EECS2030 Lab 01

Thu Sep 15, Mon Sep 19, and Tue Sep 20

Due dates:
Section A Mon Sep 26 before 23:59
Section E LAB1 Mon Sep 26 before 23:59
Section E LAB2 Tue Sep 27 before 23:59

You may work in groups of up to four students from the *same* lab section. If you choose to work in a group, then please pay close attention to the submission instructions for group work.

Please wait until Monday September 19 to submit your work. We need to set up some software infrastructure to handle the submissions so that you receive immediate feedback on your work.

Introduction

The goals of this lab are to:

- learn about the Java style rules you are expected to use in this course
- implement a small utility class
- implement a small regular class
- test your implementations using JUnit
- use your classes to produce pictures of shapes produced by a famous toy called the Spirograph™.

Style Rules

The style rules are not overly restrictive in EECS2030.

1. Your programs should use the normal Java conventions (class names begin with an uppercase letter, variable names begin with a lowercase letter, **`public static final`** constants should be in all caps, etc.).

---

2. In general, use short but descriptive variable names. There are exceptions to this rule; for example, traditional loop variables are often called i, j, k, etc.

Avoid very long names; they are hard to read, take up too much screen space, and are easy to mistype.

---

3. Use a consistent indentation size. Beware of the TAB vs SPACE problem: Tabs have no fixed size; one editor might interpret a tab to be 4 spaces and another might use 8 spaces. If you mix tabs and spaces, you will have indenting errors when your code is viewed in different editors.

---

4. Use a consistent brace style:

```
// left aligned braces

class X
{
  public void someMethod()
  {
    // ...
  }

  public void anotherMethod()
  {
    for (int i = 0; i < 1; i++)
    {
      // ...
    }
  }
}
```

or

```
// ragged braces

class X {
  public void someMethod() {
    // ...
  }

  public void anotherMethod() {
    for (int i = 0; i < 1; i++) {
      // ...
    }
  }
}
```

---

5. This one always causes problems for students. Insert a space around operators (except the period ".").

The following is

```
    // some code somewhere
    boolean isBetween = (x > MIN_VALUE) && (x > MAX_VALUE);
    int someValue = x + y * z;
```

much easier to read than this

```
    // AVOID DOING THIS

    // some code somewhere
    boolean isBetween=(x>MIN_VALUE)&&(x>MAX_VALUE);
    int someValue=x+y*z;
```

---

6. Avoid using "magic numbers". A magic number is a number that appears in a program in place of a named constant. For example, consider the following code:

```
int n = 7 * 24;
```

What do the numbers **7** and **24** mean? Compare the code above to the following:

```
final int DAYS_PER_WEEK = 7;
final int HOURS_PER_DAY = 24;
int n = DAYS_PER_WEEK * HOURS_PER_DAY;
```

In the second example, the meaning of **7** and **24** is now clear (better yet would be to also rename **n**).

Not all numbers are magic numbers. You can usually use the values **1** and **2** without creating a named constant. If you ever find yourself doing something like:

```
final int TEN = 10;
```

