CSC-223
Lab 2
Work in a Group of 3

Geometry Fact Computer Input: Points and Segments

This assignment implements scaled-down versions of foundational classes that we will use to represent a geometry figure: points and segments. The classes you will implement in this assignment will fall under the input system of our *geometry fact computer*; see Figure 1 for `src` folder project structure. Your project will have a parallel `test` source folder containing junit tests as well.

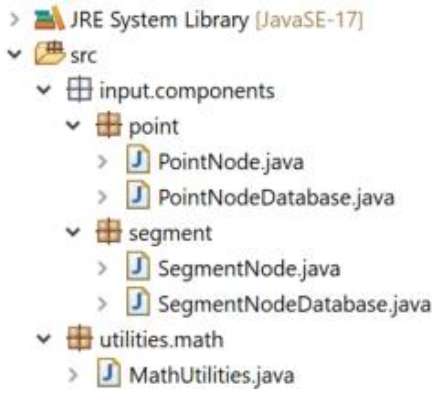

Figure 1: The project structure for this assignment.

What You Need to Do: Project Creation

One group member should create the initial project structure with provided files and make it accessible to the rest of the group via github.

Note that `utilities.MathUtilities` and skeletons of `PointNode and SegmentNode` have been provided and should be incorporated into the project.

What You Need to Do: `PointNode` Class

A point is a two-dimensional geometry object defined by its coordinates `x` and `y`, often denoted as the ordered pair `(x,y)`. A point may also be named. For example, if a geometry figure labels the origin in the Cartesian Plane as `A`, we might denote it is as `A(0, 0)`. If a name is not specified, we use the private `ANONYMOUS` string constant.

Follow the provided skeleton to complete this class.

What You Need to Do: `PointNodeDatabase` Class

The goal of the `PointNodeDatabase` class is to store all input points in a geometry figure by allowing us to (1) add new points to the database as well as (2) look up points by name or by coordinate values.

This 'database' implementation will provide basic read / write capability for `PointNode` objects, but not much more. Underlying our database will be a `Set`, in particular, a `LinkedHashSet` to ensure efficient lookup. (Note how the `PointNode` class overrides the `hashCode` method facilitating use of a hashing structure like `LinkedHashSet`.)

You are to implement the methods listed in the UML class diagram below.

PointNodeDatabase	
#_points	: Set<PointNode>
+PointNodeDatabase()	
+PointNodeDatabase(List<PointNode>)	
+put(PointNode)	: void
+contains(PointNode)	: boolean
+contains(double x, double y)	: boolean
+getName(PointNode)	: String
+getName(double x, double y)	: String
+getPoint(PointNode)	: PointNode
+getPoint(double x, double y)	: PointNode

The `getPoint(PointNode)` method may seem redundant, but its purpose is to acquire a stored database object to mitigate copies of point objects.

What You Need to Do: `SegmentNode` Class

We are using the geometric definition of a line segment: a line bounded by two points (two `PointNode` objects). A partial implementation of the `SegmentNode` class has been provided. You will need to implement an `equals` method; it would also benefit your to implement a `toString` method.

What You Need to Do: `SegmentNodeDatabase` Class

Our segment database is not a traditional database in which we directly store `SegmentNode` objects. Instead, this database will store *adjacency lists* as a map: a fundamental structure when representing a graph. While these terms may sound foreign, you can intuitively understand these concepts by considering the geometry figure in Figure 2.

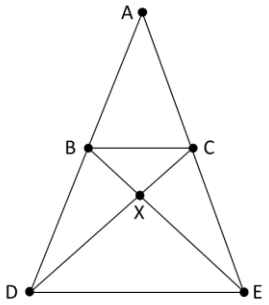

Figure 2: A sample geometry figure.

The goal of our *adjacency map* is to represent all the segments in a geometry figure (edges in an undirected graph). An adjacency map is a dictionary structure in which the key is a point in the geometry figure and the value is a list of points which are connected to the key. For example, point `A` is directly connected to point `B` and point `C`. The corresponding adjacency list for `A` is `A : {B, C}`. The adjacency map corresponding to the geometry figure in Figure 2 is shown in Figure 3.

A : {B, C}
B : {A, C, X, D}
C : {A, B, X, E}
X : {B, C, D, E}
D : {B, X, E}
E : {D, X, C}

Figure 3: The adjacency map corresponding to the geometry figure in Figure 2.

Observe that the geometry figure in Figure 2 contains 10 fundamental segments (`AB`, `AC`, etc.). However, the adjacency map representation in Figure 3 indicates 20 segments because each of the segments are stored two times: once for each of the endpoints.

Using the adjacency map structure discussed above, you are to implement the `SegmentNodeDatabase` class listed in the UML class diagram below.

SegmentNodeDatabase
#_adjLists : Map<PointNode, Set<PointNode>>
+SegmentNodeDatabase()
+SegmentNodeDatabase(Map<PointNode, Set<PointNode>>)
+numUndirectedEdges() : int
-addDirectedEdge(PointNode, PointNode) : void
+addUndirectedEdge(PointNode, PointNode) : void
+addAdjacencyList(PointNode, List<PointNode>) : void
+asSegmentList() : List<SegmentNode>
+asUniqueSegmentList() : List<SegmentNode>

What You Need to Do: Testing

You are to implement a set of unit tests for each method of each class. A build method has been provided in the `SegmentNodeDatabaseTest` class as an example of how to construct a test with a geometric figure: it is a bit painful.

Use the naming and testing paradigm we established in the first lab: our test classes will always be defined in a *parallel* `test` source folder in Eclipse.

When you execute your `junit` tests, ***no output*** should be produced (never print to `System.out` or any other output stream unless it is a file-based logging system); we are seeking only a 'green' output indication in Eclipse. Make sure to use the drop-down menu to show all tests have been executed and are successful.

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*.