CSC-223
Lab 6
Work in a Group of 2 or 3. 3 is preferred.

Geometry Objects and Input Façade

Our goal in this assignment is to transition from the input system to our geometry fact computer for downstream processing.

What You Need to Do: `InputFacade` Implementation

It has taken some effort to implement the input system. With complex, multi-package subsystems, it is often difficult for another programmer to figure out how best to extract necessary information. To provide a clean interface to the input subsystem we will create a façade class. A façade is a *design pattern* in which the goal is to provide a simplified interface to a library, a framework, or any other complex set of classes.

You are required to complete the implementation of the `InputFacade` class according to the Javadoc comments in the provided source code. Observe in Figure 1 that the `InputFacade` class is placed in the input package in a position of prominence: where we might place a *main* procedure for a subsystem.

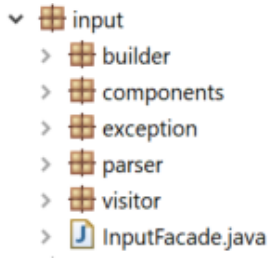

Figure 1: The placement of the input façade implementation in the input package.

Geometry Object Structure

The `InputFacade` class is a bridge to the geometry fact computer which will begin to process and interpret an input geometry figure. Before you can implement the façade, you need to implement some geometry objects that will serve to represent elements (1) we receive from input (e.g., points and segments) and (2) compute downstream (e.g., angles, triangles, etc.). We need to create a high-level package for our geometry objects; this is shown in Figure 1Figure 2.

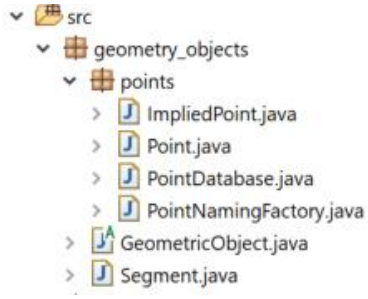

Figure 2: The structure and contents of the geometry objects package.

Currently, the `GeometricObject` abstract class will be empty; this may change in the future. The rest of the classes will be described in turn.

What You Need to Do: `Point` Class

You have already implemented a class to represent a point (`PointNode`). However, the geometric interpretation of a point is significant enough that we want to distinguish the idea of an 'input' point and a

point as a geometric object. While this may seem to lead to redundancy in code, our geometric point implementation (`Point`) will be more extensive than our input-focused implementation (`PointNode`).

Again, a point is a two-dimensional geometry object defined by its coordinates `x` and `y`, often denoted as the ordered pair `(x,y)`. Follow the skeleton including the docstring comments to complete this class. You will notice that `Point` implements the `Comparable` interface. Recall that the `Comparable` interface requires that we implement a `compareTo` method. We will use lexicographic ordering where we first consider the values of `x` coordinates and then the values of the `y` coordinates. Hence, $(2, 5) < (3, 5)$ since $2 < 3$. Also $(0, 10) < (0, 12)$ since $0 = 0$ and $10 < 12$.

For our purposes, points may also be named. For example, if a geometry figure labels the origin as `A`, we might denote it is as `A(0, 0)`. However, a name is not required by the `Point` class (even though the `PointDatabase` will enforce naming). If a name is not specified, we use the private `ANONYMOUS` string constant (as we have done before).

What You Need to Do: `PointDatabase` Class

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

This is similar to the functionality you implemented in the `input` package functionality on a previous assignment; however, our underlying data structure will be more complex. `PointDatabase` is a misnomer because the intent of this class is as a delegator to the real database implementation provided by `PointNamingFactory`. Follow the skeleton for guidance on the desired functionality to be implemented for `PointDatabase`, noting that much of the purported functionality of `PointDatabase` is implemented in the delegate `PointNamingFactory` implementation.

What You Need to Do: `PointNamingFactory` Class

All points in our database must be named. If a name is not provided, the `PointNamingFactory` will generate one. This type of 'factory' structure is a *design pattern* in which we use a consistent, simple interface to generate names *as needed* and not each time a point is encountered. Here is a link to the general idea of a factory method; be aware that our implementation will deviate from a classic implementation.

The underlying data structure we will use to represent our `Point` objects is a `Map` (a.k.a. `Dictionary`). We conveniently implemented such a data structure in a previous assignment. However, in this case, we will use the Java-defined `LinkedHashMap`.

A `Map` is actually an odd data structure to use for our database because a map requires `<Key, Value>` pairs and we only have 'keys' and not corresponding 'values'. It seems like it would be more convenient to use a `Set`. However, `HashSet`, for example, does not define a `get` (method to look up and return an object) whereas the `Map` interface does require an implementing class define `get`. For our implementation, each `<Key, Value>` pair will be such that `Key == Value`; this implies `Key` and `Value` refer to the same object since the code snippet refers to address-based comparison.

Follow the skeleton for guidance on the desired functionality to be implemented for `PointNamingFactory`.

What You Need to Do: Testing

You are to implement appropriate unit tests for:

- `Point`

- `PointDatabase`
- `PointNamingFactory`
- `InputFacade`

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. The template project provided on github already defines this project structure.

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.

What You Need to Do: Review Provided Source Files

Before starting take stock of the files that have been provided. This includes a geometric `Segment` class (to parallel the geometry point class you will complete).

Commenting

Your code should be well documented, including docstring comments of methods, blocks of code, and header comments in each file. Junit tests should have reasonable and *meaningful* messages printed, if failure occurs (e.g., use the string-based assert functions such as `assertTrue(java.lang.String message, boolean condition)`).

Header Comments

Your program must use the following standard comment at the top of *each source code file*. Copy and paste this comment and modify it accordingly with your files.

```
/**
 * Write a succinct, meaningful description of the class here. You should avoid wordiness
 * and redundancy. If necessary, additional paragraphs should be preceded by <p>,
 * the html tag for a new paragraph.
 *
 * <p>Bugs: (a list of bugs and / or other problems)
 *
 * @author <your name>
 * @date   <date of completion>
 */
```

Inline Comments

Comment your code with a *reasonable amount of comments* throughout the program. Each non-obvious method should have a Javadoc comment that includes information about input, output, overall operation of the function, as well as any limitations that might raise exceptions. Javadoc comments can be inherited.

Each *block* of code (3-4 or more lines in sequence) in a function should be commented. Comments go above and before the code they are referencing.

It is ***prohibited*** to use *long* comments to the right of lines of source code; attempt 80 character-wide text in source code files. Succinct comments (a few words at maximum can be fine).

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*.