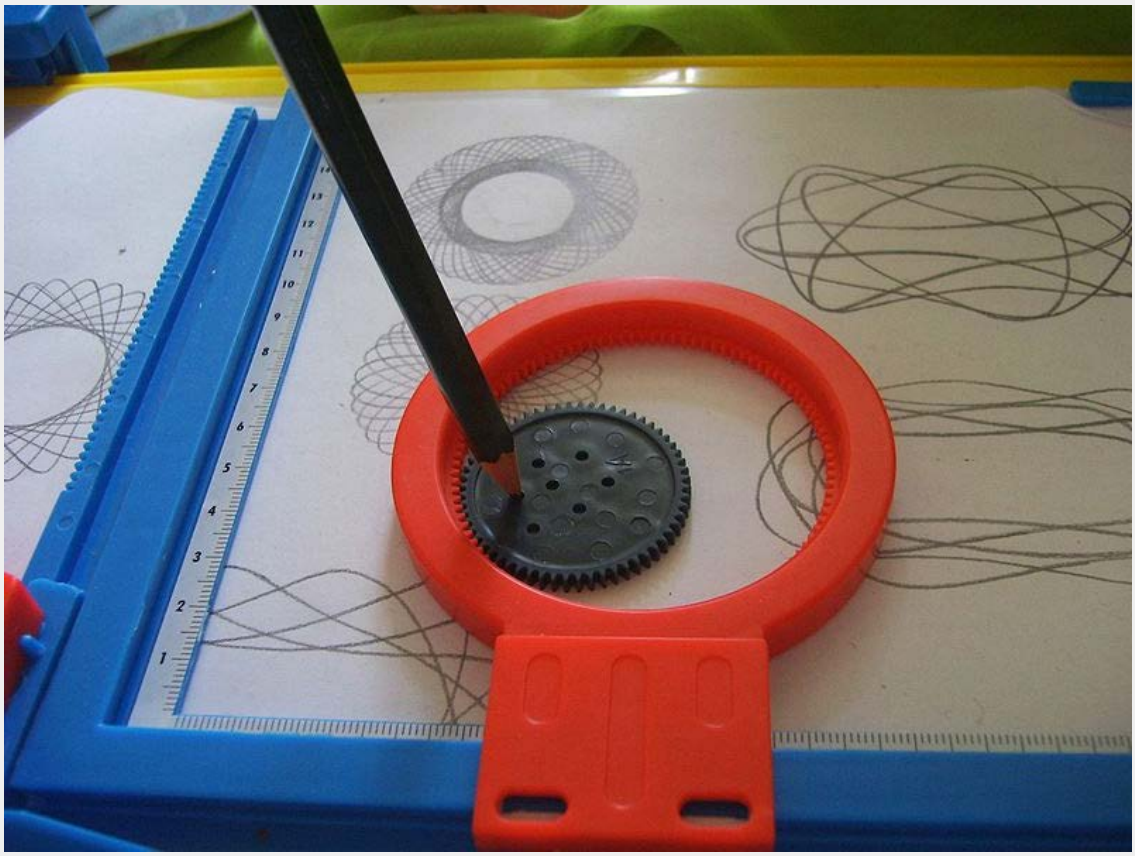
then you are probably better off using **10** and explaining its meaning in a comment.

---

7. A good IDE (integrated development environment) such as eclipse will correct many style errors for you. In eclipse, you can select the code that you want to format, right click to bring up a context menu, and choose *Source -> Format* to automatically format your code.

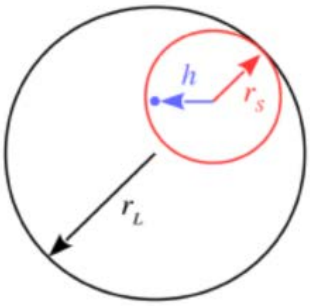
Spirograph™

Spirograph™ is a famous toy that lets the user draw shapes called hypotrochoids (and epitrochoids) using a colored pencil and special wheels (see the image below by Kungfuman)



The user spins the smaller wheel inside the larger wheel using a colored pencil. Usually, the smaller wheel must travel around the larger wheel many times before the pattern is complete. The curves generated by this process are called hypotrochoids. Here is a video showing a similar toy in action.

For our purposes, we will describe hypotrochoids using the following equations:



radius of the larger wheel (shown in black):

$$r_L = 1$$

radius of the smaller wheel (shown in red):

$$r_S$$

where $0 <= r_S <= r_L$

x and y coordinates of the curve:

$$x = (r_L - r_S)\cos(\theta) + h\cos\left(\frac{r_L - r_S}{r_S}\theta\right)$$

$$y = (r_L - r_S)\sin(\theta) - h\sin\left(\frac{r_L - r_S}{r_S}\theta\right)$$

where θ is an angle in degrees, and h is the distance between the pencil tip (shown in blue) and the center of the smaller wheel. Because the pencil tip sits inside the smaller wheel, it must be the case that $0 <= h < r_S$.

