the `Point` class to create a new type of `Circle` class and augment it with new capability; and to solve the problem of maximum disc coverage using both `Point` and `Circle`.

Grading

This lab is not graded and it will not be used for the project. However, the concepts used here will be beneficial for our project. Unless you are already familiar with the learning objectives above, you are highly encouraged to do this lab and submit to CodeCrunch.

CS2030 Programming Methodology II

Semester 4 2018/2019

javac; java; codecrunch

1. Augmenting the Point Class

Augment the `Point` class with the following public methods and constructors. The static methods provided by `java.lang.Math`[1] may be very useful.

1.1 Static Method to Construct Mid Point

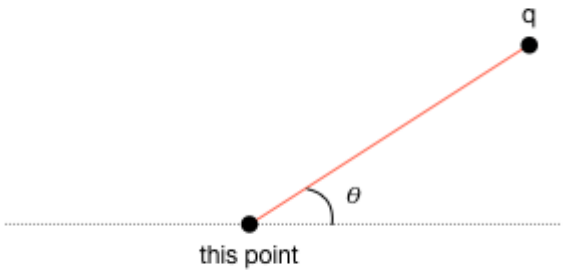
Implement a method with the following function signature. Given two points `p` and `q`, create and return the midpoint of `p` and `q`.

```
public static Point midpoint(Point p, Point q);
```

1.2 Angle Between Points

Implement a method with the following function signature. Given another point `q`, compute and return the angle between the current (`this`) point and point `q`. Consider the figure below, the angle returned should be θ. We can use the function `atan2()` to compute the angle.

```
public double angle(Point q);
```



Consider the following code fragment:

```
1  Point p = new Point(0, 0);
2  p.angle(new Point(1, 1));
```

The code fragment should return `0.7853981633974483` which is equivalent to $\pi/4$. The following code `p.angle(new Point(1, 0));` should return `0.0`. Please refer to the footnote to find more information about `java.lang.Math` functions including `atan2`. The information about `atan2` is reproduced for our convenience. Note that for future labs, midterm, or final, the information given may not be this specific.

`static double`	**`atan`**`(double a)`	Returns the arc tangent of a value; the returned angle is in the range *-pi*/2 through *pi*/2.
`static double`	**`atan2`**`(double y, double x)`	Returns the angle *theta* from the conversion of rectangular coordinates (x, y) to polar coordinates (r, *theta*).

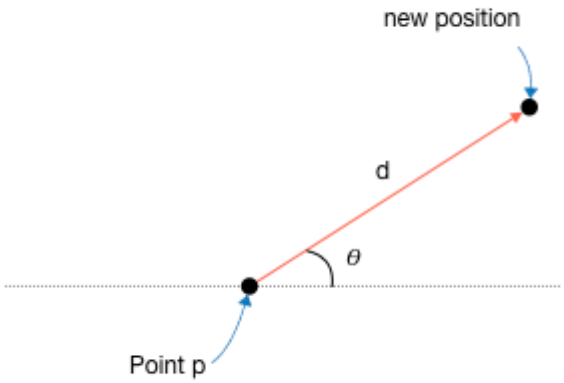
[1] https://docs.oracle.com/javase/9/docs/api/java/lang/Math.html

1.3 Moving a Point

Implement a method with the following function signature. Given an angle `theta` (in radian) and a distance `d`, we should move the point by the given distance at the direction of the given angle.

```
public void move(double theta, double d);
```

Consider the figure below. The new point should have the coordinate $(x + d\cos\theta, y + d\sin\theta)$. We need to use sine and cosine. Given the code `p.move(p.angle(q), p.distance(q));` the point `p` should coincide with the point `q`.



