intro2cs/introcsp/introcst

Ex4: Object Based Programming

In this exercise you will practice implementing classes according to given APIs, and using these classes and other classes provided for you.

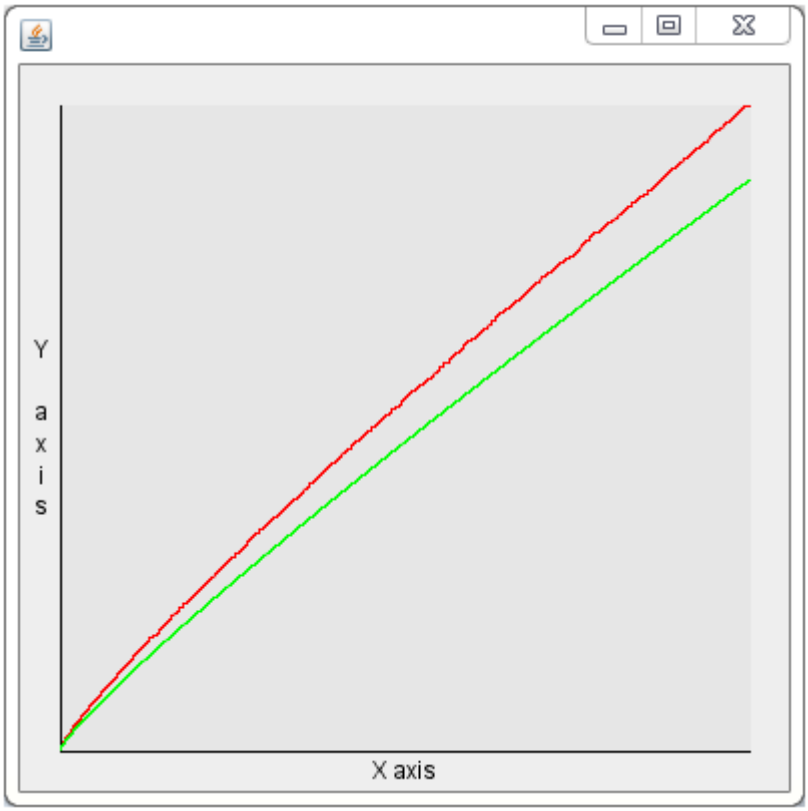

Figure 1: Expected ex4 output for n=10,000

Part 1: Implementing classes

The first part of the exercise requires implementing two classes: the Point class, which represents a two dimensional point, and the PrimesEnumerator, which is a class that can perform calculations on prime numbers. You should implement both classes according to the given APIs.

Part 2: Visualizing the density of prime numbers

In this part of the exercise you will use the two classes that you implemented in Part 1 of the exercise, and another class provided by us, to visualize the density of prime numbers. The class you are provided with is Plotter, which allows you to plot curves to a graphical interface. You will implement the class PrimesDensityVisualizer, which plots a graph with two curves. One of them will be π(x), that is, the number of primes smaller than or equal to x, and the other will be the approximation of π(x), given by the formula x/ln(x). For example, π(2)=1, π(11)=5, while the approximation for π(2) and π(11) is roughly 2.885 and 4.587, respectively.

The colors used to plot the functions will be the constants of the java.awt.Color class: `java.awt.Color.green` and `java.awt.Color.red`. The function π(x) will be plotted in red, and the approximation function to π(x) in green. Each of the two curves will contain values for x in the range [2, number] where "number" is the argument provided to your program (see below for details about command-line arguments).

As a side note regarding the reason for this range requirement, note that ln(1)=0 and therefore the approximation function is not well defined for this case. For the curious among us, further explanations regarding these functions may be found on the relevant Wikipedia page.

It is important to note that π(x) is only defined on natural numbers, and therefore ideally it should be presented as a dot-graph or as bars. However, for aesthetic reasons, in this exercise we shall perform a linear interpolation between the values of π(x) and π(x+1) for all x, and visually present the graph as a continuous line. This task is taken care of by the Plotter class, which is given a sequence of points (by you) which are then processed to present a continuous graph on the screen.

In addition, even though the approximation function is continuous (unlike π(x)), we require that it too will be calculated only on natural numbers, and a linear interpolation will be performed here as well by the Plotter class. For large enough numbers of points, the interpolated graph looks almost identical to a "real" plot of the approximation function, depending on the computer screen's pixel resolution.

The class PrimesDensityVisualizer will contain at least the following methods: **`drawPrimesCountingFunction, numberOfPrimesBelow, approximateNumOfPrimesBelow,`** as well as a **`main`** method. Please refer to the provided Javadoc for exact definitions of the first three methods.