Getting started

To get started, you should do the following in eclipse:

1. Create a new Java Project (perhaps called **lab1**)
2. Create a new package called **eecs2030.lab1**
3. In the package **eecs2030.lab1** create a new Java utility class named **SpiroUtil** (but don't add any new code to the class yet).
4. In the package **eecs2030.lab1** create a new Java class named **Point2**. Add code to your class so that it appears like so:

```
package eecs2030.lab1;

/**
 * A simple class for representing points in 2D Cartesian
 * coordinates. Every Point2D instance has an
 * x and y coordinate.
 *
 * @author EECS2030 Fall 2016
 *
 */
public class Point2 {

    private double x;
    private double y;

    public double getX() {
        return this.x;
```

```
    }

    public double getY() {
        return this.y;
    }

}
```

Inspect the API

1. Study the API for **SpiroUtil**. Recall that the API for a class typically shows all of the **public** features of a class. You should see that **SpiroUtil** has one **public static final** field and three **public** methods. Make sure that you can find the value of the field and that you understand the API for each method.

Add the fields and methods to SpiroUtil

1. To the class **SpiroUtil**, add the **public static final** field; make sure that it is spelled correctly and has the correct value.
2. Add the following temporary implementations of the three methods (save your work when you've added the methods):

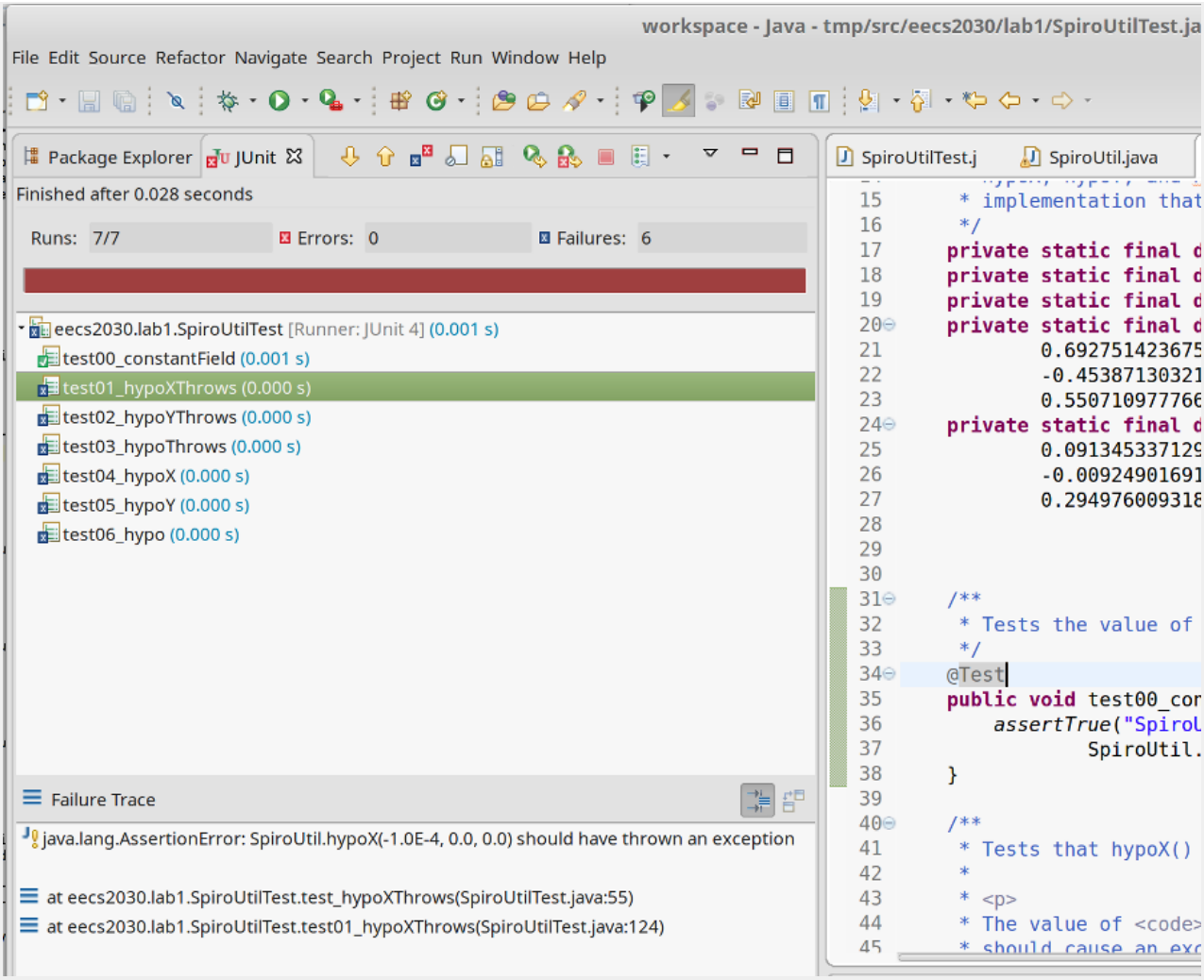
```
public static double hypoX(double wheelRadius, double pencilRadius, double degrees) {

  return 0.0;
}


public static double hypoY(double wheelRadius, double pencilRadius, double degrees) {

  return 0.0;
}


public static Point2 hypo(double wheelRadius, double pencilRadius, double degrees) {

  return new Point2();
}
```