2. Augmenting the Circle Class

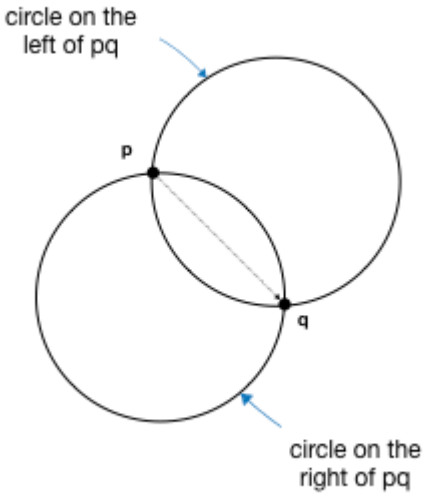
Augment the `Circle` class with the following methods and constructors.

2.1 Constructor

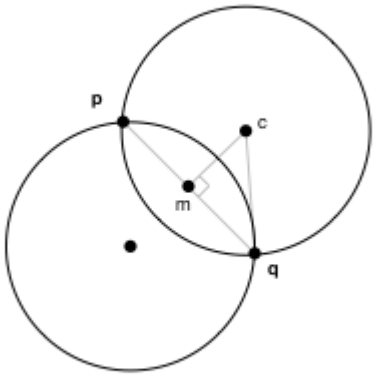
Implement a constructor with the following function signature. Given two points `p` and `q`, as well as the radius `r`, create and return a circle that passes through both `p` and `q`, with a radius `r`.

```
public Circle(Point p, Point q, double r);
```

There are two possible circles as we can see from the figure below. From the figure, if we *walk* in a clockwise manner from point `p` to `q`, we will create the circle on the left of `pq`. The other circle will be considered by walking either anti-clockwise or walking clockwise from point `q` to `p`. For our current problem, we will only consider the circle on the left generated by `new Circle(p, q, r)` as we can always generate the circle on the right by executing `new Circle(q, p, r)`.



If the distance between `p` and `q` is no greater than 2x`r`[2]. To find the center `c` of the new circle, we can first find the midpoint `m` of the line `pq`. We can then find the length of line `mc` and the angle between `m` and `c` using the methods in `Point` that we have written. We also know that the length `cq` is the radius `r` as we can see from the figure below.



Consider another case where the distance between `p` and `q` is either greater than 2x`r` or zero. Such `Circle` objects are invalid. In such cases, we should return a `Circle` with radius `Double.NaN`[3] and center point (0,0).

2.2 Valid Circles

Implement a method with the following function signature. The method returns `true` if the `Circle` object is valid, otherwise it returns `false`. We may either use `Double.isNaN` to check for `NaN` value or use another field.

```
public boolean isValid();
```

[2] If the distance between `p` and `q` is *exactly* 2x`r`, then the two circles are one and the same.
[3] https://docs.oracle.com/javase/9/docs/api/java/lang/Double.html

CS2030 Programming Methodology II

Semester 4 2018/2019

javac; java; codecrunch

3. Maximum Disc Coverage

We are now going to use the `Circle` and `Point` class to solve the maximum disc coverage problem. In this problem, we are given a set of points on a 2D plane and a unit disc (i.e., a circle of radius 1). We want to place the disc so that it covers as many points as possible. Maximum disc coverage problem asks what is the maximum number of points that we can cover with the disc at any one time?

We will use the following simple (non-optimal) algorithm. First, some observations:

- A disc that covers the maximum number of points must pass through at least two points.
- For every pair of points that is of no more than distance `2.0` away from each other, there is at most two unit discs that have their perimeter passing through the two points. You have written a constructor to find such circles.

Ergo, the algorithm simply goes through every pair of points, and for each circle that passes through them, count how many points are covered by each circle. Let's rewrite that in a more structured way called pseudocode:

1 For every pair of points
   a. Create two circles
   b. For each of the two circles:
      i. *Count number of points covered*
      ii. If the number of points is greater than current maximum, update the maximum
2 Return the maximum

Notice that the step 1b(i) is complex. This is a good candidate for *modularizing* your code. The essence of modularity is ***wishful thinking***. For instance, it would be nice if we have a function that can count the number of points covered by a circle.

