CS 455 Programming Assignment 1

Fall 2023 [Bono]
Due date: Wednesday, Sep. 13, 11:59pm Pacific Time

Introduction

In this assignment you will write a graphics-based program to draw a "rectangular" spiral, centered in the window. This is based on Programming project 6.21 in our textbook.

This assignment will give you practice with creating and implementing classes, using loops and conditionals, doing console-based IO, drawing to a graphics window, and writing a unit test. Also you'll get practice in general program development. Please read over the whole assignment before beginning: in particular, do not wait until the assignment is due to read about submitting it: you'll want to do your first submit before the final deadline, because when you submit it we do some checks on your code, and you will be able to resubmit if you fail those checks (but not if you are out of time).

This document may seem voluminous, because it includes some instructional material, and hints on how to do certain things. You'll need to read it or parts thereof more than once, possibly skimming some parts on the first reading. To help you find things as you are working on the assignment, here is a table of contents for the main sections:

Table of Contents

Resources

- Horstmann, Section 2.9, 2.10, 3.8, How-to 3.2 Programs that draw stuff
- Horstmann, Chapter 3, Implementing Classes
- Horstmann, Section 4.3.1, Reading input
- Horstmann, Chapter 5, Decisions
- Control Structures Videos

The programming environment for this assignment

Reminder: The first time you access the assignment on Vocareum, you will need to go through the link on d2l (More detailed directions about this were given on Lab 1). Any subsequent accesses to the same assignment can be made via labs.vocareum.com (or through the d2l link).

In the normal Vocareum configuration, you have a Linux terminal, but no way to run a program with a graphical user interface (GUI). For this assignment we are using a different Vocareum configuration that will allow you to open multiple windows, including a separate one to run your GUI program. With this configuration, when you start up Vocareum for this assignment, it will not start up a terminal in the workbench window (i.e., the usual one you use), but you use a virtual Linux desktop instead.

How to start up a virtual Linux Desktop in Vocareum
The way you get to a virtual linux desktop in this assignment is to go to a menu that's on the upper right of the workbench window: choose Actions--> Applications --> Desktop.

That will open a linux desktop in another tab in your browser. If it starts with a pop-up dialog, choose "Use default configuration." There are a few ways to open a terminal window in this desktop. You can use the Applications menu at the top left of the screen, and choose "Terminal Emulator". Or you can right click anywhere on the desktop, and choose "Open Terminal Here".

Warning: depending on how you started up the terminal, it might not start out in your home directory (i.e., "work"), but rather starts in the root directory ("/") or somewhere else. So the first thing you should do is

```
cd
```

to get into your home directory. (One way to check if you are in your home directory is you will see the ~ (tilde) right before the $ in the Linux shell prompt.)

Your home directory will be populated with the starter files we are providing you. Part of what we provided is source code for a complete sample Java GUI program there, so you can try out compiling and running such a program in this environment before you write code for your own program. Compile and run this program:

```
javac CarViewer.java
java CarViewer
```

More about this car example (from Section 3.8 of the textbook) later.

You can switch between these two tabs in your browser to switch between editing (normal Vocareum window), and compiling and running (Linux desktop). To make it easier to see your compile errors at the same time as you view your source code, you can put the Vocareum tab in a different browser window altogether.

Another option with the desktop is to use one of the other source code editors available within the desktop itself. I saw `emacs` and `vim` (Rt-click on desktop-->Applications-->Accessories). I'm not sure how fast these work on this platform, so if you end up using one of these, let me know how it goes. (I only opened emacs there briefly once; it started up pretty fast, so that's a good sign.) Both emacs and vim are a little different than other editors you are used to, so you probably would want to take a look at an online tutorial on the web before using them. Eclipse is also available there; it may be somewhat slower than running it locally.

You can disconnect from the Desktop by closing the tab, or in the main Vocareum window (upper right) do: Actions-->Applications-->Stop App. Then you can restart later the same way you did the first time.

Using another IDE for this assignment
If you don't want to use Vocareum and its Linux desktop as your development environment, you can use another IDE running locally on your own machine.

If you choose go this other route, you would do the following:

1. Install and try out another IDE. There are a few tutorials in the Java section of the Documentation page to help you get started with IntelliJ or Eclipse.
2. Download the starter files for the assignment into a folder on your laptop (e.g., call it `pa1`). The easiest way to do this is, from the Vocareum PA1 assignment workbench do: Actions-->Download starter code (menu on the upper right).
3. Get your program working, tested, and completely documented on your laptop IDE, and when you are ready to turn it in . . .
4. Upload your complete program into Vocareum. In the Vocareum workbench, click Upload (button in the upper-left area), and you will get a file browser that will allow you to select multiple files to upload. You will only need the source code files (and README), no `.class` or project files. *Do not wait until the final due date/time is imminent before uploading and testing it on Vocareum.*
5. Compile and completely test your program again on Vocareum. You want to make sure what you wrote will run in the Vocareum environment, because it is the grading environment. This also helps you to make sure that you are submitting the correct version of your solution.
6. **Submit your assignment on Vocareum.** Don't forget this step! Please read the details about submitting this in the section on it at the end of this document.