Notice that method implementations are clearly incorrect (because they either always return zero or a default **Point2** reference); however, they do compile. Such temporary method implementations are called *method stubs*. Stubs are useful in test-driven software development because you can write unit tests for the methods that will compile and run; obviously, the unit tests will fail, but you can now use the unit tests to help guide the development of the methods.

Add a unit tester

Unit testing is the act of testing the smallest functional units of a program. For most Java programs, the smallest functional units are the methods of the classes. JUnit is a unit testing framework that helps programmers unit test their programs. In this lab, you are provided with a simple unit tester:

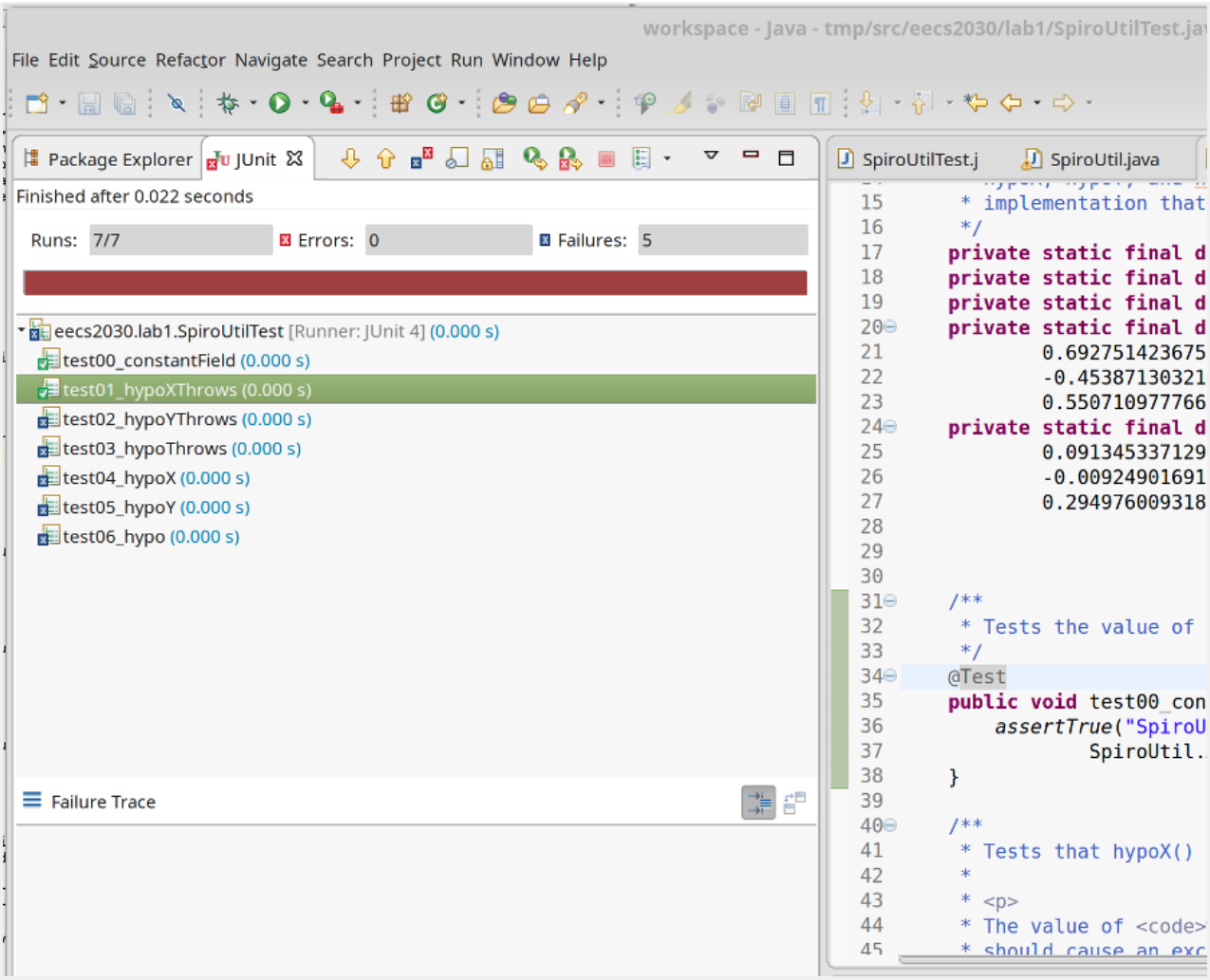
1. Create a new class for the unit tester. Do this by right-clicking on **SpiroUtil.java** in the *Package Explorer* on the left-hand side of the eclipse IDE, and selecting *New -> JUnit test case*. A dialog window will appear with the title *New JUnit Test Case*; the name of the test case should be **SpiroUtilTest**. Click the *Finish* button.
2. Another dialog window should appear with the message *JUnit 4 is not on the build path. Do you want to add it?*. You do want to add the JUnit 4 library to the build path. Click the *OK* button to do so.
3. Replace the automatically created contents of **SpiroUtilTest** with the contents of this file. Use cut-and-paste, don't try typing it all in!
4. Run the unit tester by clicking the green play button in the eclipse tool bar (under the main menu). You should see the following:



5. Notice that 6 of 7 tests have failed. If you click on a test, you can see why the test failed. In the figure above, we see the message *"java.lang.AssertionError: SpiroUtil.hypoX(-1.0E-4, 0.0, 0.0) should have thrown an exception"*. From the message we can infer that the test failed because our implementation of **hypoX** does not throw an exception when the test value for the **wheelRadius** is negative (**-0.0001** in this case).
6. Go back and look at the API for **SpiroUtil**. In the **hypoX** method, we see that the method should throw an exception when either **wheelRadius** or **pencilRadius** has an illegal value. Fix the method by adding the following code (and then save your work):

```
public static double hypoX(double wheelRadius, double pencilRadius, double degrees) {
    if (wheelRadius < 0.0) {
        throw new IllegalArgumentException("wheel radius is negative");
    }
    if (wheelRadius > SpiroUtil.BIG_WHEEL_RADIUS) {
        throw new IllegalArgumentException("wheel radius is greater than SpiroUtil.BIG_WHEEL_RADIUS");
    }
    if (pencilRadius < 0.0) {
        throw new IllegalArgumentException("pencil radius is negative");
    }
    if (pencilRadius > wheelRadius) {
        throw new IllegalArgumentException("pencil radius is greater than wheel radius");
    }


    return 0;
}
```

7. Go back to the unit test file **SpiroUtilTest** and run it again. This time you should see the following:



8. Notice that the test which previously failed now passes (indicated by the green check mark).
9. Complete the remaining methods. Every time you make a change to a method, save your work and run the unit tester. Keep making changes until all of the tests *except* the last one passes. You need to complete the **`Point2`** implementation before the last test passes.