3.1 Skeleton File

The skeleton of main class, called `LabZero.java` is provided. The file `MaxDiscCover.java` is also given and this is where we are going implement the maximum disc coverage solution. Place these two files in the same directory as `Circle.java` and `Point.java`.

3.2 Input and Output

The skeleton file `LabZero.java` reads a set of points from the standard input in the following format:

- The first line is an `integer`, indicating the number of points n where $n \geq 3$ in the file.
- The next n lines contains the x and y coordinates of n points, one point per line. Each line has two `double`s separated by a space. The first `double` is the x coordinate and the second `double` is the y coordinate.

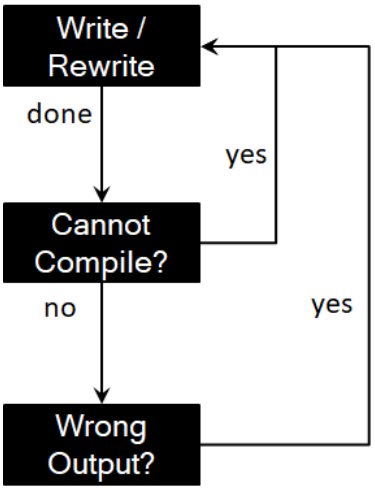
We assume that the format of the input is always correct as specified and there is always at least two points with distance less than `2.0` between them. Study the way we perform input reading using the `Scanner` class[4]. We use the `System.out.println` method for our printing[5].

[4] https://docs.oracle.com/javase/9/docs/api/java/util/Scanner.html
[5] https://docs.oracle.com/javase/tutorial/essential/io/formatting.html

3.3. Process

Complete the program by implementing the maximum disc coverage algorithm above. Print the maximum number of points covered to standard output. You may add additional methods and fields for `Point` and `Circle` if necessary.

3.4 Compilation and Executing



The programming workflow can be described by the diagram on the right. We first start by writing the program using IDE of our choice. Once the program is written, we save the program into a file. For instance, we can save the program into a file called `Prog.java`.

We can compile the program using the command `javac Prog.java`. If you have written your program in a different name, simply change `Prog.java` to the name of your file. If there are no message, it means the compilation is successful. We have a mantra, "*no news is good news*".

We can also compile multiple Java file with the command `javac Prog1.java Prog2.java ...`. We may need to specify all the Java file necessary. Alternatively, if we are in a folder that contains all the necessary Java file, we can simply issue the command `java *.java` to compile all files with extension `.java`.

If the compilation is unsuccessful, simply go back to the top and rewrite the program. On the other hand, if the compilation is successful, we will have a file called `Prog.class`. This is Java bytecode. We can execute this Java bytecode using the command `java Prog`. Note that we use the class name instead of the file name. Common mistakes include `java Prog.java` and `java Prog.class`.

If the execution produces the wrong output, simply go back to the top and rewrite the program. However, if the execution *looks like* it is producing the correct output, it may still produce incorrect output due to *invisible characters* such as whitespaces and/or newlines. The next step will show how we can have more confidence in the correctness of our program.

3.5 Comparing with Test Cases

In every lab session, we will provide *public* test cases to check for correctness. At the very least, it should verify the correctness of the output format. To run the program such that the given test case is provided as input, we can use the command `java Prog < Test1.in`. The `<` operator redirects the file into the standard input. It is as if there is a ghost typist that types all the content of the file when running the program.

Since we can redirect input, we can also redirect the output for comparison. The command for comparing the output with the given test case is `java Prog < Test1.in | diff - Test1.out`. diff is a Unix program to check for differences. Again, "*no news is good news*".

In Windows, we cannot write this in one line. Instead, we use multiple lines to achieve the same effect.

```
java Prog < Test1.in > Test1.tmp
```

```
FC Test1.tmp Test1.out
```

FC is a Windows program to check for file differences. Using this method we can be more confident in the correctness of our program especially if our program passes all the *public* test cases. We can then submit to CodeCrunch for *private* test cases which we will not release. **You should test your code with your own data.**