The assignment files

The files in **`bold`** below are ones you create and/or modify and submit. The ones *not* in bold are files you will use, but that you should not modify. The ones with a * to the left are starter files we provided.

- * `CarViewer.java`, `CarComponent.java`, and `Car.java`. The code for the example in Section 3.8 of the textbook. For more about why these are in the starter files, see the section on The Car example.
- * **`SpiralGenerator.java`** Your `SpiralGenerator` class. The public interface is provided. You will be completing the implementation and a test driver for it.

- **`SpiralViewer.java`** Your `SpiralViewer` class. You create this file and class.
- **`SpiralComponent.java`** Your `SpiralComponent` class. You create this file and class.
- **`SpiralGeneratorTester.java`** Your unit test program (a.k.a., test driver) for your `SpiralGenerator` class. You create this file and class.
- * `SpiralViewer.list` A list of the .java files for compiling the `SpiralViewer` program. For more information about this, see the subsection on compiling multi-file programs.
- * **`README`** for more about what goes in this file, see the section on README file. Before you start the assignment please read the following statement which you will be "signing" in the README:

> "I certify that the work submitted for this assignment does not violate USC's student conduct code. In particular, the work is my own, not a collaboration, and does not involve code created by other people or AI software, except for the resources explicitly mentioned in the CS 455 Course Syllabus. And I did not share my solution or parts of it with other students in the course."

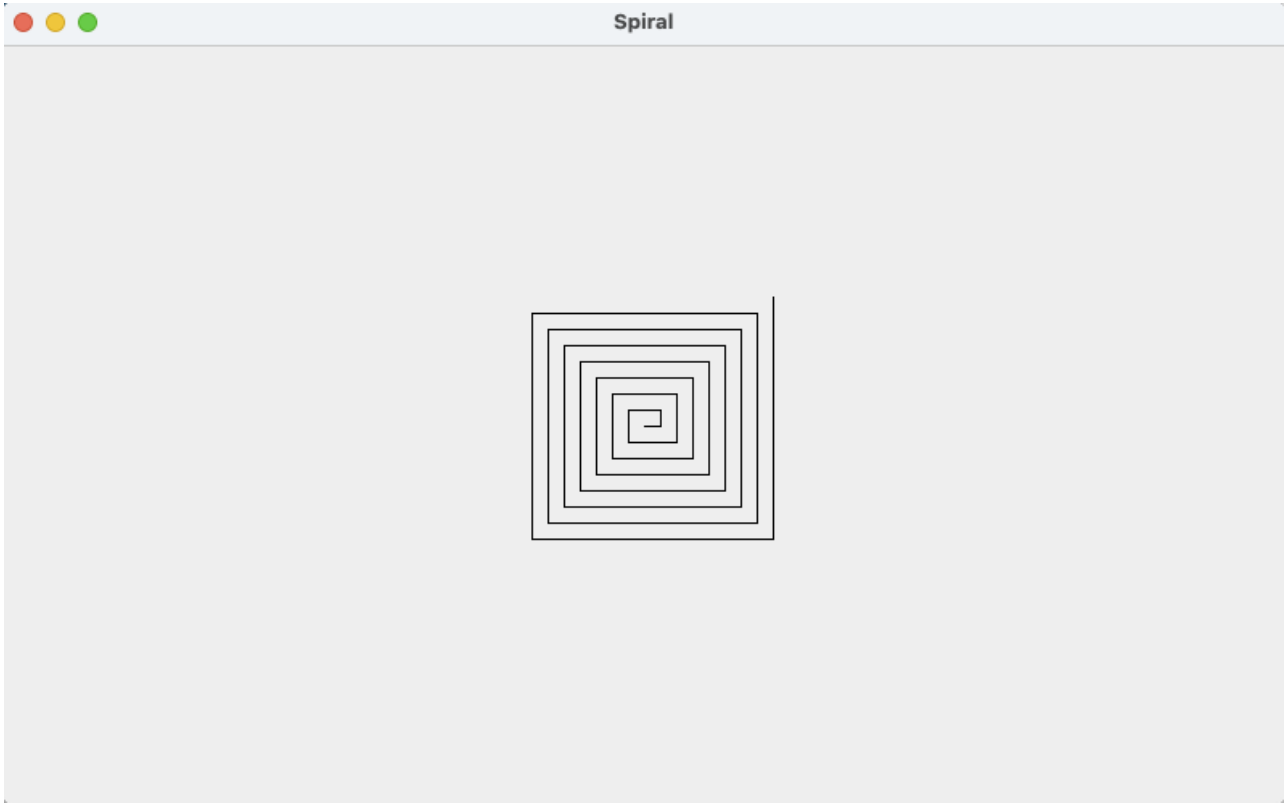
For more information about the classes mentioned above see the section on class design.

