This programming assignment will be due at the beginning of class on Monday, September 24.

To submit your work log in on MathLAN and open a terminal window, create a new directory within your home directory, copy or move the files you want to submit into that directory, and then run the command

/home/reseda/executables/submit-207 directory-name

putting the name of the directory you created in place of directory-name.

The sieve of Eratosthenes is an algorithm for identifying prime numbers by generating the integers, in ascending order beginning with 2, and then repeatedly marking the least previously unmarked number as prime and marking all of its subsequent multiples as composite. The first pass marks 2 as prime and 4, 6, 8, and all the larger even numbers as composite. The least unmarked number is 3, so on the second pass we mark it as prime and 6, 9, 12, and so on as composite. Now the least unmarked number is 5, so on the third pass we mark it as prime and 10, 15, 20, and so on as composite, and so on. Every prime number eventually appears as the least unmarked one, and every composite is marked as soon as its least prime factor is encountered.

This is a very widely known algorithm. Many Java implementations are available on the World Wide Web, including some that run right in the browser window. You can download and study them, if you like.

The Java implementations that I’ve seen, however, use some kind of a data structure to retain the generated integers, and don’t really reflect or take advantage of the object-oriented model. Since we haven’t yet studied Java’s data structures and could use some practice with the basics of objects and message-passing, we’ll implement this algorithm in a different way in this exercise.

Let’s call the objects that your program will use sieves. The purpose of each sieve is to filter out the multiples of some specified number; for instance, once the program gets rolling, there will be a 2-sieve that filters out even numbers, a 3-sieve that filters out multiples of 3, a 5-sieve that filters out multiples of 5, and so on. Each of these will be a different object, but they will all be instances of the class Sieve.

Here’s how the filtering will work. Each sieve will store the number whose multiples it is trying to block in a field called factor. The Sieve class will support a method called next, which takes no arguments. Each time it is invoked, next will return an integer, one that is guaranteed not to be divisible by factor.

How does next compute the integers that it returns? It sends a message to another Sieve, asking it for a “release candidate.” If the candidate is not divisible by factor, next returns it immediately; otherwise, it asks for another candidate, and so on. (Actually, it turns out that no more than two requests are ever needed, although the mathematical reason for this isn’t obvious.)

How does next find the sieve to which it sends its requests? We’ll store a reference to that “source” sieve in another field of the Sieve class. In other words, the Sieve class will be defined recursively: Each sieve will have another sieve (or, more precisely, a reference to another sieve) inside it.

To avoid circularity, this configuration of sieves within sieves must eventually terminate. We manage this by designing one special Sieve, the base sieve, that has a null reference in the field that all other sieves use for their respective source sieves. Instead of applying a divisibility test,
the next method in this base sieve will do the work of generating the integers in ascending order, starting with 2. That is, the first time it is invoked, the next method of the base sieve will return 2; the next time it is invoked, it will return 3; the next time, 4; and so on, however many invocations are needed.

We’ll set things up so that the 2-sieve will have the base sieve as its source and filter out the even numbers from 4 on. The 3-sieve can then have the 2-sieve as its source, and the 5-sieve will have the 3-sieve as its source, and so on, each sieve receiving a sequence of release candidates from which multiples of all smaller factors have already been removed.

Since the base sieve doesn’t apply a divisibility test, its factor field can be used for a different purpose, namely, to keep track of the greatest integer that next has so far returned. In the base sieve, the next procedure should just increment the value stored in this field and then return the result.

Write and compile a Java class definition that implements this design. The class definition will include declarations for the two fields (factor and source), two constructors (one, with zero arguments, for the base sieve and the other, with two arguments, for all other sieves; the two arguments are the number whose multiples are to be filtered and the source sieve from which the release candidates will be drawn), and at least one method, next. The next method should be able to tell whether it is running in the base sieve or in some other sieve by testing whether source is a null reference.

Next, turn this class definition into a program by adding a main method. This method should begin by creating the base sieve and assigning it to a local variable, say sifter. Sending the next message to sifter will yield a prime number. Supply this prime number and the sieve that is currently stored in sifter to the second constructor, which will build a new sieve that filters out multiples of prime. Now assign this new sieve to sifter and repeat the process to get the next prime number.

Arrange for your main program to compute and print out the first five hundred prime numbers, ten to a line, right-justified in fields seven columns wide, and then stop. Compile and execute the program and collect the output in a file. Make sure to submit both the finished version of the .java file and the file containing the collected output.