CSC-223
Lab 7
Work in a group of 2.

Geometry Fact Computer Preprocessor

It is more complex to compute the characteristics of a geometry figure than you might think. Figure 1 shows a geometry figure we have considered when describing several past assignments.

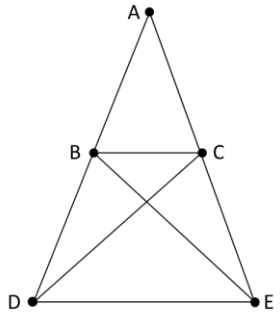

Figure 1: A sample geometry figure.

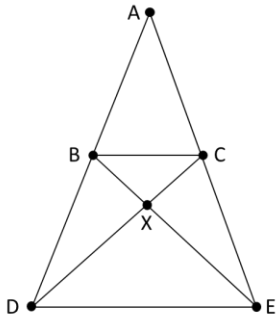

Figure 2: The geometry figure from Figure 1 with labeled intersection point and thus larger set of minimal segments.

Consider the labeled points in Figure 1. We can enumerate the set of all segments between those labeled points:

`AB, AC, BC, BD, BE, CD, CE, DE.`

Now let us label the point at which `BE` and `CD` intersect as `X` as shown in Figure 2. We refer to point `X` as an *implicit point* since there are meaningful geometric components attributed to it (e.g., segments, angles, triangles, pentagons, etc.) even though it is not labeled. It is also the case that a student might label the point en route to a solution of a problem related to the figure. Thus, our complete list of segments for Figure 2 can be enumerated as:

`AB, AC, BC, BD, BX, CX, CE, DX, DE, XE.`

We refer to this list of segments for Figure 2 as the set of *minimal segments.* A minimal segment is a segment that does not contain any other point (explicit or implicit) between its two endpoints.

While minimal segments are descriptive of a figure, it should be clear that Figure 2 contains more than the 10 segments. For example, `AD` and `AE` are valid segments in Figure 2, but are not minimal. Given a complete set of minimal segments, we can then compute all non-minimal segments:

`AB + BD = AD, AC + CE = AE, BX + XE = BE, CX + XD = CD.`

Computing all non-minimal segments is a non-trivial task. Consider Figure 3 which contains 4 minimal segments and 6 non-minimal segments. In this example it should be relatively clear that we can determine the total number of segments in Figure 3 using a combination: $\binom{5}{2} = 10$.

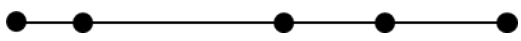

Figure 3: A geometry figure with 4 minimal segments and 6 non-minimal segments.

What You Need to Do: `Preprocessor` and `Segment` Implementations

Considering the JSON representation of the geometry figure in Figure 1, it is clear that we cannot expect input geometry figures to provide the complete set of minimal segments for a geometry figure. Instead, we must compute the set of minimal segments as well as non-minimal segments.

The focus of the `Preprocessor` class will be to compute all segments by performing the following computations in sequence.

1. Compute the set of implied points.
2. Use the implicit points to compute the implicit minimal segments.
3. Identify the complete set of minimal segments (accounting for implicit points).
4. Construct the set of all non-minimal segments (based on minimal segments).
5. Populate a container all non-minimal segments.
6. Populate a container with all segments; this is our 'database'.

This algorithm has been provided in code in the `Preprocessor` class; your task is to implement the missing methods. To organize and isolate functionality, the `ImplicitPointPreprocessor` class should compute the set of the implicit points and incorporate into the project as shown in Figure 4.

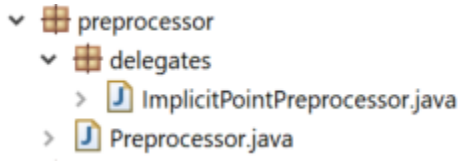

Figure 4: Structure of the preprocessor package.

To facilitate preprocessing, several files have been provided that implement computational geometry methods to determine intersection points: see Figure 5 along with a new utility class (`OutTriple`) in Figure 6. Your preprocessor code will have to interact with the `Segment` class, but will not have to directly interact with any of the delegate computational geometry functionality. Please review the provided `Segment` class as it implements additional functionality not needed in previous assignments; *the* `Segment` *class also requires you complete a few methods.*

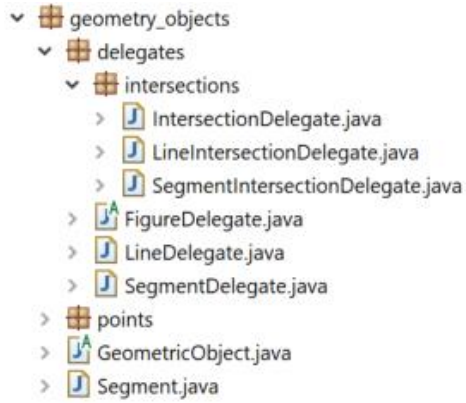

Figure 5: The structure of the provided computational geometry processing delegate files.

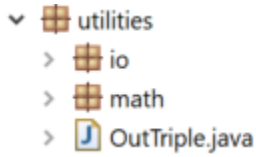

Figure 6: An additional utility class to facilitate returning 3 objects from a method: `OutTriple`.

Testing

Write thorough junit tests for all of your implemented functionality. A large test has been provided in `PreprocessorTest`. Test along the way with smaller, tighter tests.

Commenting

Comment well. See old lab instructions for details.

Submitting: Source Code

For this lab, you will demonstrate your source code to the instructor, in person. Be ready to demonstrate (1) successful execution of all junit tests and (2) the github repository which includes commented source code (see above) and a clear commit history with meaningful commit messages *from all group members*.