The assignment

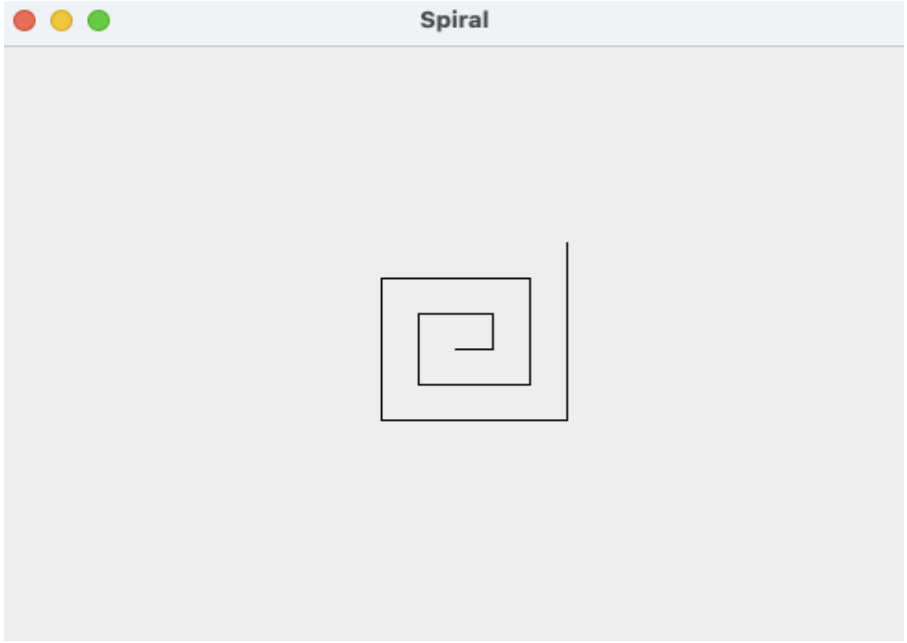
Initially your program will do some I/O at the console, described here. It will prompt for the length of the initial (i.e., smallest) segment in the spiral, error checking that a positive value is entered. Then it will prompt for the number of line segments to display in the spiral, also error checking that a positive value is entered. (More details about error checking here.) This part of the program will be console-based, to keep things simpler.

Then it will display a 500 tall by 800 wide pixel window with the spiral displayed on it, such that the start of the first segment is centered in the window.

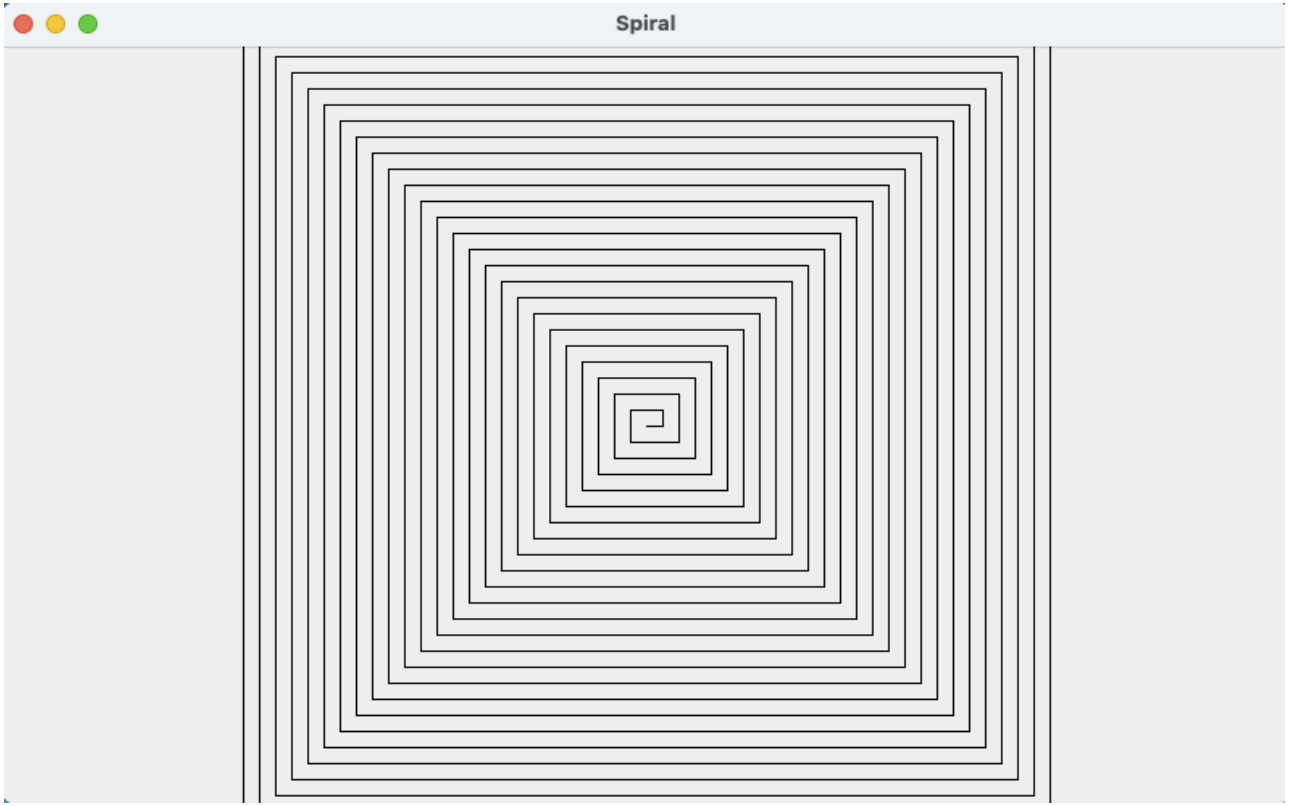
Here is a screen-shot of output from one run of our solution to this assignment, where our responses to the console prompts were to use a 10-pixel initial segment length, and draw a 30-segment spiral (shown in the (500, 800 window)).