Complete the `Point2` class

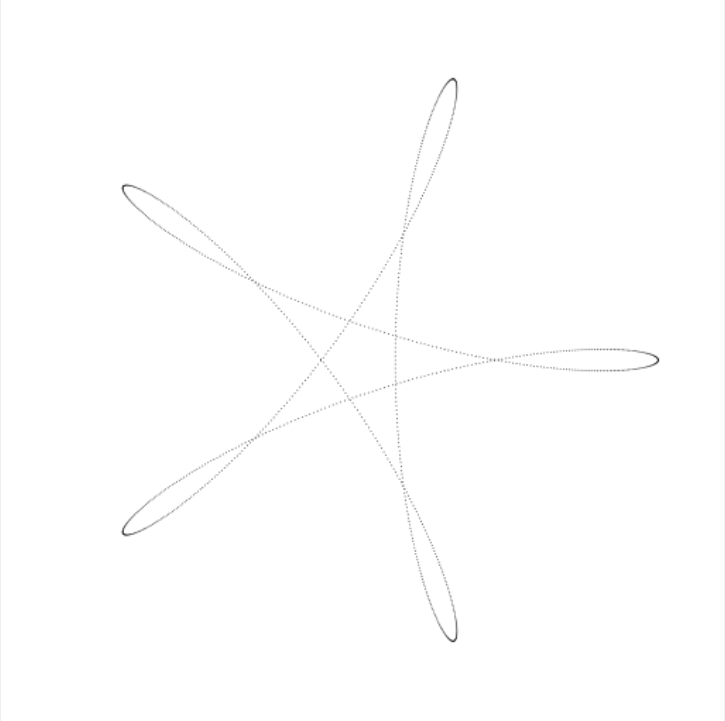
1. Add a new JUnit tester for the **`Point2`** class. Do this by right-clicking on **`Point2.java`** in the *Package Explorer* on the left-hand side of the eclipse IDE, and selecting *New -> JUnit test case*. A dialog window will appear with the title *New JUnit Test Case*; the name of the test case should be **`Point2Test`**. Click the *Finish* button.
2. Replace the automatically created contents of **`Point2Test`** with the contents of this file.
3. Study the API for **`Point2`**. You should see that **`Point2`** has two constructors and six **`public`** methods (**`getX`** and **`getY`** are already complete and correct). Make sure that you understand the API for each constructor and method.
4. Complete the constructors and methods using the unit tester to help you.

Make some pictures

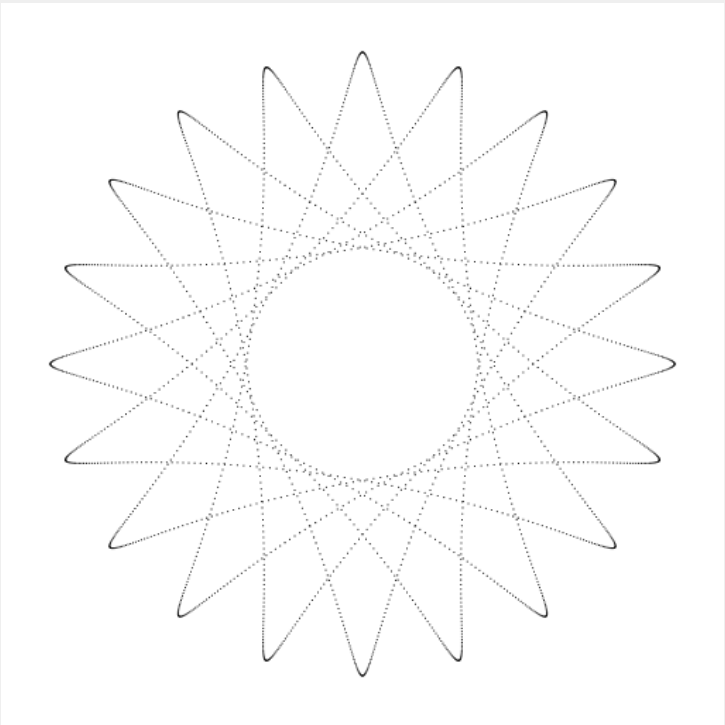
1. Download this jar file and add it to your project:
    1. select the **`Project`** menu
    2. select **`Properties`**
    3. on the left side of the dialog that appears, select **`Java Build Path`**
    4. on the right, select the **`Libraries`** tab
    5. click the **`Add External JARs...`** button
    6. add the jar file you downloaded
2. Add a new class named **`Spirograph`** to the **`eecs2030.lab1`** package. Replace the automatically created contents of **`Spirograph`** with the contents of this file.
3. Run the program entering values for the wheel radius and pencil radius when prompted.