To run the program, a user will pass a single argument to it from the command-line. This is done by appending a space followed by a number string when running the program. For example, a user might run:
`java PrimesDensityVisualizer 1000`
In this case, we say that the argument received is 1000. We will learn more about this in the course soon. For now, it suffices to say that the argument given will be passed to the **`main`** method via the **`String[] args`** parameter. The first (and only in this case) argument is contained in the String **`args[0]`**. The function **`Integer.parseInt()`** can be used to convert this String into an integer, as follows: **`Integer.parseInt(args[0])`** .
In this exercise you may safely assume that when we run our tests, a single parameter that contains a number string is passed to your program, and that the call to **`Integer.parseInt(args[0])`** will not fail. However, you must verify that the returned input number belongs in the legal expected range of numbers for the program.

The **`main`** method expects to receive exactly one argument, a number n. The function π and its approximation will be plotted for all natural numbers in the (inclusive) range [2,n]. Upon encountering illegal input (i.e. if the argument is less than 2), the program will output the following string and terminate:
`Usage: PrimesDensityVisualizer x, where x is a natural number greater than 1.`
In order to exit, please use **`return`** (and **not** **`System.exit()`**).

Please make sure that your output for n=10,000 looks like the figure that appears in this document.

Notes

In your code, the usage of any form of tables (arrays, etc.) that contain constant pre-calculated prime numbers or approximations of prime numbers is **strictly prohibited**. All values must be calculated by your code during run-time. Do not try and fool the automatic tests; remember that your code is inspected manually as well. Any violation of this rule will result in severe point loss, up to possible disqualification of your exercise.

When trying to parse the command-line argument, you may see “index out of range” exceptions or “number format exception” being thrown and crashing your program. This might be the result of forgetting to pass an argument to the program, or trying to call **`Integer.parseInt()`** on an argument string that cannot be parsed as an integer. If you see any of these two errors, make sure that you are running the program correctly. Needless to say, when given proper correct input (i.e. a single parameter that can be parsed as an integer), your program must never crash!

In order to allow us to test your code thoroughly, we require that the class **`PrimesDensityVisualizer`** will contain a private static **`Plotter`** data member. We also require that a public static function called **`setPlotter()`** is defined, and this way our tester

can use a specialized testing-Plotter class. The exact description of how the variable and function should be defined is given in the exercise Javadoc, as part of the **`setPlotter()`** function description. Please pay close attention to this requirement: **do not create new `Plotter` instances in your code**, instead use this single static `Plotter` instance.

In your code, don't forget to call **`plotter.openWindow()`** in order to show the plot window.

The Plotter implementation can be found here. In order to use the Plotter class you need to extract this file's contents next to your code and compile your code together with the Plotter source code. Note that Plotter is not included in the intro.jar file, so you'll have to get it from the link above. Please do not submit Plotter.java as part of your solution!

Documentation

In this exercise you are requested to use Javadoc when documenting your code. Please do not submit the html files.

School solution

An executable version of the school solution can be invoked from the lab computers by typing: `~introcsp/bin/ex4/primesDensityVisualizer x` , where x is a natural number greater than 1.

Testing

It is recommended to check your JAR file by copying it to an empty directory, and running the automatic testers by executing the command: `~introcsp/bin/testers/ex4 ex4.jar` and verifying that all the tests pass successfully.
You may also download the tests JAR from here. Extract the contents of the JAR file using the shell command: `jar xvf ex4testing.jar` and follow the instructions in the file called **`TESTING.`**

Submission

Before you submit, remember to use constants where appropriate and refrain from duplicating code. Make sure to thoroughly test all the classes you implemented.
Don't forget to document your code (including javadoc) and follow all course coding style guidelines.

Submit a JAR file called **`ex4.jar`** containing **only** the files **`Point.java`**, **`PrimesEnumerator.java`**, **`PrimesDensityVisualizer.java`** and **`README.`**

Food for Thought (Completely Optional)

The following is **not** needed to get a full score, will **not** give you any bonus, is **not** needed for the test, but will enrich your knowledge.

1. Add a private field to **`PrimesEnumerator`** that counts the number of times **`isPrime`** has been called so far, and a public method named **`getNumberOfIsPrimeCalls`** that retrieves the value of this field.
2. At the end of your main function, print the return value of **`PrimesEnuemrator.getNumberOfIsPrimeCalls()`**.
3. How many times is **`PrimesEnumerator.isPrime`** called when running your program with n=10? How about n=100? n=1000? Can you explain your findings?
4. Can you think of a way to implement **`PrimesDensityVisualizer.drawPrimesCountingFunction`** that would cause no more than 10 calls to **`PrimesEnuemrator.isPrime`** when your program is run with n=10, no more than a 100 calls when being called with n=100, more than a 1000 calls when being called with n=1000, etc.?