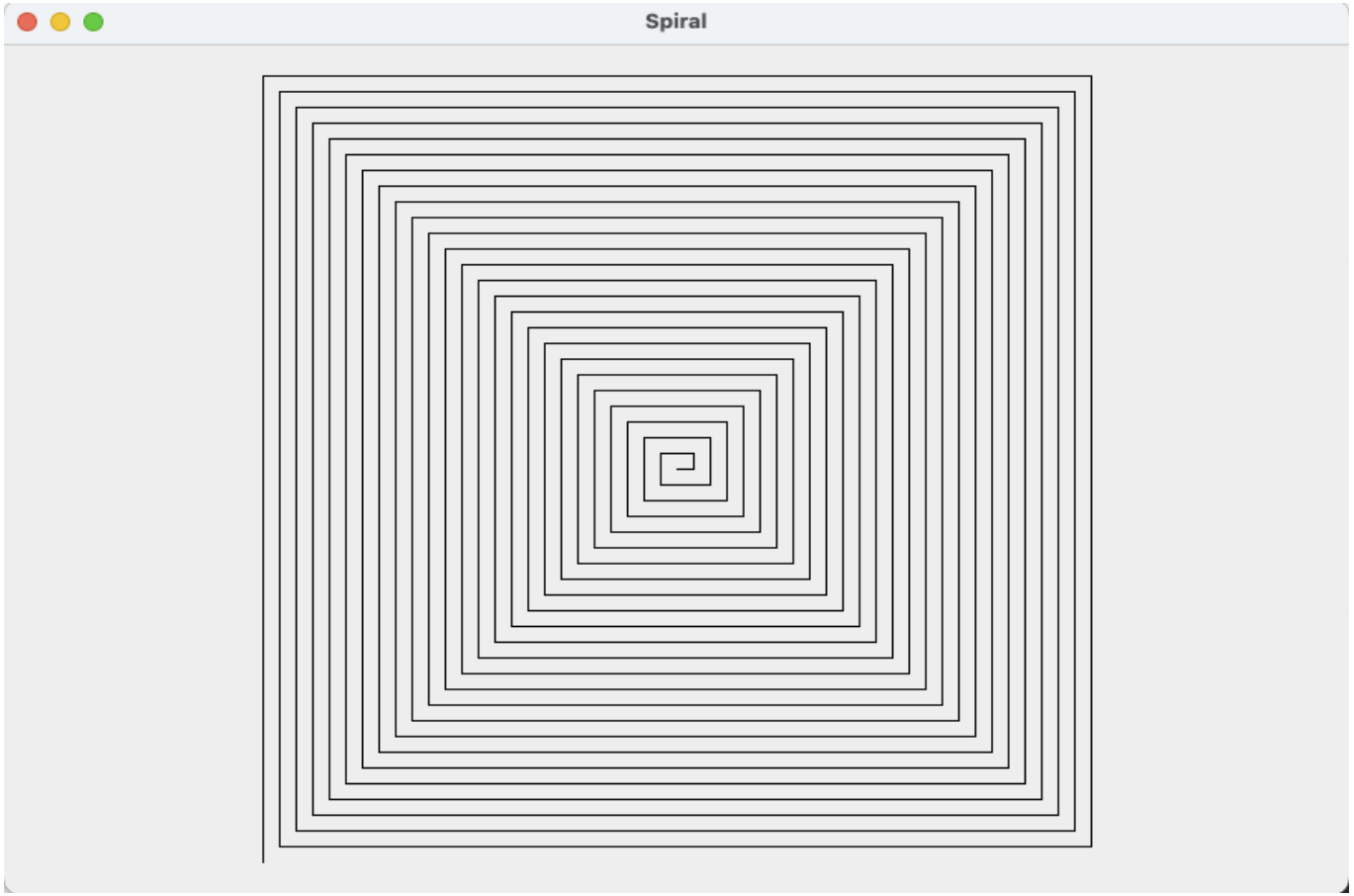
Here's a different spiral to make it easier to see how each segment is arranged (initial segment = 20, number of segments = 10). Also, I shrunk the window before doing this screenshot, so it would be a bigger part of the window.