Here are some sample solutions using different values for the command line arguments:

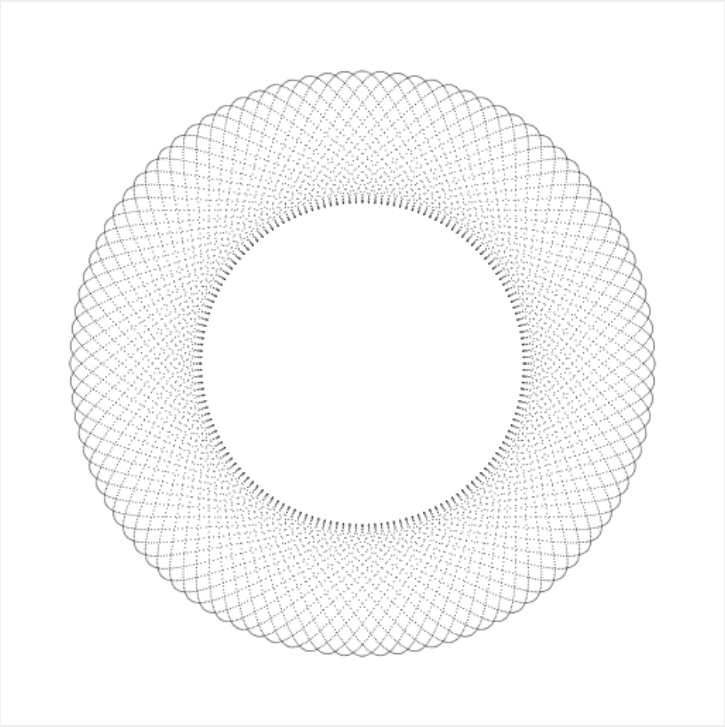
wheel radius: 0.6, pencil radius 0.5



wheel radius: 0.35, pencil radius 0.3



wheel radius: 0.31, pencil radius 0.2



Submit your work

Please wait until Monday September 19 to submit your work. We need to set up some software infrastructure to handle the submissions so that you receive immediate feedback on your work.

Submit for students NOT working in a group

If you are *not* working in a group, submit your solution using the **`submit`** command. Remember that you first need to find your workspace directory, then you need to find your project directory. In your project directory, your files will be located in the directory **`src/eecs2030/lab1`**

```
submit 2030 lab1 SpiroUtil.java Point2.java
```

Submit for students working in a group

If you are working in a group, nly one student from the group needs to submit the lab work. Create a plain text file named **`group.txt`**. You can do this in eclipse using the menu *File -> New -> File*. Type your login names into the file with each login name on its own line. For example, if the students with login names **`rey`**, **`finn`**, and **`dameronp`**, worked in a group the contents of **`group.txt`** would be:

```
rey
finn
dameronp
```

Submit your solution using the **`submit`** command. Remember that you first need to find your workspace directory, then you need to find your project directory. In your project directory, your files will be located in the directory **`src/eecs2030/lab1`**

```
submit 2030 lab1 SpiroUtil.java Point2.java group.txt
```

Submit from outside of the lab

The process for submitting from outside of the Prism lab involves the following steps:

1. transfer the files from your computer to the undergraduate EECS server red.eecs.yorku.ca
2. remotely log in to your EECS account
3. submit your newly transferred files in your remote login session
4. repeat Steps 1 and 3 as required

Windows users will likely need to install additional software first. Mac users have all of the required software as part of MacOS.

Detailed instructions are here.