If some of the segments to be displayed based on the input values given would go outside the window, that part of the spiral will not be visible (unless you resize the window manually as you are running the program): this is the expected behavior. For example, here's a spiral with initial segment length of 10, and number of segments = 100, shown without resizing the window:



As mentioned in the textbook, every time a window gets resized or iconified and de-iconified `paintComponent` gets called again. Here's a later screen-shot created during the same run shown above, but after the user dragged the corner to resize the window to show the rest of the spiral. Note that the spiral is centered in the new window dimensions.



More about the graphics library methods necessary to get these results in the section on Graphics programming.

There are a few other requirements for the assignment discussed in the following sections. To summarize here, the other requirements are:

- you must create and use the classes mentioned in the section on Class design.
- you must create a working unit test program for your `SpiralGenerator` class, the requirements of which are discussed in the section on Testing the `SpiralGenerator` class.
- you must edit and submit the README file discussed in the section about that.
- your program will also be evaluated on style and documentation. More about this in the section on grading criteria.

More details of the error-checking

As mentioned in the earlier section, when your program prompts for the length of the initial segment, and also when it prompts for the number of segments to display, you will error check that a positive value is entered. More specifically, we mean that on an invalid value the program will print out an informative error message and then prompt and read again until the user enters a valid value. Example with the segment length prompt (user

input shown in italics):

```
Enter length of initial segment: -5
ERROR: value must be > 0
Enter length of initial segment: -1
ERROR: value must be > 0
Enter length of initial segment: 0
ERROR: value must be > 0
Enter length of initial segment: 5
```

Your program does not have to handle non-numeric input. (We will not test it on that case.)

Class design

To help you make your program object-oriented, we are giving you the general class design for this program. You are required to use the following classes (ones in bold are ones you will be creating yourself or implementing):

- **`SpiralViewer`**. Contains the `main` method. Prompts for the initial segment length and number of segments, and creates the `JFrame` containing the `SpiralComponent`. Besides `SpiralComponent`, this class does not depend on any of the other classes mentioned here (e.g., if one of those other classes changed, `SpiralViewer` would not have to change.) The later section on communicating information between objects will be useful when developing this and the next class listed. We recommend you use `CarViewer.java` as a starting point for the code in your viewer (more about that example in the next section.)

- **`SpiralComponent`**. Extends `JComponent`. Constructor initializes any necessary data. Overrides `paintComponent` to draw the spiral, using a `SpiralGenerator` object to generate the segments in the spiral to draw. Here also you can also use the component class from the car example as a starting point.

- **`SpiralGenerator`**. A spiral generator can generate segments in a rectangular spiral, one by one, starting from the center of the spiral and moving outward. It has a constructor that has parameters for the location of the start of the initial segment, and the length of the initial segment, which is also known as the "unit length", because the length of the other segments are all in relation to the length of this segment. It has one other method, `nextSegment` which returns the next segment in the spiral. *This class does no I/O: neither to the console nor to the graphics window.*

  We are giving you the exact interface to use for this class. By *interface*, we mean what clients need to know about the class to use it, i.e., the class comment, the public method headers and associated method comments. Do not change the public interface when you incorporate it into your own program. We have provided skeleton code for `SpiralGenerator.java`. Please read the class and method comments in that file for its exact specification.

  Note: For all work in this class, when we say do not change the interface for a class, we mean no changing the class name or the provided method headers, no adding public methods, no removing public methods. As part of the grading process, we will be using our own test programs with such classes, and if you change the public interface your code might not even compile with our test programs.

`SpiralGenerator` does not depend on any of the other classes you are creating here. It does depend on the Java library classes `Point` and `Line2D`, see the section on lines and points for more about these classes.

- **`SpiralGeneratorTester`**. A program to test your `SpiralGenerator` class independently from its use in the `SpiralViewer` program. It will have its own `main` method. This class is described in more detail in the section on testing the `SpiralGenerator` class.

Note: this list doesn't include all the java library classes that will be used in the program; for example `SpiralComponent` will need `java.awt.Graphics`.

The Car example
Our program follows the conventions of graphical classes used in the textbook (see Resources, near the beginning of this document, for relevant textbook readings). In particular, the design for this assignment overall follows the car example from Section 3.8 of the textbook that has a viewer and a component. One difference is the car example also has a graphical object that can get instantiated multiple times and drawn in different locations on the screen (the `Car` class). We have no such analogous class in our design. You will do all the drawing in the `SpiralComponent` class itself.

We provided you with the source code for that example as part of the starter files for this assignment. You can use the code in CarViewer.java as a starting point for your SpiralViewer class for this assignment.

In addition to examining the general structure of the car example code, you can use it to test out running a GUI program in the Vocareum virtual Linux Desktop before developing your own code. When you run it there, you can also see how the display changes when you change the size of the window in which the CarViewer application is running and the corresponding code that gets that to happen (the display for *your* program will also change when the window is resized).

We also modified CarComponent.java a little bit from the version of the code from the text: we instrumented the code so it prints a message to the console every time paintComponent gets called, to help you better understand when the Java Swing graphics framework calls paintComponent. To see this, once you start running the program, make sure you can see the terminal window, as well as the CarComponent window; then try resizing the CarComponent window, and minimizing it and opening it again.

Incremental development and unit testing

Any program of non-trivial size will be developed faster, with fewer bugs, using the technique of incremental development, which means developing, and testing, pieces of the program incrementally. The incremental aspect is that your program may gradually grow until it includes the complete functionality. (Other people use different names for the same thing. Sometimes it's called building subsets.)

A desirable feature of individual classes is that they are as independent as possible from a program that uses them. Some classes, such as `String`, or `ArrayList` (which we will see soon) are general-purpose and can be used in many different programs. Other classes are more special-purpose, such as `SpiralGenerator`, but still are modules that can be separated from a particular program that uses them. We can test such a module using a unit-test, which is a program specially designed to test the module.

We often unit-test one (or more) classes, and then once we are convinced that unit is working correctly, we can integrate that class with other code that uses it. If this larger code base is now buggy, we can feel fairly certain that the bug is in the new code we added, since we already tested the first class. So any time we find bugs, it's in a small program: much easier than locating bugs in large programs.

Similarly, if we make *later* changes or enhancements to our application our code will be more robust in the face of these changes because, in our unit-test, we tested the module in ways not specific to how it was used in this application. (As you have experienced as a user, software is always getting changed over time, e.g., the latest version of Windows is in the double-digits.) For example, in our `SpiralViewer` we would not be able to see if we had the correct results for spirals that start using negative coordinates.

For this assignment, the final product will not be a very large program, but we want to get you in the practice of using incremental development, so you will still be successful when you are trying to develop and debug much larger programs. Even in this program there are at least two distinct issues to deal with: (1) figuring out how to create the correct sequence of segments (2) figuring out how to do a graphics-based program in Java. It will be much easier you we can deal with these issues one at a time, so you can isolate bugs related to each one more easily. For this assignment you are required to write a console-based Tester class to test your SpiralGenerator class. This test program is described in more detail in the next section.

Testing the `SpiralGenerator` class

You are actually going to submit two programs for this assignment, both of which use your `SpiralGenerator` class either directly or indirectly. One is `SpiralViewer`, described earlier, that has a graphical display. The other is a console-based program, `SpiralGeneratorTester`, expressly written to test your `SpiralGenerator` class, without including the drawing functionality of the `SpiralViewer` program. The rationale for unit tests was discussed in the previous section. First, here's more information about compiling Java code:

How to compile and run multi-file Java programs on the command line Often you can just list the file that contains `main` in the compile command and `javac` figures out what other classes are used in that program and compiles those as well. However, sometimes the Java compiler gets confused when you only have modified some of the source files since the original compile. For *running* a program that uses multiple class files, the only class name you give as the argument to the `java` virtual machine is the one containing `main`.

When you are compiling and running your test program you should be able to do it as follows:

```
javac SpiralGenerator*.java
java SpiralGeneratorTester
```

The wild-card ("*" symbol) in the compile command will match the two files `SpiralGeneratorTester.java` and `SpiralGenerator.java`.

For the larger program we are doing for this assignment (for that one `main` is in `SpiralViewer.java`), you can either list all of the files it uses on the command line or use the following convenient shorthand:

```
javac @SpiralViewer.list
java SpiralViewer
```

The `SpiralViewer.list` file (one of your starter files) just consists of the list of files to compile for the program. The `@` on the command line tells java to look in the file that follows it to find out what java files to compile. An alternate is to use `*.java` instead in the compile command, although that one would also attempt to compile `SpiralGeneratorTester.java` as well as the code for the car example.

As mentioned in the previous section a test program like `SpiralGeneratorTester` is called a unit test; we have discussed such unit tests in lecture, and they are also discussed in Section 3.4 of the textbook. One goal of this test program is for you to test the functionality of the SpiralGenerator class.

So, what should you put in your `SpiralGeneratorTester`? This will be a console-based program -- i.e., no GUI. It will be a *non-interactive program* (i.e., fixed data, nothing read in from the user), that tests every method multiple times, printing informative output to the console with the results of each operation.

For a `SpiralGenerator` with given parameters you can predetermine what all the output will be. But instead of comparing with reference output (that you'd likely have to compute by hand), in our tester we're going to just test a subset of what has to be true for a spiral.

Here are the requirements for your test program:

- test creating and using various SpiralGenerator objects with different parameters, printing out information about that SpiralGenerator object, and printing out each segment generated (see section on printing lines and points for more information). There's an example of what your output should look like below (it doesn't show all the output from a test program run).
- for each of these objects, test all the methods multiple times.
- include boundary cases in your tests
- for each segment generated, test that it is horizontal or vertical (e.g., a segment that goes at a 45 degree angle would fail this test)
- and that for each pair of two sequential segments generated:
    - they are connected, i.e., they share an endpoint
    - and they are perpendicular to each other (note, you can assume that you have already tested horiz / vert, so you do not need to test general perpendicularity)
- use the exact output format shown below

For the tests on the segments generated and pairs of segments generated, if one of the conditions isn't satisfied, print an informative failure message about how it failed. Here is an example of partial output from a SpiralGeneratorTester running on a SpiralGenerator that is not currently working correctly (i.e., it fails some of the tests):

```
. . .
Making a spiral starting from java.awt.Point[x=200,y=300]
with a unit length of 5, and made up of 10 segments:
Segment #1: Point2D.Double[200.0, 300.0] Point2D.Double[205.0, 300.0]
Segment #2: Point2D.Double[205.0, 299.0] Point2D.Double[205.0, 294.0]
FAILED: last two segments are not connected
. . .
Segment #4: Point2D.Double[195.0, 295.0] Point2D.Double[205.0, 305.0]
FAILED: segment is not horizontal or vertical.  Skipping pair tests.
. . .
Segment #6: Point2D.Double[210.0, 305.0] Point2D.Double[210.0, 290.0]
Segment #7: Point2D.Double[210.0, 290.0] Point2D.Double[210.0, 305.0]
FAILED: last two segments are not perpendicular
. . .
```

Any time a spiral generator fails a test, print out an informative error message, but keep running the rest of your tests. If a pair of segments fails multiple conditions, print message about all the conditions it fails. However, you can skip pair tests if the segment generated is not horizontal or vertical. We will test your SpiralGeneratorTester with our own *buggy* SpiralGenerator classes.

Note: When you test a method such as the `SpiralGenerator` constructor, which has a restriction on its parameter (in this case the restriction is that the `unitLength` must be greater or equal to one) it means that the behavior of the method is undefined if that precondition is not met. Your test program should not attempt any tests that violate preconditions on methods.

Hints on graphics programming

The graphics primitives you will need for this program are covered in the graphics sections at the end of Chapters 2 and 3 of the textbook, except for a few things we will discuss here. You will not need to go hunting through the online documentation or random web sites to figure out how to do the necessary drawing. More specifically: how to draw a line is shown in section 2.10.2, and there is a concrete example of drawing lines alien face example in textbook section 2.10.4; we also provided the code from another example from the textbook (see the Car example section for more on that.)

The `JComponent` methods, `getWidth()` and `getHeight()`, which get the width and height of the component, will come in handy here. Since `SpiralComponent` is a subclass of `JComponent` you can directly call those methods from your component object. For an example of such calls, see the CarComponent class included in the starter code (and discussed further here).

Lines and Points

As discussed in the textbook (and in lecture, with respect to the similar `Rectangle` class), the shape classes in the java graphics library do not have drawing capabilities themselves, but just store data about something you'll be drawing. We'll be using lines and points here, but there are several line and point classes that we'll try to disentangle here:

- When you see the types `Line2D` and `Point2D`, these are "abstract" classes, which means they are sort of placeholders for concrete subclasses. You are not allowed to create instances of abstract classes directly (they don't have constructors).
- The concrete subclass of `Line2D` that we are using is `Line2D.Double`. When you *create* a line object (e.g., inside `nextSegment`), you will have to use the concrete type, but because it's a subclass of Line2D, nextSegment can use the return type `Line2D`. This is because of the substitution rule for subclasses: we can always substitute a subclass object where some code asks for the superclass type.
- The concrete subclass of `Point2D` we are using is `Point`. (e.g, one of the parameters to the `SpiralGenerator` constructor). Another such subclass is `Point2D.Double`.

Printing lines and points

Most Java classes have a `toString` method defined that returns a String form of the object, useful for printing its contents; even better, if we stick an object that has `toString` defined in a print or println statement, this toString method will get implicitly called. However, for some reason `Line2D.Double` does *not* have a toString that does what we want, so if you want to print a line, you'll have to print the points it contains. Here's some

example code to print a segment returned by our `nextSegment()` method. `getP1` and `getP2` are `Line2D` methods (which means they also work for `Line2D.Double`).

```
Line2D currSeg = gen.nextSegment();
System.out.println(currSeg.getP1() + " " + currSeg.getP2());
```

Here's some sample output from that statement:

```
Point2D.Double[215.0, 310.0] Point2D.Double[215.0, 285.0]
```

How to communicate information between objects

There are several techniques to communicate information between classes and methods of classes, including via parameters and return values of methods. In particular, here you have the issue of receiving some information in `main` in `SpiralViewer`, that is, the initial segment length to use and the number of segments to draw, but needing to use that information in the component. To do this, your `SpiralComponent` class will need to have its own constructor (Note: this is different than the simpler component examples in the book). From `main` you can pass the information to that constructor, and then, if you also needed access to it in other methods, you would save it in an instance variable.

Recall that you never will be calling `paintComponent` yourself, nor are you allowed to change the parameters to it.

README file

For this and all other programs you will be required to submit a text file called `README` with your assignment. In it you will initial the certification we mentioned earlier. This is also the place to document known bugs in your program. That means you should describe thoroughly any test cases that fail for the program you are submitting. (Not your bug history -- just info about the version you are submitting.) You should also document here what subset your solution implements if you weren't able to complete the whole program (more about that in the next section). If you used any outside allowable resources for your code you give your references for that here too (it is not necessary include references to the Ch. 2 and 3 graphics examples). You can also use the `README` to give the grader any other special information, such as if there is some special way to compile or run your program (this would be unusual for students who complete the assignment).

Grading criteria

This program will be graded based on correctness, style, and testing. Programs that do not compile will get little or no credit. However, an incomplete program can get some correctness points if it has partial functionality (you document the partial functionality in the `README`, discussed above). This grading policy is to encourage frequent testing of subsets (discussed earlier in the section on incremental development). Also, for incomplete programs, the style score will be scaled according to how much is completed.

We have published a more complete set of style guidelines for the course on the assignments page, but here are a few things to pay particular attention to for this program:

- documentation. You need to supply an overall comment for each class, and detailed comments about the interface of each method (so called method comments). For the main program you need to supply a comment describing what the program does, and how to run it: this would go above the main class definition. (We have already provided the class and method comments for the `SpiralGenerator` class since we specified that interfaces for you.) This was described in more detail in lecture and the textbook (see Section 3.2.4). Use in-line comments where necessary to explain any confusing code. "This is a for loop" type comments are just clutter, and not useful).
- named constants. There are some numbers mentioned in the assignment description as well as other values that are described to be "fixed" in the program. Each of these should be given a descriptive name in the program so it would be easy to change the value later. Named constants in Java are discussed in section 4.1.2 and programming tip 4.1 in the textbook.
- private data. You should never need public data. Rather, clients should only be able to access data through methods. The rationale for this is discussed in section 3.1.3 of the textbook and in lecture.
- good identifier names. Use descriptive names for variables, parameters, and methods. Also use Java naming conventions. Details in item 8 of course style guidelines. Sections 2.2.3 and 4.1.2 of the textbook discuss more about naming.
- good/consistent indenting. Use the conventions from the textbook or lecture.

For this program only, you do not have to worry too much about method length (guideline #9), and while you should document any instance variables that are not obvious from their names, you do not have to worry about representation invariants (item #17) -- we will be discussing those later this semester.

Implementing the required class design will also be part of your style/documentation score.

How to turn in your assignment

Make sure your name, NetID, course, assignment, and semester are at the top of each file you submit (for source files, they would be inside of comments) for any assignment you submit for this course. You will lose a point on any assignment for which this information is missing. Note: your NetID is the part of your USC email address before the @

The files you need to submit are the ones shown in bold in the earlier section on assignment files.

No matter where/how you developed the code, we will be grading it on Vocareum using the java compiler and virtual machine there (Version 1.8, aka Java 8).

If you developed your program outside of Vocareum, for example, using Eclipse on your laptop, you'll need to upload your code to Vocareum and recompile and retest it completely before you submit it. *Do not wait until the final due date/time is imminent before testing it on Vocareum*. Please read the earlier section on using another IDE for more details on this.

How to submit your program When you are ready to submit the assignment press the big "Submit" button in your PA1 Vocareum work area. *Do not wait until the final due date/time is imminent before attempting to submit for the first time*. You are allowed to submit as many times as you like, but we will only grade the last one submitted.

What happens when you click submit. Vocareum will check that you have the correct files in your work area and whether they compile. It will also check whether the segments generated by your `SpiralGenerator` class are correct for a very small spiral. *Passing* these submit checks is not

necessary or sufficient to submit your code (the graders will get a copy of what you submitted either way). (It *would* be necessary but not sufficient for getting full credit.) However, if your final submitted code does not pass all the tests we would expect that you would include some explanation of that in your README. One situation where it might fail would be if you only completed a subset of the assignment (and your README should document what subset you completed.)

The results of the submit checks will appear on your terminal window. You can also access them by going to the "Details" menu, and choosing "View Submission Report". (Like we did on Lab 1.) Please read this report to see if you passed the tests, so you can fix your program and resubmit if necessary. Make sure you scroll down to the bottom of the report so you don't miss anything.

If you are unsure of whether you submitted the right version, there's a way to view the contents of your last submit in Vocareum after the fact: see the item in the file list on the left called "Latest Submission".
